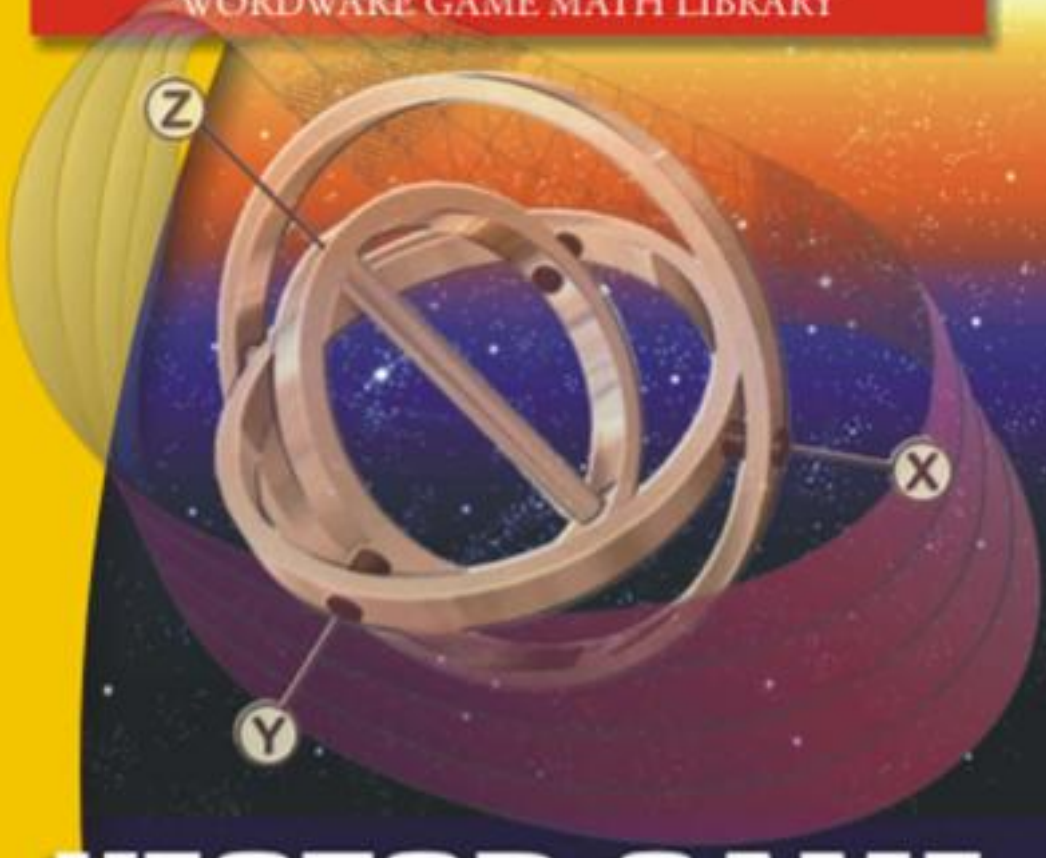
WORDWARE GAME MATH LIBRARY



VECTOR GAME Math Processors

Companion CD-ROM Included

James C. Leiterman

Vector Game Math Processors

James Leiterman

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Leiterman, James.
Vector game math processors / by James Leiterman.
p. cm.
Includes bibliographical references and index.
ISBN 1-55622-921-6
1. Vector processing (Computer science). 2. Computer games--Programming.
3. Supercomputers--Programming. 4. Computer science--Mathematics.
5. Algorithms. I. Title.
QA76.5 .L446 2002
004'.35--dc21 2002014988
CIP

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-921-6

10 9 8 7 6 5 4 3 2 1
0211

All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Contents

Preface

(or, So Why Did He Write This Book?)

All my life I have loved working with numbers, except for that time in high school when I took algebra, but I will not get into that. As you will read near the end of this preface, I have eight children. That is 2^3 kids, not the 2.3 children in a typical-sized family in the United States: the size of a perfect binary cube (2x2x2). Numbers are easy for me to remember, but names are something else. For example, I worked for LucasArts for over four years, and people would always come into my office to ask me to help solve their problem or pass me in the hall with the standard greeting, "Hi Jim!" I would then think to myself, "Who was that?" I would have to go through the company yearbook to figure out who it was.

A portion of this book was originally going to be in an X86 optimization book I had been writing, *X86 Assembly Language Optimization in Computer Games.* It was designed around Intel Pentium processors and all the various Pentium superset instruction sets of that time.

Four years ago, the timing for this book was perfect as the 3DNow! chip had been out for a while and Intel's Katmai chip had not been released. I wrote the first half related to general-purpose programming to near completion, less the floating-point, and had several publishers interested in it; but after several months of review, they all liked the book but passed on it. The typical response was that they still had X86 books in inventory that were not selling. So the only copies of that book in existence are those I gave to my technical friends for review that they refused to give back, as they liked it too much. So I retired the book and moved on to better things. Several years later, I had an idea for the book you are reading and found a publisher that liked both books and had the same insight and vision as me.

Well, the timing was great for this book, and at the time of publication, there were none out there specific to the topic of vector processors. There were a few books on parallel processing, but they typically covered high-performance processors in commercial systems. The PlayStation 2 game console from Sony, Xbox from Microsoft, and GameCube from Nintendo had recently been shipped to consumers. They each contain a vector processor, putting super computer power in the hands of consumers at a reasonably low cost. This opens the door for processor manufacturers to lower their costs and make it reasonable for computer manufacturers to start shipping vector processors with their computers. Of course, this now requires someone to program these things, thus this book!

One last comment: Not everyone will be happy with a result. All programmers have their favorite software development tools, their favorite processor, and their own ideas of how things should be put together. By all means, please let me know what you think (in a nice way). If you want to be "thorough" about it, write your own book.

Writing is hard work. Technical book authors typically spend an extremely large part of their free time writing their books when not doing their regular paid work. They have many sleepless nights so that the book can be published before the information becomes dated and, hence, redundant. Their children and spouse tend to not see much of their resident author and family member either. Authors do it for the fun of it, as well as the name recognition, and in some rare cases, the money.

I wish to thank those who have contributed information, hints, testing time, etc., for this book: Paul Stapley for some console testing and technical overview recommendations; my old-time friend from back in my Atari days, Jack Palevich, for his review of my book and insight into vertex shaders; Bob Alkire and Steve Saunders, also from my Atari days, for their technical check; Wolfgang F. Engel for his technical check of my chapter on the vertex shader; Ken Mayfield for some 3D computer art donations; Michael Robinette for setting up my original Code Warrior development project environment on Macintosh under OS9 and for some G3/G4 testing; Adrian Bourke down under in Australia for some G3/G4 Macintosh testing and OSX usage tips; Matthias Wloka with nVIDIA for some technical vertex shader programming help; John Hogan and Chao-Ying Fu with MIPS for some MIPS V and MIPS-3D coding support; Allan Tajii with Hitachi for some SH4 coding support; Fletcher Dunn for some preliminary technical checking; and others that I have not mentioned here for their contributions.

And most of all, I would like to thank my wife for not balking too much when I bought that new laptop, G4 Macintosh, top-of-the-line video card, and other computer peripherals. Although I should note that every time she discovered a new piece of equipment, her rhetorical question was, "That is the last one, right?"

I finally wish to thank Jim Hill from Wordware Publishing, Inc. for seeing the niche that this book would fill, Wes Beckwith for not asking the question I frequently hear from my children, "Is it done yet? Is it done yet?", and Paula Price for making sure those checks arrived just in time when I needed them.

So get up from the floor or chair in the bookstore in which you are currently reading this book, as you know you will need this book for work. Besides, I filled it with so much stuff you might as well stop copying it into that little notebook, grab a second copy for use at home, walk over to that check stand, and buy them both. Tell your friends how great the book is so they will buy a copy, too! Insist to your employer that the technical book library needs a few copies as well. This book is an instruction manual and a math source library, all rolled into one.

My eight children and outnumbered domestic engineering wife will be thankful that we will be able to afford school clothes as well as Christmas presents this year! Unlike the title of that old movie's implication that kids are *Cheaper by the Dozen*, they are not! They eat us out of house and home!

Keep an eye out for any other book by me because since this one is finished, my focus has been on their completion.

For any updates or code supplements to any of my books, check my web site: http://www.leiterman.com/books.html.

Send any questions or comments to books@leiterman.com.

My brother Ranger Robert Leiterman is the writer of mystery-related nature books that cover diverse topics as natural resources, as well as his Bigfoot mystery series. Buy his books too, especially if you are a game designer and interested in crypto zoology or natural resources. If it was not for him sending me his books to help proofread, I probably would not have started writing my own books (so blame him!).

ISBN: 0595141757

ISBN: 0595203027

He did not implement all my editing recommendations, so do not blame me for any of the grammar problems you find in his book!

Chapter 1
Introduction

Vector math processors have, up until recently, been in the domain of the supercomputer, such as the Cray computers. Computers that have recently joined this realm are the Apple Velocity Engine (AltiVec) coprocessor of the PowerPC G4 in Macintosh and UNIX computers, as well as IBM's Power PC-based Gekko used in the GameCube and Digital Signal Processing Systems (DSP). MIPS processors, such as the Toshiba TX-79, and the Emotion Engine (EE) and Vector Units (VUs) used in the Sony PlayStation 2 are also in this group. The X86 processors, such as Intel's Pentium III used in the Xbox, and all other X86s including the Pentium IV and AMD's 3DNow! extension instructions used in PCs are other recent additions. Both fixed-point as well as floating-point math is being used by the computer, video gaming, and embedded worlds in vector-based operations.

3D graphic rendering hardware has been going through major increases in the numbers of polygons that can be handled by using geometry engines as part of their rendering hardware to accelerate the speed of mathematical calculations. There is also the recent introduction of the programmable vertex and pixel shaders built into newer video cards that use this same vector functionality. These work well for rendering polygons with textures, depth ordering z-buffers or w-buffers, and translucency-controlled alpha channels with lighting, perspective correction, etc. at relatively high rates of speed. The problem is that the burden of all the other 3D processing, culling, transformations, rotations, etc. are put on the computer's central processing unit (CPU), which is needed for artificial intelligence (AI), terrain following, landscape management, property management, sound, etc. Well, you get the idea. For those of you looking for work, keep in mind that this new technology has created a surplus of processor power that is being filled with the new high-growth occupation of AI and physics programmers.

Recent microprocessor architectures have been updated to include vector math functionality, but the processor was limited to small

sequences of a vector math calculation; such implementations include Multimedia Extensions (MMX), AMD's 3DNow! Professional, or the Gekko chip, where only half vectors are dealt with at any one time. These advances, however, have been a boon for engineers on a budget as their vector-based math used in scientific applications can run faster on these newer computers when properly coded due to their vector math ability. The "catch" here is that vector processors have special memory requirements and must use math libraries designed to use that special vector functionality of the processor, not that of the slower standard floating-point unit (FPU), which is still present on the chip. Third-party libraries tend to be biased toward a favorite processor or are just written with generic code and thus will not run efficiently on some processors and/or take advantage of some instruction-based shortcuts.

A full vector processor can be given sequences and arrays of calculations to perform. They typically have their own instruction set devoted to the movement of mathematical values to and from memory, as well as the arithmetic instructions to perform the needed transformations on those values. This allows them to be assigned a mathematical task and thus free the computer system's processor(s) to handle the other running tasks of the application.

The cost of a personal supercomputer was out of range for most consumers until the end of 2000 with the release of the PlayStation 2 console (PS2) by Sony. Rumor has it that if you interconnect multiple PS2 consoles as a cluster, you will have a poor man's supercomputer. In fact, Sony announced that they would be planning to manufacture the "GSCube," a product based upon interconnecting 16 emotion engines and graphic synthesizers.

Actually, if you think about it, it is a pretty cool idea. A low-budget version would mean that each console on a rack boots their cluster CD/DVD with their TCP/IP network connection and optional hard disk, and their network link becomes a cheap mathematical number-crunching supercomputer cluster slave.

The vector processor is the next logical step for the microcomputer used in the home and office, so in this book, we will discuss the instruction sets that they have as well as how to use them. This book is targeted at programmers who are less likely to have access to the expensive supercomputers, but instead have access to licensed console development boxes, console Linux Dev Kits, cheap unauthorized (and possibly illegal) hacker setups, or the new inexpensive embedded DSP vector coprocessors coming out in the market as you read this.

I cannot come out and blab what I know about a proprietary processor used in a particular console as much as I would like to. Although it would definitely increase the sales of this book, it could possibly mess up my developer license. I do discuss some tidbits here and there, utilizing the public domain GNU C Compiler (GCC) and related access to inline assembly. Certain console manufacturers are afraid of hackers developing for their systems and have closed public informational sources, except for some product overviews. But you are in luck! Some engineers love to promote their achievements. For example, technical details related to Sony's PS2 processing power and overview were made public at a 1999 Institute of Electrical and Electronics Engineers (IEEE) International Solid-State Circuits Conference by Sony and Toshiba engineers (TP 15.1 and 2) with slides. In 2001, Sony released to the general public their Japanese Linux Development Kit; the LDK will be released in 2002 for some other countries, of which the EE and VU will be especially useful to you potential PS2 developers. Manuals are listed in the references section at the back of this book.

In addition, game developers often release debug code in their games containing development references, typically due to the haste of last-minute changes prior to shipping. I also lurk on hacker web sites and monitor any hacking breakthrough that may come about. All that information is in a public domain and thus can be selectively discussed in this forum. You will find some references and Internet links at the back of this book.

I have pushed the information written in this book (pretty close) to the edge of that line etched in the sand by those manufacturers. What this book does is pool processor informational resources together from publicly available information, especially when related to consoles.

Hint: AltiVec and 3DNow! are two of the publicly documented instruction sets that have similarities to other unpublished processors.

One thing to keep in mind is that some of the processor manufacturers typically have published technical information for processors that have behaviors and instruction sets similar to those that are proprietary. One of these is Motorola with their AltiVec Velocity Engine, which makes its information freely available. This is a superset of almost all functionality of current consumer-based vector processors.

This is not an AltiVec programming manual, but understanding that particular processor and noting the differences between the other public processors with their vector functionality embedded in their

Multimedia Extensions (MMX), Single Instruction Multiple Data (SIMD) features, and Streaming SIMD Extensions (SSE) gives an excellent insight into the functionality and instruction sets of processors with unpublished technical information. So read this book between the lines.

Another topic covered in this book is vertex and pixel shaders. They are touched on lightly, but the vector math is accented and thus brought to light. The new graphics cards have the programmable graphics processors with vector functionality.

Hint: Check out my web site at http://www.leiterman.com/books.html for additional information, code, links, etc. related to this book.

This book is not going to teach you anything about programming the game consoles due to their proprietary information and the need for one or more technical game development books for each. You need to be an authorized and licensed developer to develop for those closed architectural platforms or have access to a hobbyist development kit, such as the Linux Dev Kit for PS2. The goal of this book is to give you the skills and the insight to program those public, as well as proprietary, platforms using a vector-based mind-set. Once you have mastered a publicly documented instruction set like AltiVec, being afraid of a processor or finding vector processors, such as Sony's VU coprocessor, difficult to program for should be a thing of the past, as it will not seem as complicated and will be a snap!

GOAL

A better understanding of CPU and Graphics Processor Unit (GPU) with vector-based instruction sets!

One other thing to keep in mind is that if you understand this information, it may be easier for you to get a job in the game or embedded software development industry. This is because you will have enhanced your programming foundations and possibly have a leg up on your competition.

That's enough to keep the console manufacturers happy, so let's get to it! I know a number of you like technical books to be like a resource bible, but I hate for assembly books (no matter how detailed) or books of the same orientation to be arranged in that fashion because:

1. It takes me too long to find what I am looking for!
2. They almost always put me to sleep!

Hint: This book is divided into chapters of functionality.

This book is not arranged like a bible. Instead, it is arranged as chapters of functionality. If you want that kind of organization, just look at the

index of this book, scan for the instruction you are looking for, and turn to the page. I program multiple processors in assembly and occasionally have to reach for a book to look up the correct mnemonic—quite often my own books! Manufacturers almost always seem to camouflage them. Depending on the processor, the mnemonics shifting versus rotating can be located all over the place. For example, the x86; {psllw, pslld, psllq, ..., shld, shr, shrd} is a mild case due to the closeness of their spellings, but for Boolean bit logic; {and, ..., or, pand, ..., xor} are all over the place in an alphabetical arrangement. When grouped in chapters of functionality, one merely turns to the chapter related to what is required and then leafs through the pages. For these examples, merely turn to Chapter 6, "Bit Mangling," or Chapter 7, "Bit Wrangling." Okay, okay, so I had a little fun with the chapter titles, but there is no having to wade through pages of extra information trying to find what you are looking for. In addition (not meant to be a pun), there are practical examples near the descriptions and not in the back of this book, which is even more helpful in jogging your memory as to its usage. Even the companion CD for this book uses the same orientation.

Since the primary (cross) development computer for most of you is a PC and not necessarily a Macintosh, and the target platform is a PC and not necessarily an Xbox, GameCube, or PS2, the bulk of the examples are for the X86, but additional sample code is on the companion CD for other platforms. I tried to minimize printed computer code as much as possible so that the pages of the book do not turn into a mere source code listing! Hopefully, I did not overtrim and make it seem confusing. If that occurs, merely open your source code editor or Integrated Development Environment (IDE) to the chapter and project on the companion CD related to that point in the book you are trying to understand.

The book is also written in a friendly style to occasionally be amusing and thus help you in remembering the information over a longer period of time. What good is a technical book that is purely mathematical in nature, difficult to extract any information from, and puts you (I mean, me) to sleep? You would most likely have to reread the information once you woke up—and maybe again after that! The idea is that you should be able to sit down in a comfortable setting and read the book cover to cover to get a global overview. Then go back to your computer and using the book as a tool, implement what you need or cut and paste into your code, but use at your own risk! You should use this book as an appendix to more in-depth technical information to gain an understanding of that information.

The code on the CD is broken down by platform, chapter, and project, but most of the code has not been optimized. I explain this later, but briefly: Optimized code is difficult to read and understand! For that reason, I tried to keep this book as clear and readable as possible. Code optimizers such as Intel's VTune program are available for purposes of optimization.

This book, as mentioned, is divided into chapters of functionality. (Have I repeated that enough times?) If you are lacking a mathematical foundation related to these subjects of geometry, trigonometry, or linear algebra, I would recommend the book *3D Math Primer for Graphics and Game Development* by Fletcher Dunn and Ian Parberry or a visit to your local university bookstore. The book you are now reading (and hopefully paid for) is related to the use of vector math in games or embedded and scientific applications. With that in mind, there is:

- A coding standards recommendation that this book follows
- An overview of vector processors being used in games and not specific to any one processor, so the differences between those processors covered are highlighted

Once the foundations are covered, similar to a toddler, there is crawling before one can walk. Thus, the following is covered:

- Bit masking and shifting
- Ability to convert data to a usable form
- Addition/subtraction (integer/floating-point)
- Multiplication/division (integer/floating-point)
- Special functions
- Trigonometric functionality

...and then, finally, flight!

- Advanced vector math
  - Matrices
  - Quaternions
- Use in tools (programmer versus artist wars)
- Use in graphics
  - Vertex shaders
  - Pixel shaders
- Use in FMV (Full Motion Video)
- Debugging

Hint: Write vector algorithms in the pure C programming language using standard floating-point functions, and then rewrite using vector mnemonics.

Just as it is very important to write functions in C code before rewriting in assembly, it is very important to write your vector math algorithms using the regular math operations. Do not write code destined for assembly code using the C++ programming language because you will have to untangle it later. Assembly language is designed for low-level development, and C++ is a high-level object-oriented development language using inheritance, name mangling, and other levels of abstraction, which makes the code harder to simplify. There is, of course, no reason why you would not wrap your assembly code with C++ functions or libraries. I strongly recommend you debug your assembly language function before locking it away in a static or dynamic library, as debugging it will be harder.

This allows the algorithm to be debugged and mathematical vector patterns to be identified before writing the vector algorithm. In addition, the results of both algorithms can be compared to verify that they are identical, and thus the vector code is functioning as expected. At any time throughout the process, a vectorizing C compiler could be used as a benchmark. When the specialized compiler encounters control flags, it examines the C code for patterns of repetition that can be bundled as a series of parallel operations and then uses the vector instructions to implement it. The use of such a compiler would be the quickest method to get vector-based code up and running. The results are not always optimal, but sometimes examining the compiler output can give insight into writing optimized vector code.

Book Legend

CD Files

This book has a companion CD, which contains sample code with SIMD functionality. Each chapter with related sample code will have a table similar to the following:

CD Workbench Files: /Bench/*architecture*/chap02/*project*/*platform*

architecture		*project*		*platform*
PowerPC	/vmp_ppc/	**Ram Test**	**/ramtest/**	/mac9cw
X86	/vmp_x86/	**ram**	**/ram/**	/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

By substituting the data elements in a column into the CD Workbench Files: path, it will establish a path as to where a related project is stored. There are multiple sets of file sample code:

- PowerPC/AltiVec for Macintosh OS9
- PlayStation 2 devTool for official developers, HomeBrew for hobbyists, and PS2 Linux Kit
- X86 for Win32 and official Xbox developers

The goal of the CD is to be a pseudo vector math library that has been split open to display its internals for observation and learning. Each module has its own initialization routine to set up function prototype pointers so that, depending on the processor running the sample code, the correct set of functions best suited toward that processor are assigned to the pointers. If you are one of those (unusual) people running two or more processors in a multiprocessor computer, and they are not identical, then I am sorry! The good news is that the samples are single threaded, so you should theoretically be okay! There is one other item. There are many flavors of processors. Although the basis of each function call was a cut and paste from other functions, the core functionality of the function was not. So there may be some debris where certain comments are not exactly correct, so forgive me. There is a lot of code on that CD to have to wade through and keep in-sync with the various platforms and their flavors of processors. It is not a gourmet meal, as it is only steak and potatoes, but they still have taste!

This book uses the following SIMD operator table to indicate the level of support for a particular processor. An item in bold in the table indicates that an instruction is supported. An item in italics indicates an unsupported one.

AltiVec	**MMX**	*SSE*	**SSE2**	*3DNow*	*3DMX+*	*MIPS*	*MMI*

Each SIMD operator will have a table showing the organization of the mnemonic as well as the type of data and bit size of data for each processor supported.

AltiVec	Op	*Dst*,	*aSrc*,	*bSrc*	Unsigned	128
MMX	Op	*mmDst*,	*mmDst*,	*mmSrc*	Signed	64
3DNow!	Op	*mmDst*,	*mmDst*,	*mmSrc*	Signed	64
3DMX+	Op	*mmDst,*	*mmDst,*	*mmSrc*	Signed	64
SSE	Op	*xmmDst*,	*xmmDst*,	*xmmSrc*	Single-Precision	128
SSE2	Op	*xmmDst*,	*xmmDst*,	*xmmSrc*	Single-Precision	128
MIPS	Op	*Dst*,	*aSrc*,	*bSrc*	Unsigned	128
MMI	Op	*Dst*,	*aSrc*,	*bSrc*	Unsigned	128

Some operators take two arguments, some three, and others four. For processors such as X86, *sourceB* is the same as the *destination*. For other processors, such as PowerPC, the source arguments are separately detached from the destination arguments. When the SIMD operator is being explained, both arguments will be used, such as, "The Op instruction operates upon the sources *aSrc (xmmSrc)* and *bSrc (xmmDst)*, and the result is stored in the destination *Dst (xmmDst)*."

Each of the SIMD operators will have a table entry that contains one of these declarations indicating the type of data that it supports:

- Unsigned — Unsigned integer
- Signed — Signed integer
- [Un]signed — Sign neutral, thus integer can be either signed or unsigned
- Single-Precision — Single-precision floating-point
- Double-Precision — Double-precision floating-point

Note: Technically, an integer that is considered signless, meaning without sign (sign neutral), is called signless in languages such as Pascal or [Un]signed. Normally, however, there is typically no third case because if it is not signed, it is unsigned. In regards to this book, however, I needed this third case to represent that a value could be either signed or unsigned, thus [Un]signed is utilized.

Hint: Watch for declarations of Pseudo Vec!

Not all processors handle the same set of parallel instructions. Some, such as the Hitachi SH7750 and MIPS-3D, have minimal SIMD support, while some, such as the AltiVec, MIPS-MMI, and MIPS-VU, are heavily loaded. Located throughout this book are emulation code samples to help explain functionality and alleviate any missing SIMD

operators as well as give you, as a developer, ideas of implementing your own algorithm on a computer.

Pseudo Vec

A pseudo vector declaration will have an associated C language macro or functions to emulate the functionality of a SIMD operator.

The source code on the companion CD contains various sets of code for the different processors. Almost none of the functions actually return a value; instead, the results of the equation are assigned to a pointer specified by the first (and sometimes second) function parameters. They will always be arranged in an order similar to:

```
void vmp_functionPlatform(argDest, argSourceA, argSourceB)
```

For the generic C code on the companion CD, the term "generic" will be embedded, such as in the following:

```
void vmp_FAddGeneric( float *pfD,                  // Dest
                      float fA, float fB )         // SrcA

void vmp_QVecAddGeneric(vmp3DQVector * const pvD,  // Dest
                  const vmp3DQVector * const pvA,  // SrcA
                  const vmp3DQVector * const pvB)  // SrcB
```

An important item to note is the actual parameter, {pvD, pvA, pvB}, {pbD, phD, pwD}, etc. The "F" in FAdd represents a scalar single-precision float.

Table 1-1: Float data type declaration

"F" in vmp_FAdd	Scalar single-precision float
"Vec" in vmp_VecAdd	Single-precision (3) float vector
"QVec" in vmp_QVecAdd	Single-precision (4) float vector
"DVec" in vmp_DVecAdd	Double-precision (4) float vector

Table 1-2: Data size of an element

"v" in pvD	128-bit vector, typically (SP Float)
"b" in pbD	8-bit [un]signed packed
"h" in phD	16-bit [un]signed packed
"w" in pwD	32-bit [un]signed packed

Really complicated, don't you think? Of course, when you selectively cut and paste some of this library code into your functions and change them just enough to make them your own, you will most probably use your own conventions and not mine (whatever works for you!).

Pseudo Vec (PowerPC) (AltiVec) (X86) (3DNow!) (3DNow!+) (MMX) (MMX+) (SSE) (SSE2) (MIPS) (MMI) (VU)

Declarations that are followed by a processor label specify alternate assembly code specifically for use by that processor to emulate a missing SIMD operation.

Graphics 101

An algorithm typically found in graphics programming, the 101 is a play on a university beginner-level class number.

Graphics 101 (X86) (3DNow!) (MMX) (SIMD) (PowerPC)

This is the graphics algorithm implementation for the specified processor instruction set.

Algebraic Laws

This item is self explanatory. Algebraic laws are used in this book as well as in other books related to mathematics. The first time one of the laws is encountered, it will be defined similar to what is shown here as a reminder to the rules of algebra learned back in school.

Identity	$n1 = 1n = n$	$n + 0 = 0 + n = n$

I-VU-Q

This is an abbreviation to represent the answer to an interview question that has been encountered and its solution. I hate tests. Even in college. Delivering fantastic completed projects was very challenging for me, but I would perform poorly on the midterms and finals. There is one not-so-fond memory of a midterm for a digital electronics class where the professor prepared two tests to alleviate any possible cheating. The questions on each page were interwoven, so the answer to one problem was used as part of the equation for the next. The problem is that a mistake was made in an equation on the first page of one of the tests and, unfortunately, that was the one I was given. Unable to get the values to balance was frustrating, and I wound up chewing on the bone and not letting go. Fellow classmates who did not check their answers did well, and those who did check did not do so well. When I noticed the elapsed time, there were only 15 minutes left of the two hours to complete the

test. Needless to say, I was more then a little ticked off that we were not allowed to retake the test!

This is the same as programming tests during job interviews, over the phone or in an e-mail form, and especially during an on-site interview gauntlet where any slight goof-up gets one filed in the round file for years! I do not do well unless I have my research library within arm's reach and plenty of time to respond with a well-thought-out answer. Once a company has drawn a conclusion about you, it seems to be their opinion for years. Besides, these tests are useless, as the person doing the testing judges the answers based upon his or her experiences (or lack thereof) and skill level. During a gauntlet or marathon session, quite often the alpha male of the pack wants to make sure that everyone in the room, especially the interviewee, knows just how superior he thinks he is, so he will ask the most complicated or obscure questions he can that will make the interviewee look bad. In fact, a good number of those questions have nothing to do with game programming (not that I am bitter or anything!). There was also a rumor that a fellow employee at a company I worked for actually had other people do his pre-interview programming test for him. Nobody attempted to verify his knowledge, and it took about six months for management to acknowledge his lack of skill level. Of course, not all the interviewers are of this sort, but I have personally encountered quite a few that are, at least from my perspective.

The point is that if one was interviewing test pilots, one would want to test them for their quick responses to any environmental concern that might arise in the cockpit. If you were hiring someone to design and build the plane, you would not want someone who would slap the plane together from memory (without looking at the instructions, like a couple of my children) but, rather, would take their time, do research, design it, and build it properly.

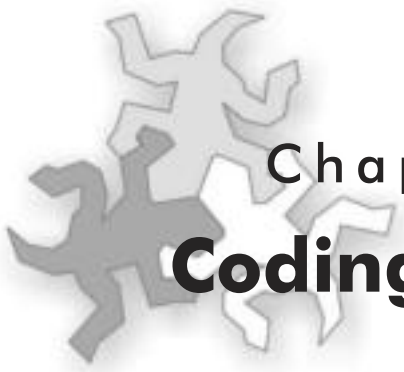
To make a long story short (too late), I have thrown in some interview questions and their answers into this book that have come up time to time during my interviews over the years. Hopefully, they will help you when you have to handle a programming test, as it seems that some of those same questions seem to be continuously passed around from company to company. It is sort of my personal crusade to abolish interview testing! (Keep an eye out for one of my next books, *Programming Interview Questions from HELL!* or *Introduction to Vertex and Pixel Shaders*.)

Insight

One summer, I taught a video game design class for a "College for Kids" program. For some reason, it was one of the more popular classes they had. About halfway into the program a couple of students came up and asked when they would get to play video games. (Apparently, they had not read the syllabus.) They merely added two-plus-two and calculated three as a result.

Throughout this book, you may be thinking, "Well, when are we coming to the game building part?" This is not that kind of book. There are plenty of good books already covering that subject. This book is more related to the foundations of the mathematical portions of the engine, arranging those to work in parallel, and the tuning of that engine to make it run more efficiently.

Okay, that should be enough ground-breaking (soapbox) material. Let's get on with it!



Chapter 2
Coding Standards

I am not going to bore you to death with coding standards, as there are a multitude of books on the subject already and that is not what this book is about. To become a better programmer, however, you should adopt and rigidly follow a set of coding standards.

I just said that I was not going to bore you with coding standards, and this chapter is called "Coding Standards?" What gives?

The source code on the companion CD uses terms such as const and assert, as well as naming conventions, which should be a standard that you adopt into your own style of programming, if you have not already. The following tools cost very little, only a little extra time to drop them into your code, but they will save you time in the long run, especially when dealing with vector code; that is why this chapter is here! They primarily entail the use of const and assert, as well as memory alignment. It should be noted that due to page width limitations, they are mostly only used in the first few functions in print and then not at all. This is not an indicator that they are not needed, only that I did not want the pages in the book to appear to be too verbose in printed computer code. You will find that the equivalent functions on the CD will be fully loaded with the use of const and the appropriate assertions.

CD Workbench Files: /Bench/*architecture*/chap02/*project*/*platform*

	architecture		*project*	*platform*
PowerPC	/vmp_ppc/	**Ram Test**	**/ramtest/**	/mac9cw
X86	/vmp_x86/	**ram**	**/ram/**	/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Constants

To put it simply, the const can essentially be thought of as a write protection of sorts.

```
void ramSetup(const CPUInfo * const pInfo);
```

In this example, the contents of the data structure (first const) and the pointer to the structure (second const) cannot be altered. This function merely reads information contained within that structure and does not alter it or its pointer in any fashion. The const guarantees it! Of course, that does not stop the programmer from casting the pointer to a const-less pointer and then modifying the data. But that would be bad!

The placement of the second const protecting the pointer is very specific, but the placement of the first const is a little more liberal. It can occur before or after the data type declaration, such as in the following:

```
void ramSetup(const CPUInfo       * const pInfo);
void ramSetup(      CPUInfo const * const pInfo);
```

Data Alignment

This brings to mind a story of a recent event in which I was the lead in a skunk works division of my employer, and they had me assigned to a task involving loaning my assembly optimization expertise to another company for an open source project that the company was sponsoring. I was in the process of doing a code merge of some optimized PowerPC assembly to the main project core when a problem arose. I tracked this down to a memory alignment conflict with the new AltiVec code. It seems that a programmer working with an X86-based platform accidentally shifted a (byte-aligned) vector buffer within a data structure by pre-appending a couple of new data members to a C data structure. Because the vector buffers had been defined as byte arrays, the compiler obliged and so they were no longer 16-byte aligned, but single-byte aligned. The AltiVec did not care because it guarantees data alignment by ignoring the lower four address bits; thus, the data was being accessed improperly skewed. The reason a problem was known to occur was that the decompressed image had become corrupted in a test run after the code merge.

Processors work at their most efficient state when working with properly aligned data. In the case of a 128-bit processor, such as the

AltiVec, the data is forced to an alignment. In the case of Intel's Pentium III with its SSE or better instruction set, there is not one 128-bit load, but two. Processors have been designed for efficient operations so, internally, the data is not loaded misaligned; it is 128-bit aligned, but in the case of the SSE, it is corrected by shifting two halves of two 128-bit loads to adjust for the requested 128 bits of data. This misalignment is very inefficient!

The first item on the agenda is the alignment of data values. Pointers are typically four byte; 64-bit requires eight byte; 128-bit requires 16 byte.

```
#define ALIGN2( len ) (( len + 1 ) & ~1 )    // round up to 16 bits
#define ALIGN4( len ) (( len + 3 ) & ~3 )    // round up to 32 bits
#define ALIGN8( len ) (( len + 7 ) & ~7 )    // round up to 64 bits
#define ALIGN16( len ) (( len + 15 ) & ~15 ) // round up to 128 bits
```

Some of you may note that the basic equation of these macros:

$$(A, X) \quad ((A + X) \& \sim X)$$

relies on a byte count of 2^N so that a logical AND can be taken to advantage and could possibly be replaced with:

$$(A, X) \quad ((A + (X - 1)) \% X)$$

and be easier to read. But that would be giving too much credit to the C compiler, as some will do a division for the modulus and some will see the binary mask and take advantage with a mere logical AND operation. Even though this latter code is clearer, it may not be compiled as fast code! If you are not sure what your compiler does, merely set a breakpoint at a usage of a macro. Then either expand the macro or display mixed C and assembly code. The division or AND will be right near where your program counter is pointing to your place in the code!

A macro using this alternate method would be:

```
#define ALIGN( len, bytes )((len + (bytes–1)) % bytes)
```

This one is a little obscure and typically unknown by non-console developers, but CD sector size alignment is needed for all files destined to be loaded directly from the CD, as they are typically loaded by number of sectors, not number of bytes, and this is typically 2048 or 2336. All of these require some sort of alignment correction jig.

```
    // round up 1 CD Sector
#define ALIGN2048( len ) (( len + 2047 ) & ~2047 )
#define ALIGN2336( len ) (( len + 2335 ) % 2336 )
```

The correction is as simple as:

```
void *foo( uint nReqSize )
{
   uint nSize;
   nSize = ALIGN16( nReqSize );
:
 :    // Insert your other code here!
}
```

The requested size is stretched to the appropriate sized block. This really comes in handy when building databases in tools for use in games.

GOAL

Ensure properly aligned data.

I have had several incidents over the years with compilers and misaligned data. Calls to the C function malloc() returned memory on a four-byte alignment, but when working with 64-bit MMX or some 128-bit SSE instructions, there would be unaligned memory stall problems. Some processors cause a misalignment exception error, while others just cause memory stalls, but either way, neither method is good for your application.

The first half of the remedy is easy. Just make sure your vectors are a data type with a block size set to 16 bytes and your compiler is set to 16-byte alignment and not one byte. The smart move is to ensure that all data structures are padded so they will still be aligned properly, even if the compiler alignment gets set to one byte. This will make your life easier, especially if code is ever ported to other platforms, especially UNIX.

Notice the Struct member alignment field in the following property page for Project Settings in Visual C++ version 6. The default is 8 Bytes *, denoted by the asterisk, but we really want 16 bytes for vector programming.

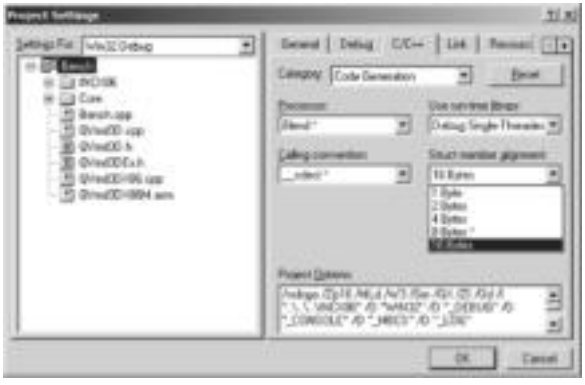


Figure 2-1: Visual C++ version 6 memory alignment property page

Pancake Memory LIFO Queue

Using an alternative memory management scheme, such as pancaking, where a base (or sub-base) level is set and the next available memory pointer is merely advanced by the amount of memory needed, there is no memory free function, as memory is merely disposed of by resetting the memory available back to its original base (in essence, abandoning the memory and merely making sure the base is on a 16-byte alignment). This is like a bottom-based processor stack. A free memory pointer is preset to the bottom of the stack at the physical base level of that memory. As data is loaded into memory, the free pointer is moved higher up in memory, and when it is decided it is time to release that memory, all allocated objects are instantly thrown away by merely resetting the free pointer to the base level again. Game consoles sometimes use this method to conserve code space from having to deal with individual deallocations!

Obviously, since there is no need for reallocations, or freeing of memory, there is no need for a header either.

There are other schemes, but make sure your memory is 16-byte aligned. Now that any possible memory allocation alignment problems have been taken care of up front, it is time to move on to the good stuff.

Stack

Never, never, never pass packed data on the stack as a data argument. Always pass a pointer to the packed data instead. Ignoring any possible issues such as faster consumption of a stack, the need for more stack space, or security issues, such as code being pushed onto the stack, there is no guarantee that the data would be properly aligned. (Okay, there is an exception with AltiVec, but I strongly recommend against it, as it is extremely unportable to other platforms!) Again there are exceptions, but it is not portable. Do not declare local stack arguments for packed data. In assembly language, the stack register can be "corrected" and then "uncorrected" before returning. A better (portable cross-platform) solution would be to declare a buffer large enough for 16-byte aligned data elements and padded with an extra 12 bytes of memory. Vectors are aligned to 128-bit (16-byte) data blocks within that buffer.

This is only one example of aligning memory: Adding fixed sizes to allow for modulus 16 to the buffer will correct the alignment and improve processing speed as well!

3D Vector (Floating Point)

```
typedef struct vmp3DVector
{
  float x;
  float y;
  float z;
} vmp3DVector;

  // Three 96-bit vectors aligned to 128 bits each, thus four floats
  // each so 3 x 4 = 12 bytes, but add three extra bytes (+3) to handle
  // a possible misaligned offset of { 0, 4, 8, 12 }. Once the first
  // is aligned, all other four byte blocks will be aligned as well!

float vecbuf[3  *4 +3];  // enough space +3=15 bytes
vmp3DVector *pvA, *pvB, *pvD;

  // Force proper alignment

pvA = (vmp3DVector*) ALIGN16((int)(vecbuf));
pvB = (vmp3DVector*) ALIGN16((int)(pvA+1));
pvD = (vmp3DVector*) ALIGN16((int)(pvB+1));
```

I realize that it initially appears to be crude, but it delivers the functionality you require through the use of pointers, and it is cross-platform compatible.

3D Quad Vector (Floating Point)

Of course, if you are dealing with quad vectors, then align the first one. All the others, which are the same data type and already 16 bytes in size, will automatically be aligned.

```
typedef struct vmp3DQVector
{
  float x;
  float y;
  float z;
  float w;
} vmp3DQVector;

vmp3DQVector *pvC, *pvE, *pvF;

 // Force proper alignment

pvC = (vmp3DQVector*) ALIGN16((int)(vecbuf));
pvE = pvC+1;
pvF = pvE+1;
```

The same applies for 4x4 matrices. The following is a "quick and dirty" demonstration of aligned three-element vector data structures.

```
 // Copy vectors to aligned memory
pvA->x=vA.x;  pvA->y=vA.y;  pvA->z=vA.z;
pvB->x=vB.x;  pvB->y=vB.y;  pvB->z=vB.z;
```

```
vmp_SIMDEntry();     //x86 FPU/MMX switching

   // if (most likely) non-aligned memory
vmp_CrossProduct0( &vD, &vA, &vB );

   // if (guaranteed) aligned memory
vmp_CrossProduct( pvD, pvA, pvB );

vmp_SIMDExit();      //x86 FPU/MMX switching
```

Note the convention of the appended zero used by vmp_CrossProduct0 and vmp_CrossProduct. The zero denotes that the function is not guaranteed to be aligned to (*n* mod 16) with a zero remainder.

Another item to keep in mind is that a vector is 12 bytes in length, which is made up of three floats; a float is four bytes in size, but it is read/write as 16 bytes on a processor with a 128-bit data width. The extra four-byte float must be preserved, and if the tricks of 128-bit memory allocation are utilized, an out-of-bounds error will not occur, since the data is being advanced in 16-byte blocks.

There are always exceptions to the rule, and that occurs here as well. The compiler for the AltiVec instruction set can use the following local argument stack declaration:

```
void Foo( void )
{
  vector float vD, vA, vB;
}
```

The following vector declaration automatically aligns the data to a 16-byte alignment. For the MIPS processor, the GCC can generate the following definition:

```
typedef float sceVu0FVECTOR[4] \
                          __attribute__((aligned (16)));

void Foo( void )
{
  sceVu0FVECTOR vD, vA, vB;
}
```

I am sorry to say that there is only one method for the 16-byte aligned stack frame of data within the Visual C++ environment for the X86-based Win32 environment, but unfortunately this only works with version 7.0 or higher or version 6 and a processor pack. The following is a snippet from a DirectX header file d3dx8math.h:

```
#if _MSC_VER >= 1300   // Visual C++ ver.7
#define _ALIGN_16 __declspec(align(16))
#else
#define _ALIGN_16      // Earlier compiler may not understand
#endif                 // this, do nothing.
```

So the following could be used:

```
vmp3DVector vA;
__declspec(align(16)) vmp3DVector vB;
_ALIGN_16 vmp3DVector vC;
```

The alignment of vA cannot be guaranteed, but vB and vC are aligned on a 16-byte boundary. Codeplay's Vector C and Intel's C++ compilers also support this declaration.

There is, however, the Macro Assembler (masm), which has the following:

```
align 16
```

followed by the 16-byte aligned data declaration for Real4:

```
vHalf  Real4  0.5,0.5,0.5,0.5
```

Another task that can easily be done from within assembly language is a stack correction for 16-byte memory alignment. The stack pointer works by moving down through memory while adding items to the stack. So by using a secondary stack frame pointer, the stack can be corrected!

```
push ebx
mov ebx,esp           ; ebx references passed arguments
and esp,0fffffff0h    ; 16 byte align

; Insert your code reference by [esp-x]

mov esp,ebx
pop ebx
```

Hint: Use assertions in your code to trap errors early in the development process.

Assertions

The second tool used within the source code on the companion CD is the use of an assertion. Typically, at the top of your file should be the inclusion of the following header file declaration:

```
#include    <assert.h>
```

Sorry to bore some of you who religiously use these! An assertion is essentially a test for a not true, thus a false, condition. That is, if the condition is false, an assertion will occur whereby the processor IP is halted at the offending line of code. This is very similar to the following:

```
if (!(3==a))                    // assert(3==a)
{
    // System Break
    DebugBreak();               // Win32, Xbox
}
```

As mentioned, the condition must fail. The variable *a* needs to be assigned the value of 3 to be successful; otherwise, it would halt the system. There are many philosophies on the subject of assertions. In my own particular case, I believe in four types of assertions: Fatal — Debug, Fatal — Release, Non-Fatal — Debug, and Non-Fatal — Release.

- A **Fatal — Debug** is a programmer error. This is something that should never, ever, ever occur, such as passing a null or an obviously bad pointer, passing an out-of-range argument to a function, or passing a misaligned vector or matrix. It halts the debug version of the code and forces the programmer to fix it right away. This is the one assertion that should occur most of the time in your debug code while it is being developed, and never in the shipped code.
- A **Fatal — Release** is an unrecoverable error in a shipped version of the application that is totally out of memory, so there is not even enough memory to display an error message (a catastrophic error). This should preferably never be used. There are even ways in an application to stop the application logic, jettison memory, and put up your, "Sorry I'm dead! Call your Customer Support Person." You do not want your customers to ever get the "Blue Screen of Death!" (You Windows people know what I am talking about!)
- **Non-Fatal — Debug** and **Non-Fatal — Release** are recoverable errors, or errors that can be worked around without killing the application. These are typically not able to load a resource from the media, missing or contaminated files, missing sound resources, etc. Your application can be crippled in such a way to allow it to progress. Even during development of the game, you may want to alert the developer that table sizes do not match the number of defined indexes for that table. The idea is to not kill the running application, but to annoy the programmer enough to get him to fix the problem. I like that so much I am going to state that again!

Hint: The idea of non-fatal assertions is to not kill the running application, but to annoy the programmer enough to get him to fix the problem.

On a personal note, please do not get annoyed and turn them off at the source instead of fixing the real problem. (I once worked with a programmer who did just that, which caused many Homer Simpson-type errors to creep into the code that would have been caught immediately otherwise. Doh!) On the inverse to that, a supervisor once threatened to chop off my fingers because the assertions in the code to catch errors were doing that and breaking the build. A better method is to spend a couple minutes catching the errors early in the development cycle, rather than a long time during crunch mode at the tail of the project trying to track them down or never knowing about them until after the product has shipped.

For purposes of this book, only a debug assertion will be utilized. In essence, this is if an application were to be defined as a debug model; the assertion would then exist in the code, thus the code would run a little slower but considerably safer. In a release model, it is stubbed out, thus it does not exist and your code will magically run a bit faster. On the companion CD, you will find the following assertions in use for the generic assertion ASSERT().

Listing 2-1: Assertion type definitions

```
#ifdef USE_ASSERT   // Active Assertions

#define ASSERT( arg1 )        assert( (arg1) )
#define ASSERT_PTR( arg1 )    assert(NULL!=(arg1))
#define ASSERT_PTR4( arg1 )   assert((NULL!=(arg1)) \
                                    && (0==(((long)(arg1))&3)))
#define ASSERT_PTR8( arg1 )   assert((NULL!=(arg1)) \
                                    && (0==(((long)(arg1))&7)))
#define ASSERT_PTR16( arg1 )  assert((NULL!=(arg1)) \
                                    && (0==(((long)(arg1))&15)))
#define ASSERT_FNEG( arg1 )   assert(0.0f<=(arg1));
#define ASSERT_FZERO( arg1 )  assert(0.0f!=(arg1));
#define ASSERT_NEG( arg1 )    assert(0<=(arg1));
#define ASSERT_ZERO( arg1 )   assert(0!=(arg1));

#else         // Assertions stubbed to nothing
              // (empty macro)
#define ASSERT( arg1 )
#define ASSERT_PTR( arg1 )
#define ASSERT_PTR4( arg1 )
#define ASSERT_PTR8( arg1 )
#define ASSERT_PTR16( arg1 )
#define ASSERT_FNEG( arg1 )
#define ASSERT_FZERO( arg1 )
#define ASSERT_NEG( arg1 )
#define ASSERT_ZERO( arg1 )
#endif
```

You would merely insert your custom assertion.

```
uint nActor = 5;
ASSERT(nActor < MAX_ACTOR);
```

The pointer assertion ASSERT_PTR4() does two things. First, it guarantees that the pointer is not assigned to null. Second, memory pointers must or should be (depending upon the processor and exception settings) referencing memory at least four-byte aligned so that pointer alignment is checked for. Even if only referencing strings, it should be recognized that if they are at least four-byte aligned, algorithms can be made more efficient when dealing with those string components in parallel.

```
void *pApe;
ASSERT_PTR4(pApe);
```

This is a good way to ensure that integers and single-precision floating-point values are properly aligned.

The pointer assertion ASSERT_PTR16() is virtually the same, except it guarantees that the pointer is not null and is referencing data on a 16-byte alignment, which is necessary for vector math processing.

```
ASSERT_PTR16(pApe);
```

Different processors have different behaviors, which could become a camouflaged problem. Asserting on a non-aligned 16-byte data reference finds these problems fast. These last two deal with asserting upon invalid values of zero or negative single-precision floating-point numbers.

```
float f;
ASSERT_FNEG(f);       // Assert if a negative.
ASSERT_FZERO(f);      // Assert if absolute zero
```

Memory Systems

The second part of coding standards is the memory alignment check for any compiled code. This is to verify that malloc() is indeed allocating properly aligned memory for the superset Single Instruction Multiple Data (SIMD) instruction sets. You can test this by executing a simple algorithm, which is discussed in the following section.

RamTest Memory Alignment Test

Listing 2-2: \chap02\ramTest\Bench.cpp

```
#define RAM_TBL_SIZE        4096

int main( int argc, char *argv[] )
{
  unsigned int n, nCnt, bSet, bTst, bMsk, a, b;
  void *pTbl[RAM_TBL_SIZE];

    // Allocate a series of incr. size blocks

  for (n=0; n<RAM_TBL_SIZE; n++)
  {
    pTbl[n]=(byte *) malloc(n+1);
    if (NULL==pTbl[n])
    {
      cout << "low memory.  (continuing)..." << endl;
      break;
    }
  }
  nCnt = n;             // # of entries allocated

    //  Test memory for alignments

  bSet = 16;            // Preset to 128 bit (128/16)
  bTst = bSet - 1;
  bMsk = ~bTst;

  for (n=0; n<nCnt; n++)
  {
    a = (unsigned int) pTbl[n];
    do {                // round up to 'bSet' bits
      b = (a + bTst) & bMsk;
      if (a==b)
      {
        break;          // okay
      }
                        // Unaligned...
      bSet >>= 1;       // reduce by a bit
      bTst = bSet - 1;
      bMsk = ~0 ^ bTst;
    } while(1);
  }

    // Release all of memory to cleanup

  for (n=0; n<nCnt; n++)
  {
    free(pTbl[n]);
  }

  cout << "Ram Alignment is set to " << bSet;
  cout << " bytes (" << (bSet<<3) << " bits).\n";
  cout << flush;
  return 0;
}
```

Please note that it loops up to 4096 times, slowly increasing the size of the allocation just in case the memory manager issues properly aligned memory for a short period of time before allocating any that might be skewed. Also, you will most likely get a memory low message, but that is okay; you are only allocating about eight meg or so. If everything is fine, there will be a match between your processor and the following table. Please note the Gekko and PS2 information is from their manufacturers' own web sites, as to the hardware capabilities of their machines.

Table 2-1: SIMD instruction set with data width in bits and bytes

SIMD Instruction Set (Data Width)	Bits	Bytes
AltiVec	128	16
AMD 3DNow!	64	8
AMD 3DNow! Extensions	64	8
AMD 3DNow! MMX Extensions	64	8
AMD 3DNow! Professional	64/128	8/16
Gekko	64	8
MMX	64	8
SSE	128	16
SSE2	128	16
MIPS - MMI	64/128	8/16
PS2 - VU0/1	128	16
TX-79, TX-99	64/128	16

If there is a mismatch, then you have an alignment error problem. This can be rectified by using code similar to that found in the following section. This function is designed to wrap the standard function call to malloc() or new. Do not forget to add the assertion as a good programming practice!

Memory Header

The following header is hidden at the true base of memory allocated by our function. In essence, memory is slightly overallocated. This is not really needed by Macintosh, as it properly aligns data to 4 or 16 bytes, based upon if AltiVec code is being compiled or not. I should probably point out that AltiVec is the velocity engine put out by Motorola as a super instruction set for the PowerPC that works with 128-bit data. The function malloc is in essence routed to the correct core allocation function.

Listing 2-3: \chap02\ram\ram.cpp

```
typedef struct RamHeadType
{
  uint32  nReqSize;   // Requested size
  uint32  extra[3];   // padding to help align to 16 byte.
} RamHead;
```

Allocate Memory (Malloc Wrapper)

Listing 2-4: \chap02\ram\ram.cpp

```
void * ramAlloc( uint nReqSize )
{
  void *pMem;
  RamHead *pHead;
  uint nSize;

  ASSERT_ZERO(nReqSize);

  // Force to 16-byte block + room for header

  nSize = ALIGN16(nReqSize) + sizeof(RamHead);
  pHead = (RamHead *) malloc(nSize);
  pMem = pHead;

  if (NULL==pHead)
  {                     //  Allocation Error
  }
  else
  {                     // Save Req Size
    pHead->nReqSize = nReqSize + sizeof(RamHead);
    pHead->extra[0] = 1;
    pHead->extra[1] = 2;
    pHead->extra[2] = 3;

      // Align by adj header +4 to +16 bytes

    pMem = (void *) ALIGN16( ((uint)pMem)
                        + sizeof(uint32) );
  }
  return pMem;
}
```

It functions by aligning the amount of memory requested to the nearest 16-byte boundary. This will assist in maintaining memory to a 16-byte block size. An additional 16 bytes are allocated as the head. This is useful for two reasons:

- The memory passed to the calling function can be forced to the proper alignment.
- A side benefit of storing the requested size is that size adjustments similar to a realloc() can be issued, and the calling function does not

have to know the current size when releasing that memory back to the pool.

Hidden in the beginning of the allocated memory is a header where the requested size is stored in the first 32-bit word, and the other three words are set to the values of {1, 2, 3}. The pointer is then advanced to a 16-byte alignment and passed to the calling function.

When releasing memory back to the system, the returned pointer needs to be unadjusted back to the true base address; otherwise, a memory exception will occur. The following function wraps the release function free().

Release Memory (Free Wrapper)

Listing 2-5: \chap02\ram\ram.cpp

```
void ramFree( const void * const pRaw )
{
  uint32 *pMem;

  ASSERT_PTR4(pRaw);
  ASSERT_PTR(*pRaw);

  pMem = (uint32 *)pRaw;
  if (*(--pMem)<sizeof(RamHead))
  {
    pMem -= *pMem;
  }
    //  pMem original (unadjusted) pointer
  free(pMem);
}
```

This may seem wasteful, but the base address of the memory being allocated by new or malloc is unknown. With current malloc libraries, it tends to be 4- or 8-byte aligned, so there is a need to allocate for a worst case.

The memory release occurs by decrementing the word pointer by one 4-byte word. If that location contains a value between one and three, the pointer is decremented by that value so that it points at the size information when cast to a RamHead pointer. This is the true memory base position and the pointer that gets returned to the system function free().

For C++ fans, the new and delete operators can be overloaded to this insulating memory module. I also recommend one final item. The memory allocation and release functions should require a pointer to be passed. This will allow the release function to nullify the pointer, and in future enhancements, each pointer could be considered the “owner” of

the memory and thus adjusted for any garbage collection algorithms instituted for a heap compaction in a flat memory environment.

Allocate Memory

A pointer is passed as ppMem and set.

Listing 2-6: \chap02\ram\ram.cpp

```
bool ramGet( byte ** const ppMem, uint nReqSize )
{
  ASSERT_PTR4(ppMem);

  *ppMem = (byte *) ramAlloc(nReqSize);
  return (NULL!=*ppMem) ? true : false;
}
```

Allocate (Cleared) Memory

Listing 2-7: \chap02\ram\ram.cpp

```
bool ramGetClr(byte **const ppMem, uint nReqSize)
{
  bool ret;

  ASSERT_PTR4(ppMem);

  ret = false;
  *ppMem = (byte *)ramAlloc(nReqSize);
  if (NULL!=*ppMem)
  {
    ramZero(*ppMem, 0, nReqSize);
    ret = true;
  }

  return ret;
}
```

Free Memory — Pointer is Set to NULL

Listing 2-8: \chap02\ram\ram.cpp

```
void ramRelease( byte ** const ppMem )
{
  ASSERT_PTR4(ppMem);

  ramFree(*ppMem);
  *ppMem = NULL;
}
```

Exercises

1. What is an assertion?
2. How many assertion types does the author recommend? What are they and what do they do?
3. Create another memory allocation scheme with code to allocate and release that memory for allocation of any size but still be properly aligned.
4. What should one do when allocating memory for a file load to guarantee that no memory access by a processor could accidentally get a memory exception for accessing past the end of allocating memory without special handling?
5. If a raw ASCII text file was loaded into memory, how would it be loaded and what is the easiest method of treating it as one large string?

Chapter 3
Processor Differential Insight

This chapter discusses the similarities and differences between the various processors used in video games, most of which are used in this book, along with a couple of proprietary ones that are not. This book is not an assembly language manual, although the initial focus is from an assembly level point of view. As such, it only lightly touches upon optimization issues, as these are very manufacturer and processor revision specific. With that in mind, this book will not go into detail as to the functionality of a particular processor, except to help bring to light those vector supporting differences.

CD Workbench Files: /Bench/*architecture*/chap03/*project*/*platform*

	architecture		*project*	*platform*
PowerPC	/vmp_ppc/	**CPU Id**	**/cpuid/**	/mac9cw
X86	/vmp_x86/			/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Floating-Point 101

Remember this is not rocket science, and so minor deviations will occur in the formulas since, for example, a single-precision float is only 32 bits in size, which is the data size that this book will predominately use. For higher precision, 64-bit double-precision or 80-bit double-extended precision floating-point should be used instead. These floating-point numbers are based upon a similarity to the IEEE 754 standards specification. Unfortunately, the 80-bit version is only available in a scalar form on an X86's FPU, and the 64-bit packed double-precision is only available on the SSE2 processor.

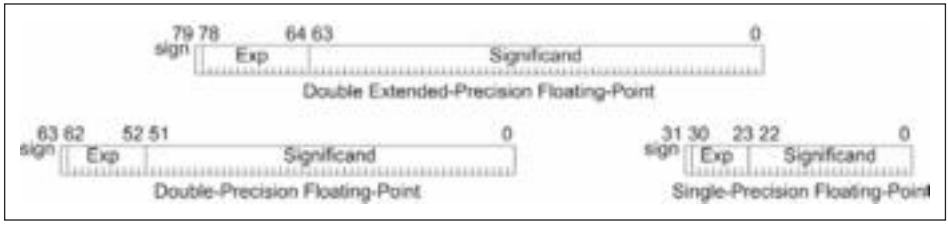


Figure 3-1: Floating-point bit configurations

Most programmers only know a floating-point value from using a declaration, such as a float, double, real4, or real8, etc. They know that there is a sign bit that, if set, indicates the value is negative and, if clear, indicates that the value is positive. That is about it, as it is pretty much a black box, and they have never had a need to dig into it further.

For this book, you will be required to understand a little bit more. The exponent is a base-2 power representation stored as a binary integer. The significand (mantissa) really consists of two components, a J-bit and a binary fraction.

For the single-precision value, there is a hidden "1" leading the 23 bits of the mantissa, thus making it a 24-bit significand. The exponent is 8 bits, thus having a bias value of 127. The magnitude of the supported range of numbers is 2×10^{-38} to 2×10^{38}.

For double-precision values, there is a hidden "1" leading the 52 bits of the mantissa, thus making it a 53-bit significand. The exponent is 11 bits, thus having a bias value of 1023. The magnitude of the supported range of numbers is 2.23×10^{-308} to 1.8×10^{308}.

For the 80-bit version, the extra bits are primarily for protection against precision loss from rounding and over/underflows. The leading "1" is the 64th bit of the significand. The exponent is 15 bits, thus having a bias value of 32767. The magnitude of the supported range of numbers is 3.3×10^{-4932} to 1.21×10^{4932}.

The product of the exponent and significand result in the floating-point value.

Table 3-1: Single-precision floating-point to hex equivalent

Value	Hex	Sign	Exp	Significand
–1.0	0xBF800000	1	7F	000000
0.0	0x00000000	0	00	000000
0.0000001	0x33D6BF95	0	67	56BF95
1.0	0x3F800000	0	7F	000000
2.0	0x40000000	0	80	000000
3.0	0x40400000	0	80	800000
4.0	0x40800000	0	81	000000

Programmers are also usually aware that floats cannot be divided by zero or process a square root of –1 because an exception error would occur:

```
NaN – Not A Number
```

Now that floating-point values have been examined, we can move on to comparisons of floating-point values.

Floating-Point Comparison

Do not expect the resulting values from different calculations to be identical. For example, 2.0 x 9.0 is about 18.0 and 180.0/10.0 is about 18.0, but the two 18.0 values are not guaranteed to be identical.

Let's examine a range of values 10^n and compare a displacement of ±0.001 versus ±0.0000001.

Table 3-2: Note the single-precision loss between the ±0.001 displacement as the number of digits goes up in the base number. As the base number gets larger, fewer decimal places of precision can be supported. The bold hexadecimal numbers are where the precision was totally lost!

Base Number	–0.001	+0.0	+0.001
1.0	0x3F7FBE77	0x3F800000	0x3F8020C5
10.0	0x411FFBE7	0x41200000	0x41200419
100.0	0x42C7FF7D	0x42C80000	0x42C80083
1000.0	0x4479FFF0	0x447A0000	0x447A0010
10000.0	0x461C3FFF	0x461C4000	0x461C4001
100000.0	**0x47C35000**	**0x47C35000**	**0x47C35000**
1000000.0	**0x49742400**	**0x49742400**	**0x49742400**
10000000.0	**0x4B189680**	**0x4B189680**	**0x4B189680**
100000000.0	**0x4CBEBC20**	**0x4CBEBC20**	**0x4CBEBC20**

Table 3-3: In this single-precision table, the displacement is between ±0.0000001. Notice the larger number of hexadecimal numbers in bold, indicating a loss of precision.

Base Number	–0.0000001	+0.0	+0.0000001
1.0	0x3F7FFFFE	0x3F800000	0x3F800001
10.0	**0x41200000**	**0x41200000**	**0x41200000**
100.0	**0x42C80000**	**0x42C80000**	**0x42C80000**
1000.0	**0x447A0000**	**0x447A0000**	**0x447A0000**
10000.0	**0x461C4000**	**0x461C4000**	**0x461C4000**
100000.0	**0x47C35000**	**0x47C35000**	**0x47C35000**
1000000.0	**0x49742400**	**0x49742400**	**0x49742400**
10000000.0	**0x4B189680**	**0x4B189680**	**0x4B189680**
100000000.0	**0x4CBEBC20**	**0x4CBEBC20**	**0x4CBEBC20**

Okay, let's do one more table for more clarity.

Table 3-4: Note that accuracy of the precision of the numbers diminishes as the number of digits increases.

Base Number	+0.001	+0.002	+0.003
1.0	0x3F8020C5	0x3F804189	0x3F80624E
10.0	0x41200419	0x41200831	0x41200C4A
100.0	0x42C80083	0x42C80106	0x42C80189
1000.0	0x447A0010	0x447A0021	0x447A0031
10000.0	0x461C4001	0x461C4002	0x461C4003
100000.0	**0x47C35000**	**0x47C35000**	**0x47C35000**
1000000.0	**0x49742400**	**0x49742400**	**0x49742400**

This means that smaller numbers, such as those that are normalized and have a numerical range from –1.0 to 1.0, allow for higher precision values, but those with larger values are inaccurate, thus not very precise. For example, the distance between 1.001 and 1.002, 1.002 and 1.003, etc. is about 0x20c4 (8388). This means that about 8,387 numbers exist between those two samples. A number with a higher digit count, such as 1000.001 and 1000.002, support about 0x11 (17), so only about 16 numbers exist between those two numbers. A number around 1000000 identifies 1000000.001 and 1000000.002 as the same number. This makes for comparisons of floating-point numbers with nearly the same value very tricky. This is one of the reasons why floating-point numbers are not used for currency, as they tend to lose pennies. Binary Coded Decimal (BCD) and fixed-point (integer) are used instead.

When working with normalized numbers {–1.0 ... 1.0}, a comparison algorithm with a precision slop factor (accuracy) of around 0.0000001 should be utilized. When working with estimated results, a much smaller value should be used. The function in Listing 3-1 returns a Boolean true : false value to indicate that the two values are close enough to be considered the same value. Normally, you would not compare two floating-point values, except to see if one is greater than the other for purposes of clipping. A comparison of the same value is almost never shown here. It is only used in this book for purposes of comparing the results of C code to assembly code to see if you are getting results from your algorithms in the range of what you expected. Listing 3-1 compares two single-precision floating-point values and determines if they are equivalent based upon the precision factor or if one is less than or greater than the other.

Listing 3-1: vmp_IsFEqual()

```
bool vmp_IsFEqual(float fA, float fB, float fPrec)
{
     // The same so very big similar numbers or very small
     // accurate numbers.
  if (fA == fB) return true;

     // Try with a little precision slop!
  return (((fA-fPrec)<=fB) && (fB<=(fA+fPrec)));
}
```

Making the call for single-precision floating-point numbers is easy:

```
#define SINGLE_PRECISION 0.0000001f
if (!vmp_IsFEqual(f, f2, SINGLE_PRECISION))
```

Here is a fast algorithm that uses estimation for division or square roots rather than merely reducing the precision to something less accurate:

```
#define FAST_PRECISION   0.001f
```

This book will discuss these fast estimate algorithms in later chapters. For vector comparisons, this book uses the following code.

Listing 3-2: Comparing two {XYZ} vectors using a specified precision factor

```
bool vmp_IsVEqual( const vmp3DVector * const pvA,
      const vmp3DVector * const pvB, float fPrec )
{
  ASSERT_PTR4(pvA);  // See explanation of assert macros
  ASSERT_PTR4(pvB);  // later in this chapter!

  if (   !vmp_IsFEqual(pvA->x, pvB->x, fPrec)
      || !vmp_IsFEqual(pvA->y, pvB->y, fPrec)
      || !vmp_IsFEqual(pvA->z, pvB->z, fPrec))
  {
    return false;
  }

  return true;
}
```

When dealing with quad vectors (vmp3DQVector), an alternative function is called.

Listing 3-3: Comparing two {XYZW} vectors using a specified precision factor

```
bool vmp_IsQVEqual(const vmp3DQVector *const pvA,
                   const vmp3DQVector *const pvB,
                   float fPrec )
```

This tests a fourth element {.w}:

```
|| !vmp_IsFEqual(pvA->w, pvB->w, fPrec)
```

Processor Data Type Encoding

Table 2-1 gave a listing of the data widths of the various processors with SIMD instruction sets that are discussed in this book, along with a few others for good measure. It is important to understand that to be considered a multimedia processor, there is a minimum of "necessary" instructions required:

- Boolean bit logic
- At least some arithmetic, such as integer addition/subtraction

As the processor gets more complicated, it will contain SIMD features, such as:

- Integer multiplication and division
- Single/double-precision floating-point
  - Addition, subtraction, multiplication, and division
- Min, max, average, comparison, etc.

Of course, all the processors will require a set of instructions not related to SIMD, but for general operations and data conversion. These will not be discussed in this book. For more information, utilize documentation related specifically for the processor being targeted.

Different manufacturers use different concepts of word sizes and processor type data encoding. Note the bit size of a processor as well as the associated data types and letter prefix from the following tables. For example, the Gekko is a {b,h,w}, so in the rest of this book when inspecting SIMD instructions, keep those letters and their data size correlation in mind.

Note: X86 programmers, please note the absence of half-words represented by h in the following tables.

Table 3-5: PC – X86 – MMX, MMX+, 3DNow!, 3DNow!+ (64-bit)

C Type		procType	Bytes	Bits
char	b	byte	1	8
short	w	word	2	16
int	d	dword	4	32
long	q	qword	8	64
SSE, SSE2 (128-bit)				
long long			16	128

Table 3-6: Xbox – X86 – PIII – MMX (64-bit)

C Type		procType	Bytes	Bits
char	b	byte	1	8
short	w	word	2	16
int	d	dword	4	32
long	q	qword	8	64
long long			16	128
SSE (128-bit)				
long long			16	128

Table 3-7: GameCube – PowerPC – Gekko (64-bit)

C Type		procType	Bytes	Bits
char	b	byte	1	8
short	h	half-word	2	16
int	w	word	4	32

Table 3-8: G4 PowerPC – AltiVec (128-bit)

C Type		procType	Bytes	Bits
char	b	byte	1	8
short	h	half-word	2	16
int	w	word	4	32

Table 3-9: MIPS (64/128-bit)

C Type		procType	Bytes	Bits
char	b	byte	1	8
short	h	half-word	2	16
int	w	word	4	32
long	d	dword	8	64

Those of you targeting X86-based processors should take heed. It originated as a 16-bit processor with an 8/16-bit data path over 20 years ago, and so its data type sequencing is a little off from the others to maintain its downward compatibility. The other processors are more recent and typically based upon the wider 32/64/128-bit data paths, and so they use the term "half-word," represented by an "h," which skews the entire data type definition table. This changes the meaning of "w," "d," and "q." This book reflects this later mindset and use of half-word, so be careful.

X86 and IBM Personal Computer

Each of the game platforms listed have their architecture frozen except for one: the X86. This is the most complicated of all the processors due to its life span and constant upgrades and enhancements since its first introduction to the marketplace on August 12, 1980, as an 8088 processor in an IBM computer. If the 1983 Charlie Chaplin promoter of the IBM personal computer were to see it now, he would be proud and astonished at all its architectural changes over the years.

For a processor to survive, it must be enhanced to meet the demands of technology, find a second life in an embedded marketplace, or die. Intel and AMD have done just that, but unfortunately in the process, the technology has forked over time and there are now a multitude of flavors of the original 8086 processor core in existence. In addition, AMD has started to remerge the technologies of the 3DNow! extensions and SSE and form the 3DNow! Professional instruction sets.

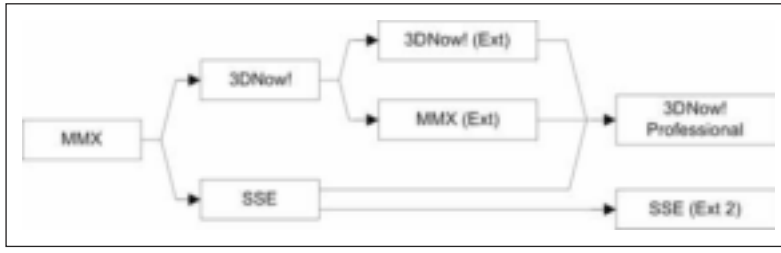


Figure 3-2: Flavor tree of X86 processors

The point is that now there are several X86 SIMD feature sets, and not all processors have them all. The first step is to resolve this. Intel initially did this by developing an instruction with the mnemonic CPUID along with a set of sample processor detection code. AMD adopted the same instruction with their set of sample code. As the technology forked further apart, each company's sample CPUID code emphasized their processors, so programmers have had to merge both companies' code a bit, although AMD's was more diverse.

To make it even more complicated, AMD has recently put out 3DNow! Professional. This is a hybrid of all the X86 instruction sets of the AMD and Intel technologies except the SSE (Extension 2) — at least at the time of the writing of this book! Because of the confusion factor, this book has done them both one better. As this book is about vector programming and not general purpose programming, it does not list all the programming instructions in the source code printed on these

pages. The book would have been too thick, and I would probably have never finished it. In addition, the cross-platform flavor of the book would be lost. However, it is necessary for your programs to know which processor is running so the companion CD contains code related to these chapters.

The function essentially enumerates the processor level and sets SIMD feature bits so that a single application can differentiate between them more easily. With that in mind, here is a breakdown. Note that the following definitions are defined for purposes of differentiation and enumerated use and are standard for this book.

Listing 3-4: \inc???\CpuAsm.h

```
X86 CPU vendors
typedef enum
{
   CPUVEN_UNKNOWN   = 0, // Unknown
   CPUVEN_INTEL     = 1, // Intel
   CPUVEN_AMD       = 2, // AMD
   CPUVEN_CYRIX     = 3, // Cyrix
   CPUVEN_CENTAUR   = 4, // IDT Centaur (WinChip)
   CPUVEN_NATIONAL  = 5, // National Semiconductor
   CPUVEN_UMC       = 6, // UMC
   CPUVEN_NEXGEN    = 7, // NexGen
   CPUVEN_RISE      = 8, // Rise
   CPUVEN_TRANSMETA = 9  // Transmeta
} CPUVEN;

PowerPC CPU vendors
typedef enum
{
   CPUVEN_UNKNOWN   = 0, // Unknown
   CPUVEN_MOTOROLA  = 1, // Motorola
   CPUVEN_IBM       = 2  // IBM
} CPUVEN;

X86 CPU bit flags
typedef enum
{
   CPUBITS_FPU       = 0x0001, // FPU flag
   CPUBITS_MMX       = 0x0002, // MMX flag
   CPUBITS_3DNOW     = 0x0004, // 3DNow! flag
   CPUBITS_FXSR      = 0x0008, // Fast FP Store
   CPUBITS_SSE       = 0x0010, // SSE
   CPUBITS_SSE2      = 0x0020, // SSE (Ext 2)
   CPUBITS_3DNOW_MMX = 0x0040, // 3DNow! (MMX Ext)
   CPUBITS_3DNOW_EXT = 0x0080, // 3DNow! (Ext)
   CPUBITS_3DNOW_SSE = 0x0100  // 3DNow! Professional
} CPUBITS;

PowerPC CPU bit flags
typedef enum
{
   CPUBITS_ALTIVEC   = 0x00000001, // AltiVec
```

```
   CPUBITS_GEKKO     = 0x00000002  // Gekko
} CPUBITS;

MIPS CPU bit flags
typedef enum
{
   CPUBITS_EE        = 0x00000001, // Emotion Engine
   CPUBITS_VU0       = 0x00000002  // VPU0
} CPUBITS;

CPU information
typedef struct CpuInfoType
{
    uint   nCpuId;  // CPU type Identifier
    uint   nFpuId;  // floating-point Unit Id
    uint   nBits;   // Feature bits
    uint   nMfg;    // Manufacturer
} CpuInfo;

void CpuDetect( CpuInfo * const pInfo );
```

The listed information can be obtained by using the included function CpuDetect(). However, from your point of view, who manufactured the CPU is not nearly as important as the bits that CPUBITS has listed above. Each of those bits being set indicates the existence of the associated functionality. Your program would merely check the bit and correlate the correct set of code. If the processor sets the CPUBITS_3DNOW bit, it would need to vector to the 3DNow!-based algorithm. If the CPUBITS_SSE bit is set, it would vector to that set of code. Keep in mind that when I first started this book, neither existed on the same CPU, but while I was writing it, AMD came out with the 3DNow! Professional. This is a union of the two superset families (excluding the SSE2), for which there is also a CPU bit definition. However, that can easily change in the future. My recommendation would be to rate their priority from highest to lowest performance in the initialization logic of your program based upon your application's criteria. Use SSE instructions whenever possible for scalar as well as vector processing. Use 3DNow! for estimate instructions, as they are designed for higher speed calculations despite their lower accuracy. In that way, you will always have the best performance for your code, even on newer machines released after you ship your code. With this in mind, use a mechanism similar to that, which follows and can be found on the companion CD.

Depending on which instruction superset is being used, there are a variety of registers available.

The X86 processor has a dual mode in relationship to the X86 processor's MMX and FPU registers. In these particular cases, whenever there is a need to switch back and forth, the appropriate instruction

needs to be called. In addition, there is a difference between the AMD instruction *femms* and the Intel instruction *emms*. In the case of non-X86 code, the function is not needed, and thus needs to stub out to an empty macro. The same applies when using the SSE instructions, as they use the registers XMM and not MMX or FPU.

Listing 3-5: vmp_SIMDEntry and SIMDExit Definitions

```
vmp_SIMDEntry
#ifdef CC_VMP_WIN32
typedef void (*vmp_SIMDEntryProc)(void);
extern vmp_SIMDEntryProc vmp_SIMDEntry;
#else
#define vmp_SIMDEntry()
#endif

vmp_SIMDExit
#ifdef CC_VMP_WIN32
typedef void (*vmp_SIMDExitProc)(void);
extern vmp_SIMDExitProc vmp_SIMDExit;
#else
#define vmp_SIMDExit()
#endif
```

Note that if your floating-point code gets erratic and appears to have unexpected QNAN or infinity in this book or illegal values, then look for a usage of a FPU or MMX instruction while the other mode was thought to be in effect.

The actual initialization code for a module requires all the function vectors to be set accordingly. In this manner, the code can vector to the correct function handler for the particular processor it is running on instead of having to do a processor type comparison and branch.

Listing 3-6: SIMD function pointers setup

```
CpuInfo cinfo;

char szBuf[ CPU_SZBUF_MAX ];

CpuDetect(&cinfo );               // Detect CPU
cout << CpuInfoStr(szBuf, &cinfo) << endl;

if (CPUBITS_3DNOW_SSE & cinfo.nBits)
  {
    vmp_SIMDEntry = vmp_SIMDEntryAsm3DNow;
    vmp_SIMDExit  = vmp_SIMDExitAsm3DNow;

Insert other 3DNow! and SSE function pointers here
  }
else if (CPUBITS_SSE & cinfo.nBits)
  {
    vmp_SIMDEntry = vmp_SIMDEntryAsmMMX;
    vmp_SIMDExit  = vmp_SIMDExitAsmMMX;
```

```
Insert other SSE function pointers here
  }
else if (CPUBITS_3DNOW & cinfo.nBits)
  {
    vmp_SIMDEntry = vmp_SIMDEntryAsm3DNow;
    vmp_SIMDExit  = vmp_SIMDExitAsm3DNow;

Insert other 3DNow! function pointers here
  }
else if (CPUBITS_MMX & cinfo.nBits)
  {
    vmp_SIMDEntry = vmp_SIMDEntryAsmMMX;
    vmp_SIMDExit  = vmp_SIMDExitAsmMMX;

Insert other MMX function pointers here
  }
else    // Generic Code
  {
    vmp_SIMDEntry = vmp_SIMDEntryGeneric;
    vmp_SIMDExit  = vmp_SIMDExitGeneric;

Insert generic function pointers here
  }
```

With the masking of other bits, such as AltiVec for other processors:

```
if (CPUBITS_ALTIVEC & cinfo.nBits)
  {
  }
else
  {
  }
```

The point is that whatever mechanism you put into place — switch-case statements, lookup tables, etc. — you want to have the best (fastest) set of code available for that processor. The trick, however, is not to use up valuable memory supporting all those combinations. Fortunately, consoles are fixed targets, which can assist you in being more selective. In fact, you can get away with absolute function calls and not function pointers, but that would really be up to you! It all depends upon how you implement your cross-platform capabilities. The code samples on the companion CD use a platform-specific file to connect the dots, so to speak, so it can be abstracted out easily to a platform-specific implementation. Of course, if using direct calls, you would want to have two flavors of function prototypes in the header files, those set up to be function pointer based and those as standard function prototypes.

To be a little more clear:

Function pointer:

```
typedef void (*vmp_QVecAddProc)(vmp3DQVector * const pvD,
                                const vmp3DQVector * const pvA,
                                const vmp3DQVector * const pvB);
extern vmp_QVecAddProc vmp_QVecAdd;
```

```
extern vmp_QVecAddProc vmp_QVecAdd0;
```

Function prototype:

```
void vmp_QVecAddAsm3DNow(vmp3DQVector * const pvD,
                         const vmp3DQVector * const pvA,
                         const vmp3DQVector * const pvB);
```

...and so assignment is simply:

```
vmp_QVecAdd = vmp_QVecAddAsm3DNow;
```

Registers

The following registers and their ranges are for dealing with the SIMD instruction sets directly. They do not include system registers.

Table 3-10: SIMD instruction set with register names and bit widths

SIMD Instruction Set	Registers	Range	Bits
PowerPC	r#	(0...31)	32
	fr#		32/64
PowerPC - AltiVec	r#	(0...31)	32
	fr#		32/64
	vr#		128
PowerPC - Gekko	r#	(0...31)	64
	fr#		32/64
MMX	mm#	(0...7)	64
SSE	xmm#	(0...7)	128
SSE2	xmm#	(0...7)	128
3DNow!	mm#	(0...7)	64
3DNow! Extensions (3DNow+)	mm#	(0...7)	64
3DNow! MMX Extensions (MMX+)	mm#	(0...7)	64
PS2 – EE, TX-79, TX-99 (MMI)	MIPS	(1...31)	8/128
PS2 – VU0/1	vi# vf#	(0...31)	16
		(1...31)	128

Destination and Source Orientations

One more difference between the platforms has to do with the format of the assembly instructions. Depending upon the processor, there are typically two orientations.

One allows three values to be operated upon simultaneously. In C programming, this is a form similar to:

```
D = A + B
```

Proc: PowerPC – AltiVec
MIPS – (Hint: Toshiba)

mnemonic *destination*, *sourceA*, *sourceB*

vaddub vr1,vr1,vr2

The other format uses the destination as one of the sources. In C programming, this is similar to:

```
D += A
```

Proc: X86 – MMX, SSE, SSE2, 3DNow!, etc.

mnemonic destination, source

paddb mm1,mm2

Big and Little Endian

A very important processor specification one needs to be aware of is the endian orientation. This drastically affects how byte ordering affects data orientation. X86 processors are little endian, but MIPS and PowerPC processors, as a default, tend to be big endian.

Table 3-11: SIMD instruction set with endian orientation

SIMD Instruction Set	Endian
PowerPC	Big
X86	Little
MIPS	Little

Dealing with endian orientation can sometimes feel like a pretzel, especially if you work primarily in little endian and need to convert data to the big endian form. Little endian is linear, just like memory, so the more significant byte would be the next (incremental) addressed one in memory. For the size of a data word in big endian, the more significant byte would be the previous (decremental) addressed one in memory.

Big endian has a difference. The most significant byte is first in memory with a progression down to the least significant byte. The cycle then repeats for the next block. In the following diagram, the data in memory is blocked into groups of 128 bits (16 bytes).

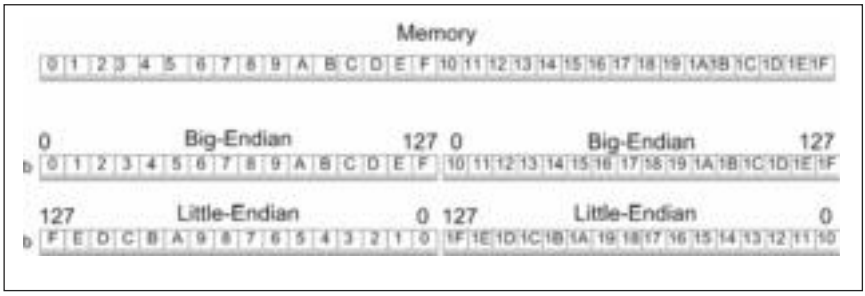


Figure 3-3: Big endian and little endian byte orientations in relationship to memory

Hint: Use descriptive big versus little endian macros to simplify endian conversion.

In the C language, using the following shift to the left by one for a 32-bit data word makes perfect visual sense for big endian because the fourth byte contains the least significant bit (LSB), and data is shifted to the left toward the most significant bit (MSB):

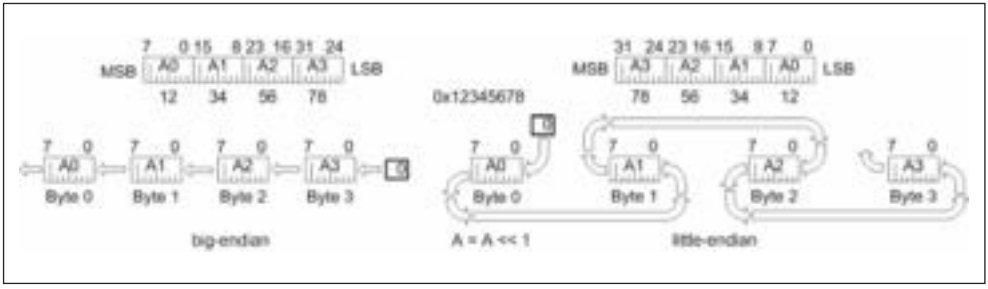


Figure 3-4: Visible connections between individual bytes and the left shift of a 32-bit data element

For little endian, the same shift in C magnifies the value by a factor of two for each bit, but visually it makes no sense because the LSB is on the left. By using a mirrored reflection, it then becomes clear. One last item to remember is that on a per-byte basis, whether big or little endian, bit #7 is always the MSB on the left and bit #0 is the LSB on the right. The endian differentiation is for the byte encoding for the data sizes larger than a byte.

If your application is designed to be multi-platform, then possibly having multiple endian declarations might make your life easier. Having Endian16(), Endian32(), Endian64(), and Endian128() conversion functions are one thing, but having extra declarations, such as BigEndian64() versus LittleEndian64(), that map to either a stub macro or an Endian64 converter will save you some time. The data file being

read will be in a known endian orientation. The target platform knows what it needs, so if the label of big endian is used when the source data is known to be big endian and vice versa for little endian, the use of that declaration will resolve any confusion. This will work for any platform!

Table 3-12: Correlation between little and big endian orientation and whether a byte swap or a stub function is implemented

Source Data	Big Endian Machine	Little Endian Machine
BigEndian( )	stub	byte swap
LittleEndian( )	byte swap	stub

X86 little endian:

```
#define VMP_LITTLE_ENDIAN
```

PowerPC big endian:

```
#define VMP_BIG_ENDIAN
```

This is oversimplifying it, and there are other methods, such as the *bswap* instruction on the X86, but here are some (slow) macros to do the job.

```
#define VMP_ENDIAN32( a )  (((a>>24)&0x000000ff) |
                            ((a>>8)&0x0000ff00) | ((a<<8)&0x00ff0000) | (a<<24))

#define VMP_ENDIAN16( a )  (((a>>8)&0x00ff) | (a<<8))
```

Replacing these macros with some inline code would be very efficient, especially if properly aligned. For example, note that this little function, though not too fast, works with unaligned memory much faster than those macros.

Listing 3-7: Fast generic 32-bit endian conversion

```
int32 VMP_ENDIAN32(int32 val)
{
  uint8 buf[4];

  buf[ 0 ]=*(((uint8*)&val)+3);   // = [3]
  buf[ 1 ]=*(((uint8*)&val)+2);   // = [2]
  buf[ 2 ]=*(((uint8*)&val)+1);   // = [1]
  buf[ 3 ]=*(((uint8*)&val)+0);   // = [0]
  return *(int32*)buf;
}
```

The type of casting camouflages it a bit, but it is merely a byte read-write with inverse offsets. I will leave the actual endian implementation up to you! Just remember that it is preferable to have the tools handle your endian conversion so that the game does not have to. Since tools

exercise the same data over and over for the length of a project, you might as well make them as efficient as possible.

For cross-platform compatibility, I refer to the following as a little pretzel logic. It looks a little twisted, but if you dig a little deeper, it becomes what it is: slicker'n snail snot!

Processor (Big/Little) Endian to (Big/Little) Endian Data Relationship Macros

Listing 3-8: KariType.h

```
#ifdef VMP_BIG_ENDIAN    // Big endian processor

         // Big endian data on big endian processor
     #define VMP_BIG_ENDIAN32                    //stub
     #define VMP_BIG_ENDIAN16                    //stub

         // Little endian data on big endian processor
     #define VMP_LITTLE_ENDIAN32   VMP_ENDIAN32
     #define VMP_LITTLE_ENDIAN16   VMP_ENDIAN16

#else                    // Little endian processor

         // Big endian data on little endian processor
     #define VMP_BIG_ENDIAN32      VMP_ENDIAN32
     #define VMP_BIG_ENDIAN16      VMP_ENDIAN16

         // Little endian data on little endian processor
     #define VMP_LITTLE_ENDIAN32                 //stub
     #define VMP_LITTLE_ENDIAN16                 //stub

#endif
```

Note that same endian to same endian assignment merely stubs out the macro, so no conversion is needed or implemented! One only needs to know what byte ordering the data is in and what ordering is needed to use the appropriate macro. It will then be cross-platform compatible to all other platforms as long as the endian flag is set properly for that platform.

Neat, huh? No extra #ifdef cluttering up the code!

If you require more speed on an X86 and wish to be more daring, investigate the 32-bit *bswap* instruction.

MIPS Multimedia Instructions (MMI)

There are a multitude of flavors for the MIPS processor. They originally started with a MIPS I and have slowly been enhanced over time with each successor supporting the previous set of instructions. At the time of this book's publication, the MIPS instruction set was up to the level

of a MIPS64 processor. This is inclusive of the MIPS V instruction set where Paired Single (PS) precision instructions had been introduced, as well as a MIPS-3D superset for some extra functionality. This operates upon two single-precision values that are stored in two halves of a 64-bit floating-point register. The sample MIPS floating-point code in this book can be considered MIPS64 code but is labeled as MIPS V for purposes of clarity.

For the MIPS V instructions, the PS is appended to the scalar instructions, as in the following sampling of MIPS V supported instructions.

Table 3-13: Naming comparison of scalar single-precision, scalar double-precision, and paired single-precision floating-point values supported by the MIPS V specification

Scalar SPFP	Scalar DPFP	Paired SPFP
abs.s fd, fs	abs.d fd, fs	abs.ps fd, fs
add.s fd, fs, ft	add.d fd, fs, ft	add.ps fd, fs, ft
c.cond.s cc, fs, ft	c.cond.d cc, fs, ft	c.cond.ps cc, fs, ft
madd.s fd, fr, fs, ft	madd.d fd, fr, fs, ft	madd.ps fd, fr, fs, ft
mov.s fd, fs	mov.d fd, fs	mov.ps fd, fs
msub.s fd, fr, fs, ft	msub.d fd, fr, fs, ft	msub.ps fd, fr, fs, ft
mul.s fd, fs, ft	mul.d fd, fs, ft	mul.ps fd, fs, ft
neg.s fd, fs	neg.d fd, fs	neg.ps fd, fs
nmadd.s fd, fr, fs, ft	nmadd.d fd, fr, fs, ft	nmadd.ps fd, fr, fs, ft
nmsub.s fd, fr, fs, ft	nmsub.d fd, fr, fs, ft	nmsub.ps fd, fr, fs, ft
sub.s fd, fs, ft	sub.d fd, fs, ft	sub.ps fd, fs, ft

Other optimized floating-point instructions were added, such as estimated reciprocals and reciprocal square roots with optional precision corrections, which support single (32-bit), double (64-bit), and paired-single (two 32-bit floating-point values packed into a 64-bit register). At the time of this book's publication, the MIPS-3D had recently been introduced and only supported 64-bit processors, but it should just be a matter of time for it to be enhanced for the 128-bit vector model similar to the Toshiba implementation.

Table 3-14: The 13 floating-point instructions used by MIPS processors supporting the MIPS-3D Graphics Extensions

ADDR	Floating-point reduction add
MULR	Floating-point reduction multiply
RECIP1	Estimated reciprocal (first step low precision)
RECIP2	Reciprocal second step for standard precision
RSQRT1	Estimated square root (first step low precision)
RSQRT2	Square root second step for standard precision
CVT.PS.PW	Convert (paired) 32-bit integers to floating-point

CVT.PW.PS	Convert (paired) floating-point to 32-bit integers
CABS	Absolute comparison of floating-point
BC1ANY2F	Branch on any two floating-point (false) conditions
BC1ANY2T	" (true)
BC1ANY4F	Branch on any four floating-point (false) conditions
BC1ANY4T	" (true)

Since the register size of a 64-bit processor is 64-bit, the Boolean operations {*and, nor, or, xor*} are 64 bit as well!

All other operations, however, are still 32 bit, such as using the instruction *add* for a 32-bit addition but the instruction *dadd* for a 64-bit addition!

The original Sony PlayStation used a MIPS R3000A, while the PlayStation 2 uses a MIPS R5400. This latter processor was jointly designed by Toshiba and Sony Computer Entertainment, Inc. and uses a Toshiba TX79 core. On April 2, 2001, Toshiba spun off the company ArTile MicroSystems, Inc., which markets the TX79 Embedded Processor core as a model C790 superscalar microprocessor. It does not contain the VU coprocessor components, but it does have the 64-bit MIPS IV instruction set architectures, as well as the 128-bit multimedia instruction set of which the document TX System RISC – TX79 Core Architecture is available for download from Toshiba's Japanese web site as the file TX79Architecture.pdf.

The next generation MIPS processor TX-99 code named Amethyst is due for release in 2003; it was announced in February 2002 by Toshiba to be a joint venture between themselves and MIPS. It is planned to contain the MIPS-3D ASE (Application Specific Extension) instruction set.

The MIPS processor with multimedia instructions is very similar to that of the SSE2 processor. There is a catch, however. A MIPS processor with an earlier series number than R4000 actually executes the instruction following a memory load, especially a branch. If it's a load, the data is still being loaded and thus not stable, and so the data in the register will not be reliable if an attempt is made to use it right away. In this particular case, there is a need for an extra *NOP* to delay that access. When branching, there is a need for a non-destructive instruction, such as a *NOP*. In the sample code for this book, the macros LDELAY (for load delay) and BDELAY (for branch delay) are utilized. Since the PS2 is an R5400, the LDELAY is actually stubbed in this book as an empty macro, as those latter processors delay if instructed to access a register that is still being loaded. If this is an original PlayStation, however, the *NOP* would indeed be needed, as it uses a MIPS R3000. Some newer

compilers handle the *NOP* automatically, but the macro placement makes this safer and more portable between compilers and assemblers.

Coprocessors are supported and labeled as:

Coprocessor 0	CP0	System control coprocessor
Coprocessor 1	CP1	Floating-point coprocessor
Coprocessor 2	CP2	Application specific (VU for a PS2)
Coprocessor 3	CP3	FPU in MIPS64 architecture

One last point to make has to do with the naming of registers. The compiler is a GNU C/C++ that supports inline assembly, and thus is not processor aware in terms of the proper naming conventions of the registers, so a series of definitions are used to convert from the appropriate MIPS naming convention to the compiler register declaration of $0...$31! Also keep in mind that register $0 is a constant, permanently assigned a value of zero.

Table 3-15: MIPS general registers

zero=$0	$at=$1			ZERO & Assembler tmp.
v0=$2	v1=$3			Function return
a0=$4	a1=$5	a2=$6	a3=$7	Function arguments
t0=$8	t1=$9	t2=$10	t3=$11	Temporary (scratch)
t4=$12	t5=$13	t6=$14	t7=$15	
t8=$24	t9=$25			
s0=$16	s1=$17	s2=$18	s3=$19	Subroutine register vars.
s4=$20	s5=$21	s6=$22	s7=$23	
s8=$30				(frame pointer)
k0=$26	k1=$27			Interrupt/Trap
gp=$gp	sp=$sp			Global/Stack

The floating-point processor is coprocessor #1 (COP1).

Table 3-16: (COP1) MIPS floating-point registers

$f0	$f1	$f2	$f3	Function return
$f4	$f5	$f6	$f7	Temporary (scratch)
$f8	$f9	$f10	$f11	
$f12	$f13	$f14	$f15	Incoming args.
$f16	$f17	$f18	$f19	Temporaries
$f20	$f21	$f22	$f23	Saved temporaries
$f24	$f25	$f26	$f27	
$f28	$f29	$f30	$f31	

For those embedded programmers dealing with an older MIPS I or MIPS II processor, you will find that they do not have 32 floating-point registers, but 16. Single-precision is accessed as even numbers $f0, $f2, etc., and double-precision is loaded in halves, $f0 for the lower 32 bits and $f1 as the upper 32 bits, etc.

Data can be moved between general-purpose registers and floating-point registers:

```
mtc1  t0,$f4          ; Move GPR  to  (COP1) register - $f4.
mfc1  t0,$f4          ; Move GPR from (COP1) register - $f4.

add.s $f4,$f8,$f8     ; Add single-precision floating-point
add.d $f4,$f8,$f8     ; Add double-precision floating-point

lq     t1,0(t0)       ; Move 128-bit mem[t0]  to  register t1.
sq     t1,0(t0)       ; Move 128-bit mem[t0] from register t1.
```

With some compilers, such as those available from Green Hills Software, scalar floating-point arguments are not passed through a general-purpose register, such as (a0 ... a3); they are instead passed through a floating-point register, such as ($f12 ... $f15). Thus, there would be no need for an *MTC1* instruction, as the floating-point register could be used directly.

```
add.s $f4,$f12,$f12    ; Add single-precision floating-point
```

PS2 VU Coprocessor Instruction Supposition

The MIPS processors follow the MIPS guidelines and instruction sets for COP0, COP1, and COP2. Each load/save instruction for MIPS processors append a numerical digit, such as *LD, LQ, MFC1, MTC1, MFC2, MTC2, LDC1, SDC2, LQ2, SQ2*, etc., to move data values from memory to the correct processor and back.

For the MIPS-MMI superset, *LQ* loads 128-bit data and *SQ* saves 128-bit data. Please note that MMI is an acronym that I came up with to represent the Emotion Engine (C790) Multi-Media Instructions for MIPS. Currently, MIPS has a 3D interface standard, but companies have licensed MIPS with their own superset instruction sets. With the advent of MMX, AltiVec, and other SIMD-based instruction sets, packed integer instructions are becoming a mainstay on MIPS. Toshiba has just jumped this convention from the MIPS IV 64 bit to 128 bit.

The VU0 coprocessor is the COP2, and it has four 32-bit single-precision floating-points, thus a 128-bit data path. As a COP2, *LQC2* loads 128-bit vector floating-point by transferring from memory to coprocessor #2 (C2 i.e., *LQC2*), then *SQC2* saves 128-bit vector floating-point by transferring from coprocessor #2 (C2 — i.e., *SQC2*) to memory.

These are vector floating-point registers, thus the reference $vf#.

Table 3-17: (COP2) MIPS vector single-precision floating-point registers

vf0=$vf0	vf1=$vf1	vf2=$vf2	vf3=$vf3
vf4=$vf4	vf5=$vf5	vf6=$vf6	vf7=$vf7
vf8=$vf8	vf9=$vf9	vf10=$vf10	vf11=$vf11
vf12=$vf12	vf13=$vf13	vf14=$vf14	vf15=$vf15
vf16=$vf16	vf17=$vf17	vf18=$vf18	vf19=$vf19
vf20=$vf20	vf21=$vf21	vf22=$vf22	vf23=$vf23
vf24=$vf24	vf25=$vf25	vf26=$vf26	vf27=$vf27
vf28=$vf28	vf29=$vf29	vf30=$vf30	vf31=$vf31

```
lqc2   vf4,0(t0)     ; Move mem[t0]  to  (COP2) Register – vf#.
sqc2   vf4,0(t0)     ; Move mem[t0] from (COP2) Register – vf#.

vadd.xyzw vf4,vf5,vf6    ; Vector Add
```

For more information, check out the MIPS and PS2 references in the back of this book.

Gekko Supposition

Before we move on any further, now is a good time to make some suppositions as to the Gekko superset instruction set flavored PowerPC processor used in the GameCube by Nintendo. I am not an authorized game developer for this platform, so I have to use my best guess as to this processor. Those of you who are actually in the loop, bear with me! Keep in mind, however, that the principles you learn here can be applied to your code.

Nintendo and IBM have both been tight-lipped about all but a few superficial details as to the Gekko's capabilities. However, in two different interviews, some information critical to our concerns and this book were let slip by Mike West, Multimedia Architect and Peter Sandon, PowerPC Performance Manager.

> Mike West: *"You have to understand what's done with the floating-point unit. It is the conventional 64-bit PowerPC floating-point unit, but the adaptations we made to it allow two simultaneous 32-bit calculations to occur. Basically, each instruction is completed every cycle." December 12, 2001*
>
> *"One of the modifications it made was to cut the 64-bit floating-point unit in half, allowing it to do two 32-bit floating-point operations every cycle." May 17, 2001*

> Peter Sandon: *"I should say that it is a significant effort to implement the paired-single floating-point function beyond what is there in the standard PowerPC." December 12, 2001*

I have underlined the key items. The first item to remember is that the PowerPC supports 32, 32-bit general-purpose registers (GPRs) {r0...r31} and 32, 64-bit floating-point registers (FPRs) that support both single-precision floating-point as well as 64-bit double-precision floating-point. Each data type requires a unique load/save instruction.

8-bit byte	lbz rD,d(rA)
16-bit half-word	lhz rD,d(rA)
32-bit word	lwz rD,d(rA)
32-bit single-precision floating-point	lfs frD,d(rA)
64-bit double-precision floating-point	lfd frD,d(rA)

Since a different load instruction is used to differentiate between the integer, single-precision floating-point, and double-precision floating-point, both floating-point types use the same 64-bit registers, and the comments in the interviews discussed cutting the 64-bit floating-point unit in half, it makes perfect sense (at least to me) that the same floating-point register (fr#) is used. Thus, a new method for loading packed single-precision floating-point is needed to differentiate it from the other data types.

This can be done through either an operational mode change or a new instruction. Regardless of the mechanism, a new set of instructions for dealing with paired single-precision floating-point is needed. Since manufacturers seem to use *v* for vector or *p* for (paired/packed), a prefix of *p* seems appropriate here. This is similar to what MIPS did for the MIPS-3D specification.

Since the data is in 64-bit pairs, pay attention to a similar 64-bit processor in this book, such as the AMD 3DNow! instruction set, and its dealing with single-precision floating-point.

One last item: In all the interviews and minimal technical publications, I found no mention of the packed integer capability for the Gekko chip. If this is indeed the case, keep in mind the general PowerPC code in this book for simulating integer-based SIMD instructions, as that functionality is needed.

Of course, those few of you who are authorized by Nintendo to be privy to the unpublished Gekko documentation by IBM will know the correct instruction mnemonics, but the principles of this book should apply just the same.

Function Wrappers

Dealing with aligned memory access to 128-bit registers is very convenient, but when accessing unaligned memory, extra wrapper code must be inserted to deal with reading and writing of that data for the calculation. The following are some of the standard function wrappers used for each processor and/or platform. Typically, integer and floating-point must be handled differently internally due to their different instruction sets, but they both follow the same function declaration and argument naming conventions that have already been discussed in Chapter 1.

```
void vmp_FUNCTION( void * const pvD,   // Destination
             const void * const pvA,   // Source A
             const void * const pvB ); // Source B
```

These wrappers are being defined here, as they are not shown in detail in later chapters within this book so as not to clutter up the pages or bore you with the same stuff over and over again. In this way, I am trying to help you get your money's worth for this book. If you are ever in doubt, confused, or just plain annoyed, look at the printed function in its entirety on the companion CD. I tried to keep each file under 1000 lines of code.

You should note the placement of assembler/compiler commands, such as alignment, and processor declarations. Also note the inclusion of the const declaration and assertions.

Integer Function Wrappers

It is important to understand the basic shells that will be needed when writing functions for a particular processor, whether the data is aligned or not or uses four element vectors or only three. This section may seem redundant for now, but it will make understanding in later chapters easier. You might also find it easier when it comes to the task of writing your own code.

Sometimes the generic code is all that is supported, and so any optimized code is best, such as the following generic X86 assembly code. I realize that it is only emulated vector processing, but sometimes that is the best way to go to replace missing functionality.

vmp_FUNCTION (Generic X86 Assembly)

Remember that there are so many flavors of X86 that inline assembly commands are needed to turn on and off particular features, such as indicated in bold, on a function-by-function basis. A compiler cannot typically do this; therefore, I personally prefer to use an assembler, such as masm, instead of inline assembly code with a C language compiler.

Listing 3-9: ???X86.asm

```
  align 16
  .586
        public  vmp_FUNCAsm
vmp_FUNCAsm    proc    near, pvD:ptr, pvA:ptr, pvB:ptr
  push  ebp
  push  ebx
  push  esi
  push  edi

  mov   eax,[esp+16+8]  ;pvA
  mov   ebx,[esp+16+12] ;pvB
  mov   ebp,[esp+16+4]  ;pvD

ASSERT_PTR4(eax)        ;pvA
ASSERT_PTR4(ebx)        ;pvB
ASSERT_PTR4(ebp)        ;pvD

    ; Read lower 64bits
  mov   ecx,[eax+0]     ; Read  A Bits {31...0}
  mov   esi,[eax+4]     ;       A Bits {63...32}
  mov   edx,[ebx+0]     ; Read  B Bits {31...0}
  mov   edi,[ebx+4]     ; Read  B Bits {63...32}
```

Insert instructions here for lower 64-bit calculation.

```
  FUNC  ecx,edx         ;      Lower Bits  {31...0}
  FUNC  esi,edi         ;      Upper Bits  {63...32}

  mov   [ebp+0],ecx     ; Write lower bits {31...0}
  mov   [ebp+4],esi     ;   "          "   {63...32}

    ; Read upper 64bits
  mov   ecx,[eax+8]     ; Read  A Bits {95...64}
  mov   esi,[eax+12]    ;       A Bits {127...96}
  mov   edx,[ebx+8]     ; Read  B Bits {95...64}
  mov   edi,[ebx+12]    ;       B Bits {127...96}
```

Insert instructions here for upper 64-bit calculation.

```
  FUNC  ecx,edx         ;      Lower Bits  {95...64}
  FUNC  esi,edi         ;      Upper Bits  {127...96}

  mov   [ebp+8],ecx     ; Write upper bits {95...64}
  mov   [ebp+12],esi    ;   "          "   {127...96}

  pop   edi
  pop   esi
```

```
  pop   ebx
  pop   ebp
  ret
vmp_FUNCAsm  endp
```

vmp_FUNCTION (MMX-X86 Assembly) (Un)aligned

The MMX instruction set can work with misaligned memory, thus the reasoning for the (un)aligned declaration! Note the instruction *emms* is commented out. This is used to switch modes between MMX and FPU operational mode. It is normally needed, but since the idea is to replace most of the FPU calls with MMX calls, it is not needed at this function level. Instead, it would be moved up into a higher layer where entire sections of code would be of one type or the other. The instruction takes too long to be called each time a function is called, therefore a need arises for well-organized code to help maximize processor efficiency.

Listing 3-10: ???X86.asm

```
  align 16
  .MMX
        public  vmp_FUNCAsmMMX
vmp_FUNCAsmMMX     proc    near pvD:ptr, pvA:ptr, pvB:ptr
;;;         emms
ASSERT_PTR4(pvA)
ASSERT_PTR4(pvB)
ASSERT_PTR4(pvD)

  mov   ecx,pvB
  mov   eax,pvA
  mov   edx,pvD

  movq  mm0,[ecx+0]     ; Read B Data Bits {63...0}
  movq  mm1,[ecx+8]     ;                  {127...64}
  movq  mm2,[eax+0]     ; Read A Data Bits {63...0}
  movq  mm3,[eax+8]     ;                  {127...64}
```

Insert instructions here for lower and upper 64-bit calculation.

```
  FUNC  mm0,mm2         ;      Lower Bits  {63...0}
  FUNC  mm1,mm3         ;      Upper Bits  {127...64}

  movq  [edx+0],mm0     ; Write Data Bits  {63...0}
  movq  [edx+8],mm1     ;                  {127...64}

;;;         emms
  ret
vmp_FUNCAsmMMX  endp
```

vmp_FUNCTION (SSE2-X86 Assembly) Aligned

Here is where things get more interesting. First, the SSE has no 128-bit integer support, as it was primarily for 128-bit floating-point and the addition of a needed 64-bit integer MMX instruction. Therefore, the SSE2 is presented here instead, as a full series of 128-bit integer instructions were added to double the data width of the 64-bit MMX instructions. An assertion will occur if data is misaligned, and the 128-bit integer memory movement instruction *movdqa* is used to access that memory. By replacing that instruction with *movdqu*, the memory can be misaligned. The problem is that the function will run slower due to the extra time required for the CPU to automatically realign that memory. Also replace the following macro assertion ASSERT_PTR16() with ASSERT_PTR4() to allow a lower 32-bit alignment instead of the 128-bit.

Listing 3-11: ???X86M.asm

```
  align  16
  .XMM
        public  vmp_FUNCAsmSSE2
vmp_FUNCAsmSSE2     proc    near pvD:ptr, pvA:ptr, pvB:ptr

ASSERT_PTR16(pvA)
ASSERT_PTR16(pvB)
ASSERT_PTR16(pvD)

  mov    ecx,pvB
  mov    eax,pvA

  movdqa xmm0,[ecx]     ; Read B Data Bits {127...0}
  movdqa xmm1,[eax]     ; Read A Data Bits {127...0}

  mov    edx,pvD

Insert instruction here for 128-bit calculation.

  FUNC  xmm0,xmm1       ;             Bits  {127...0}

  movdqa [edx],xmm0     ; Write D Data Bits {127...0}
  ret
vmp_FUNCAsmSSE2  endp
```

vmp_FUNCTION (PowerPC) (Un)aligned

Some Macintosh computers only support the PowerPC processor (G3) and some support the AltiVec instruction set (G4 and above). For those that do not have AltiVec capability, the functionality must be emulated. In this particular case, a function such as follows is defined.

Listing 3-12: ???PPC.cpp

```
void vmp_FUNCPPC(  void * const pvD,   // r4 – Destination
             const void * const pvA,   // r5 – A Source
             const void * const pvB )  // r6 – B Source
{
  unsigned int register a0, a1, a2, a3, b0, b1, b2, b3;

#ifndef USE_ASSERT
#pragma unused(pvD, pvA, pvB)
#endif

  ASSERT_PTR4(pvB);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvD);

__asm {
  lwz   a0,0(r4)        // Read A Bits {0...31}
  lwz   a1,4(r4)        //      "      {32...63}
  lwz   a2,8(r4)        //      "      {64...95}
  lwz   a3,12(r4)       //      "      {96...127}

  lwz   b0,0(r5)        // Read B Bits {0...31}
  lwz   b1,4(r5)        //      "      {32...63}
  lwz   b2,8(r5)        //      "      {64...95}
  lwz   b3,12(r5)       //      "      {96...127}
```

Insert instructions here for the four, 32-bit block calculations.

```
  FUNC  a0,a0,b0        //       Bits  {0...31}
  FUNC  a1,a1,b1        //       Bits  {32...63}
  FUNC  a2,a2,b2        //       Bits  {64...95}
  FUNC  a3,a3,b3        //       Bits  {96...127}

  stw   a0,0(r3)        // Write Bits  {0...31}
  stw   a1,4(r3)        //      "      {32...63}
  stw   a2,8(r3)        //      "      {64...95}
  stw   a3,12(r3)       //      "      {96...127}
  }
}
```

Also note that the PowerPC uses 32-bit registers, and thus the memory can be aligned down to 32 bits. Here it is very easy to use inline code, so the standard C compiler can be utilized. Note the #pragma. If assertions are turned off, the compiler will complain about the arguments not being used. Since those arguments are actually equated to the register r4...r6, they are used instead, as the functions use register passing.

Another item to note is the "unsigned int register" declaration that tells the compiler to use registers instead of stack memory, which helps to keep the function as fast as possible. In keeping with good programming practices, remember to use the const as well as assertions. Also note the bits comment is reversed, as this is a big endian implementation!

vmp_FUNCTION (AltiVec Assembly) Aligned

The nice thing about AltiVec is that there really is not a need to write it in assembly code, as a nice set of functions already exists in library form. So it is merely a matter of calling the library, especially if the data is aligned properly. An item to note (of which I will continuously remind you) is that there is no memory alignment exception error. The lower four address lines are wired to zero so that an unaligned memory access merely accesses the wrong memory! Also, since the data is being converted from this book's cross-platform declarations of vmp3DQVector to the AltiVec libraries, a type cast to (vector float *) is required. A union could have just as easily been used, but I needed to keep this book visual and understandable.

This can be done using VPREM and shifting for the load, but writing data back gets very messy. Four-byte alignment can almost be guaranteed, thus making the following a better function wrapper.

Listing 3-13: ???Altivec.cpp

```
void vmp_FUNCBVAltivec( uint8 * const pbD,
                  const uint8 * const pbA,
                  const uint8 * const pbB)
{
  ASSERT_PTR16(pbD);
  ASSERT_PTR16(pbA);
  ASSERT_PTR16(pbB);

  *(vector unsigned char *)pbD = vec_FUNC(
                            (*(vector unsigned char *)pbA),
                            (*(vector unsigned char *)pbB));
}
```

vmp_FUNCTION (AltiVec Assembly) Unaligned

As mentioned previously, unaligned memory needs special handling. The data must be moved to aligned memory, then loaded, processed, and written back out a field at a time to access the proper data. Note the use of local stack arguments. (The vector declaration forces the local arguments to be 128-bit aligned on the stack!)

Listing 3-14: ???Altivec.cpp

```
void vmp_FUNCBVAltivec0(uint8 * const pbD,
                  const uint8 * const pbA,
                  const uint8 * const pbB)
{
  vector unsigned char vD, vA, vB;

  ASSERT_PTR4(pbD);
  ASSERT_PTR4(pbA);
```

```
  ASSERT_PTR4(pbB);

  // Load each A & B 8-bit array into 128-bit aligned stack.
  *(((uint32 *)&vA)+0) = *(((uint32 *)pbA)+0);
  *(((uint32 *)&vA)+1) = *(((uint32 *)pbA)+1);
  *(((uint32 *)&vA)+2) = *(((uint32 *)pbA)+2);
  *(((uint32 *)&vA)+3) = *(((uint32 *)pbA)+3);

  *(((uint32 *)&vB)+0) = *(((uint32 *)pbB)+0);
  *(((uint32 *)&vB)+1) = *(((uint32 *)pbB)+1);
  *(((uint32 *)&vB)+2) = *(((uint32 *)pbB)+2);
  *(((uint32 *)&vB)+3) = *(((uint32 *)pbB)+3);
```

Insert instruction here for 128-bit calculation.

```
  vD = vec_FUNC( vA, vB );

    ; Save each 8-bit result into unaligned memory.
  *(((uint32 *)pbD)+0) = *(((uint32 *)&vD)+0);
  *(((uint32 *)pbD)+1) = *(((uint32 *)&vD)+1);
  *(((uint32 *)pbD)+2) = *(((uint32 *)&vD)+2);
  *(((uint32 *)pbD)+3) = *(((uint32 *)&vD)+3);
}
```

vmp_FUNCTION (MIPS-MMI) Aligned

Aligned MIPS code is very simple to manipulate, as it is merely loaded, processed, and written to memory.

Listing 3-15: ???MMI.s

```
  .align 3
  .global vmp_FUNCMMI
vmp_FUNCMMI:

ASSERT_PTR16(a1)
ASSERT_PTR16(a2)
ASSERT_PTR16(a0)

  lq   t1, 0(a1)        // pvA Read Data Bits {127...0}
  lq   t2, 0(a2)        // pvB                {127...0}
  LDELAY                // NOP - Load Delay Slot
```

Insert instruction here for 128-bit calculation.

```
  FUNC  t0,t1,t2        //       Bits  {127...0}

  sq   t0, 0(a0)        // pvD Write Data Bits {127...0}
  j    ra               // return
  BDELAY                // NOP = Branch Delay Slot
```

Did you notice the LDELAY and BDELAY? Refer back to the "MIPS Multimedia Instructions (MMI)" section for review.

vmp_FUNCTION (MIPS-MMI) Unaligned

That was pretty clean, but what happens when data is unaligned on a MIPS processor? It is not fun. A MIPS processor that supports 128-bit MMI instructions with 128-bit registers typically has a 64-bit general-purpose programming instruction set that uses the lower 64 bits of those registers. When data is misaligned, there are two loads to load the left and right halves of misaligned 64-bit memory using the instructions *ldl* (load left) and *ldr* (load right). There is no misaligned 128-bit access, so the data needs to be loaded 64 bits at a time in fragments and then merged. There is one saving grace when dealing with Boolean logic, however. Since each bit in Boolean logic is isolated from each other and the general-purpose instructions deal with 64 bits (the lower 64 bits of the 128-bit register), there is no need to merge the two realigned 64-bit halves back into an aligned 128-bit value just to have to split it again. So the following displays this faster Boolean dual 64-bit solution:

Listing 3-16: ???MMI.s

```
  .align 3
  .global vmp_FUNCMMI0
vmp_FUNCMMI0:

ASSERT_PTR4(a1)
ASSERT_PTR4(a2)
ASSERT_PTR4(a0)

  ldl   t1, 7(a1)       // pvA Read Bits {63...0}   (left)
  ldr   t1, 0(a1)       //  "           "           (right)
  ldl   t3, 15(a1)      //  "            {127...64} (left)
  ldr   t3, 8(a1)       //  "           "           (right)

  ldl   t2, 7(a2)       // pvB Read Bits {63...0}   (left)
  ldr   t2, 0(a2)       //  "           "           (right)
  ldl   t4, 15(a2)      //  "            {127...64} (left)
  ldr   t4, 8(a2)       //  "           "           (right)

Insert instructions here for lower and upper 64-bit calculation.

  FUNC  t0,t1,t2        //         Bits  {63...0}
  FUNC  t1,t3,t4        //         Bits  {127...64}

  sdl   t0, 7(a0)       // pvD Write Bits {63...0}   (left)
  sdr   t0, 0(a0)       //  "            "           (right)
  sdl   t1, 15(a0)      //  "             {127...64} (left)
  sdr   t1, 8(a0)       //  "            "           (right)
  j     ra              // return
  BDELAY                // NOP = Branch Delay Slot
```

If the data is not Boolean but packed data, then it needs to be dealt with in a 128-bit form. In this particular case, replace the previous lines

marked in bold with the following code. The *pcpyld* command essentially copies the lower bits of one register into the upper bits of the other, thus merging the data back into a 128-bit form. The *pcpyud* command copies the upper 64 bits into the lower 64 bits of another register, thus reversing the process.

```
pcpyld t1, t3, t1     // A Data Bits {127...0}
pcpyld t2, t4, t2     // B           {127...0}
```

Insert instruction here for 128-bit calculation.

```
FUNC  t0,t1,t2        //       Bits  {127...0}  D=f(A,B)

pcpyud t1, t0, t0     // D  {63...0  127...64}

// t0=lower 64bits  t1=upper 64bits
```

Single-Precision Function Quad Vector Wrappers

Single-precision floating-point function wrappers are very similar to those used for integer-based functions, but some processors treat the read and write of floating-point data differently than that of integer data. Another difference is that in the case of packed integers, a full 128 bits of data is processed. In the case of floating-point, the vector is either a 96-bit three-float vector or a 128-bit four-float (quad) vector. If you recall the three-float vector, which is referred to as just a vector, it is typically contained individually or as an array of structures. The latter, a QVector, is typically used in data organized as a structure of arrays. For the sake of simplicity, the QVector will be presented first.

vmp_FUNCTION (3DNow!-X86 Assembly) (Un)aligned

The 3DNow! instruction set uses the MMX registers for processing single-precision floating-point in parallel. Although the register size is only 64 bits, the pipelining architecture handles two registers simultaneously, effectively giving the 128-bit calculation. But both halves have to be handled individually. Memory can be misaligned, but a stall will occur. The .K3D indicates that the code contains 3DNow! instructions, and the *femms* instruction is AMD's fast *emms* instruction. It is commented out, as the code in this book was written so that FPU/MMX mode is switched at a higher level. Note the use of assertions. Also note the indicators of the field {XYZW} and which ones are being used for that particular instruction in the comment area. Also, in terms of

efficiency, the general register ecx should be used instead of ebx, as it has to be preserved on the stack. But for purposes of visual explanation, the ebx register is typically used for the code snippet portions in the book. The following, however, are not code snippets but full functions, so ecx is used here instead so the code would not be cluttered with an extra push and pop of the ebx register.

Listing 3-17: QV??3DX86M.asm

```
  align 16
  public    vmp_QVec FUNC Asm3DNow
vmp_QVec FUNC Asm3DNow  proc near , vD:ptr, vA:ptr, vB:ptr
  .K3D
;;;        femms
ASSERT_PTR4(vD)
ASSERT_PTR4(vA)
ASSERT_PTR4(vB)

  mov   eax,vA          ; Vector A_xyzw
  mov   ecx,vB          ; Vector B_xyzw
  mov   edx,vD          ; Vector Destination_xyzw

  movq  mm0,[eax+0]     ;vA.xy   {A_y A_x}
  movq  mm2,[ecx+0]     ;vB.xy   {B_y B_x}
  movq  mm1,[eax+8]     ;vA.zw   {A_w A_z}
  movq  mm3,[ecx+8]     ;vB.zw   {B_w B_z}
```

Insert instructions here for lower and upper 64-bit calculation.

```
  FUNC  mm0,mm2         ;       Bits  {63...0}
  FUNC  mm1,mm3         ;       Bits  {127...64}

  movq  [edx+0],mm0     ;   {D_y=f(A_y,B_y) D_x=f(A_x,B_x)}
  movq  [edx+8],mm1     ;   {D_w=f(A_w,B_w) D_z=f(A_z,B_z)}
;;;     femms
        ret
vmp_QVec FUNC Asm3DNow endp
```

vmp_FUNCTION (SSE-X86 Assembly) Aligned and Unaligned

This is very similar to the packed integer instructions, but in the case of single-precision floating-point, a different set of memory movement instructions needs to be used. For aligned memory *movaps* (Move Aligned Single-Precision) and unaligned *movups* (Move Unaligned Single-Precision), do not forget to replace the 128-bit assertion ASSERT_PTR16 with the 32-bit assertion ASSERT_PTR4.

Listing 3-18: QV??3DX86M.asm

```
  align  16
  .XMM
       public  vmp_QVec FUNC AsmSSE
vmp_QVec FUNC AsmSSE  proc  near vD:ptr, vA:ptr, vB:ptr
;;;        emms
ASSERT_PTR16(vA)
ASSERT_PTR16(vB)
ASSERT_PTR16(vD)

  mov    ecx,vB          ; Vector B
  mov    eax,vA          ; Vector A
  movaps xmm1,[ecx]      ;vB.xyzw {Bw Bz By Bx}
  movaps xmm0,[eax]      ;vA.xyzw {Aw Az Ay Ax}
  mov    edx,pvD

Insert instruction here for 128-bit calculation.

  FUNC  xmm0,xmm1        ;       Bits  {127...0}

  movaps [edx],xmm0      ; { Dw=f(Aw,Bw) Dz=f(Az,Bz)
  ret                    ;   Dy=f(Ay,By) Dx=f(Ax,Bx)}
vmp_QVec FUNC AsmSSE  endp
```

vmp_FUNCTION (AltiVec Assembly) Aligned

The nice thing about AltiVec is that there is really no need to write it in assembly code, as a nice set of functions already exists in library form. So it is merely a matter of calling the library, especially if the data is aligned properly. An item to note (of which I will again continuously remind you) is that there is no memory alignment exception error. The lower four address lines are wired to zero so that an unaligned memory access merely accesses the wrong memory. Also, since the data is being converted from this book's cross-platform declarations of vmp3DQVector to the AltiVec libraries, a type cast to (vector float *) is required. Déjà vu!

Listing 3-19: QV??3DAltivec.cpp

```
void vmp_QVec FUNC Altivec( vmp3DQVector * const pvD,
                    const vmp3DQVector * const pvA,
                    const vmp3DQVector * const pvB)
{
  ASSERT_PTR16(pvD);
  ASSERT_PTR16(pvA);
  ASSERT_PTR16(pvB);

  *(vector float *)pvD = vec_FUNC((*(vector float *)pvA),
                                  (*(vector float *)pvB));
}
```

vmp_FUNCTION (AltiVec Assembly) Unaligned

As mentioned previously, unaligned memory needs special handling. The data must be moved to aligned memory, then loaded, processed, and written back out a field at a time to access the proper data. Note the use of local stack arguments. (The vector declaration forces it to be 128-bit aligned on the stack for AltiVec.)

Listing 3-20: QV??3DAltivec.cpp

```
void vmp_QVec FUNC AltivecO(vmp3DQVector * const pvD,
                    const vmp3DQVector * const pvA,
                    const vmp3DQVector * const pvB)
{
  vector float vD, vA, vB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

    ; Load each A & B float into 128-bit aligned stack.
  ((vmp3DQVector *) &vA)->x = pvA->x;
  ((vmp3DQVector *) &vA)->y = pvA->y;
  ((vmp3DQVector *) &vA)->z = pvA->z;
  ((vmp3DQVector *) &vA)->w = pvA->w;

  ((vmp3DQVector *) &vB)->x = pvB->x;
  ((vmp3DQVector *) &vB)->y = pvB->y;
  ((vmp3DQVector *) &vB)->z = pvB->z;
  ((vmp3DQVector *) &vB)->w = pvB->w;
```

Insert instruction here for 128-bit calculation.

```
  vD = vec_FUNC( vA, vB );

    ; Save each resulting float into unaligned memory.
  pvD->x = ((vmp3DQVector *) &vD)->x;
  pvD->y = ((vmp3DQVector *) &vD)->y;
  pvD->z = ((vmp3DQVector *) &vD)->z;
  pvD->w = ((vmp3DQVector *) &vD)->w;
```

vmp_FUNCTION (MIPS-VU0 Assembly) Aligned

Since this is the first single-precision floating-point calculation for MIPS, it will appear to be obscure due to it actually having been embedded within a GCC (GNU C Compiler), but it should become apparent that it is using similar techniques, except for one. This processor has the capability of writing data files {XYZW} depending on how flags are set. The other fields are in essence preserved, making it pretty cool (and much easier in some ways) to program.

For more details, check out your PS2 Linux Kit or devTool manual.

This processor is really nice, as if working with properly aligned memory; the only difference between a tri-float vector and a four-float quad vector is that the fourth element {W} is merely attached to the field specifier: .xyz versus .xyzw.

First, load 128-bit vectors from memory into VU registers (vector float #) vf4 and vf5. Note that %1 and %2 are pointers *pvA* and *pvB*, and the 0x0(%1) means the contents of memory are referenced by *pvA* with a displacement of 0 bytes. 0x0(%2) means the same zero offset but with the *pvB* reference. Note that memory must be aligned, or an exception will occur!

```
lqc2      vf4,0x0(%1)      #pvA
lqc2      vf5,0x0(%2)      #pvB
```

If you recall from earlier, *lqc2* is used to load a quad word into a register in the coprocessor #2.

This will be pretty standard across all the vector code in this book.

Listing 3-21: QV??3DVU.c

```
void vmp_QVecAddVU(vmp3DQVector * const pvD,
             const vmp3DQVector * const pvA,
             const vmp3DQVector * const pvB)
{
  ASSERT_PTR16(pvD);
  ASSERT_PTR16(pvA);
  ASSERT_PTR16(pvB);

asm __volatile__("
  lqc2  vf4,0x0(%1)    # Read vA_xyzw
  lqc2  vf5,0x0(%2)    # Read vB_xyzw
Insert instruction here for 128-bit calculation.
  FUNC vf6,vf4,vf5     # D.xyzw = f(A.xyzw, B.xyzw)

  sqc2  vf6,0x0(%0)    # Write vD_xyzw
  ":
   : "r" (pvD) , "r" (pvA), "r" (pvB)
  );

}
```

vmp_FUNCTION (MIPS-VU0 Assembly) Unaligned

The data here is also loaded a field at a time into a 128-bit aligned data structure and then loaded, processed, and written back a field at a time.

Listing 3-22: QV??3DVU.c

```
void vmp_QVec FUNC VU0(vmp3DQVector * const pvD,
               const vmp3DQVector * const pvA,
               const vmp3DQVector * const pvB)
{
  sceVu0FVECTOR vD, vA, vB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  vA[0] = pvA->x;
  vA[1] = pvA->y;
  vA[2] = pvA->z;
  vA[3] = pvA->w;

  vB[0] = pvB->x;
  vB[1] = pvB->y;
  vB[2] = pvB->z;
  vB[3] = pvB->w;

asm __volatile__("
  lqc2  vf4,0x0(%1)
  lqc2  vf5,0x0(%2)
```

Insert instruction here for 128-bit calculation.

```
  FUNC vf6,vf4,vf5     # D.xyzw = f(A.xyzw, B.xyzw)

  sqc2  vf6,0x0(%0)
  ":
   : "r" (vD) , "r" (vA), "r" (vB)
  );

  pvD->x = vD[0];
  pvD->y = vD[1];
  pvD->z = vD[2];
  pvD->w = vD[3];
}
```

Double-Precision Function Quad Vector Wrappers

vmp_FUNCTION (SSE2-X86 Assembly) Aligned and Unaligned

This is very similar to the single-precision instructions, but in the case of double-precision floating-point, a different set of memory movement instructions needs to be used. For aligned memory *movapd* (Move Aligned Double-Precision) and unaligned *movupd* (Move Unaligned Double-Precision), do not forget to replace the 128-bit assertion ASSERT_PTR16 with the 64-bit assertion ASSERT_PTR8.

Listing 3-23: QDV??3DX86M.asm

```
  align  16
  .XMM
       public  vmp_QDVec FUNC AsmSSE2
vmp_QDVec FUNC AsmSSE2  proc  near vD:ptr, vA:ptr, vB:ptr
;;;        emms
ASSERT_PTR16(vA)
ASSERT_PTR16(vB)
ASSERT_PTR16(vD)

  mov    ecx,vB          ; Vector B
  mov    eax,vA          ; Vector A
  movapd xmm2,[ecx]      ;vB.xy {By Bx}
  movapd xmm0,[eax]      ;vA.xy {Ay Ax}
  movapd xmm3,[ecx+16]   ;vB.zw {Bw Bz}
  movapd xmm1,[eax+16]   ;vA.zw {Aw Az}
  mov    edx,pvD

Insert instruction here for 2x128-bit calculation.

  FUNC  xmm0,xmm2        ;       Bits  {127...0}
  FUNC  xmm1,xmm3        ;       Bits  {255...128}

  movapd [edx],xmm0      ; { Dy=f(Ay,By) Dx=f(Ax,Bx)
  movapd [edx+16],xmm1   ; { Dw=f(Aw,Bw) Dz=f(Az,Bz)
  ret
vmp_QDVec FUNC AsmSSE2  endp
```

Single-Precision Function Vector Wrappers

These wrappers are kind of interesting because a vector has three floats and the fourth value is not guaranteed to be a float, nor can it be trusted to be a floating-point value. Also, the fourth float of the destination needs to be protected from change. So it almost does not matter if the data is aligned, as it has to be copied to an aligned data block, the fourth value has to be set to something neutral like a value of zero or one, and the result not written back to the destination. Some processors merely return an invalid result for the bad element and some cause an exception.

vmp_FUNCTION (3DNow!-X86 Assembly) (Un)aligned

The three-float vector is easy to preserve with the 3DNow! instruction set, as the fourth value {.w} is merely loaded as a single float, thus setting a zero into the {.w} position for calculation purposes. The data is then written back as a 64-bit {.xy} and 32-bit {.z} only.

Listing 3-24: V??3DX86M.asm

```
  align  16
  public  vmp_Vec FUNC Asm3DNow
vmp_Vec FUNC Asm3DNow  proc  near , vD:ptr, vA:ptr, vB:ptr
  .K3D
;;;        femms
ASSERT_PTR4(vA)
ASSERT_PTR4(vB)
ASSERT_PTR4(vD)

  mov   eax,vA                ; Vector A
  mov   ecx,vB                ; Vector B
  mov   edx,vD                ; Vector Destination

  movq  mm0,[eax]                     ;vA.xy   {Ay Ax}
  movq  mm2,[ecx]                     ;vB.xy   {By Bx}
  movd  mm1,(vmp3DVector PTR [eax]).z ;vA.z0   {0 Az}
  movd  mm3,(vmp3DVector PTR [ecx]).z ;vB.z0   {0 Bz}

Insert instructions here for lower and upper 64-bit calculation.

  FUNC  mm0,mm2               ;       Bits  {63...0}
  FUNC  mm1,mm3               ;       Bits  {127...64}

  movq  [edx],mm0             ; { Dy=f(Ay,By) Dx=f(Ax,Bx)}
  movd  (vmp3DVector PTR [edx]).z,mm1  ; {0  Dz=f(Az,Bz)}
;;;        femms
        ret
vmp_Vec FUNC Asm3DNow endp
```

vmp_FUNCTION (SSE-X86 Assembly) Aligned and Unaligned

As this is a three-float function, the data represented by the fourth float must be protected; therefore, the data is loaded as four floats, the calculation takes the place of the original {.w} value that is protected by bit masking and blended back into the data, and a four-float value is written back to memory. It needs to be kept in mind as to whether this fourth value is really a float or an integer value and protected accordingly. Remember, for unaligned memory, substitute *movups* for *movaps* as well as use ASSERT_PTR4 instead of ASSERT_PTR16.

Listing 3-25: V??3DX86M.asm

```
  align  16
  public  vmp_VecFUNCAsmSSE
vmp_VecFUNCAsmSSE  proc  near , vD:ptr, vA:ptr, vB:ptr
  .XMM
;;;        emms
ASSERT_PTR16(vD)
ASSERT_PTR16(vA)
ASSERT_PTR16(vB)

  mov   edx,vD                  ; Vector Destination
```

```
  mov   ecx,vB                    ; Vector B_xyz
  mov   eax,vA                    ; Vector A_xyz

  movaps xmm2,[edx]               ;vD.###w {D_w # # # }
  movaps xmm0,[ecx]               ;vB.xyz# {# B_z B_y B_x}
  movaps xmm1,[eax]               ;vA.xyz# {# A_z A_y A_x}
  andps  xmm2,OWORD PTR himsk32          ; {D_w 0 0 0 }
```

Insert instruction here for 128-bit calculation.

```
  FUNC  xmm0,mm1                  ;       Bits  {127...0}

  andps  xmm0,OWORD PTR lomsk96   ; {0 f_z f_y f_x}
  orps   xmm0,xmm2                ; {D_w f_z f_y f_x}
  movaps [edx],xmm0               ; {D_w f_z f_y f_x}
;;;       emms
          ret
vmp_VecFUNCAsmSSE endp
```

vmp_FUNCTION (AltiVec Assembly) (Un)aligned

There really is no difference for this three-float vector, whether it is aligned or not. The fourth float is unknown and cannot be trusted as a float. It also must be protected, so each field needs to be loaded into a safe-aligned quad vector. The fourth {.w} float is ignored and left as is. Only the first three floats of the calculated result are stored back to memory, thus protecting the fourth element.

Listing 3-26: V??3DAltivec.cpp

```
void vmp_VecFUNCAltivec(vmp3DVector * const pvD,
                  const vmp3DVector * const pvA,
                  const vmp3DVector * const pvB)
{
  vector float vD, vA, vB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  ((vmp3DQVector *) &vA)->x = pvA->x;
  ((vmp3DQVector *) &vA)->y = pvA->y;
  ((vmp3DQVector *) &vA)->z = pvA->z;

  ((vmp3DQVector *) &vB)->x = pvB->x;
  ((vmp3DQVector *) &vB)->y = pvB->y;
  ((vmp3DQVector *) &vB)->z = pvB->z;
```

Insert instruction here for 128-bit calculation.

```
  vD = vec_FUNC( vA, vB );

  pvD->x = ((vmp3DQVector *) &vD)->x;
  pvD->y = ((vmp3DQVector *) &vD)->y;
  pvD->z = ((vmp3DQVector *) &vD)->z;
}
```

vmp_FUNCTION (MIPS-VU0 Assembly) Aligned

This processor is the easiest of all discussed in this book for dealing with three-float vectors. By merely specifying Function.xyz, the three floats are modified and the fourth float {.w} is protected from change (kind of snazzy)!

Listing 3-27: V??3DVU.c

```
void vmp_VecAddVU(vmp3DVector * const pvD,
            const vmp3DVector * const pvA,
            const vmp3DVector * const pvB)
{
  ASSERT_PTR16(pvD);
  ASSERT_PTR16(pvA);
  ASSERT_PTR16(pvB);

asm __volatile__("
  lqc2     vf4,0x0(%1)
  lqc2     vf5,0x0(%2)
  lqc2     vf6,0x0(%0)     # .xyzw
```

Insert instruction here for 128-bit calculation.

```
  FUNC.xyz vf6,vf4,vf5     # D.w |= f(A.xyz, B.xyz)

  sqc2     vf6,0x0(%0)     # .xyzw
  ":
   : "r" (pvD) , "r" (pvA), "r" (pvB)
  );
}
```

This may be a little verbose, and there are other flavors, but those function shells should give you an idea of how to implement your own functions, as well as those included in the pages of this book.

Double-Precision Function Vector Wrappers

vmp_FUNCTION (SSE2-X86 Assembly) Aligned and Unaligned

As this is a three-float function, the data represented by the fourth float must be protected; therefore, the data is loaded as four floats and the calculation takes the place of the original {.w} value that is protected by bit masking and blended back into the data, and a four-float value is written back to memory. Keep in mind as to whether this fourth value is really a float or an integer value and protected accordingly. Remember to substitute *movupd* for *movapd* for unaligned memory as well as use the ASSERT_PTR8 instead of ASSERT_PTR16.

Listing 3-28: DV??3DX86M.asm

```
  align  16
  .XMM
       public  vmp_DVec FUNC AsmSSE2
vmp_DVec FUNC AsmSSE2  proc  near vD:ptr, vA:ptr, vB:ptr
;;;        emms
ASSERT_PTR16(vA)
ASSERT_PTR16(vB)
ASSERT_PTR16(vD)

  mov    edx,pvD
  mov    ecx,vB          ; Vector B
  mov    eax,vA          ; Vector A

  movapd xmm2,[ecx]      ;vB.xy {By Bx}
  movapd xmm0,[eax]      ;vA.xy {Ay Ax}
  movsd  xmm3,[ecx+16]   ;vB.z  {0 Bz}
  movsd  xmm1,[eax+16]   ;vA.z  {0 Az}
```

Insert instruction here for 128-bit calculation.

```
  FUNC  xmm0,xmm2        ;       Bits  {127...0}
  FUNC  xmm1,xmm3        ;       Bits  {255...128}

  movapd [edx],xmm0      ; { Dy=f(Ay,By) Dx=f(Ax,Bx) }
  movsd  [edx+16],xmm1   ; {              Dz=f(Az,Bz) }
  ret
vmp_DVec FUNC AsmSSE2  endp
```

Exercises

1. How much accuracy does the vmp_IsFEqual() function allow for with a single-precision definition? Fast precision?
2. AltiVec supports 1/4096 precision with estimated reciprocal and square root functions. What would the accuracy value passed to the third argument need to be to support an estimated result for the function vmp_IsFEqual()?
3. Does vmp_IsFEqual() accept a negative value for the third argument? Should there be an assertion? Why or why not?
4. In this chapter, vmp_IsVEqual() uses an ASSERT_PTR4(). What assertion would be used instead to force a 16-byte alignment?

5. Write C functions to support double-precision for:
   a. vmp_IsDEqual() scalar double-precision
   b. vmp_IsDVEqual() vector double-precision
   c. vmp_IsDQVEqual() quad vector double-precision
6. Given:

   0x12, 0x56, 0x89, 0x23, 0xEF, 0x89, 0x28, 0xC3
   0xE2, 0xFF, 0x04, 0x23, 0x49, 0x41, 0x74, 0x3F
   0x56, 0x89, 0xAA, 0xB2, 0xC7, 0x38, 0x28, 0x2A
   0x28, 0x28, 0x42, 0x73, 0x82, 0xDE, 0xF3, 0x28

   Show 128-bit data in proper endian order for 8-bit, 16-bit, 32-bit, and 64-bit block sizes for:
   a. SSE
   b. AltiVec
   c. MIPS-MMI
7. 128 bits = four single-precision floats. How wide would the next generation processor have to be for quad vector double-precision? Write a memory handler for this new hypothetical-type processor.
8. Future super matrix processors will be able to number-crunch entire matrices at a time. How wide would the data path be? Write an alignment macro for that width. Hint: There are two primary solutions!

Chapter 4
Vector Methodologies

This chapter discusses the methodology behind the use of vector mathematics by following a preliminary step by step as to whether one can, or even should, write an equation in a vector form.

- Target processor
- Type of data (integer/floating-point)?
- Signed or unsigned?
- What is parallel?
- Distinct or parallel?

This will be followed by some elementary methods, such as how to build vector equations.

CD Workbench Files: /Bench/*architecture*/chap04/*project*/*platform*

	architecture		*project*	*platform*
PowerPC	/vmp_ppc/	**A.O.S.**	**/aos/**	/mac9cw
X86	/vmp_x86/	**Tally**	**/tally/**	/vc6
MIPS	/vmp_mips/	**Core 3D**	**/vmp3d/**	/vc.net
				/devTool

Target Processor

The target processor is obviously the planned destination of your program. When writing video games, embedded applications, etc., it is primarily a single target, but quite often there will be multiple skews of a product. The personal computer is in a special class of its own, due to its multitude of flavors of X86 processors, memory footprint, etc. The differences and similarities between the various processors was discussed in Chapter 3, "Processor Differential Insight," and should be kept in mind. Some SIMD instructions that exist on one processor may not on another, thus an algorithm may be written using different methodologies based upon its required functionality and processor capabilities.

Type of Data

Mostly, it does not matter that a parallel method of programming is going to be utilized. The limits of your source data tend to be a known constant. The limits of the resulting data may be an unknown value, but the worst-case limits can be calculated and accounted for.

The integer values tend to be accurate (ignoring saturation and overflow) but have restrictive limits concerning whether they should be signed or unsigned.

Most of the following limits should already be known to you but have been included here for reference.

Table 4-1: Bits and signed/unsigned limits

Limits	8	16	32	64 Bits
Unsigned Hi	255	65535	4294967295	18446744073709551615
Unsigned Lo	0	0	0	0
Signed Hi	127	32767	2147483647	9223372036854775807
Signed Lo	–128	–32768	–2147483648	–9223372036854775808

The floating-point values tend to not have any restrictive limits, but there is a loss of precision. Switching from a single-precision to double-precision helps alleviate that problem and increase accuracy, but there is still a loss.

Whatever method is used, keep in mind that the more bits used to store the data and perform the calculation, the more memory resources and calculations on a batch of numbers will be required, which will extend pressing time.

AoS

Data organization in a discrete (non-SIMD) environment is that of an Array of Structures (AoS) and is typically organized as follows:

Listing 4-1: \chap04\AOS\aos.h

```
typedef struct AoS_Type
{
  float x;
  float y;
  float z;
  float w;
} AoS;

AoS adata[500];
```

As each adjacent structure within the array is sequentially accessed, the penalty caused by excessive caching of memory pages is minimized.

SoA

Data organization in a SIMD environment is that of a Structure of Arrays (SoA) and is typically organized as follows:

Listing 4-2

```
typedef struct SoA_Type
{
  float x[500];
  float y[500];
  float z[500];
  float w[500];
} SoA;

SoA sdata
```

SIMD instructions perform at their best when processing the same operation on like data. In this particular case, multiple elements of each array are handled simultaneously with elements of other arrays. The problem here, though, is that to load multiple items from the *x* array and then load multiple items from the adjacent *y* array, a memory access penalty will occur due to cache loading.

A cache is used for efficient processing of data. Data is transferred from system memory to high-performance cache memory for high-speed access.

Note: A typical Intel processor will have a limited number of pages, each typically 4 KB in size. An oversized "load" degrades the performance that would have been a beneficial increase. See your individual processor optimization manuals for more information.

Each SIMD type register is in effect an array of data, such that four 32-bit floats would be as follows:

```
float x[4];    // 32bit * 4 = 128bits
```

and a grouping of these registers could be considered an SoA type, such as the following:

```
struct {
  float X[4];
  float Y[4];
  float Z[4];
  float W[4]
} SoA;
```

A Possible Solution?

Different formulas will have different methodologies to organize data in a SIMD friendly orientation. One possible solution is to have data organized in a memory cache friendly organization, such as the AoS has to offer but transposing them into an SoA format upon demand before performing the actual calculation. This adds extra time to the total performance but will still have an overall acceptable response time.

Equation 4-1: AoS (Array of Structures) to SoA (Structure of Arrays) mapping in little endian

AoS			
W_0	Z_0	Y_0	X_0
W_1	Z_1	Y_1	X_1
W_2	Z_2	Y_2	X_2
W_3	Z_3	Y_3	X_3

⇨

SoA			
X_3	X_2	X_1	X_0
Y_3	Y_2	Y_1	Y_0
Z_3	Z_2	Z_1	Z_0
W_3	W_2	W_1	W_0

Another method would be the organization of memory into small blocks, sort of a hybrid between SoA and AoS. If memory cache blocks are 4KB in size, by using a structure of 4 x 8 x 4 = 128 bytes and an array size of 4096/128 = 32 entries, the best of both worlds can be used for SIMD operations. If multiple blocks are concatenated, even better performance is rewarded.

4KB Cache / 32 = 128-byte blocks

Listing 4-3: \chap04\AOS\AoS.h

```
#define SOA_ARY_MAX 8
#define SOA_BLK_MAX 32

typedef struct
{
  float x[SOA_ARY_MAX];
  float y[SOA_ARY_MAX];
  float z[SOA_ARY_MAX];
  float w[SOA_ARY_MAX];
} SoA8;

typedef struct SoABlk_Type
{
  SoA8 blk[SOA_BLK_MAX];
} SoABlk;
```

The problem is finding an algorithm that will allow the data to be organized in such a manner. With that in mind, the following code can be used to convert 4096 bytes from an Array of Structures to a Structure of Arrays block:

```
for (n=k=0; k<SOA_BLK_MAX; k++)  //32
{
  for (j=0; j<SOA_ARY_MAX; j++)  //8
  {
    pSoA->blk[k].x[j] = pAoS[n].x;
    pSoA->blk[k].y[j] = pAoS[n].y;
    pSoA->blk[k].z[j] = pAoS[n].z;
    pSoA->blk[k].w[j] = pAoS[n++].w;
  }
}
```

It does the job, but it is extremely slow due to all the array lookups. An alternative would be to use pointers and simple pointer arithmetic with fixed offsets. With this design, an array of 1...*n* structures can be converted. Obviously, if it were to be written in assembly code, it would be beneficial to take advantage of register pipelining to optimize any throughput. A destination working memory buffer would need to be specified and would need a byte count of nSize, which is rounded up to the nearest 4096-byte block size. The following C code demonstrates this technique.

```
#define SOA_MAX   (SOA_ARY_MAX*SOA_BLK_MAX) //256

nSize=((nCnt+(SOA_MAX-1)) & ~(SOA_MAX-1))<<4;
```

Convert AoS to 4K Block SoA Quickly

Listing 4-4: \chap04\AOS\Bench.cpp

```
void vmpXlatAoS_4K(
    SoABlk *pDAry,    // Destination Block
    const AoS *pSAry, // Source Array of Struct
    uint nCnt)        // Number of structures
{
  uint n;
  float *pD, *pS;

  ASSERT_PTR4(pSAry);
  ASSERT_PTR4(pDAry);
  ASSERT_ZERO(nCnt);
  pS = (float*)pSAry;
  pD = (float*)pDAry;

  if (8 <= nCnt)       // 8 or more
  {
    n = nCnt >> 3;
    nCnt &= 0x07;
```

```
    do {
      *(pD+0) =*(pS+0);    *(pD+8) =*(pS+1);
      *(pD+16)=*(pS+2);    *(pD+24)=*(pS+3);
      *(pD+1) =*(pS+4);    *(pD+9) =*(pS+5);
      *(pD+17)=*(pS+6);    *(pD+25)=*(pS+7);
      *(pD+2) =*(pS+8);    *(pD+10)=*(pS+9);
      *(pD+18)=*(pS+10);   *(pD+26)=*(pS+11);
      *(pD+3) =*(pS+12);   *(pD+11)=*(pS+13);
      *(pD+19)=*(pS+14);   *(pD+27)=*(pS+15);
      *(pD+4) =*(pS+16);   *(pD+12)=*(pS+17);
      *(pD+20)=*(pS+18);   *(pD+28)=*(pS+19);
      *(pD+5) =*(pS+20);   *(pD+13)=*(pS+21);
      *(pD+21)=*(pS+22);   *(pD+29)=*(pS+23);
      *(pD+6) =*(pS+24);   *(pD+14)=*(pS+25);
      *(pD+22)=*(pS+26);   *(pD+30)=*(pS+27);
      *(pD+7) =*(pS+28);   *(pD+15)=*(pS+29);
      *(pD+23)=*(pS+30);   *(pD+31)=*(pS+31);
      pS += 32;
      pD += 32;
    } while(--n);
  }

  if (nCnt)           // Move any remaining source elements.
  {
    do {
      *(pD+0) =*(pS+0);    *(pD+8) =*(pS+1);
      *(pD+16)=*(pS+2);    *(pD+24)=*(pS+3);
      pS += 4;
      pD++;
    } while(--nCnt);
  }
}
```

With the data rearranged in the blocked SoA format, the full power of the SIMD instruction sets can be taken advantage of. Once completed, the data would need to be converted to something usable. A smart programmer using a software render would allow it to import data in this format. If a hardware render or library requiring the AoS format is being used, then once all the transformations have completed, the data would need to be converted back into that AoS format. The following pointer-based C code can be used. It is very similar to the previous code sample but with the source and destination data types reversed.

Just in case you were wondering why I use an if (nCnt) do{} while (--nCnt) instead of a while(nCnt--), it is typically more efficient. This book isn't about code optimization, although it contains multiple helpful recommendations, such as this one. If you either do not believe me or are curious, peek at the generated assembly code sometime!

Convert 4K SoA to AoS Quickly

Listing 4-5: \chap04\AOS\Bench.cpp

```
void vmpXlatSoA_4K(
    AoS *pDAry,          // Dst. Array of Struct.
    const SoABlk *pSAry, // Source Block
    uint nCnt )          // Number of structures
{
  uint n;
  float *pD, *pS;

  ASSERT_PTR4(pSAry);
  ASSERT_PTR4(pDAry);
  ASSERT_ZERO(nCnt);

  pS = (float*)pSAry;
  pD = (float*)pDAry;

  if (nCnt >= 8)          // 8 or more?
  {
    n = nCnt >> 3;
    nCnt &= 0x07;

    do {
      *(pD+0) =*(pS+0);    *(pD+1) =*(pS+8);
      *(pD+2) =*(pS+16);   *(pD+3) =*(pS+24);
      *(pD+4) =*(pS+1);    *(pD+5) =*(pS+9);
      *(pD+6) =*(pS+17);   *(pD+7) =*(pS+25);
      *(pD+8) =*(pS+2);    *(pD+9) =*(pS+10);
      *(pD+10)=*(pS+18);   *(pD+11)=*(pS+26);
      *(pD+12)=*(pS+3);    *(pD+13)=*(pS+11);
      *(pD+14)=*(pS+19);   *(pD+15)=*(pS+27);
      *(pD+16)=*(pS+4);    *(pD+17)=*(pS+12);
      *(pD+18)=*(pS+20);   *(pD+19)=*(pS+28);
      *(pD+20)=*(pS+5);    *(pD+21)=*(pS+13);
      *(pD+22)=*(pS+21);   *(pD+23)=*(pS+29);
      *(pD+24)=*(pS+6);    *(pD+25)=*(pS+14);
      *(pD+26)=*(pS+22);   *(pD+27)=*(pS+30);
      *(pD+28)=*(pS+7);    *(pD+29)=*(pS+15);
      *(pD+30)=*(pS+23);   *(pD+31)=*(pS+31);
      pS += 32;
      pD += 32;
    } while(--n);
  }

  if (nCnt)               // Have any remainders?
  {
    do {
      *(pD+0)=*(pS+0);    *(pD+1)=*(pS+8);
      *(pD+2)=*(pS+16);   *(pD+3)=*(pS+24);
      pS++;
      pD += 4;
    } while(--nCnt);
  }
}
```

One last detail: If there is a need to translate from an AoS to a SoA, keep in mind that a write to memory takes longer than a read from it. Your algorithm should read the data into a usable form and perform an initial calculation before writing it to memory in the SoA arrangement. Perform the calculations, but upon completion of the last calculation, write it back to memory in the destined format, be it AoS or SoA.

There are more efficient methods of data conversion depending on processor type, which will be discussed in the next chapter, "Vector Data Conversion."

Packed and Parallel and Pickled

Okay, not "pickled," but it got your attention, right? The SIMD instruction set is as its name indicates; that data is arranged in a series of sets of parallel bits. Examine the following forms of parallel data: characters, shorts, integers, longs, floats, and doubles. Keep in mind the use or absence of the data type term "half-word," which represents the 16-bit arrangement. This is used throughout the book. The data types handled are the same ones that you should already be familiar with on an individual basis. The only difference is that there are multiples of them.

In the following drawing, note that each SIMD contains two or more bit blocks contained within each set of 64/128 SIMD bits. The mathematical operations that are performed are done in parallel. For example, in 128 bits are 16 packed blocks, each containing 8 bits each, and so there are 16 separate calculations performed simultaneously. This same parallel logic occurs for all SIMD instructions, whether they are integer or floating-point based. The following numbers are going to be evaluated using an adder with and without a SIMD method of summation. For purposes of simplicity, four numbers of 16 bits each will be processed in a visual hex form and the factor of big/little endian being ignored.

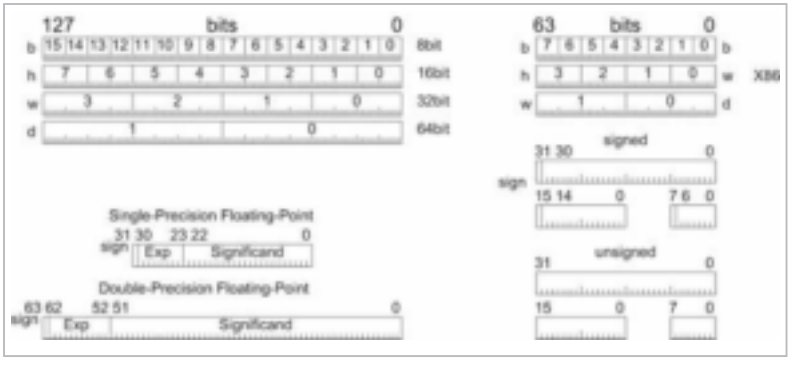


Figure 4-1: Miscellaneous (un)packed integer and floating-point data sizes represented in little endian

The following unsigned decimal numbers are summed using four separate calculations:

32766	12886	254101	21922
+63540	+59189	+29012	+10563
96306	72075	54422	32485

Add again, but in a 16-bit hex form, and see them summed individually again:

7FFE	3256	6342	55A2
+F834	+E735	+7154	+2943
1 7832	1 198B	D496	7EE5

But our computer has 32-, 64-, or 128-bit capability. Why waste time waiting for four individual calculations when there are bits to spare each time? Suppose we do not care about the carry.

Please note the carry that occurred in the first and second calculations, as their values exceeded the maximum value of 65535 containable within a 16-bit value. The following summation on the right is done with a single 64-bit addition, so instead of four calculations, there is only one. On the left, a SIMD operation handles each 16-bit element individually as in the addition, but the unwanted carry is thrown away.

63...48	47...32	31...16	15...0	63...0
7FFE	3256	6342	55A2	7FFE3256634255A2
+F834	+E735	+7154	+2943	+F834E73571542943
7832	198B	D496	7EE5	7833198BD4967EE5

Aha! Note the difference. A hex 7832 result in the upper 16 bits versus that of a 7833 in that first summation! The carry from the second equation affected the first equation by being summed as part of the calculation. The point is that the equations were isolated from each other and so did not affect each other. The bonus is that the CPU time was only spent for one calculation and not four, since they were done simultaneously! If there was a concern about the overflow, such as the lost carry, some processors have the ability to saturate; that is not to exceed the maximum allowable value. This would be similar to MIN(val, 0xFFFF) for an unsigned 16-bit value. If the value exceeded 65535, it would stick upon 65535, thus that same SIMD calculation with saturation would be:

63...48	47...32	31...16	15...0
7FFE	3256	6342	55A2
+F834	+E735	+7154	+2943
FFFF	FFFF	D496	7EE5

In a subtraction, a simple borrow, such as in the following example on the right, would borrow from each adjacent bit, whether it be in the same data block or an adjacent one. However, with a SIMD operation where each block of 16 bits is handled individually, as in the subtraction below on the left, each block is isolated. So adjacent blocks do not affect each other and processing time is saved handling the calculations in parallel, instead of four consecutively.

63...48	47...32	31...16	15...0
7FFE	3256	6342	55A2
–F834	+E735	+7154	+2943
87CA	4B21	F1EE	2C5F

63...0
7FFE3256634255A2
–F834E73571542943
87C94B20F1EE2C5F

If this same method was used on the 128-bit form where there are 16 8-bit values handled simultaneously versus waiting for 16 calculations to be performed, the savings in CPU time becomes obvious. This is like pickled sardines packed in parallel in a tin, do you not think? (Okay, I digress!) At this point, you should have a better idea of SIMD parallelism. More on this will be discussed throughout the mathematical operations chapters of this book.

Discrete or Parallel?

Before anyone even thinks of the possibility of vectorizing their code, they must first write the function using a normal high-level language compiler, such as C. (Note that I did not say C++, C#, or Java, as those languages are too high level! But there is no reason why a high-level library could not wrap assembly-based vector code!) This will allow the algorithm to be refined and locked down as to its functionality to minimize any lost time due to changes or disposal of an equation. This will create a quick changeable code base foundation. The vectorizing of the code should only be thought of as a post-process method of accelerating code using the same reasoning as one would for writing a function in assembly code to replace that of a high-level language.

Hint: Vectorize code to make it go faster!

Okay, to most of you, that hint is a no-brainer, but, nevertheless, it is important to remember.

Hint: Vectorize only frozen code!

There is no need to spend programming time writing a custom low-level vector function for a parallel method of operation if the code is not locked in a frozen state. If that statement is ignored, time may be wasted having to alter an existing parallel algorithm instead of modifying a discrete one. Think about it! Which of the following would be more preferable to your project timelines?

Table 4-2: Problem-algorithm solution development

Waiting to Vectorize	Do It Now
Analyze problem.	Analyze problem.
Write distinct algorithm.	Write distinct algorithm.
Test algorithms.	Test algorithms.
Design change: adjust algorithms.	Lay out in parallel.
Test algorithms.	Write using parallel methods.
Analyze algorithm.	Test parallel algorithms.
Layout in parallel.	Design change: adjust algorithms. (**If change significant, go to beginning of this column and start again.**)
Write using parallel methods.	Test parallel algorithms.
Test parallel algorithms.	

The above table is visually misleading. Making equation changes to a function is much simpler in the distinct model than it is in a parallel one, thus waiting until after any possible changes need to be done before writing the code using a parallel method is much more advisable. You should, of course, keep in mind how you would go about writing the algorithm in parallel, even while still writing the version.

Some code can be made efficient by using parallel methods, and some cannot! An example of code not recommended for parallelism would be that which has dependencies upon the results of a previous scalar equation. This does not work well in parallel unless multiple datasets need to be evaluated.

```
A = 1;
B = 2;
C = B + A;
D = C + A;
E = D + C;
F = B + D;
```

A Secure Hash Algorithm (SHA-1) is such an algorithm. In parallel, one cannot solve an equation unless each of the source operands has a valid value. The following would definitely result in a wrong solution:

A +B C	A +C D	C +D E	D +B F

How could the operand C be used in the second block as a source operand if the first block has not resolved its value? The same goes for C and D in the third block, as well as D in the fourth block.

With some rearrangement for parallelism, something similar to the following is obtained. Note the shaded areas of interest.

A +B C	D +0 D	E +0 E	F +0 F
C +0 C	A +C D	E +0 E	F +0 F
C +0 C	D +0 D	C +D E	D +B F

Unfortunately, out of 12 operations, only four were effective. The remaining eight were wasted, but there was a savings of one operator cycle since only three cycles needed to be done instead of four. The algebraic law of identity was used to keep the placeholder operands intact.

Algebraic Law:

Additive Identity	$n + 0 = 0 + n = n$
Multiplicative Identity	$n1 = 1n = n$

Since this is a summation problem by adding a value of zero, the operand remains intact. If this equation resolved a product, a value of one would be used instead.

Even though the end result is better than using a distinct method, it is not really worth the effort. Now, if operands A through F were actually separate arrays of numbers instead of individual discrete values:

```
A = {1,3,5,7};
B = {2,4,6,8};
C = B + A;
D = C + A;
E = D + C;
F = B + D;
```

...then:

A[0]	A[1]	A[2]	A[3]	1	3	5	7
+B[0]	+B[1]	+B[2]	+B[3]	+2	+4	+6	+8
C[0]	C[1]	C[2]	C[3]	3	7	11	15
A[0]	A[1]	A[2]	A[3]	1	3	5	7
+C[0]	+C[1]	+C[2]	+C[3]	+3	+7	+11	+15
D[0]	D[1]	D[2]	D[3]	4	10	16	22
C[0]	C[1]	C[2]	C[3]	3	7	11	15
+D[0]	+D[1]	+D[2]	+D[3]	+4	+10	+16	+22
E[0]	E[1]	E[2]	E[3]	7	17	27	37
D[0]	D[1]	D[2]	D[3]	4	10	16	22
+B[0]	+B[1]	+B[2]	+B[3]	+2	+4	+6	+8
F[0]	F[1]	F[2]	F[3]	6	14	22	30

...each of the operands would be resolved in parallel, thus no cross dependencies. This increases the operator cycles from three to four, but out of 16 operations, all 16 are effective; this is in the same amount of time that only four discrete operators could be handled. There are no cycles wasted in this particular case. The dependencies are not required until the next operation. In this particular case, the array is only four elements in length, but if there were more, one would merely loop while using standard optimization methodologies.

Algorithmic Breakdown

We have not discussed any of the particular SIMD instructions for any particular processor since the "big picture" was painted using some broad strokes. Let's start with a simple problem.

Array Summation

A terrain construction tool imports polygon objects from a 3D mesh file for export to a Dedicated Relational Database, but it first needs to do some analysis on the database to calculate memory requirements, such as the number of objects in the database, the minimum and maximum polygon face counts of each individual object, the maximum number of

faces used by all objects in the database, the minimum and maximum sizes as well as pixel resolution for each of the coordinates of a vector point, etc.

Okay, okay, I know. Those of you who are more experienced game programmers are now screaming, "You've got to be kidding! I'd never do it that way!!!" I admit that neither would I. However, I needed a simple problem for this exercise. There are different methods to solve this phantom problem, but pretend that the terrain tool does lots of face splitting, culling, etc. and so it is not easy to keep accurate face counts. A list (array) of face counts for each object is generated and a tally of all the faces is needed in the landscape. The formula would be something similar to the following equation:

```
FaceTally = Ary[0] + ... + Ary[nArtCnt-1];
```

The first concern is to pick a data size to store the face counts that are known to not be overflowed. A buffer bit size of 32 bits will contain each object's face count nicely. To simplify this problem, an assumption is that the object count is a factor of four with a minimum of eight, so there could be {8, 12, 16, 20, 24, etc.} objects. Obviously, a real-life problem would have no such limitation.

```
uint Ary[] = {
  826, 998, 968, 276,  353, 127, 927, 324,
  467, 754, 345, 942,  667,  36, 299, 894,
  581, 373, 579, 325 };
uint nAryCnt = sizeof(Ary) / sizeof(*Ary); //=20
```

Ary Tally

With the list Ary[] data in place, a discrete test is needed. The following can be used as a control function, and its results will be used in conjunction with a vector algorithm for testing purposes to verify correctness and accuracy.

Listing 4-6: \chap04\Tally\Tally.cpp

```
uint AryTally( const uint *pAry, uint nCnt )
{
  uint nTally = 0;

  ASSERT_PTR4(pAry);

  if (nCnt)
  {
    do {
      nTally += *pAry++;
    } while(--nCnt);
  }
```

```
  return nTally;
}
```

Ary Tally (Quad)

This pseudo sample code is needed to help visualize this algorithm running on a 128-bit processor, so the first step to vectorize the code is to have four calculations occur simultaneously. The number of loops is adjusted to process four 32-bit calculations at a time. The following sample does just that! Four "buckets" are used to tally the data separately and then sum the remaining number of vectors.

Listing 4-7: \chap04\Tally\Tally.cpp

```
uint AryTallyQ( const uint *pAry, uint nCnt )
{
  uint nVecCnt, T[4];

  ASSERT_PTR4(pAry);

  nVecCnt = nCnt >> 2;          // # of 4-word vectors

  ASSERT(2<=nVecCnt);           // only allow {8,12,16,...}

  T[0] = *(pAry+0)+ *(pAry+4); //1st+2nd 128 bits
  T[1] = *(pAry+1)+ *(pAry+5);
  T[2] = *(pAry+2)+ *(pAry+6);
  T[3] = *(pAry+3)+ *(pAry+7);
  pAry += 8;
  nVecCnt -= 2;

  if (nVecCnt)
  {
    do {
      T[0] += *(pAry+0);        // next 128 bits
      T[1] += *(pAry+1);
      T[2] += *(pAry+2);
      T[3] += *(pAry+3);
      pAry += 4;
    } while( --nVecCnt );
  }

  return T[0] + T[1] + T[2] + T[3];
}
```

The following is a step-by-step trace that results from the previous function with the sample data Ary[]:

```
T[0]=826+353; T[1]=998+127; T[2]=968+927; T[3]=276+324;
T[0]+=467;    T[1]+=754;    T[2]+=345;    T[3]+=942;
T[0]+=667;    T[1]+=36;     T[2]+=299;    T[3]+=894;
T[0]+=581;    T[1]+=373;    T[2]+=579;    T[3]+=325;

nTally = T[0] + T[1] + T[2] + T[3];  //=11061
```

Vector — 32-bit Unsigned Word

The following sample works with 128-bit vectors subdivided into 32-bit unsigned words, and so the following data structure will be cast.

```
typedef struct VectorUW
{
  uint x;
  uint y;
  uint z;
  uint w;
} VectorUW;
```

Next, a vector library function is needed, such as:

```
void VecAddUW( VectorUW *dst, VectorUW *srcA,
                              VectorUW *srcB );
```

...so as to add the second 128-bit vector argument *srcA* to the third 128-bit vector argument *srcB* and store the result in the first 128-bit vector argument *dst*.

Add Unsigned 4x32-bit Word Vector

The following code sample simulates the summation of four parallel 32-bit unsigned words within a 128-bit vector.

Listing 4-8: \chap04\Tally\Tally.cpp

```
void VecAddUW( VectorUW * const vDst,
         const VectorUW * const vSrcA,
         const VectorUW * const vSrcB )
{
  ASSERT_PTR4(vDst);
  ASSERT_PTR4(vSrcA);
  ASSERT_PTR4(vSrcB);

  vDst->x = vSrcA->x + vSrcB->x;
  vDst->y = vSrcA->y + vSrcB->y;
  vDst->z = vSrcA->z + vSrcB->z;
  vDst->w = vSrcA->w + vSrcB->w;
}
```

Ary Tally (Vector)

The following is the vector-based summing function:

Listing 4-9: \chap04\Tally\Tally.cpp

```
uint AryTallyV( const uint *pAry, uint nCnt )
{
  uint nVecCnt;
  VectorUW vT, *vA;

  ASSERT_PTR4(pAry);
```

```
  vA = (VectorUW *)pAry;
  nVecCnt=nCnt >> 2;           // # of 4-word vectors

  ASSERT(2 <= nVecCnt);        // only allow {8,12,16,...}

  VecAddUW( &vT, vA, vA+1 );   // 1st+2nd 128 bits
  vA += 2;
  nVecCnt -= 2;

  if (nVecCnt)
  {
    do {                       // next 128 bits
      VecAddUW( &vT, &vT, vA++ );
    } while( --nVecCnt );
  }

  return vT.x + vT.y + vT.z + vT.w;
}
```

At this point, the functions such as VecAddUW() would be optimized to be as quick as possible, such as using SIMD instructions. That is where this book comes in. So, to recap:

- Know which processors are targeted.
- Pick your data type and number of bits required.
- Write a discrete code version.
- Modify discrete code into multiple like calculations per pass.
- Modify "like" version into vector functions with SIMD.

Thinking Out of the Box (Hexagon)

Think out of the box!

This section discusses thinking out of the box. Unfortunately, this is an overused expression these days, but thinking out of the hexagon just does not cut it! Anyway, in thinking in this manner, you should build a reputation among your peers for finding new and innovative methods for solving programming problems. Got it? Good!

Not all algorithms can be easily converted into a vector-based algorithm. Programmers used to a general-purpose programming mentality actually program in the box. What needs to be done is use creative methods to handle your algorithms, and thus leave the box.

Vertical Interpolation with Rounding

Typically, I use the following code as a simple example of a method to handle vector processing with a non-vector processor. This is a great hobby of mine that I fondly refer to as "pseudo vector processing." It is the process of inventing new, faster optimized code using new methods of implementation. The following code is the inner loop of an algorithm that vertically averages a 16x16 byte array, where adjacent rows (scan lines) would be averaged together using an equation similar to (A+B+1)/2. Examine the averaging of 16 pairs of 8-bit numbers by summing with 16 other 8-bit numbers, plus a value of one, and then dividing each pair by two to obtain the resulting average. The values are interlaced to minimize any CPU register dependencies.

```
byte *pD, *pA, *pB;

*(pD+0)  = ((*(pA+0)  + *(pB+0)  +1) >> 1);
*(pD+1)  = ((*(pA+1)  + *(pB+1)  +1) >> 1);
           :             :
           :             :
*(pD+14) = ((*(pA+14) + *(pB+14) +1) >> 1);
*(pD+15) = ((*(pA+15) + *(pB+15) +1) >> 1);
```

If two 8-bit values are summed, a carry will result, thus the creation of a ninth bit. In this implementation, a larger number of bits is needed to handle the calculation and the division by two resulting logically from the right shift. For the averaging of two rows of 16 bytes, 16 additions followed by 16 divisions would be required. In the following example, each byte would need to be unpacked into a larger data container, the addition and division processed, and the result repacked — a time-consuming operation.

```
byte *pD, *pA, *pB;
uint A, B;

A=(uint)*(pA+0);
B=(uint)*(pB+0);
*(pD+0)=(byte)((A+B+1)>>1);

A=(uint)*(pA+1);
B=(uint)*(pB+1);
*(pD+1)=(byte)((A+B+1)>>1);
          :             :
          :             :
A=(uint)*(pA+14);
B=(uint)*(pB+14);
*(pD+14)=(byte)((A+B+1)>>1);

A=(uint)*(pA+15);
B=(uint)*(pB+15);
*(pD+15)=(byte)((A+B+1)>>1);
```

An alternative to that would be to first reduce the values, sum the results, and add an error correction value back onto the final result by using the algebraic law of distribution:

Algebraic Law:

Distributive	a(b+c) = ab+ac	(b+c)/a = b/a + c/a

With this in mind, let's note the equation so that the 8 bits of both *A* and *B* get the LSB shifted into oblivion.

```
(*(pA+0) >> 1) thus (AAAAAAAA >> 1) = 0AAAAAAA
(*(pB+0) >> 1) thus (BBBBBBBB >> 1) = 0BBBBBBB
```

This means the end result has a loss of resolution. To recap, without a "correction," there is data loss due to the initial logical right shift, so look into this from a binary point of view. Since two numbers are being summed, only the last bit needs to be looked at:

```
  A31, A30,..., A1, A0
+ B31, B30,..., B1, B0
```

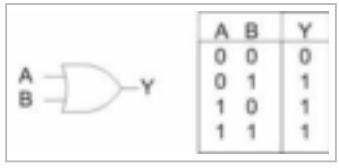
```
  A31, A30,..., A1, A0
  B31, B30,..., B1, B0
(+                  1) >> 1
```

On the left in the following example, all the combinations of the result of the summation of the least significant bit of *A* and *B* with the resulting carry [Carry | Bit#0] are shown. On the right is the averaging (with rounding) result of the least significant bits of two numbers. Note there is no carry!

A + B	**0**	**1**
0	00	01
1	01	10

(A+B+1) /2	**0**	**1**
0	00	01
1	01	01

The LSB operation on the right for the averaging operation above uses the same logic as the following logical "|" OR function. Again, note that there is no carry from the LSB due to the divide by two.



A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 4-2: Logical OR

Thus:

$$\frac{(A+B+1)}{2} = (A+B+1)>>1 = ((A>>1)+(B>>1)) + ((A\&1) \mid (B\&1))$$

Do you notice the same result between the two previous examples? By shifting the LSB out for both *A* and *B*, summing the result, then using a logical OR on the LSB of the original *A* and *B* values and adding that error correction to the previous summation, the same result can be obtained as the original equation. This appears to be more complicated, but note that there is no carry between 8-bit sections! So now comes the payoff. Four 8-bit values can be handled simultaneously by shifting 32 bits in the form of four 8-bit values to the right similar to the following:

```
aaaaaaaabbbbbbbbccccccccdddddddd >> 1
```

Shift the original value right by one! Note "M" and "L" denote most and least significant bits in big or little endian and 8 bit is endianless.

```
M       LM       LM       LM       L
0aaaaaaaabbbbbbbbccccccccddddddddd
         ^        ^        ^        ^
```

Note that each LSB has been shifted into the MSB of the adjacent byte, thus "^" contaminating that byte. This is rectified by masking each byte with a hex 7F (01111111); the MSB eighth bit of each is left empty so that the results of each LSB when shifted right will not contaminate the MSB of the adjacent byte.

```
M       LM       LM       LM       L
01111111011111110111111101111111       7F mask
0aaaaaaa0bbbbbbb0ccccccc0ddddddd
```

When this result is summed for *A* and *B*, any resulting carry will occupy the bit position containing the "0." Even if both bytes contained 7F+7F, the result would be a FE, leaving room for one bit — the result of the LSB of the original ((A&1) | (B&1)). The following code demonstrates this latter solution.

```
#define eMASK01 0x01010101       // Mask 01
#define eMASK7F 0x7f7f7f7f       // Mask 7f

unsigned int32 *pD, *pA, *pB;

*(pD+0) = ((*(pA+0) >> 1)      & eMASK7F)
        + ((*(pB+0) >> 1)      & eMASK7F)
        + ((*(pA+0) | *(pB+0)) & eMASK01);
*(pD+1) = ((*(pA+1) >> 1)      & eMASK7F)
        + ((*(pB+1) >> 1)      & eMASK7F)
        + ((*(pA+1) | *(pB+1)) & eMASK01);
*(pD+2) = ((*(pA+2) >> 1)      & eMASK7F)
        + ((*(pB+2) >> 1)      & eMASK7F)
        + ((*(pA+2) | *(pB+2)) & eMASK01);
*(pD+3) = ((*(pA+3) >> 1)      & eMASK7F)
        + ((*(pB+3) >> 1)      & eMASK7F)
        + ((*(pA+3) | *(pB+3)) & eMASK01);
```

By inserting this code into a simple loop and advancing *pD*, *pA*, *pS* with their appropriate strides accordingly, the original 16x16 function can be recreated. Simply note that by using an alternative form of the algebraic law of distribution and handling any loss that may occur, a new, faster implementation of the algorithm is found. One last observation is that the (+1) for rounding was implemented by merely using a logical OR of the original LSB of each byte. Again, that's just thinking out of the box!

Exercises

1. What is AoS? Give an example organization to store four quad single-precision floats within a 2048-byte block.
2. What is SoA? Give an example organization to store four quad single-precision floats within a 2048-byte block.
3. A = 5, B = 7, C = 9

   D = A + B, E = A + C, F = B + D, G = C + 8, H = G + B

   Diagram the above as tightly packed as possible for quad vector processing.

Chapter 5
Vector Data Conversion

(Un)aligned Memory Access

As discussed earlier, it is very important to have proper data alignment on all data. There are, however, times when this is not possible. In those cases the data would still need to be accessed quickly, so before we get into the heavy-duty instructions that can be handled, let us look at a method of handling the pre- and post-preparation of the data for them.

Pseudo Vec (X86)

Misaligned MMX and 3DNow! (64-bit)

For 64-bit access, there is a memory stall for a misaligned access, but no consequential problems. The instruction *movq* is a 64-bit move (Quad Word) and merely copies 64 bits to MMX and XMM registers. But in our particular case of vectors, it is used as follows:

Move Quad Word

MMX	movd *mmDst, mmSrc(mm/m64)*	[Un]signed	64
	movq *mmDst*, *mmSrc(mm/m64)*		
3DNow!	movq *mmDst*, *mmSrc(mm/m64)*	Single-Precision	64

```
movq  mm0,mem64      // Read INT64 from memory
movq  mem64,mm0      // Write INT64 to memory

movq  mm0,mem64      // Read SPFP from memory
movq  mem64,mm0      // Write SPFP to memory
```

Keep in mind that the 3DNow! floating-point uses the same MMX registers as the integer instructions and thus the same *movq* instruction.

Misaligned SSE(2) (128-bit)

For SSE and SSE2, things get a bit more complicated. The XMM registers are used by the SSE and SSE2 instructions and primarily for single-precision vector floating-point for SSE and both double precision and 128-bit integer for the SSE2 instructions. They are not interchangeable. In addition, there are different memory movement instructions depending on if the memory is aligned or not. There are other memory load/save instructions besides these, but these are the ones of interest to us in this book.

Move Unaligned Double Quad Word

SSE2	movdqu x*mmDst*, x*mmSrc(xmm/m128)* movdqu (x*mmDst/m128)*, x*mmSrc*	[Un]signed	128

Move Aligned Double Quad Word

SSE2	movdqa x*mmDst*, x*mmSrc(xmm/m128)* movdqa (x*mmDst/m128)*, x*mmSrc*	[Un]signed	128

Move Scalar (4x Single-Precision FP)

SSE	movss x*mmDst*, x*mmSrc(xmm/m128)* movss (x*mmDst/m128)*, x*mmSrc*	Single Precision	128

Move Low Packed (2x Single-Precision FP)

SSE	movlps x*mmDst*, *m64* movlps *m64*, x*mmSrc*	Single Precision	128

Move High Packed (2x Single-Precision FP)

SSE	movhps x*mmDst*, *m64* movhps *m64*, x*mmSrc*	Single Precision	128

Move Low to High Packed (2x Single-Precision FP)

SSE	movlhps x*mmDst*, *xmmSrc*	Single Precision	128

Move High to Low Packed (2x Single-Precision FP)

SSE	movhlps x*mmDst*, *xmmSrc*	Single Precision	128

Move Unaligned Packed (4x Single-Precision FP)

SSE	movups x*mmDst*, x*mmSrc(xmm/m128)* movups (x*mmDst/m128)*, x*mmSrc*	Single Precision	128

Move Aligned Packed (4x Single-Precision FP)

SSE movaps x*mmDst*, x*mmSrc(xmm/m128)* Single Precision 128
movaps (x*mmDst/m128)*, x*mmSrc*

Move Unaligned Packed (2x Double-Precision FP)

SSE2 movupd x*mmDst*, x*mmSrc(xmm/m128)* Double Precision 128
movupd (x*mmDst/m128)*, x*mmSrc*

Move Scalar (1x Double-Precision FP)

SSE2 movsd x*mmDst*, x*mmSrc(xmm/m128)* Double Precision 128
movsd (x*mmDst/m128)*, x*mmSrc*

Move Low Packed (1x Double-Precision FP)

SSE2 movlpd x*mmDst*, *m64* Double Precision 128
movlpd *m64*, x*mmSrc*

Move High Packed (1x Double-Precision FP)

SSE2 movhpd x*mmDst*, *m64* Double Precision 128
movhpd *m64*, x*mmSrc*

Move Aligned Packed (2x Double-Precision FP)

SSE2 movapd x*mmDst*, x*mmSrc(xmm/m128)* Double Precision 128

SSE 128-bit:

```
movups  xmm0,mem128   // Read SPFP from unaligned memory
movups  mem128,xmm0   // Write SPFP to unaligned memory

movaps  xmm0,mem128   // Read SPFP from aligned memory
movaps  mem128,xmm0   // Write SPFP to aligned memory
```

SSE (inclusive SSE) 128-bit:

```
movdqu  xmm0,mem128   // Read INT from unaligned memory
movdqu  mem128,xmm0   // Write INT to unaligned memory

movdqa  xmm0,mem128   // Read INT from aligned memory
movdqa  mem128,xmm0   // Write INT to aligned memory

movupd  xmm0,mem128   // Read DPFP from unaligned memory
movupd  mem128,xmm0   // Write DPFP to unaligned memory

movapd  xmm0,mem128   // Read DPFP from aligned memory
movapd  mem128,xmm0   // Write DPFP to aligned memory
```

Pseudo Vec (PowerPC)

The PowerPC (G3) allows aligned 4-byte access, and so all the generic code on the companion CD will work on that platform. The AltiVec (G4), however, requires all memory to be aligned properly on 16-byte boundaries, and thus it requires something a little different, which is discussed following the PowerPC (G3) code. In some cases, data needs a specific alignment, and so this is done manually.

First, the amount of shift needs to be determined and the address corrected for proper 16-byte alignment:

```
andi. r0,r4,3
cmpwi r0,0
beq   goodalign            // Branch if four byte aligned
```

Now handle bad alignment:

```
li    dy,-4                // =FFFFFFFC  {1,2,3} byte offset
slwi  lshift,r0,3          // x8  1=8, 2=16, 3=24 bits to shift
and   r4,r4,dy             // Set pbSrc to mod 4 = 0
```

Then the masks and amount of right shift need to be determined:

```
li    rshift,32
li    rmask,-1             //  =FFFFFFFF
sub   rshift,rshift,lshift // 24=32-8  16=32-16  8=32-24
srw   lmask,rmask,rshift   // 8=000X  16=00XX  24=0XXX
xor   rmask,rmask,lmask    // 8=XXX0  16=XX00  24=X000
```

Now load 64 bits of data (128 bits would be just more registers). The three bracketed columns on the right represent the {1,2,3} byte misaligned orientation of the data. [Digits] are in big endian and represent byte #'s, underscores are zero, and x's are "I do not care!"

```
                           //  1     2     3
lwz   c0,0(r4)             // [X012] [XX01] [XXX0] = [3...0]
lwz   c1,4(r4)             // [3456] [2345] [1234] = [7...4]
lwz   d1,8(r4)             // [789A] [6789] [5678] = [11...8]

                       // Note columns 1,2,3 are in big endian!

rlwnm c1,c1,lshift,0,31    // [4563] [4523] [4123]
rlwnm d1,d1,lshift,0,31    // [89A7] [8967] [8567]

slw   c0,c0,lshift         // [012_] [01__] [0___]
and   r0,c1,lmask          // [0003] [0023] [0123]

and   c1,c1,rmask          // [4560] [4500] [4000]
and   d0,d1,lmask          // [0007] [0067] [0567]

or    c0,c0,r0             // [0123] [0123] [0123]
or    c1,c1,d0             // [4567] [4567] [4567]
```

Now that c0 and c1 contain the aligned data, the algorithm using it can continue.

Pseudo Vec (AltiVec)

Loading a vector for AltiVec is not very difficult. It is merely calculating the number of bits needed to correct the alignment, and that is done by merely utilizing a logical AND of the address. If already aligned, a zero will be returned. Subtract the address with the number of bytes to shift to get the actual aligned address that AltiVec would load. (Remember, it ignores the lower four address lines during a memory load!)

Pseudo Vec (MIPS-MMI)

Accessing data from a MIPS processor is not very difficult either. However, more instructions are needed to correct any misalignment.

Load Quad Word

MMI lq *Dst*, #(*aSrc*) [Un]signed 128

```
lq    t0,0(a0)        // Read aligned 128 bit from memory
```

For aligned memory, the *lq* (Load Quad Word) instruction uses an offset to the address stored in the register. This is 0(a0), 16(a0), 32(a0), etc. This means that the load is based upon the memory address contained within the register a0 and the summation of the offset {0, 16, 32, etc.}.

Keep in mind that a float is four bytes in size, and a0 contains the base address of the array.

```
float array[] = {0, 1, 2, 3,  4, 5, 6, 7};

lq t0, 0(a0)    ; loads {0, 1, 2, 3}
lq t0, 16(a0)   ; loads {4, 5, 6, 7}
```

Save Quad Word

MMI sq *Dst*, #(*aSrc*) [Un]signed 128

```
sq    t0,0(a0)        // Write aligned 128 bit to memory
```

Load Double Word Left

MIPS ldl *Dst*, #(*aSrc*) [Un]signed 64

Load Double Word Right

MIPS ldr *Dst*, #(*aSrc*) [Un]signed 64

For misaligned memory, the general-purpose unaligned 64-bit instruction pair *ldl* (Load Double Word Left) and *ldr* (Load Double Word Right) are used. By loading two 64-bit data blocks and blending them with the 128-bit multimedia instruction *pcpyld* (Parallel Copy Lower Double Word), a register will contain the aligned 128 bits.

Read upper bytes from byte 7 downward and lower bytes from byte 0 upward.

```
ldl   t0,7(a1)        // Read (upper bits of) {63...0}
ldr   t0,0(a1)        // Read (lower bits of)...

ldl   t1,15(a1)       // Read (upper bits of) {127...64}
ldr   t1,8(a1)        // Read (lower bits of)...

pcpyld t0,t1,t0       // Blend Bits {127...0}  [t1:t0]
```

When dealing with non-Boolean data (as all bits are naturally isolated from each other), using the non-packed 64 bits is sometimes sufficient, thus there is no need to pack the two 64-bit blocks into a single 128-bit block.

Save Double Word Left

MIPS sdl *Dst*, #(*aSrc*) [Un]signed 64

Save Double Word Right

MIPS sdr *Dst*, #(*aSrc*) [Un]signed 64

Once the expression has been evaluated, use the *sq* instruction to save to aligned memory as previously listed, but if misaligned, something a little more complicated is needed. First the data needs to be unpacked with the multimedia instruction *pcpyud* (Parallel Copy Upper Double Word) into an upper and lower 64 bits of data. The general 64-bit instructions *sdl* (Store Double Word Left) and *sdr* (Store Double Word Right) are used to save the misaligned data to memory.

```
pcpyud t1,t0,t0       // D  {63...0  127...64} [t1:t0]

sdl   t0,7(a0)        // Write (upper bits of) {63...0}
sdr   t0,0(a0)        // Write (lower bits of)...

sdl   t1,15(a0)       // Write (upper bits of) {127...64}
sdr   t1,8(a0)        // Write (lower bits of)...
```

Again, I must emphasize the necessity of properly aligning memory. Imagine one instruction to load or save 128-bit data instead of the five instructions, as just shown.

Pseudo Vec (MIPS-VU0)

Load Packed Single-Precision

VPU lqc2 *Dst*, #(*aSrc*) Single-Precision 128

```
lqc2    t0,0(a1)        // Read aligned 128 bit from memory
```

Save Packed Single-Precision

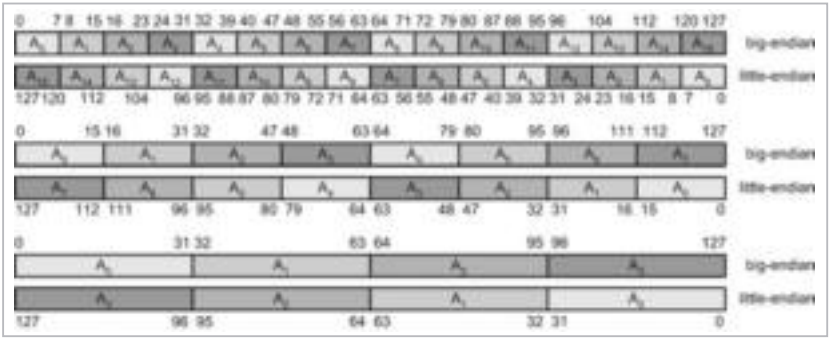
VPU sqc2 *Dst*, #(*aSrc*) Single-Precision 128

```
sqc2    t0,0(a0)        // Write aligned 128 bit from memory
```

Data Interlacing, Exchanging, Unpacking, and Merging

As we just touched on briefly, data must sometimes be interlaced to get it into a form that can be handled easily. By understanding how to interlace and de-interlace data, a most productive solution can be found for solving an expression.

The instructions in this chapter are easier to understand through visualization, and each processor has its set of instructions that it handles, but here is where the feature of endian orientation can easily be confusing: converting data from the output of one instruction and using it as the input of another.



Here is a guide to assist in remembering big versus little endian orientations using the following example byte sequence in memory:
Address 0000: 0x12,0x34,0x56,0x78,0x9a,0xbc,0xde,0xf0,0xff,
0xee,0xdd,0xcc,0xbb,0xaa,0x99,0x88

Following is the placement of those same bytes within the 128-bit data diagrams used in this chapter. Keep in mind that each 128-bit block is a repeat of the previous block.

- Little endian

  0x88, 0x99, 0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0xf0, 0xde, 0xbc, 0x9a, 0x78, 0x56, 0x34, 0x12 (8-bit)

  0x8899, 0xaabb, 0xccdd, 0xeeff, 0xf0de, 0xbc9a, 0x7856, 0x3412 (16-bit)

  0x78563412, 0xf0debc9a, 0xccddeeff, 0x8899aabb (32-bit)

- Big endian

  0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0, 0xff, 0xee, 0xdd, 0xcc, 0xbb, 0xaa, 0x99, 0x88 (8-bit)

  0x1234, 0x5678, 0x9abc, 0xdef0, 0xffee, 0xddcc, 0xbbaa, 0x9988 (16-bit)

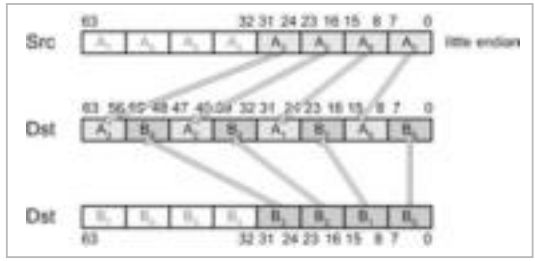
  0x12345678, 0x9abcdef0, 0xffeeddcc, 0xbbaa9988 (32-bit)

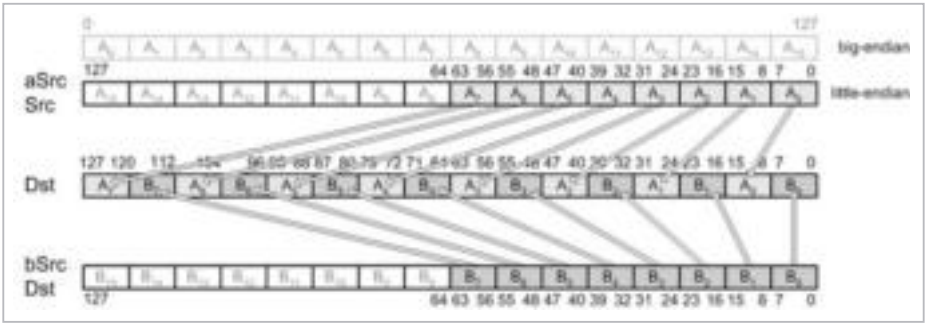
One thing to remember here is that the data elements are isolated from each other. The A_n placement of each element is related to its position. For example, when related to a quad vector, $A_0 : A_x$, $A_1 : A_y$, $A_2 : A_z$, and $A_3 : A_w$. So that means that A_w A_z A_y A_x are visually on the far right, just like A_3 A_2 A_1 A_0 for little endian, and A_x, A_y, A_z, A_w are on the far left, just like A_0, A_1, A_2, A_3 for big endian.

So as long as you get the element positions correct for your processor, the data flow represented by the arrows will be correct.

Note: The bit indicators on the diagrams in this section are in little endian byte order for functionality used by both little and big endian platforms, but when unique, the appropriate endian labeling is used. In those cases where both are supported, a "ghost" guide for big endian is provided.

Parallel Extend Lower from Byte (16x8-bit)





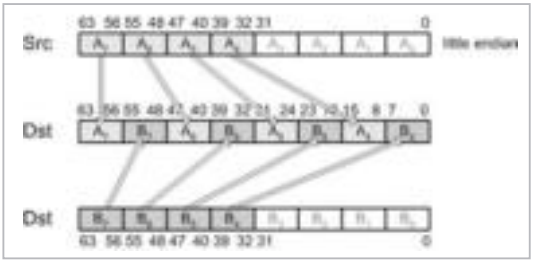
AltiVec	vmrglb *Dst, aSrc, bSrc* *vD = vec_mergel(vA, vB)*	[Un]signed 128
MMX	punpcklbw *mmDst,(mmSrc/m64)*	[Un]signed 64
SSE2	punpcklbw *xmmDst*, *(xmmSrc/m128)*	[Un]signed 128
MIPS/MMI	pextlb *Dst, aSrc, bSrc*	[Un]signed 128

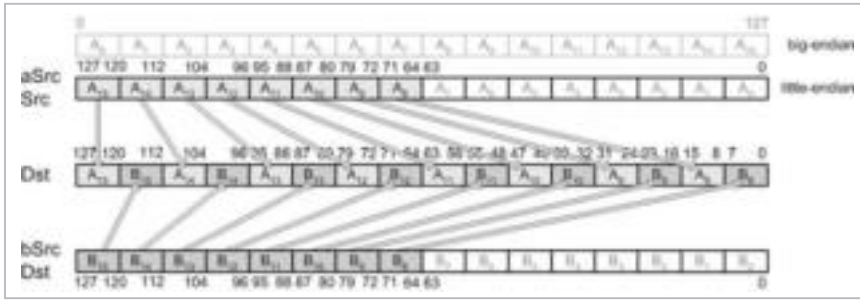
```
vmrglb    vr0,vr1,vr2
punpcklbw mm0,mm1
punpcklbw xmm0,xmm1
pextlb    t0,t1,t2
```

This is one of the more popular instructions, as it is extremely useful in the expansion of an unsigned data value. By interlacing a value of zero with an actual value, an 8-bit value is expanded to 16 bits.

```
A = 0x00000000     B = 0x44332211

  D = 00  44  00  33  00  22  00  11
        0044    0033    0022    0011
```

Parallel Extend Upper from Byte (16x8-bit)

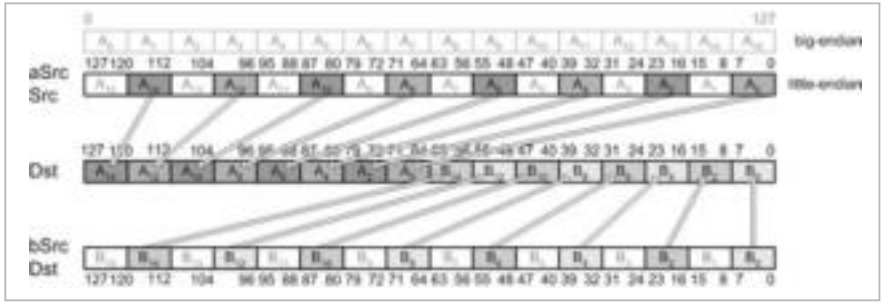




AltiVec	vmrghb *Dst, aSrc, bSrc* *vD = vec_mergeh(vA, vB)*	[Un]signed	128
MMX	punpckhbw *mmDst,(mmSrc/m64)*	[Un]signed	64
SSE2	punpckhbw *xmmDst,(xmmSrc/m128)*	[Un]signed	128
MIPS/MMI	pextub *Dst, aSrc, bSrc*[Un]signed		128

```
vmrghb    vr0,vr1,vr2
punpckhbw mm0,mm1
punpckhbw xmm0,xmm1
pextub    t0,t1,t2
```

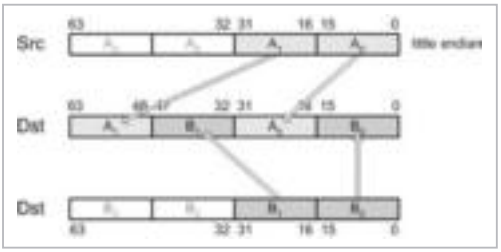
Parallel Pack to Byte (16x8-bit)

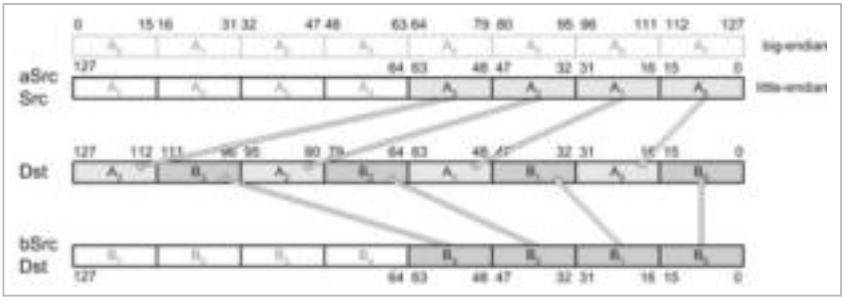


AltiVec	vpkuhum *Dst, aSrc, bSrc* *vbD = vec_pack(vhA, vhB)*	[Un]signed	128
MIPS/MMI	ppacb *Dst, aSrc, bSrc*	[Un]signed	128

```
vpkuhum vr0,vr1,vr2
ppacb   t0,t1,t2
```

Parallel Extend Lower from Half-word (8x16-bit)

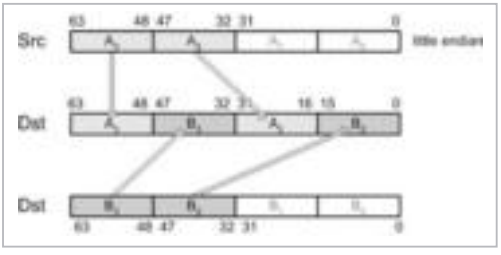


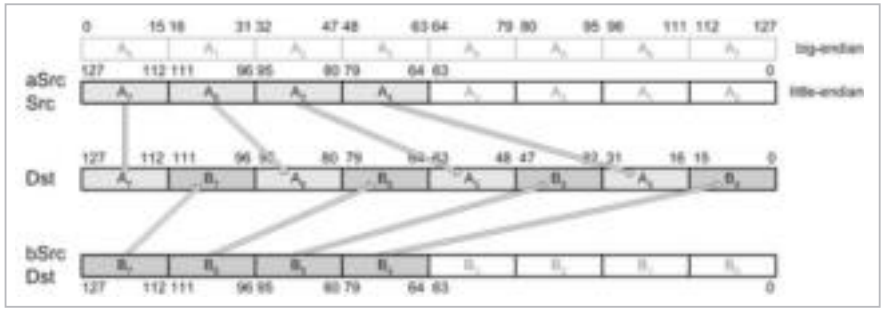


AltiVec	vmrglh *Dst, aSrc, bSrc* *vD = vec_mergel(vA, vB)*	[Un]signed	128
MMX	punpcklwd *mmDst,(mmSrc/64)*	[Un]signed	64
SSE2	punpcklwd *xmmDst*, *(xmmSrc/m128)*	[Un]signed	128
MIPS/MMI	pextlh *Dst, aSrc, bSrc*	[Un]signed	128

```
vmrglh    vr0,vr1,vr2
punpcklwd mm0,mm1
punpcklwd xmm0,xmm1
pextlh    t0,t1,t2
```

Parallel Extend Upper from Half-word (8x16-bit)

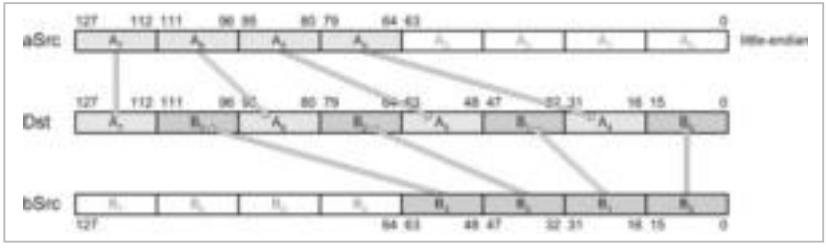




AltiVec	vmrghh *Dst, aSrc, bSrc* *vD = vec_mergeh(vA, vB)*	[Un]signed	128
MMX	punpckhwd *mmDst,(mmSrc/m64)*	[Un]signed	64
SSE2	punpckhwd *xmmDst,(xmmSrc/m128)*	[Un]signed	128
MIPS/MMI	pextuh *Dst, aSrc, bSrc*[Un]signed		128

```
vmrghh    vr0,vr1,vr2
punpckhwd mm0,mm1
punpckhwd xmm0,xmm1
pextuh    t0,t1,t2
```

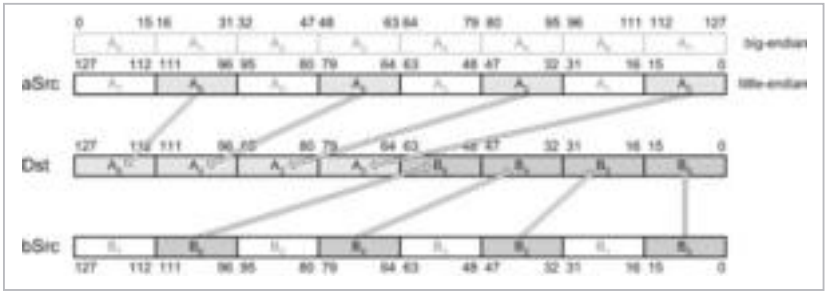
Parallel Interleave Half-word (8x16-bit)



MIPS/MMI	pinth *Dst, aSrc, bSrc*	[Un]signed	128

```
pinth t0,t1,t2
```

Parallel Pack to Half-word (8x16-bit)

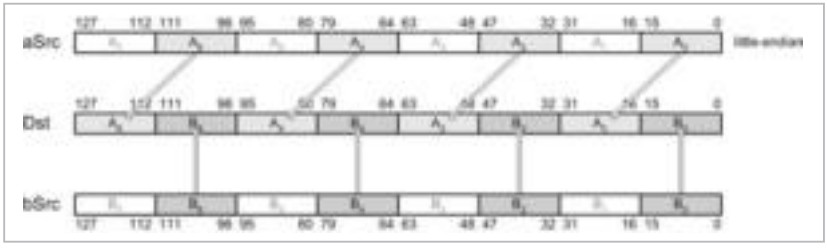


AltiVec	vpkuwum *Dst, aSrc, bSrc* *vhD = vec_pack(vwA, vwB)*	[Un]signed	128
MIPS/MMI	ppach *Dst, aSrc, bSrc*	[Un]signed	128

```
vpkuwum vr0,vr1,vr2
ppach   t0,t1,t2
```

Note that the data is halved, thus reduced in size.

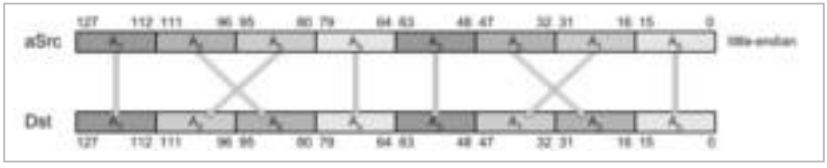
Parallel Interleave Even Half-word (8x16-bit)



MIPS/MMI	pinteh *Dst, aSrc, bSrc*	[Un]signed	128

```
pinteh t0,t1,t2
```

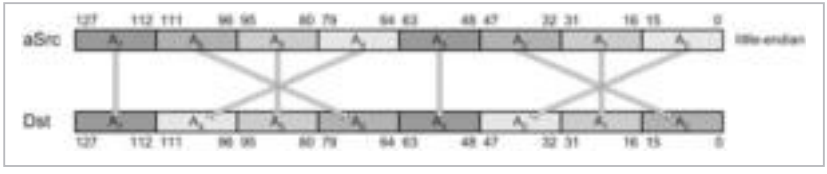
Parallel Exchange Center Half-word (8x16-bit)



MIPS/MMI	pexch *Dst, aSrc*	[Un]signed	128

```
pexch t0,t1
```

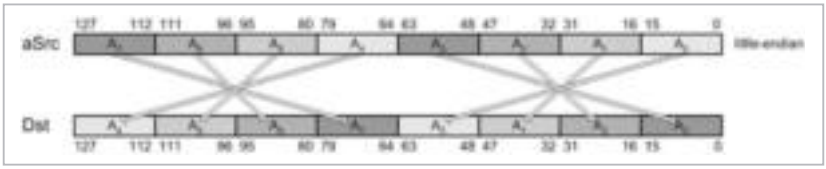
Parallel Exchange Even Half-word (8x16-bit)



MIPS/MMI	pexeh *Dst, aSrc*	[Un]signed	128

```
pexeh t0,t1
```

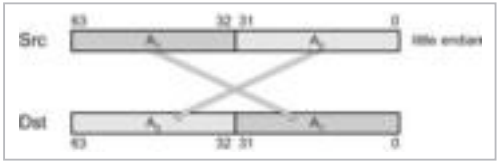
Parallel Reverse Half-word (8x16-bit)



MIPS/MMI prevh *Dst, aSrc* [Un]signed 128

```
prevh t0,t1
```

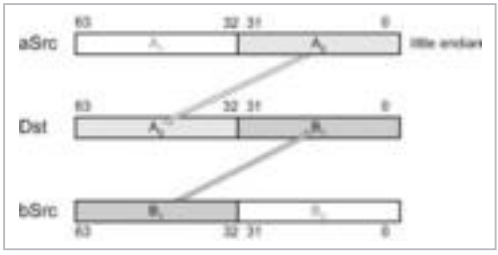
Packed Swap Double Word (2x32-bit)



3DNow!+ pswapd *mmDst,(mmSrc/m64)* [Un]signed 64

```
pswapd mm0,mm1
```

Packed Lower/Upper



MIPS V plu.ps *Dst, aSrc, bSrc* Single-Precision FP 64

```
plu.ps $f4,$f5,$f6
```

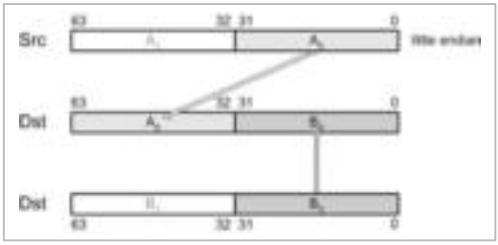
Packed Upper/Lower

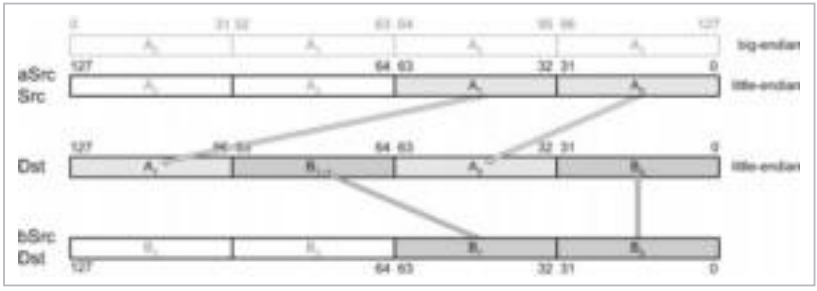
MIPS V	pul.ps *Dst, aSrc, bSrc*	Single-Precision FP	64

```
pul.ps $f4,$f5,$f6
```

Parallel Extend Lower from Word (4x32-bit)

Also Unpack and Interleave Low-Packed Single-Precision Floating-Point Values



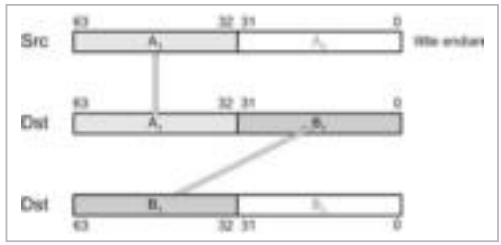


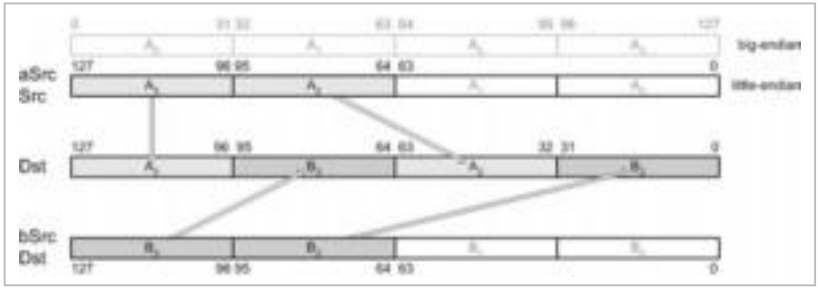
AltiVec	vmrglw *Dst, aSrc, bSrc* *vD = vec_mergel(vA, vB)*	[Un]signed	128
MMX	punpckldq *mmDst,(mmSrc/m64)*	[Un]signed	64
SSE	unpcklps *xmmDst,(xmmSrc/m128)*	Single-Precision FP	128
SSE2	punpckldq *xmmDst,(xmmSrc/m128)*	[Un]signed	128
MIPS/MMI	pextlw *Dst, aSrc, bSrc*	[Un]signed	128
MIPS V	pll.ps *Dst, aSrc, bSrc*	Single-Precision FP	64

```
vmrglw    vr0,vr1,vr2
punpckldq mm0,mm1
unpcklps  xmm0,xmm1
punpckldq xmm0,xmm1
pextlw    t0,t1,t2
pll.ps    $f4,$f5,$f6
```

Parallel Extend Upper from Word (4x32-bit)

Also Unpack and Interleave High-Packed Single-Precision Floating-Point Values

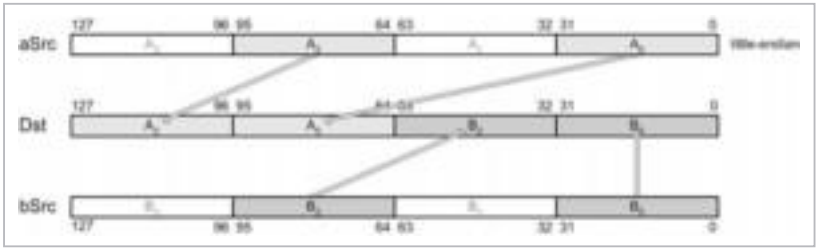




AltiVec	vmrghw *Dst, aSrc, bSrc* *vD = vec_mergeh(vA, vB)*	[Un]signed	128
MMX	punpckhdq *mmDst,(mmSrc/m64)*	[Un]signed	64
SSE	unpckhps *xmmDst,(xmmSrc/m128)*	Single-Precision FP	128
SSE2	punpckhdq *xmmDst,(xmmSrc/m128)*	[Un]signed	128
MIPS/MMI	pextuw *Dst, aSrc, bSrc*	[Un]signed	128
MIPS V	puu.ps *Dst, aSrc, bSrc*	Single-Precision FP	64

```
vmrghw    vr0,vr1,vr2
punpckhdq mm0,mm1
unpckhps  xmm0,xmm1
punpckhdq xmm0,xmm1
pextuw    t0,t1,t2
puu.ps    $f4,$f5,$f6
```

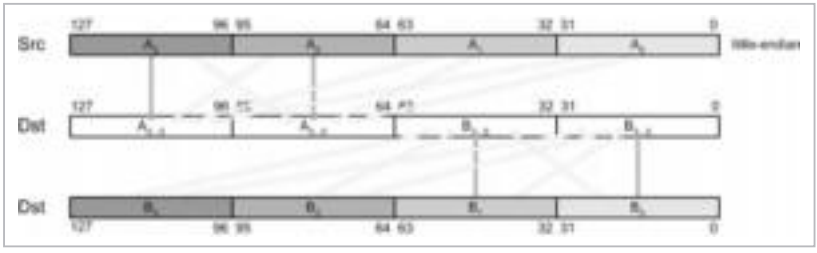
Parallel Pack to Word (4x32-bit)



MIPS/MMI ppacw *Dst, aSrc, bSrc* [Un]signed 128

```
ppacw t0,t1,t2
```

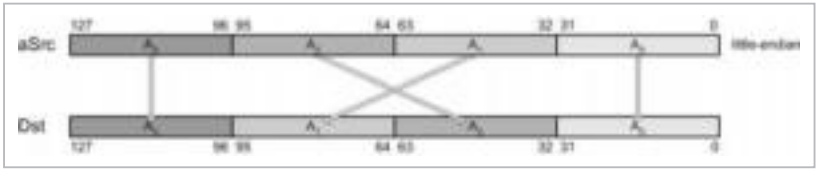
Shuffle Packed Single-Precision Floating-Point Values (4x32-bit)



SSE shufps *xmmDst, (xmmSrc/m128)*,# Single-Precision FP 128

```
shufps xmm0,xmm1,11100100b  ; 3 2 1 0     {3...0}
```

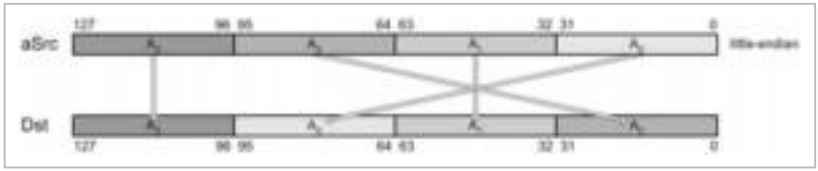
Parallel Exchange Center Word (4x32-bit)



MIPS/MMI pexcw *Dst, aSrc* [Un]signed 128

```
pexcw t0,t1
```

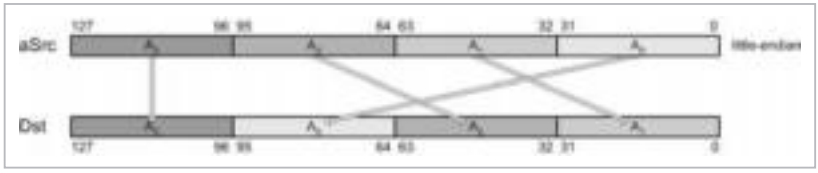
Parallel Exchange Even Word (4x32-bit)



MIPS/MMI pexew *Dst, aSrc* [Un]signed 128

```
pexew t0,t1
```

Parallel Rotate Three Words Left (4x32-bit)

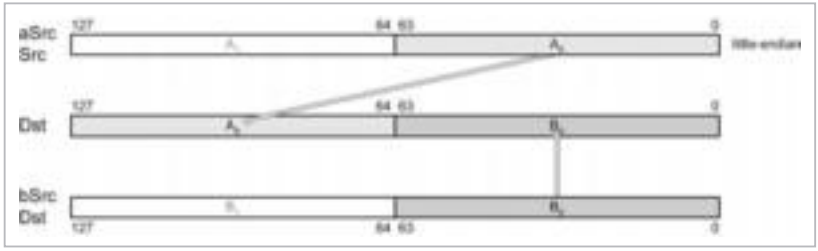


MIPS/MMI prot3w *Dst, aSrc* [Un]signed 128

```
prot3w t0,t1
```

Parallel Copy Lower Double Word (2x64 bit)

Also Unpack and Interleave Low-Packed Double-Precision Floating-Point Values



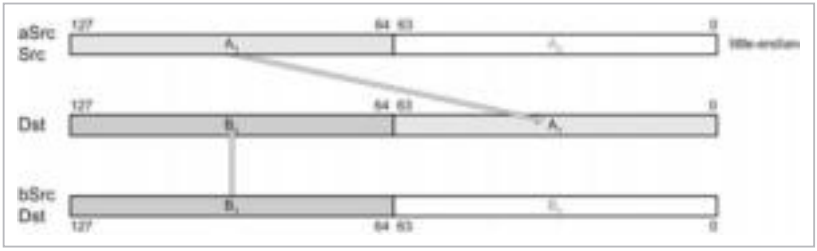
SSE2 punpcklqdq *xmmDst,(xmmSrc/m128)*
[Un]signed 128
unpcklpd *xmmDst,(xmmSrc/m128)*
Double-Precision FP 128

MIPS/MMI pcpyld *Dst, aSrc, bSrc* [Un]signed 128

```
punpcklqdq xmm0,xmm1
unpcklpd   xmm0,xmm1
pcpyld     t0,t1,t0    // {127...0}  [127...64]  [63...0]
```

Parallel Copy Upper Double Word (2x64 bit)

Also Unpack and Interleave High-Packed Double-Precision Floating-Point Values



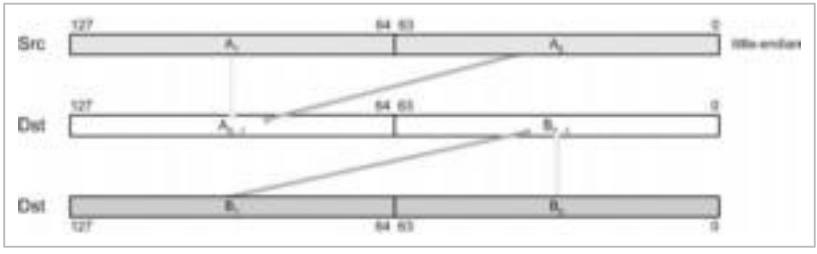
SSE2 punpckhqdq *xmmDst,(xmmSrc/m128)*
[Un]signed 128

unpckhpd *xmmDst,(xmmSrc/m128)*
Double-Precision FP 128

MIPS/MMI pcpyud *Dst, aSrc, bSrc* [Un]signed 128

```
punpckhqdq xmm0,xmm1
unpckhpd   xmm0,xmm1
pcpyud     t0,t1,t0     // {127...0}  [127...64]  [63...0]
```

Shuffle-Packed Double-Precision Floating-Point Values (2x64 bit)



SSE2 shufpd *xmmDst, (xmmSrc/m128),#*
Double-Precision FP 128

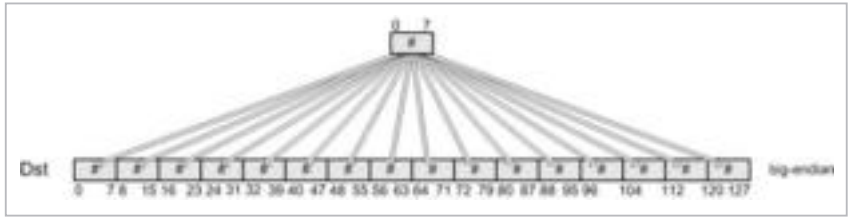
```
shufpd xmm0,xmm1,01b ; 0 1   {1...0}
```

Swizzle, Shuffle, and Splat

Various manufacturers refer to the swapping of data values by different terms: swizzle, shuffle, splat, etc. Some of these replicate a data value into two or more destination locations. In a few hybrid cases, the functions use a defined distribution or a custom-defined interlacing of source arguments, such as what was discussed in the previous section.

The splat functionally is similar to a bug hitting the windshield of an automobile at 70 miles per hour and being distributed around.

Vector Splat Immediate Signed Byte (16x8-bit)



AltiVec vspltisb *Dst,#* Signed 128

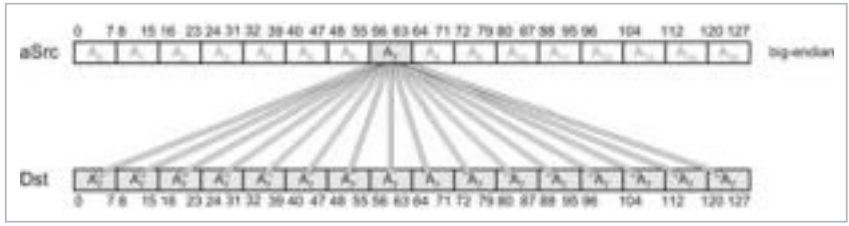
vD = vec_splat_s8(#)

vD = vec_splat_u8(#)

```
vspltisb vr0,#
```

The immediate value is a signed 5-bit value, thus having a range of –16 to 15. It gets sign extended into the upper 3 bits. The easiest way to think of this is that the upper fifth bit {D4} of the immediate value gets replicated into the upper three bits {D7...D5}, thus reflecting the sign of the value. So the –16 to 15 is distributed as hex {0xf0 to 0x0f} or the equivalent binary 111110000b to 00001111b.

Vector Splat Byte (16x8-bit)

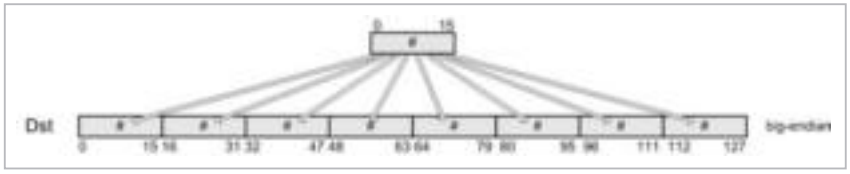


AltiVec vspltb *Dst, aSrc,#* Unsigned 128
vD = vec_splat(vA, vB, #)

```
vspltb vr0,vr1,7
```

The immediate value of {0...15} is used to select the field from aSrc and distribute (broadcast) to all of the fields in the destination Dst.

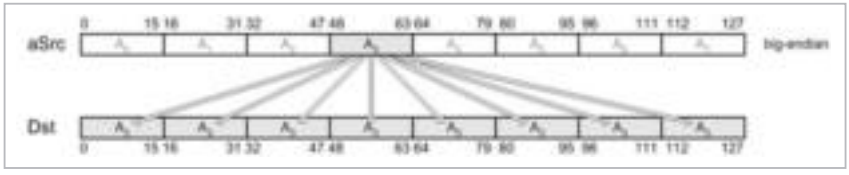
Vector Splat Immediate Signed Half-Word (8x16-bit)



AltiVec vspltish *Dst*, # Signed 128
vD = vec_splat_s16(#)
vD = vec_splat_u16(#)

```
vspltish vr0,#
```

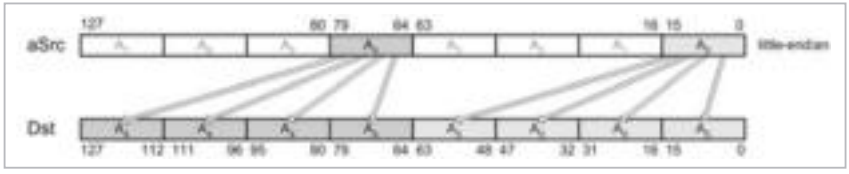
Vector Splat Half-Word (8x16-bit)



AltiVec vsplth *Dst, aSrc,#* Unsigned 128
vD = vec_splat(vA, #)

```
vsplth vr0,vr1,3
```

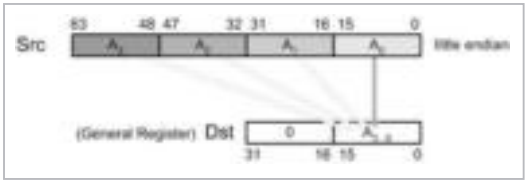
Parallel Copy Half-Word (8x16-bit)



MIPS/MMI pcpyh *Dst, aSrc* [Un]signed 128

```
pcpyh t0,t1
```

Extract Word into Integer Register (4x16-bit) to (1x16)

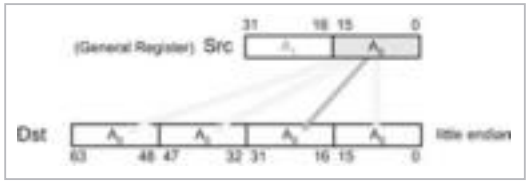


SSE pextrw *Dst32, (mmSrc/m64),#* [Un]signed 64

```
pextrw eax,mm1,00b ; {3...0}
```

One of the four 16-bit values is assigned to the lower 16 bits of the general-purpose 32-bit register and zero extended into the upper 16 bits.

Insert Word from Integer Register (1x16) to (4x16-bit)

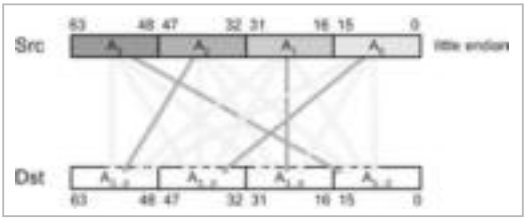


MMX+ pinsrw *(mmSrc/m64),r32,#* [Un]signed 64

```
pinsrw mm0,eax,01b ; 1   {3...0}
```

The lower 16 bits of the general-purpose register are assigned to one of the four destination 16-bit values selected by the index.

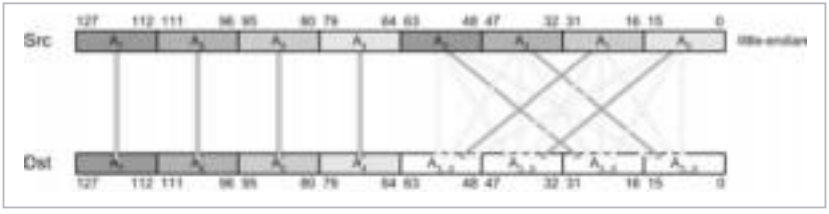
Shuffle-Packed Words (4x16-bit)



MMX+ pshufw *mmDst, (mmSrc/m64),#* [Un]signed 64
SSE pshufw *mmDst, (mmSrc/m64),#* [Un]signed 64

```
pshufw mm0,mm1,10000111b ; 2 0 1 3
```

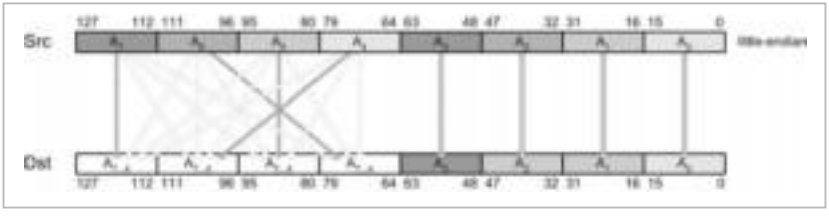
Shuffle-Packed Low Words (4x16-bit)



SSE2 pshuflw *xmmDst, (xmmSrc/m128),#* [Un]signed 128

```
pshuflw xmm0,xmm1,01001110b ; 1 0 3 2
```

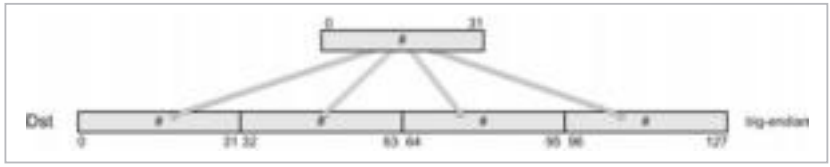
Shuffle-Packed High Words (4x16-bit)



SSE2 pshufhw *xmmDst, (xmmSrc/m128),#* [Un]signed 128

```
pshufhw xmm0,xmm1,11000110b ; 3 0 1 2
```

Vector Splat Immediate Signed Word (8x16-bit)

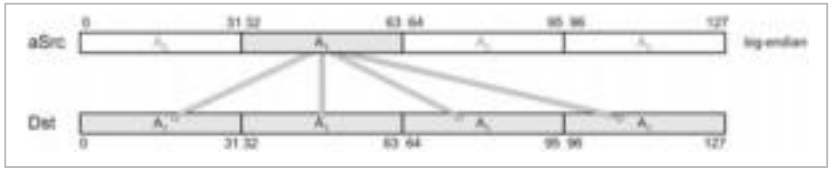


AltiVec vspltisw *Dst, #* Signed 128

vD = vec_splat_s32(#)
vD = vec_splat_u32(#)

```
vspltisw vr0,#
```

Vector Splat Word (8x16-bit)

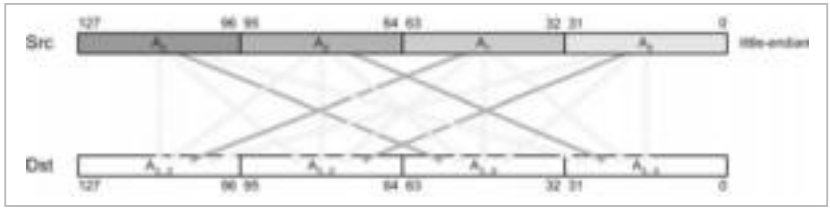


AltiVec vspltw *Dst, aSrc, #* Unsigned 128

vD = vec_splat(vA, #)

```
vspltw vr0,vr1,#
```

Shuffle-Packed Double Words (4x32-bit)

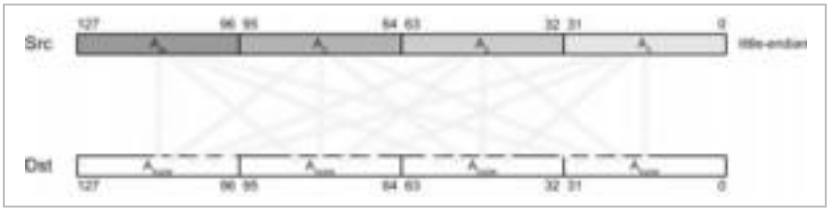


SSE2 pshufd *xmmDst, xmmSrc/m128,#* [Un]signed 128

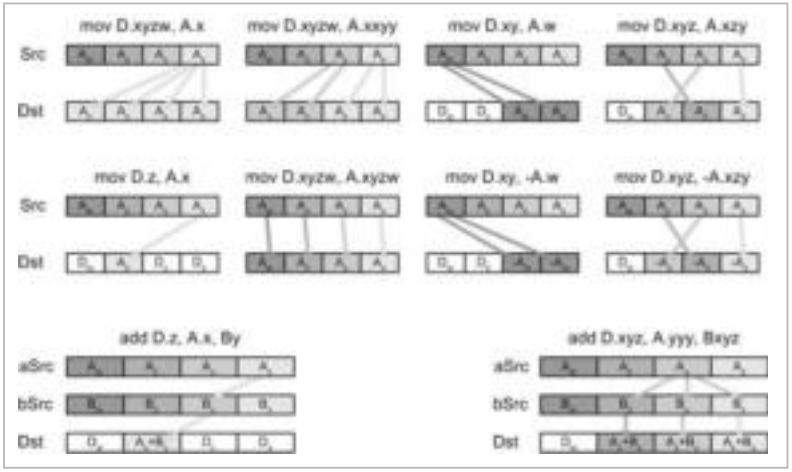
```
pshufd xmm0,xmm1,01001110b ; 1 0 3 2
```

Graphics Processor Unit (GPU) Swizzle

This will be discussed later in Chapter 15, "Vertex and Pixel Shaders." It is mentioned here because graphics processors use this swizzle method to shuffle {XYZW} elements containing single-precision floating-point values around. The PS2 VU coprocessor uses this functionality in almost the same way.



In essence, any set of elements from the source can be selectively used as a source for the computation whose results will be stored into the selected element(s) in the destination.



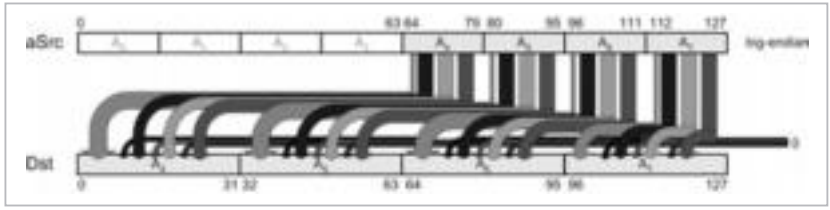
You may have noticed that two of the examples contained a negative sign on the source. This effectively inverts the source inputs to be passed into the function, and although the source can be shuffled, the destination must be kept in an {XYZW} order, and each element only occurs once. An element can be omitted (skipped over) provided there is the same number of source elements. The only exception is when no destination elements are specified, and the resulting vector is copied or the resulting scalar is replicated.

For additional information, skip ahead to the vertex and pixel shaders, or for PS2, see your Linux Kit or devTool manual.

Data Bit Expansion — RGB 5:5:5 to RGB32

One of the tedious conversions is to convert a pixel in a 5:5:5 RGB (Red, Green, Blue) format into a 24- or 32-bit 8:8:8 RGB format.This is a specialized conversion function where 16-bit 5:5:5 pixel color information is converted into 24-bit 8:8:8 plus 8-bit alpha. Though the following instructions have the same idea, they actually perform different variations.

Vector Unpack Low Pixel16 (4x16-bit) to (4x32)



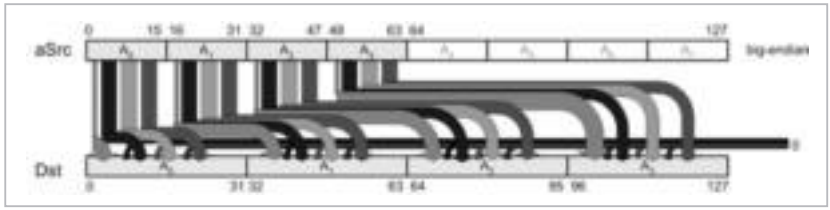
AltiVec vupklpx *Dst, aSrc* [Un]signed 128

vD = vec_unpackl(vA)

```
vupklpx vr0,vr1
```

Bit zero is sign extended to 8 bits. Bits 1...5, 6...10, 11...15 are each zero extended to 8 bits.

Vector Unpack High Pixel16 (4x16-bit) to (4x32)

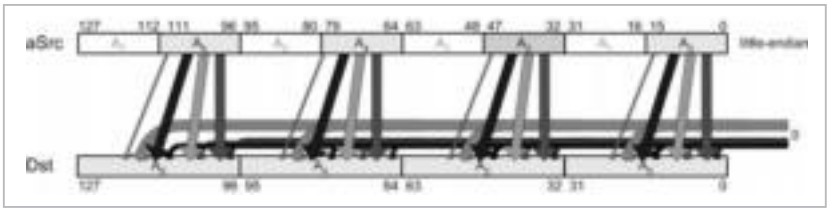


AltiVec vupkhpx *Dst, aSrc* [Un]signed 128
vD = vec_unpackh(vA)

```
vupkhpx vr0,vr1
```

Bit zero is sign extended to 8 bits. Bits 1...5, 6...10, 11...15 are each zero extended to 8 bits.

Parallel Extend from 5 Bits



MIPS/MMI pext5 *Dst, aSrc* [Un]signed 128

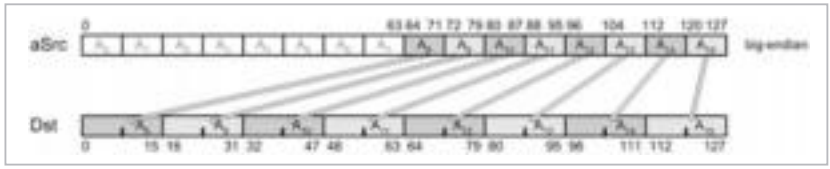
```
pext5 t0, t1
```

In this particular case, the 5 bits are shifted up 3 bits each, effectively scaling by an ×8 factor. The sign bit is used as the sign bit of the 8-bit alpha channel, but the lower 7 bits are set to zero.

Data Bit Expansion

The important item to remember is that with a regular data expansion, the enlarging of the data size from a lower bit count to a higher bit count of an unsigned number only requires a 128-bit value of zero. This needs to be interlaced with the value, and when the bit size is doubled, a zero is, in effect, moved into the upper bits. When working with signed values, instructions such as the following are needed so that the sign bit is replicated into the upper bits. In the following diagrams, note the size differential. A data element is being doubled in size (to half-word from byte or word from half-word). Also, a possible change in sign may occur, which is denoted with a from/to (±).

Vector Unpack Low-Signed Byte (8x8) to (8x16-bit)



AltiVec vupklsb *Dst, aSrc* h=(b,b) 128

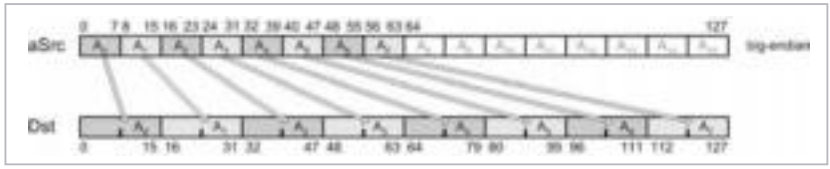
*v**h**D = vec_unpackl(v**b**A)*

```
vupklsb vr0,vr1
```

The most significant bit of each lower-signed 8-bit source value is extended into each of the upper 8 bits of the destination.

Since unpacking and extending the data in conjunction with a value of zero is used for expanding an unsigned value, this and the other following unpack instructions are used for the expansion of signed values.

Vector Unpack High-Signed Byte (8x8) to (8x16-bit)



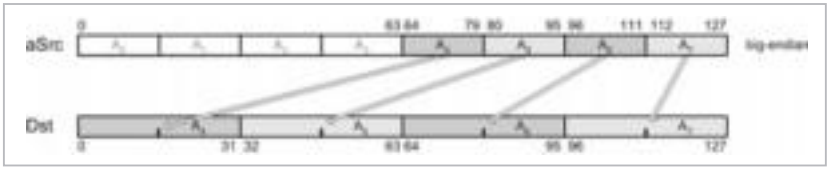
AltiVec vupkhsb *Dst, aSrc* h=(b) 128

*v**h**D = vec_unpackh(v**b**A)*

```
vupkhsb vr0,vr1
```

The most significant bit of each upper-signed 8-bit source value is extended into each of the upper 8 bits of the destination.

Vector Unpack Low-Signed Half-Word (4x16) to (4x32-bit)

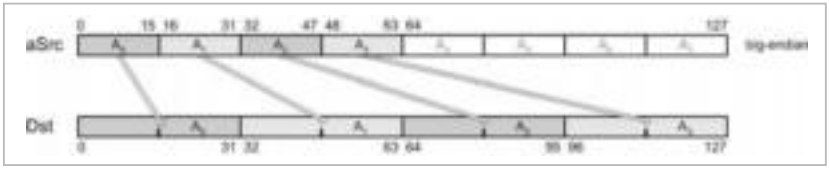


AltiVec vupklsh *Dst, aSrc* w=(h) 128
*v**w**D = vec_unpackl(v**h**A)*

```
vupklsh vr0,vr1
```

The four least significant 16-bit half-word source values are sign extended into the 32-bit destination.

Vector Unpack High-Signed Half-Word (4x16) to (4x32-bit)



AltiVec vupkhsh *Dst, aSrc* w=(h) 128
*v**w**D = vec_unpackh(v**h**A)*

```
vupkhsh vr0,vr1
```

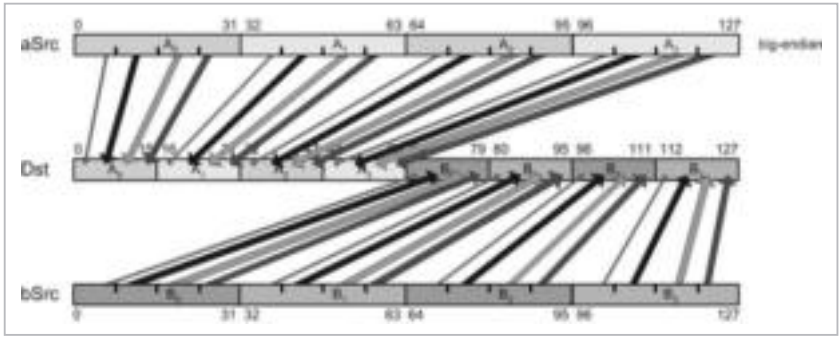
The four most significant 16-bit half-word source values are sign extended into the 32-bit destination.

Data Bit Reduction — RGB32 to RGB 5:5:5

Instructions to convert from RGB 32-bit to 5:5:5 RGB 16-bit was shown earlier. In some cases, it is necessary to convert in the other direction. Why would a game have two sets of all art textures when it could use just one and convert it down to 16 bits? Granted, there would be a loss of color information, and art load times would become longer

due to the extra processing, but in skews of tight media space, sometimes it is necessary.

Vector Pack 32-bit Pixel to 5:5:5



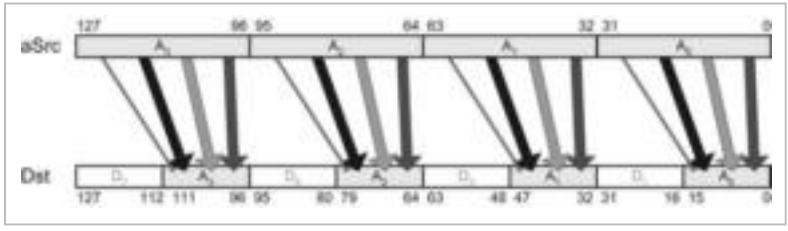
AltiVec vpkpx *Dst, aSrc, bSrc* [Un]signed 128

vD = vec_packpx(vA, vB)

```
vpkpx vr0,vr1,vr2
```

Eight 32-bit pixels can be combined simultaneously. Bit 7 of the word becomes bit 7 of the first byte. The bits 0...4 of the second through fourth bytes become bits {8...12, 16...20, and 24...28}.

Parallel Pack to 5 Bits



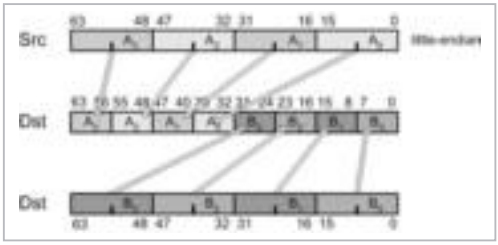
MIPS/MMI ppac5 *Dst, aSrc* [Un]signed 128

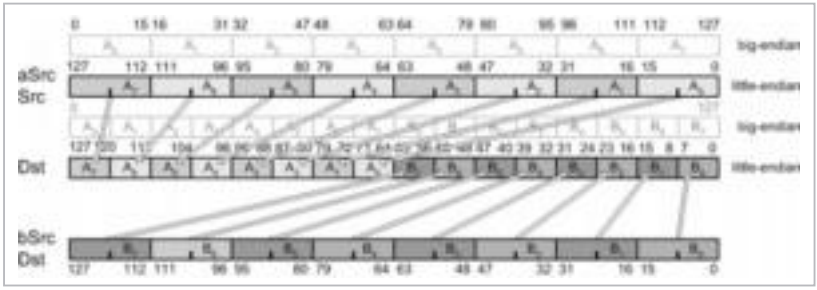
```
ppac5 t0, t1
```

In this particular case, the lower 3 bits of each color component are thrown away; thus the upper 5 bits are shifted down. In the case of the alpha channel, the sign bit of the 32-bit value becomes the sign bit of the 16-bit value.

Data Bit Reduction (with Saturation)

Vector Pack Signed Half-Word Signed Saturate





AltiVec	vpkshss *Dst, aSrc, bSrc* *v**b**D = vec_packs(v**h**A, v**h**B)*	b=(h,h)	128
MMX	packsswb *xmmDst, xmmSrc/m128*	[Un]signed	64
SSE2	packsswb *xmmDst, xmmSrc/m128*	[Un]signed	128

```
vpkshss  vr0,vr1,vr2
packsswb mm0,mm1
packsswb xmm0,xmm1
```

This instruction takes a half-word value of range {–32768 ... 32767} and saturates to a signed 8-bit range of {–128...127}.

Vector Pack Signed Half-Word Unsigned Saturate

AltiVec	vpkshus *Dst, aSrc, bSrc* *v**b**D = vec_packsu(v**h**A, v**h**B)*	$^{+}$b=(h,h)	128
MMX	packuswb *xmmDst, xmmSrc/m128*	[Un]signed	64
SSE2	packuswb *xmmDst, xmmSrc/m128*	[Un]signed	128

```
packuswb mm0,mm1
```

Visibly, this instruction uses the same diagram as the 64-bit form of the instruction *packsswb* but saturates an unsigned half-word with a range of {–32768...32767} to an unsigned 8-bit range of {0...255}.

```
vpkshus  vr0,vr1,vr2
packuswb xmm0,xmm1
```

Visibly, this instruction uses the same diagram as the 128-bit form of the instruction *packsswb* but saturates an unsigned half-word with a range of {–32768...32767} to an unsigned 8-bit range of {0...255}.

Vector Pack Unsigned Half-Word Unsigned Saturate

Visibly, this instruction uses the same diagram as the 128-bit form of the instruction *packsswb* but saturates an unsigned half-word with a range of {0...65535} to an unsigned 8-bit range of {0...255}.

AltiVec	vpkuhus *Dst, aSrc, bSrc*	$^{+}$b=$^{+}$(h,h)	128
	***vbD** = vec_packs(v**h**A, v**h**B)*	Unsigned	
	***vbD** = vec_packsu(v**h**A, v**h**B)*	Signed	

```
vpkuhus  vr0,vr1,vr2
```

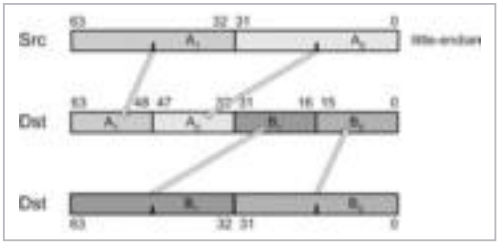
Vector Pack Unsigned Half-Word Unsigned Modulo

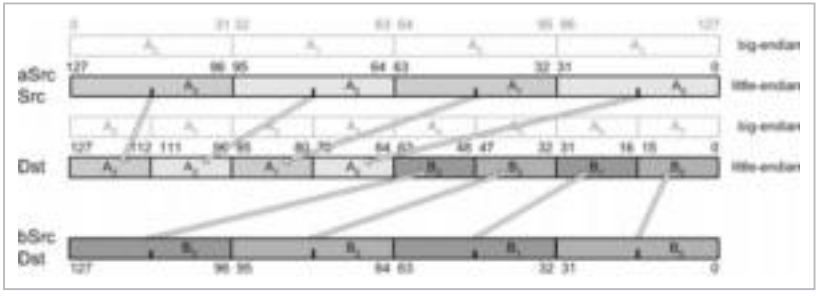
Visibly, this instruction uses the same diagram as the 128-bit form of the instruction *packsswb* but saturates an unsigned half-word with a range of {0...65535} to an unsigned 8-bit modulo with a range of {0...255}.

AltiVec	vpkuhum *Dst, aSrc, bSrc*	$^{+}$b=$^{+}$(h,h)	128
	***vbD** = vec_pack(v**h**A, v**h**B)*	[Un]signed	

```
vpkuhum  vr0,vr1,vr2
```

Vector Pack Signed Word Signed Saturate





AltiVec	vpkswss *Dst, aSrc, bSrc* *v**h**D = vec_packs(v**w**A, v**w**B)*	h=(w,w) Signed int	128
MMX	packssdw *mmDst, mmSrc/m128*	[Un]signed	64
SSE2	packssdw *xmmDst, xmmSrc/m128*	[Un]signed	128

```
vpkswss vr0,vr1,vr2
packssdw mm0,mm1
packssdw xmm0,xmm1
```

This instruction takes a value of range {–2147483648...2147483647} and saturates to a signed 16-bit range of {–32768...32767}.

Vector Pack Signed Word Unsigned Saturate

AltiVec	vpkswus *Dst, aSrc, bSrc* *v**h**D = vec_packsu(v**w**A, v**w**B)*	$^{+}$h=(w,w)	128

```
vpkswus  vr0,vr1,vr2
```

Visibly, this instruction uses the same diagram as the 128-bit form of the instruction *packssdw* but saturates a signed word with a range of {–2147483648...2147483647} to an unsigned 16-bit range of {0...65535}.

Vector Pack Unsigned Word Unsigned Saturate

AltiVec vpkuwus *Dst, aSrc, bSrc* $^{+}h=^{+}(w,w)$ 128

*v**h**D = vec_packs(v**w**A, v**w**B)*

*v**h**D = vec_packsu(v**w**A, v**w**B)*

```
vpkuwus  vr0,vr1,vr2
```

Visibly, this instruction uses the same diagram as the 128-bit form of the instruction *packssdw* but saturates an unsigned word with a range of {0...4294967295} to an unsigned 16-bit range of {0...65535}.

Exercises

1. Write code for your favorite processor to convert an image that is 256 pixels wide by 256 pixels high from 32-bit RGB data to 24-bit RGB.
2. Same as exercise #1, but convert to 5:6:5 bit RGB: 5 bits red, 6 bits green, and 5 bits blue.
3. Convert from packed 5:5:5 RGB to 32-bit RGB.
4. For {SSE2, MMI, AltiVec}, what instruction is needed to convert from a 16-bit signed number to a 32-bit signed number? From 16-bit unsigned to 32-bit unsigned?
5. For {SSE2, MMI, AltiVec}, write code snippets to pack four 128-bit vectors from packed 32-bit to packed 8 bit.
6. Write a function to convert a vector from 128-bit vector of 32-bit packed integers from big endian to little endian and vice versa.
7. Same as exercise #6, but with 32-bit single-precision floating-point values.
8. Same as exercise #6, but with 64-bit double-precision floating-point values.

Chapter 6
Bit Mangling

For about seven years, my family and I lived in the California Sierras, and during that time, I developed a passion for rustic mountain living as well as most environmental associations related to the mountains, the old west, snow, historic mining towns, coaches and wagons, treasure hunting, narrow gauge railroads, wildland fire fighting, and other miscellaneous rural benefits. Now that I have milled that old western imagery into your mind "ore" bored you to death, you can continue reading what I fondly like to refer to as the "mangling of bits." This is one of my favorite sections because with their use, I typically devise speedier methods for use in the manipulation of data. I label this as just more thinking out of the box, which has already been discussed.

Bit mangling relates to the individual "bit twiddling" of bits using Boolean logic, such as NOT, AND, NAND, NOR, OR, and XOR. Each bit is individually isolated so no bit affects any adjacent bit encapsulated by the register. These are not exactly a vector operation due to the use of the individual bits and are thus general-purpose instructions, but it is extremely vector friendly and heavily used by algorithms utilizing vectors. Included are the electronic symbols for each logical operation. Typically, I use my own books for reference from time to time, and I have found that drawing logic circuits using digital logic symbols actually makes more complex Boolean logic algorithms easier for me to simplify. Maybe it will work the same for you.

Any processor professing to contain a multimedia, SIMD, packed, parallel, or vector instruction set will contain almost all of the following instructions in one form or another. Parallel instructions typically do not have a carry flag, as found in some of the older scalar-based instruction sets such as the X86. They tend to lose overflows through the shifting out of bits, wraparound of data, or saturation. Another item to note is that not all of the displayed diagrams are used by all processors defined. Examine the pseudo vector code in those particular cases to emulate that needed functionality. Even if your processor does not have pseudo

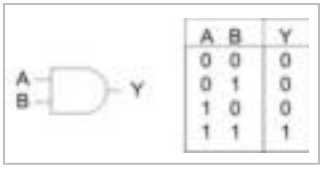
code samples, the sample code listed for other processors could be easily adapted for use with your processor. I would recommend that if you are working with a processor not listed, just pencil it into the book for your own future reference; that way you will have it all in one place.

It must be reiterated again to watch the alignment of your data objects in memory very closely. It takes extra overhead to adjust the memory into an aligned state, and it is a lot more efficient to ensure that they are aligned in the first place. Your code will be smaller and faster! This will be made obvious by the sample code included in this chapter.

CD Workbench Files: /Bench/*architecture*/chap06/*project*/*platform*

	architecture		*project*	*platform*
PowerPC	/vmp_ppc/	**Boolean Logic**	**/pbool/**	/mac9cw
X86	/vmp_x86/			/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Boolean Logical AND



AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vand *Dst*, *aSrc*, *bSrc* *vD=vec_and(vA, vB)*	[Un]signed	128
MMX	pand *mmDst*, *mmSrc(mm/m64)*	[Un]signed	64
SSE	andps *mmDst*, *mmSrc(xmm/m128)*	Single-precision FP	128
SSE2	pand *xmmDst*, *xmmSrc(xmm/m128)*	[Un]signed	128
	andpd x*mmDst*, x*mmSrc(xmm/m128)*	Double-precision FP	128
MIPS	and *Dst*, *aSrc*, *bSrc*	[Un]signed	64
MMI	pand *Dst*, *aSrc*, *bSrc*	[Un]signed	128

An AND operation means that one would need both A and B to be true to have a true result.

This multimedia extension instruction is a parallel operation that uses a Boolean AND operation upon each of the corresponding 64 or 128 bits. The source A and B are *aSrc (xmmSrc)* and *bSrc (xmmDst)*, and the result is stored in the destination *Dst (xmmDst)*. The instruction may be labeled as packed, parallel, or vector, but each bit is in reality isolated from each other, so there is no need for a data bit block arrangement. The following 32-bit example demonstrates this:

```
B      00000000000000001010010110100101b  00000a5a5h
A      00000000000000000000111111110000b  000000ff0h
A^B    00000000000000000000010110100000b  0000005a0h
```

For the SSE instruction, the *andps* is a bit-wise AND of four packed single-precision floating-point values with a bit mask.

For the SSE2 instruction, the *andpd* is a bit-wise AND of two packed double-precision floating-point values with a bit mask.

Pseudo Vec

The following C code demonstrates the functionality of a logical AND upon a 128-bit vector. The code sample logical AND the bits from vector *A* and *B* 32 bits at a time four times to effectively AND all 128 bits and then store the result in vector *D*. Note that the function stores the result pointed to by the first function argument.

Logical Packed AND D=(AB)

Listing 6-1: \chap06\pbool\PBool.cpp

```
void vmp_pand( void *pvD, void *pvA, void *pvB )
{
  uint32 *pD, *pA, *pB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;
  pB=(uint32*) pvB;
  *(pD+0) = *(pA+0) & *(pB+0);
  *(pD+1) = *(pA+1) & *(pB+1);
  *(pD+2) = *(pA+2) & *(pB+2);
  *(pD+3) = *(pA+3) & *(pB+3);
}
```

Note the assertion macro ASSERT_PTR4 since memory alignment for "generic" code only needs to be aligned to 4 bytes to support a 32-bit

value, but for proper 16-byte vector data alignment, use the following instead:

```
ASSERT_PTR16(pvD);
ASSERT_PTR16(pvA);
ASSERT_PTR16(pvB);
```

...as it will assert if the pointer is not properly aligned to 16 bytes (128 bits). Use the appropriate version in each function within your code.

Pseudo Vec (X86)

This procedure can be handled in a number of ways depending on the specific processor being utilized.

vmp_pand (X86)

This first code snippet has been constructed to take advantage of instruction pipelining and uses a general-purpose X86 assembly language logical AND. Please note any code indicated in bold. By substituting alternative mnemonics, such as the Boolean logic instructions *andc*, *or*, and *xor* that have not been discussed yet, in place of AND, the code encapsulation (code snippets) can be mostly reused!

Listing 6-2: vmp_x86\chap06\pbool\PBoolX86M.asm

```
mov  eax,pvA        ; A (Source) Vector
mov  ebx,pvB        ; B (Source) Vector
mov  edx,pvD        ; D (Destination) Vector
```

The previous *mov* instructions are common to the following code samples, and thus not replicated in those samples but need to be recognized as loading general-purpose registers in preparation for those samples.

```
mov  ebp,[eax+0]    ; Read A lower 64bits
mov  esi,[eax+4]
mov  ecx,[ebx+0]    ;   "  B   "      "
mov  edi,[ebx+4]
and  ebp,ecx        ; AND bits (31...0)
and  esi,edi        ; AND bits (63...32)
mov  [edx+0],ebp    ; Write lower 64bits
mov  [edx+4],esi

mov  ebp,[eax+8]    ; Read A upper 64bits
mov  esi,[eax+12]
mov  ecx,[ebx+8]    ;   "  B   "      "
mov  edi,[ebx+12]
and  ebp,ecx        ; AND bits (95...64)
and  esi,edi        ; AND bits (127...96)
mov  [edx+8],ebp    ; Write upper 64bits
mov  [edx+12],esi
```

An optimization worth noting is the interlacing of the registers into pairs to minimize dependencies.

vmp_pand (MMX)

In the following examples, the burden is placed upon 64- or 128-bit registers, so the 32-bit general-purpose registers are only used for memory access. With the MMX instructions, only 64 bits can be manipulated at a time, so the data is handled as upper and lower 64-bit pairs. It also helps minimize processor stalls due to register dependencies.

Listing 6-3: vmp_x86\chap06\pbool\PBoolX86M.asm

```
movq  mm0,[ebx+0]   ; Read B lower 64bits
movq  mm1,[ebx+8]   ;   "  B upper   "
movq  mm2,[eax+0]   ; Read A lower 64bits
movq  mm3,[eax+8]   ;   "  A upper   "

pand  mm0,mm2       ; AND lower 64bits
pand  mm1,mm3       ; AND upper 64bits

movq  [edx+0],mm0   ; Write D lower 64bits
movq  [edx+8],mm1   ;   "     upper   "
```

vmp_pand (SSE2) Aligned Memory

For full 128-bit handling on an X86 processor, it becomes necessary to require a minimum of a Pentium 4 with the SSE2 instruction set. The following SSE2 example is a better solution when compared to the MMX version. Despite the register dependency stall that is in this specific example, there is no FPU/MMX conflict, thus an avoidance of the MMX registers. All 128 bits are handled simultaneously, resulting in a smaller code size. There is a little problem with memory alignment with SSE and SSE2, thus two versions of the function will be needed if strict memory alignment procedures are not followed.

Listing 6-4: vmp_x86\chap06\pbool\PBoolX86M.asm

```
movdqa  xmm0,[ebx]  ; Read B Aligned 128bits
movdqa  xmm1,[eax]  ; Read A

pand    xmm0,xmm1   ; AND 128bits

movdqa  [edx],xmm0  ; Write D Aligned 128bits
```

Note the use of *movdqa* (aligned) used previously and *movdqu* (unaligned) used in the following examples. The code is virtually identical for both examples, except that if the data is misaligned and the *movdqa* instruction is utilized, an exception will occur. If memory

alignment cannot be guaranteed, the following (slightly slower version) should be used instead.

vmp_pand (SSE2) Unaligned Memory

Listing 6-5: vmp_x86\chap06\pbool\PBoolX86M.asm

```
movdqu  xmm0,[ebx]  ; Read B non-aligned 128bits
movdqu  xmm1,[eax]  ; Read A

pand    xmm0,xmm1   ; AND 128bits

movdqu  [edx],xmm0  ; Write D non-aligned 128bits
```

You see, I was not trying to pull the wool over your eyes or anything. The really nice feature for the SSE and SSE2 instructions is that for both aligned and unaligned data, the code is virtually identical except for the method of access. The only trick is to make sure it is properly aligned before using *movdqa*. If in doubt, use the instruction *movdqu*; otherwise, an exception will occur upon that misaligned access.

You may now be thinking, but why bother using *movdqa*? Why not just use *movdqu* all the time?

The answer: Your code will run slower, and that is contrary to the reason for writing your code in assembly or using vector instructions!

Pseudo Vec (PowerPC)

For the PowerPC processor, compare the C code sample previously shown in Pseudo Vec to the following PowerPC assembly using 32-bit registers in quads for full 128-bit processing.

vmp_pand (PowerPC)

Listing 6-6: vmp_ppc\chap06\pbool\PBoolPPC.cpp

```
unsigned int register a0, a1, a2, a3, b0, b1, b2, b3;

__asm {
  lwz  a0,0(r4)       // Read A  Bits  {0...31}
  lwz  a1,4(r4)
  lwz  a2,8(r4)
  lwz  a3,12(r4)      //      "        {96...127}

  lwz  b0,0(r5)       // Read B  Bits  {0...31}
  lwz  b1,4(r5)
  lwz  b2,8(r5)
  lwz  b3,12(r5)      //      "        {96...127}

  and  a0,a0,b0       // AND     Bits  {0...31}
  and  a1,a1,b1
  and  a2,a2,b2
```

```
  and  a3,a3,b3       //      "        {96...127}

  stw  a0,0(r3)       // Write D Bits  {0...31}
  stw  a1,4(r3)
  stw  a2,8(r3)
  stw  a3,12(r3)      //      "        {96...127}
}
```

The AltiVec instruction set with its 128-bit registers vastly accelerates code and minimizes data manipulation. Compare the previous PowerPC assembly to the following AltiVec library call.

vmp_pand (Altivec) Aligned Memory

Listing 6-7: vmp_ppc\chap06\pbool\PBoolAltivec.cpp

```
  *(vector unsigned int *)pvD =
        vec_and( (*(vector unsigned int *)pvA),
                 (*(vector unsigned int *)pvB) );
```

Most of what is seen is the type casting to convert from the generic void pointers of the C cross-platform wrapper of this book to the AltiVec specific library call and its data type expectation, such as (vector unsigned int *). When stripped, it effectively looks like this:

```
vD = vec_and( vA, vB );
```

This is very simple. Broken down into its inline assembly code, it looks like this:

```
vand vD, vA, vB
```

To reinforce the importance of keeping data aligned, the following code has to be alternatively used when memory alignment cannot be guaranteed.

vmp_pand (Altivec) Unaligned Memory

Listing 6-8: vmp_ppc\chap06\pbool\PBoolAltivec.cpp

```
vector unsigned int vD, vA, vB;

  *(((uint32 *)&vA)+0) = *(((uint32 *)pvA)+0);
  *(((uint32 *)&vA)+1) = *(((uint32 *)pvA)+1);
  *(((uint32 *)&vA)+2) = *(((uint32 *)pvA)+2);
  *(((uint32 *)&vA)+3) = *(((uint32 *)pvA)+3);

  *(((uint32 *)&vB)+0) = *(((uint32 *)pvB)+0);
  *(((uint32 *)&vB)+1) = *(((uint32 *)pvB)+1);
  *(((uint32 *)&vB)+2) = *(((uint32 *)pvB)+2);
  *(((uint32 *)&vB)+3) = *(((uint32 *)pvB)+3);

  vD = vec_and( vA, vB );
```

```
*(((uint32 *)pvD)+0) = *(((uint32 *)&vD)+0);
*(((uint32 *)pvD)+1) = *(((uint32 *)&vD)+1);
*(((uint32 *)pvD)+2) = *(((uint32 *)&vD)+2);
*(((uint32 *)pvD)+3) = *(((uint32 *)&vD)+3);
```

Note all the excess baggage just to load the two 128-bit vectors, *A* and *B*, from memory, perform the operation, and write the results back out to vector *D* in memory. The assertion is reduced for non-alignment, and all the data has to be transferred from non-aligned memory to aligned memory before the process takes place. The reverse has to be done once the logic operation has completed. One might as well do four separate 32-bit operations!

Pseudo Vec (MIPS)

Here is where things get interesting. If you recall, the VU chip of the PS2 has four 32-bit single-precision floating-point registers arranged as a 128-bit vector and 16-bit integer registers, but no vector integer operations are supported by the VU coprocessor's instruction set. The same applies for all of the vector-based Boolean functions in this chapter embedded within the MIPS processor.

Only the ArTile C790 (EE) with its MIPS processor and 128-bit integer multimedia registers has that ability on this platform. Interesting, though, that in job interviews, the interviewers give every indication that they do not care about that fact (at least in the interviews I have been in)! They could apparently care less about the EE, even when it is pointed out that only the MIPS processor's multimedia core has the 128-bit, integer-based parallel capabilities. They just cannot seem to get around a missing 16-bit Boolean instruction, and thus they tend to ignore the powerful integer multimedia instruction set, but that will be discussed later. With that aside, let's examine the multimedia-oriented parallel processing on the newer 64- and 128-bit advanced RISC microprocessors with MIPS instruction sets.

vmp_pand (MIPS III) Aligned Memory

I do not want to be seen as villainous for only dealing with processors used in video games, so here is a token for you embedded programmers. When dealing with 64-bit MIPS processors, merely process the data in parallel to achieve 128-bit vector handling. Two sets of 64-bit Boolean data can be handled in parallel with the use of 64-bit registers.

Listing 6-9

```
ld      t0, 0(a1)          // pvA {A63...A0}
ld      t2, 0(a2)          // pvB {B63...B0}
ld      t1, 8(a1)          // pvA {A127...A64}
ld      t3, 8(a2)          // pvB {B127...B64}

and     t0, t0, t2         // {A63∧B63 ... A0∧B0}
and     t1, t1, t3         // {A127∧B127 ... A64∧B64}

sd      t0, 0(a0)          // pvD {D63...D0}
sd      t1, 8(a0)          // pvD {D127...D64}
```

vmp_pand (MMI) Aligned Memory

When dealing with aligned memory, this is a snap.

Listing 6-10: vmp_mips\chap06\pbool\PBoolMMI.s

```
lq      t1, 0(a1)          // pvA {A127...A0}
lq      t2, 0(a2)          // pvB {B127...B0}
nop                        // NOP - Load Delay Slot

pand    t0, t1, t2         // {A127∧B127 ... A0∧B0}

sq      t0, 0(a0)          // pvD {D127...D0}
```

vmp_pand (MMI) Unaligned Memory

Since memory is unaligned, there is a choice to either load the data 32 bits at a time into an aligned stack argument or use an unaligned access, which is what is used here. An item to remember is that even if a MIPS processor with MMI supports 128-bit packed data, most of its normal general operations are 64-bit based. This is also considered the size of its integer in C. Data, therefore, needs to be loaded as unaligned 64 bit, since the nature of Boolean means that each bit is isolated no matter if the data size is 8 bits or 128 bits. Thus, there is no need to merge the separate 64-bit values into a 128-bit form for the packed parallel operations. You should notice that an *and* instruction is used instead of the *pand* instruction, as the upper 64 bits should not be trusted.

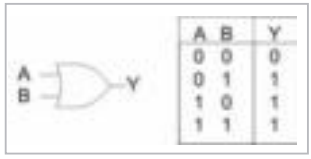
Listing 6-11: vmp_mips\chap06\pbool\PBoolMMI.s

```
ldl     t1, 7(a1)          // pvA  {A63...A0}
ldr     t1, 0(a1)
ldl     t3, 15(a1)         //      {A127...A64}
ldr     t3, 8(a1)

ldl     t2, 7(a2)          // pvB  {B63...B0}
ldr     t2, 0(a2)
ldl     t4, 15(a2)         //      {B127...B64}
ldr     t4, 8(a2)
```

```
and    t0, t1, t2        // {A63^B63 ... A0^B0}
and    t1, t3, t4        // {A127^B127 ... A64^B64}

sdl    t0, 7(a0)         // pvD {D63...D0}
sdr    t0, 0(a0)
sdl    t1, 15(a0)        //    {D127...D64}
sdr    t1, 8(a0)
```

For specific information, see a MIPS C790, PS2 Linux Kit, or PS2 devTool manual.

Boolean Logical OR



AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vor *Dst*, *aSrc*, *bSrc* *vD=vec_or(vA, vB)*	[Un]signed	128
MMX	por *mmDst*, *mmSrc(mm/m64)*	[Un]signed	64
SSE	orps x*mmDst*, x*mmSrc(xmm/m128)*	Single-Precision FP	128
SSE2	por *xmmDst*, *xmmSrc(xmm/m128)*	[Un]signed	128
	orpd x*mmDst*, x*mmSrc(xmm/m128)*	Double-Precision FP	128
MIPS	or *Dst*, *aSrc*, *bSrc*	[Un]signed	64
MMI	por *Dst*, *aSrc*, *bSrc*	[Un]signed	128

An OR operation means that one would need either A or B to be true to have a true result.

This multimedia extension instruction is a parallel operation that uses a Boolean OR operation upon each of the corresponding 64 or 128 bits. The source *A* and *B* are *aSrc (xmmSrc)* OR *bSrc (xmmDst)*, and the result is stored in the destination *Dst (xmmDst)*. The instruction may be labeled as packed, parallel, or vector, but each bit is, in reality, isolated from each other, so there is no need for a data bit block arrangement. The following 32-bit example demonstrates that.

```
B       00000000000000001010010110100101b   0a5a5h
A       00000000000000000000111111110000b   00ff0h
A∨B     00000000000000001010111111110101b   0aff5h
```

For the SSE instruction, *orps* is a bit-wise OR of four packed single-precision floating-point values with a bit mask.

For the SSE2 instruction, *orpd* is a bit-wise OR of two packed double-precision floating-point values with a bit mask.

Pseudo Vec

The following C code demonstrates the functionality of a logical OR upon a 128-bit vector. The code sample logical OR's the bits from vector A and B 32 bits at a time four times to effectively OR all 128 bits and store the result in vector D. Note that the function stores the result pointed to by the first function argument.

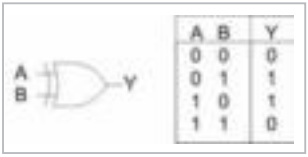
Logical Packed OR D=(A∨B)

Listing 6-12: \chap06\pbool\PBool.cpp

```
void vmp_por( void *pvD, void *pvA, void *pvB )
{
  uint32 *pD, *pA, *pB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;
  pB=(uint32*) pvB;
  *(pD+0) = *(pA+0) | *(pB+0);   // {31...0}
  *(pD+1) = *(pA+1) | *(pB+1);   // {63...32}
  *(pD+2) = *(pA+2) | *(pB+2);   // {95...64}
  *(pD+3) = *(pA+3) | *(pB+3);   // {127...96}
}
```

See the code snippets from the previous discussed instruction AND, and then substitute the instruction {*or, por, vor, vec_or()*} for the {*and*, *pand*, *vand*, *vec_and()*} accordingly.

Boolean Logical XOR (Exclusive OR)



AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI
AltiVec	vxor *Dst, aSrc, bSrc* *vD = vec_xor(vA, vB)*					[Un]signed	128
MMX	pxor *mmDst, mmSrc(mm/m64)*					[Un]signed	64
SSE	xorps *xmmDst, xmmSrc(xmm/m128)*						
						Single-Precision FP	128
SSE2	pxor *xmmDst, xmmSrc(xmm/m128)*					[Un]signed	128
	xorpd *xmmDst, xmmSrc(xmm/m128)*						
						Double-Precision FP	128
MIPS	xor *Dst, aSrc, bSrc*					[Un]signed	64
MMI	pxor *Dst, aSrc, bSrc*					[Un]signed	128

An XOR operation means that one would need either A or B to be true, but not both, to have a true result.

This multimedia extension instruction is a parallel operation that uses a Boolean XOR operation upon each of the corresponding 64 or 128 bits. The source *A* and *B* are *aSrc (xmmSrc)* XOR *bSrc (xmmDst)*, and the result is stored in the destination *Dst (xmmDst)*. The instruction may be labeled as packed, parallel, or vector, but each bit is, in reality, isolated from each other, so there is no need for a data bit block arrangement. The following 32-bit example demonstrates that.

```
B     00000000000000001010010110100101b   0a5a5h
A     00000000000000000000111111110000b   00ff0h
A⊕B   00000000000000001010101001010101b   0aa55h
```

This is typically used for the flipping of selected bits.

For the SSE instruction, *xorps* is a bit-wise XOR of four packed single-precision floating-point values with a bit mask.

For the SSE2 instruction, *xorpd* is a bit-wise XOR of two packed double-precision floating-point values with a bit mask.

Pseudo Vec

The following C code demonstrates the functionality of a logical XOR upon a 128-bit vector. The code sample logical XOR's the bits from vector A and B, 32 bits at a time, four times to effectively XOR all 128 bits and store the result in vector D. Note that the function stores the result referenced by the first function parameter pointer.

Logical Packed XOR D=(AB)

Listing 6-13: \chap06\pbool\PBool.cpp

```
void vmp_pxor( void *pvD, void *pvA, void *pvB )
{
  uint32 *pD, *pA, *pB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;
  pB=(uint32*) pvB;
  *(pD+0) = *(pA+0) ^ *(pB+0);   // {31...0}
  *(pD+1) = *(pA+1) ^ *(pB+1);   // {63...32}
  *(pD+2) = *(pA+2) ^ *(pB+2);   // {95...64}
  *(pD+3) = *(pA+3) ^ *(pB+3);   // {127...96}
}
```

See the code snippets from the previous discussed instruction AND, and then substitute the instruction {*xor*, *pxor*, *vxor*, *vec_xor()*} for the {*and*, *pand*, *vand*, *vec_and()*} accordingly.

Hint: For a Boolean NOT (one's complement), use an XOR.

Another use for this operation is as a Boolean NOT (one's complement) operator. A NOT is not typically an SIMD instruction, as bit real estate dictating the number of instructions a processor can handle is limited, and this instruction can be easily implemented by an adaptation with an exclusive OR. By using an input *A* and setting *B* permanently to a logical one, an inverse bit is achieved. Note the following image, where zero becomes one and one becomes zero!

A ⊕ B	Y
0 |	I
I |	0

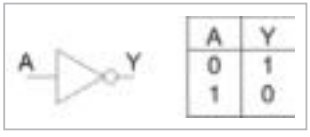


Figure 6-1: Example of using a logical XOR with a logical true bit to achieve the equivalent of a logical NOT condition

As can be seen, the input *A* and the output *Y* are exactly the same. Keep in mind that Boolean logic is bit based and bit isolated so that adjacent bits do not affect each other, such as in the following sample of 32 adjacent bits:

```
NOT 00000000000000001010010110100101b   00000a5a5h
    11111111111111110101101001011010b   0ffff5a5ah
```

Guess what you get when you flip it again?

```
NOT 11111111111111110101101001011010b   0ffff5a5ah
    00000000000000001010010110100101b   00000a5a5h
```

The original value — flip-flop, almost like tiddlywinks.

Hint: For a NEG (Negate) (two's complement), use an XOR followed by an increment.

Another use for an XOR is in an implementation for a negation (two's complement). As a refresher, this is a subtraction in a two's complement followed by an addition. By inverting all bits, a one's complement is achieved, and the next step is the two's complement by an increment (addition of one) of this value, which is then followed by the addition, effectively resulting in the subtraction.

This operation is the result of a Boolean NOT (one's complement) operator that was just discussed followed by an increment or an addition by one. Of course, this is not as slick as a reverse subtraction where...

$$-A = A \text{ ISUB } 0 \quad \text{thus} \quad -A = 0 - A$$

...but not all processors support that. Check it out in Chapter 8, "Vector Addition and Subtraction." Keep in mind though that when dealing with pipelining your code, a two-step operation may be more helpful than a single step.

```
    00000000000000001010010110100101b   00000a5a5h (42405)
NOT 11111111111111110101101001011010b   0ffff5a5ah (–42406)
INC 11111111111111110101101001011011b   0ffff5a5bh (–42405)
```

Toolbox Snippet — The Butterfly Switch

There is an interesting use for a logical XOR, and that is as a butterfly switch to allow for branchless coding. Normal coding that uses branching is typically taught to use something such as the following:

```
#define FLIP  –30
#define FLOP   47

  if ( FLIP == nf )
  {
    nf = FLOP;
  }
  else
```

```
{
  nf = FLIP;
}
```

...or...

```
nf = ( FLIP == nf ) ? FLOP : FLIP;
```

No matter which way you code it, it is the same identical code. This is fine and dandy, but the branching, especially a possible misprediction of a branch, takes time and there are two branches to contend with. If a FLIP, it flops, and if a FLOP, then it flips. Of course, instead of two branches, such as the previous code snippet, it could always be coded for one branch, such as the following:

```
nf = FLIP;
if ( FLIP == nf )
{
  nf = FLOP;
}
```

The code, if not a FLIP, as in the previous code snippet, branches around and continues on but again, there could be a misprediction.

A misprediction is as it sounds. The more advanced CPU will predict that at an if-then conditional, it will take the branch and do the conditional or branch around, thus the if-then-else. The problem is that the CPU gains efficiency by predicting that it is correct because it is pre-loading memory and, in some cases, executing instructions further down the code. The punishment comes that if it predicted wrong, that memory has to be thrown away, and the results of the calculations it processed ahead of time are disposed of. Then it needs to continue processing down the correct path. Either way, this is very time consuming, and so alternative (branchless) methods need to be devised if possible.

My favorite solution is a branchless result so there is no misprediction and the appropriate value can be selected with a butterfly switch. Let's examine these two values more closely:

```
FLOP   =    47   =   0x002F   =   0000000000101111b
FLIP   =   –30   =   0xFFE2   =   1111111111100010b
```

...and calculate the logical XOR of those two values:

```
FLIPPY =   –51   =   0xFFCD   =   1111111111001101b
```

If in our code initialization we preset a value:

```
nf = FLOP;
```

...and in place of the branching code the following snippet is used instead:

```
xor   nf,nf,FLIPPY    // FLIP / FLOP
```

...each time the code is executed it will flip to the alternate value.

Flip, flop, flip, flop, flip, flop, flip, etc.

```
   FLIP     1111111111100010b  0xFFE2  -30
⊕ FLIPPY    1111111111001101b  0xFFCD
   FLOP     0000000000101111b  0x002F   47
⊕ FLIPPY    1111111111001101b  0xFFCD
   FLIP     1111111111100010b  0xFFE2  -30
```

The best part about this is that the actual code is a single instruction, rather than a group of instructions to process a branch and decide whether to branch or not! So the code runs a lot faster, and it's smaller. This also works with non-definitions as well. Initialize with the following:

```
nf = valueA;            // First value
iFlipVal = nf ^ valueB; // Butterfly Key
```

...and select the value with the following:

```
nf = nf ^ iFlipVal;
```

... and it works great in a parallel configuration — different butterflies to control different elements, all in parallel.

If anything, at least this book is informative!

I-VU-Q

Which instruction is thought to be missing from the VU coprocessor on the PS2?

This seems to be a popular interview question, as I have encountered it numerous times. After the interviewers ask this question, they sometimes want to know how to write equivalent code. The funny thing is that they do not seem to remember the answer themselves. I will hastily draw out the following truth table from left to right and then draw the following circuit for good measure (that is, if I am not too tired and frustrated from having to answer programming questions all day long!).

A	B	Y	A·B	(A·B)'	A
0	0	0	0	1	0
0	1	1	1	1	1
1	0	1	1	1	1
1	1	0	1	0	0

Figure 6-2: A four-gate (OR, NOT-AND, AND) solution to achieve an equivalent result of a logical XOR

I will then hesitate for a second and announce, "But wait!" This takes four logical operations — an OR, AND, NOT, and AND. So instead, let's make this a bit smaller. If the ANDC type functionality is used (which has not been discussed yet):

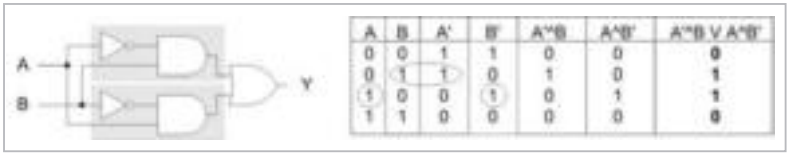


Figure 6-3: A two-gate (ANDC, OR) solution to achieve an equivalent result of a logical XOR

...notice the swapped inputs and the gate functionality similar to a logical AND. The outputs are logical OR gated together! So now it is down to two.

"But wait — there is no NOT, NOR, XOR, or ANDC on a PS2's VU coprocessor. They only exist on the EE!" So at this point, you replicate what is being presented here at your interview as a response to the question; you will either come off sounding like a really smart guy or a prima donna, depending on how you do it. So you then announce, "but wait! There is no NOT or ANDC, so how do we do it? Ever heard of a half adder?" It has the same result as that of an XOR, except it also contains a carry, and that is where the problem resides. That carry bit is contaminating an adjacent bit.

Table 6-1: A half adder solution. By ignoring the carry, a logical XOR will result.

A + B		Carry	Y
0	0	0	**0**
0	1	0	**1**
1	0	0	**1**
1	1	1	**0**

If the bits are stripped into an odd-even arrangement, the *A* and *B* (**o**dd bits) are summed and masked with the "1010" bit mask pattern. The *A* and *B* (**e**ven bits) are summed and masked with the "0101" even bit mask pattern. The results of the odd and even are logical OR'd with each other; effectively, a logical XOR is simulated. Let's examine some 16-bit data:

```
          oeoeoeoeoeoeoeoe
B         1010010110100101b   0a5a5h
A         0000111111110000b   00ff0h
A⊕B       1010101001010101b   0aa55h
```

So by a logical AND with the odd and even masks, an un-interlaced form is generated. Thus, any carry from a later summation will affect only an adjacent bit being ignored with no addition carry to a second adjacent bit. Notice that the bits indicated in bold are the usable result from the AND.

```
       B      1010010110100101b   0a5a5h
Even ∧ B      0101010101010101b   05555h (Mask)
Even's B      0000010100000101b   00505h

       B      1010010110100101b   0a5a5h
Odd  ∧ B      1010101010101010b   0aaaah (Mask)
Odd's  B      1010000010100000b   0a0a0h

       A      0000111111110000b   00ff0h
Even ∧ A      0101010101010101b   05555h (Mask)
Even's A      0000010101010000b   00550h

       A      0000111111110000b   00ff0h
Odd  ∧ A      1010101010101010b   0aaaah (Mask)
Odd's  A      0000101010100000b   00aa0h
```

Now the even and odd values of *A* and *B* are summed up separately. Note that we only care about the resulting bits in bold and not the others, as those are the individual carries, which are stripped by the logical AND of the original mask.

```
Even's B      0000010100000101b   00505h
Even's A +    0000010101010000b   00550h
              0000010010101010b   00a55h
Even      ∧   0101010101010101b   05555h (Mask)
Even    A⊕B   0000000001010101b   00055h

Odd's  B      1010000010100000b   0a0a0h
Odd's  A +    0000101010100000b   00aa0h
              1010101101000000b   0ab40h
Odd       ∧   1010101010101010b   0aaaah (Mask)
Odd     A⊕B   1010101000000000b   0aa00h
```

Now logical OR the even bits and odd bits back together for the interlaced XOR result.

```
Even    A⊕B   0000000001010101b   00055h
Odd  ∨  A⊕B   1010101000000000b   0aa00h
        A⊕B   1010101001010101b   0aa55h
```

When compared to the expected results of a "real" XOR,

```
        A⊕B   1010101001010101b   0aa55h
```

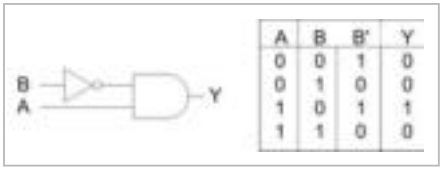
...they are exactly the same (okay, a lot more operations, but just another technique for your repertoire). Now that you have special insight into the problem, it is the start of solving this equation for yourself!

You did not think that I would be giving you all the answers, did you?

Hint: There is one more method available to do a logical not. A subtraction is a two's complement, as it is a bit flip (one's complement) followed by an increment (two's complement) and then an addition. If the result of a subtraction from zero is decremented, you effectively get a not (one's complement). A'=(0–A)–1

Does this give you any ideas? Remember that the VU only has 16-bit half-word size integers.

Boolean Logical ANDC



A	B	B'	Y
0	0	1	0
0	1	0	0
1	0	1	1
1	1	0	0

AltiVec	**MMX**	**SSE**	**SSE2**	*3DNow*	*3DMX+*	*MIPS*	*MMI*

AltiVec	vandc *Dst*, *aSrc*, *bSrc* *vD=vec_andc(vA, vB)*	[Un]signed	128
MMX	pandn *mmDst*, *mmSrc(mm/m64)*	[Un]signed	64
SSE	andnps x*mmDst*, x*mmSrc(xmm/m128)*	Single-precision FP	128
SSE2	pandn *xmmDst*, *xmmSrc(xmm/m128)*	[Un]signed	128
	andnpd x*mmDst*, x*mmSrc(xmm/m128)*	Double-precision FP	128

This instruction is a one's complement of B logical AND with A.

Altivec: D=A∧(B′) X86: Dst=Src∧(Dst′)

This multimedia extension instruction is a parallel operation that uses a Boolean NOT AND operation upon each of the corresponding 64 or 128 bits. The source *A* and *B* are *aSrc (xmmSrc)* and a one's complement of *bSrc (xmmDst)*, and the result is stored in the destination *Dst (xmmDst)*. The instruction may be labeled as packed, parallel, or vector, but each bit is in reality isolated from each other, so there is no need for a data bit block arrangement. The following 32-bit example demonstrates this:

```
B        00000000000000001010010110100101b  00000a5a5h
B′  NOT  11111111111111110101101001011010b  0ffff5a5ah
A   ∧    00000000000000000000111111110000b  000000ff0h
A∧B′     00000000000000000000101001010000b  000000a50h
```

For the SSE instruction, *andnps* is a bit-wise NOT AND of four packed single-precision floating-point values with a bit mask.

For the SSE2 instruction, *andnpd* is a bit-wise NOT AND of two packed double-precision floating-point values with a bit mask.

Pseudo Vec

The following C code demonstrates the functionality of a logical ANDC upon a 128-bit vector. The code sample logical NOT's the bits from vector *B* and then AND's these bits with vector *A*, 32 bits at a time four times, to effectively ANDC all 128 bits and then store the result in vector *D*. Note that the function stores the result referenced by the first function parameter pointer.

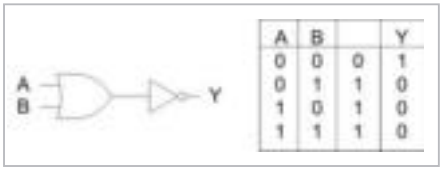
Logical Packed ANDC D=(A∧B′)

Listing 6-14: \chap06\pbool\PBool.cpp

```
void vmp_pandc( void *pvD, void *pvA, void *pvB )
{
  uint32 *pD, *pA, *pB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;
  pB=(uint32*) pvB;

  *(pD+0) = (0xffffffff ^ *(pB+0)) & *(pA+0);
  *(pD+1) = (0xffffffff ^ *(pB+1)) & *(pA+1);
  *(pD+2) = (0xffffffff ^ *(pB+2)) & *(pA+2);
  *(pD+3) = (0xffffffff ^ *(pB+3)) & *(pA+3);
}
```

See the code snippets from the previous instruction AND and substitute the instruction {*andc, pandn, vandc, vec_andc()*} for the {*and*, *pand*, *vand*, *vec_and()*} accordingly.

Boolean Logical NOR (NOT OR)



AltiVec	*MMX*	*SSE*	*SSE2*	*3DNow*	*3DMX+*	**MIPS**	**MMI**

AltiVec	vnor	*Dst*, *aSrc*, *bSrc*		
		vD=vec_nor(vA, vB)	[Un]signed	128
MIPS	nor	*Dst*, *aSrc*, *bSrc*	[Un]signed	64
MMI	pnor	*Dst*, *aSrc*, *bSrc*	[Un]signed	128

A NOT OR operation means that you would need both A and B to be false in order to have a true result.

This multimedia extension instruction is a parallel operation that uses a Boolean NOT OR operation upon each of the corresponding 64 or 128 bits. The source *aSrc* and *bSrc* are logical OR'd then one's complement, and the result is stored in the destination *Dst*. Effectively, after the logical OR, the result of each bit is inverted, thus a one's complement. The following 32-bit example demonstrates this.

```
B       00000000000000001010010110100101b  00000a5a5h
A       00000000000000000000111111110000b  000000ff0h
  A∨B   00000000000000001010111111110101b  00000aff5h
(A∨B)′  11111111111111110101000000001010b  0ffff500ah
```

The instruction may be labeled as packed, parallel, or vector, but each bit is in reality isolated from each other so there is no need for a data bit block arrangement.

This instruction has only been recently implemented as an instruction on some of the newer processors. This can also be used for its NOT functionality. By using a source of zero for one of the inputs, a NOT functionality is achieved.

Pseudo Vec

The following C code demonstrates the functionality of a logical NOR upon a 128-bit vector. The code sample logical OR's the bits from vector *A* and *B* and logical NOT's the results 32 bits at a time four times to effectively NOR all 128 bits and then store the result in vector *D*. Note

that the function stores the result referenced to by the first function parameter pointer.

Logical Packed NOR D=(A∨B′)

Listing 6-15: \chap06\pbool\PBool.cpp

```
void vmp_pnor( void *pvD, void *pvA, void *pvB )
{
  uint32 *pD, *pA, *pB;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pvA);
  ASSERT_PTR4(pvB);

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;
  pB=(uint32*) pvB;

  *(pD+0) = 0xffffffff ^ (*(pB+0) | *(pA+0));
  *(pD+1) = 0xffffffff ^ (*(pB+1) | *(pA+1));
  *(pD+2) = 0xffffffff ^ (*(pB+2) | *(pA+2));
  *(pD+3) = 0xffffffff ^ (*(pB+3) | *(pA+3));
}
```

Pseudo Vec (X86)

The X86, however, does not have NOT OR functionality, so it must be simulated with a trailing logical NOT. As explained earlier, the result of a Boolean value XOR with a logical one is a one's complement, thus the value is inverted. Keep in mind the same code wrappers used by the logical AND will work here as well!

Listing 6-16: vmp_x86\chap06\pbool\PBoolX86M.asm

```
NotMaskQ dq  0ffffffffffffffffh,0ffffffffffffffffh

vmp_pnor (MMX)

  por   mm0,mm2       ; OR lower 64bits
  por   mm1,mm3       ; OR upper 64bits
  pxor  mm0,NotMaskQ  ; Flip Bits {63...0} (NOT) for NOR
  pxor  mm1,NotMaskQ  ;           {127...64}


vmp_pnor (SSE2)

  por   xmm0,xmm1     ; OR 128bits
  pxor  xmm0,NotMaskQ ; Flip Bits for NOR
```

Pseudo Vec (PowerPC)

The PowerPC is a very versatile processor, as it supports *andc*, *nand*, *vandc*, *orc*, *nor*, and *vnor*, so see the code snippets from the previous instruction AND and substitute the instruction {*nor*, *vnor*, *vec_nor()*} for the {*and*, *vand*, *vec_and()*} accordingly.

Graphics 101 — Blit

There are different methods that one can choose to blit or bit field copy a graphics image: a pure blit where the image is merely copied pixel by pixel or a transparent copy, which is detailed here.

A transparent pixel is referred to by a variety of names: "transparent," "color key," "skip color," "invisible color," "non-displayed pixel," etc. This is a pixel containing no image color data that allows the color of the pixel directly underneath it to be displayed. It is typically set to an unusual color that helps the artists and programmers to easily identify it in relation to the rest of the colors.

If you watch the news, you see this process every day, compliments of the weatherman. He is shot on a green screen, being careful not to wear a color similar to the "color key," and the electronics make him appear in front of an image, such as a map. That composite image is transmitted to your television. If he wore a shirt that was the same shade of color as the color key, he would appear to have a big hole in the middle of his chest where you could see through his body.

In the film medium, moviemakers shoot models or actors on a blue screen, and the color blue is actually clear on the film negative. Simplifying this explanation, the non-clear areas would be converted into a mask and the images would be cookie-cut into a composite, typically using a matte backdrop.

In the digitized graphics medium in a computer, movie/game makers shoot actors on a green screen, and the images would be digitally mapped into a single image using some sort of backdrop.

Your transparency color can be any color. I typically pick a dark shade of blue. In an RGB range of (0 to 255), {red:0, green:0, blue:108}. This allows me to differentiate between the color black and transparency and still have the transparent color dark enough to not detract from the art. When I am nearly done with the image and almost ready to test it for any stray transparent pixels, I set them to a bright purple {red:255, green:0, blue:255}, as bright purple is not usually found

in my art images, and it really stands out. It does not matter which color you use, as long as the image does not contain that particular color as part of its imagery. In a 2D graphics application, there is typically a need to composite images, and so this leads to how to handle a transparent blit.

For the College for Kids program that I taught a few years ago I had put together a small program that reinforced the need for computer games to have foreign language support. This particular game was called "Monster Punch." A language would be selected and various living fruit would drop down from the top of the screen with their eyes moving around and pass out of view at the bottom of the screen. After all the fruit had fallen, the display would snap to a view of a blender, at which point all the fruit would be blended (while screaming) into a monster punch. Okay, maybe I am a little warped, but you should have been able to figure that out by now!

Copy Blit

The following sprite imagery is that of a copy blit, where a rectangular image is copied to the destination, overwriting any overlapped pixel.

Figure 6-4: Monster Punch — a copy blit of a strawberry image on the right into the blender on the left

Using efficiently optimized code, up to 8 bytes at a time can be copied with 64-bit access, which corresponds to simultaneously writing eight 8-bit pixels, four 16-bit pixels, almost three 24-bit pixels, or only two 32-bit pixels. With 128 bits, up to 16 bytes can be accessed, thus 16 8-bit pixels, eight 16-bit pixels, slightly over five 24-bit pixels, or only four 32-bit pixels.

Transparent Blit

As the following sprite image shows, all pixels from the source that match the transparent color are not copied, thus causing the sprite to be seamlessly pasted into the background.

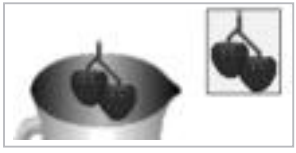


Figure 6-5: Monster Punch — a transparent blit of a strawberry image on the right into the blender on the left

Normally when dealing with transparencies, only one pixel at a time can be tested to detect if it is transparent or not, and it winds up introducing inefficiencies, such as branch mispredictions, but that is where the following sample comes in handy.

Graphics 101 — Blit (MMX)

The following code is a sample of a transparent blit, where a scan line of a count of ecx 8-bit bytes is copied from one graphic source row [esi] to a destination graphic row [edi] one pixel at a time.

Graphics Engine — Sprite Layered

This eight 8-bit transparent pixel copy using MMX code. Note that there is only one branch loop every eighth pixel.

```
tcolor qword 03f3f3f3f3f3f3fh ;03fh = transparent pixel

; esi=source  edi=destination  ecx=# of qwords

     movq    mm7,tcolor    ; Get replicated transparency

$T0: movq    mm5,[esi]     ; Get 8 source pixels
     movq    mm4,[edi]     ; Get background
     movq    mm6,mm5       ; Copy 8 source pixels

; Compare each pixel's color to transparency color and if
; a match, set each pixel in the mask to FF else 00.

     pcmpeqb mm5,mm7       ; Create masks for transparency

     add     esi,8         ; Adjust source pointer

;   Only keep the pixels in the destination that correspond
;   to the transparent pixels of the source.

     pand    mm4,mm5

; Using the same mask, flip it, then AND it with the
; source pixels keeping the non-transparent pixels.

     pandn   mm5,mm6       ; erase transparent pixels

; or the destination pixels with the source pixels.

     por     mm4,mm5       ; blend 8 pixels into art
     movq    [edi],mm4     ; Save new background

     add     edi,8         ; Adjust destination pointer
     dec     ecx           ; any pixels left?
     jne     $T0           ; Loop for eight 8-bit pixels
```

There is no transparency testing/branching, only the masking and blending of data, which makes the process of a transparent blit much faster. These two different blits (copy and transparent) are typically designed for a graphic environment, where in the image below, the background seen on the right is kept in a separate buffer like wallpaper.

Figure 6-6: Transparent copy blit of strawberry sprite and blender image background to achieve composite result of both

The background is copy blit to the working surface, as seen on the left, and then the sprite image is transparent blit in front of it. When the sprite image is animated, the area being changed is "erased" from the working surface by a rectangular copy blit of that area from the background to the working surface. Then the update sprite image has a rectangular area transparent blit in front. This is a layered approach, typically used in a video game that has a number of animated objects moving around the display.

Graphics Engine — Sprite Overlay

Another graphic sprite environment method is when the area under the sprite is remembered in a buffer attached to the sprite before the sprite image is transparent blit. This operation typically occurs simultaneously to reduce the amount of calculation work.

This is typically called an "overlay" method used by windows and some sprite engines. The drawback to this method is that overlapping of sprites needs to be minimized to erase one; all the other intersecting sprites visible above that sprite need to be erased. By replacing the image under the sprite, the list of sprites needs to be traversed, replacing the image area down to where the change needs to occur. After that point, the process is reversed, drawing the sprites back into the scene.

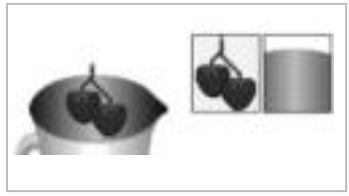


Figure 6-7: The blit of a rectangular blender image to a storage buffer and the transparent blit of a strawberry into the blender. A blit of the saved blender image back into blender effectively erases strawberry.

```
tcolor qword 03f3f3f3f3f3f3fh ;03fh = transparent pixel

; esi=source  edi=destination  ebx=buffer  ecx=# of qwords

     movq    mm7,tcolor    ; Get replicated transparency

$T0: movq    mm5,[esi]     ; Get 8 source pixels
     movq    mm4,[edi]     ; Get 8 background pixels
     movq    mm6,mm5       ; Copy 8 source pixels

; Compare each pixel's color to transparency color and if
; a match, set each pixel in the mask to FF else 00.

     pcmpeqb mm5,mm7       ; Create masks for transparency
     movq    [ebx],mm4     ; Save BGnd in buffer

; Only keep the pixels in the destination that correspond
; to the transparent pixels of the source.

     pand    mm4,mm5

; Using the same mask, flip it with the
; source pixels, keeping the non-transparent pixels.

     pandn   mm5,mm6       ; erase transparent pixels

; or the destination pixels with the source pixels.

     add     ebx,8         ; Adjust buffer pointer
     por     mm4,mm5       ; blend 8 pixels into art
     add     esi,8         ; Adjust source pointer
     movq    [edi],mm4     ; Save new background
     add     edi,8         ; Adjust destination pointer

     dec     ecx           ; any pixels left?
     jne     $T0           ; Loop for eight 8-bit pixels
```

Exercises

1. What is a half adder?
2. Make a NOT gate using a) XOR gate, b) NAND gate, and c) NOR gate.
3. Write an algorithm so that with 32 bits of source data A and B use AND masks in parallel to AND bits 31...24, XOR bits 23...16, and OR bits 15...0.
4. Write a software algorithm for the following equation:

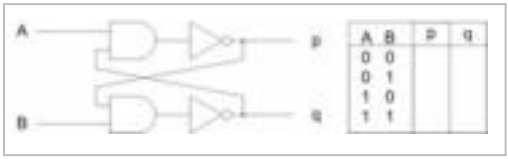
   $D=(-1-(A \wedge B)) \wedge (A \vee B)$

   What does the equation do?
5. Draw an XOR circuit using only OR and AND gates and a subtraction.

6. Make a 128-bit XOR using only bit-wise OR and AND and a subtraction.
7. Write code to change the value of tri-state data. Note the series of states is a repeating pattern {0, 1, 2, 0, 1, 2,...}. The starting state is State #0.

State	0	1	2	0...
Value	0x34	0x8A	0xE5	0x34

Extra credit:



This cross-coupled SR flip-flop will be more familiar to those of you with electronics experience. Using *A*=0 and B=0 as the starting state, finish the logic table.

How does this circuit function? What are the behaviors of p and q if the data inputs of *A* and *B* are changed to a new state? Try all 12 possibilities. How would you code it?

Chapter 7
Bit Wrangling

Think of this chapter as a continuation of the previous chapter, "Bit Mangling."

Bit wrangling actually occurs with the logical and arithmetic shifting and rotation of bits within each parallel bit range of packed bits. Just as in the scalar point of view of a similar general-purpose CPU instruction, the bits can be used for masking, base two multiplication and division, and other functionalities.

It must be reiterated that it is necessary to watch the alignment of your data objects in memory very closely. It takes extra overhead to adjust the memory into an aligned state, and it is a lot more efficient to ensure that they are aligned in the first place. Your code will be smaller and faster! This will be made obvious by the sample code included in this chapter.

CD Workbench Files: /Bench/*architecture*/chap07/*project*/*platform*

	architecture		*project*	*platform*
PowerPC	/vmp_ppc/	**Shift/Rotations**	**/prot/**	/mac9cw
X86	/vmp_x86/			/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Parallel Shift (Logical) Left

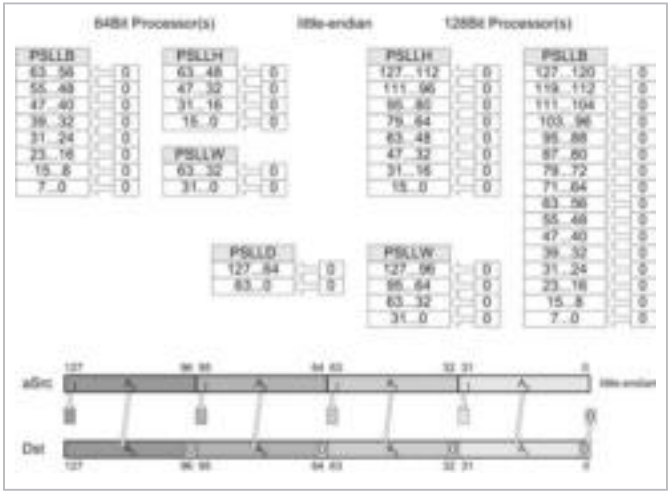


Figure 7-1: Miscellaneous examples of data types being shifted to the left by one bit

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vsl(b/h/w) *Dst*, *aSrc*, *count* *vD=vec_sl(vA, vB)*	[Un]signed	128
MMX	psll(w/d/q) *mmDst*, c*ount(#/mm/m64)*	[Un]signed	64
SSE2	psll(w/d/q) *xmmDst*, *count(#/xmm/m128)*	[Un]signed	128
MMI	psll(h/w) *Dst*, *aSrc*, *count(#)*	[Un]signed	128
	psllvw *Dst*, *aSrc*, *bSrc*	[Un]signed	128

These Multimedia and SIMD Extension instructions are parallel operations that logically left shift each of the data bit blocks by a *count* of bits. Depending on the processor, the instruction block sizes of 8, 16, 32, or 64 bits can be shifted by the specified *count*.

This is a multiplier (2^n) by shifting a 0 into the LSB of each packed value of the source *aSrc (xmmSrc)*, causing all bits to be shifted to the left and the MSB of each packed value to be lost and the result stored in the destination *Dst (xmmDst)*. There is typically no carry flag to save the bit. If the *count* indicating the number of bits to shift is more than (packed bits – 1) — 7 for (**B**ytes), 15 for (**H**alf-words), 31 for (**W**ords), or 63 for (**D**ouble words) — the destination will be typically set to a

value of zero. This is most effective when used in an integer math function where multiple numbers need to be multiplied by 2^n concurrently.

Pseudo Vec

Although logical left, right, or arithmetic shifting is supported by almost all processors, parallel shifting is not. The same parallel effect can be simulated in those particular cases of non-SIMD supporting processors. The following C code demonstrates it by concurrently shifting four packed 8-bit values left. This uses the same method demonstrated in Chapter 4, "Vector Methodologies." All 32 bits are shifted to the left in unison, but as the remaining bits become corrupted by the adjacent byte, a logical AND using a mask in the table lookup forces those vacated bits to zero.

```
#define eMASKFE 0xfefefefe       // Mask FE

uint32 val;

val = (val << 1) & eMASKFE;
```

As you saw in the previous example, an absolute shift value is the most simplistic way to go, as it is merely a matter of shifting the *n* number of bits and masking with the appropriate value.

Table 7-1: 8-bit mask for stripping bits in conjunction with shifting data (0...7) bits to the left or right

8-bit Shift	0	1	2	3	4	5	6	7
Left	FF	FE	FC	F8	F0	E0	C0	80
Right	FF	7F	3F	1F	0F	07	03	01

If the shift factor is unknown, the algorithm required becomes more complicated, as a table lookup is needed for the masking value.

```
uint32 llMaskBD[] = {  //  Left Shift 32bit 4x8bit mask
       0xffffffff, 0xfefefefe, 0xfcfcfcfc, 0xf8f8f8f8,
       0xf0f0f0f0, 0xe0e0e0e0, 0xc0c0c0c0, 0x80808080 };

uint32 rrMaskBD[] = {  //  Right Shift 32bit 4x8bit mask
       0xffffffff, 0x7f7f7f7f, 0x3f3f3f3f, 0x1f1f1f1f,
       0x0f0f0f0f, 0x07070707, 0x03030303, 0x01010101 };
```

Of course, a 16-bit shift would require a different mask.

Table 7-2: 16-bit mask for stripping bits in conjunction with shifting data (0...15) bits to the left or right

16-bit Shift	0	1	2	3	4	5	6	7
Left	FFFF	FFFE	FFFC	FFF8	FFF0	FFE0	FFC0	FF80
Right	FFFF	7FFF	3FFF	1FFF	0FFF	07FF	03FF	01FF
16-bit Shift	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Left	FF00	FE00	FC00	F800	F000	E000	C000	8000
Right	00FF	007F	003F	001F	000F	0007	0003	0001

```
uint32 llMaskHD[] = {  //  Left Shift 32bit 2x16bit mask
       0xffffffff, 0xfffefffe, 0xfffcfffc, 0xfff8fff8,
       0xfff0fff0, 0xffe0ffe0, 0xffc0ffc0, 0xff80ff80,
       0xff00ff00, 0xfe00fe00, 0xfc00fc00, 0xf800f800,
       0xf000f000, 0xe000e000, 0xc000c000, 0x80008000 };

uint32 rrMaskHD[] = {  //  Right Shift 32bit 2x16bit mask
       0xffffffff, 0x7fff7fff, 0x3fff3fff, 0x1fff1fff,
       0x0fff0fff, 0x07ff07ff, 0x03ff03ff, 0x01ff01ff,
       0x00ff00ff, 0x007f007f, 0x003f003f, 0x001f001f,
       0x000f000f, 0x00070007, 0x00030003, 0x00010001 };
```

For 32-bit shifting, no masks are needed, provided the full size of the 32-bit data registers is used, which also makes for simpler code.

So keeping it simple, first examine four 8-bit bytes being shifted simultaneously. Similar to the processors, the shift count needs to be truncated to a value of *n*–1, thus between 0...7 in this particular case. Different processors have different results, especially with the various C compilers being a feature of the C language. For example, a shift of an 8-bit byte by a value of nine on some compilers would result in a zero, some a one, and some — if directly transposing to assembly instructions — would tend to ignore the extraneous bits. For the 8-bit value, only the three least significant bits would be used. Thus, a shift value of nine would effectively be a shift by one, since it meets and/or exceeds the data width of the eight bits and only a shift value of {0...*n*–1}, thus {0...7} is allowed.

This also helps to prevent an out-of-bounds memory reference in regards to the mask lookup! The value is then shifted by the adjusted *count* and then a logical AND with the mask. This effectively processes all four values simultaneously, as mentioned!

Packed Shift Left 4x8-bit

```
uint32 vmp_psllBD( uint32 val, uint count )
{
    count &= (8-1);       // Clip count to data bit size -1
    return (val << count) & llMaskBD[count];
}
```

So looking at this from a 128-bit point of view, this function handles a logical left shift of 16 8-bit bytes simultaneously by a shift value! Of course, if it was a simple count of one, as most processors support, the function becomes much simpler since only a fixed mask is required along with the shift of a single bit.

Packed Shift Left Logical 16x8-bit by n:{0...7}

Listing 7-1: \chap07\prot\PRot.cpp

```
void vmp_psllB( uint8 * pbD, uint8 * pbA, uint count )
{
  uint32 msk, *pD, *pA;

  ASSERT_PTR4(pbA);
  ASSERT_PTR4(pbD);

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;

  msk = llMaskBD[count];
  count &= (8-1);        // Clip count to data bit size -1

  *(pD+0) = (*(pA+0) & msk) << count;
  *(pD+1) = (*(pA+1) & msk) << count;
  *(pD+2) = (*(pA+2) & msk) << count;
  *(pD+3) = (*(pA+3) & msk) << count;
}
```

Packed Shift Left Logical 8x16-bit by n:{0...15}

A shift of an 8x16 bit is just as simple. Only the count truncation as well as the mask would need to differ.

```
void vmp_psllH( uint16 * phD, uint16 * phA, uint count )
        :
        :
  msk = llMaskHD[count];
  count &= (16-1);       // Clip count to data bit size-1
```

Packed Shift Left Logical 4x32-bit by n:{0...31}

The same applies for a 4x32 bit. Only since the register size of the UINT is 32 bits, no mask is needed!

```
void vmp_psllW( uint32 * pD, uint32 * pA, uint count )
        :
        :
  count &= (32-1);       // Clip count to data bit size-1
```

Pseudo Vec (X86)

The following is an example needed for the X86 processor because it supports packed 16, 32, and 64 bits but not an 8-bit logical shift. It uses a similar table of masks for the zero-bit fill, as viewed previously.

```
llMaskBQ dq 0ffffffffffffffh, 0fefefefefefefefeh,
         dq 0fcfcfcfcfcfcfcfch, 0f8f8f8f8f8f8f8f8h,
         dq 0f0f0f0f0f0f0f0f0h, 0e0e0e0e0e0e0e0e0h,
         dq 0c0c0c0c0c0c0c0c0h, 08080808080808080h,
```

As an MMX register is only 64 bits, the 128-bit value is handled using register pairs.

vmp_psllB (MMX) 16x8-bit Vector

Listing 7-2: vmp_x86\chap07\prot\psllX86M.asm

```
mov    ecx,dword ptr count  ; # of bits to shift
mov    edx,pbD
mov    eax,pbA
and    ecx,8-1              ; Clip count to 0...7 bits

movq  mm0,[eax+0]           ; Read data
movq  mm1,[eax+8]
movd  mm2,ecx
       ; mm0=64bits mm1=64bits  mm2=# of bits to shift
psllq mm0,mm2               ; val << count
psllq mm1,mm2
pand  mm0,llMaskBQ[ecx*8]   ; Strip lower bits
pand  mm1,llMaskBQ[ecx*8]

movq  [edx+0],mm0           ; Write data
movq  [edx+8],mm1
```

Of course with this instruction, if only a shift by one is needed, the *pand* and *psllq* can be replaced with only a *paddb*. The MMX instructions support a byte add, and (A<<1) is equivalent to (A+A).

So replace this:

```
psllq mm0,1                 ; val << 1
psllq mm1,1
pand  mm0,llMaskBQ[1*8]     ; Strip lower bit
pand  mm1,llMaskBQ[1*8]
```

...with this:

```
paddb mm0,mm0
paddb mm1,mm1
```

One last thing to note is the actual operation. There is no difference between a pre-mask and post-mask. Only the mask!

((A&7F)<<1) is equivalent to ((A<<1)&FE)

```
   AAAAAAAA BBBBBBBB          AAAAAAAA BBBBBBBB C
 & 01111111 01111111        << 1 ________________
   0AAAAAAA 0BBBBBBB            A AAAAAAAB BBBBBBBC
<< 1 ________________         & 0 11111110 11111110
   AAAAAAA0 BBBBBBB0            AAAAAAA0 BBBBBBB0
```

If you compare the left and right traces, you should see that they have the same result.

As mentioned earlier, the 16- and 32-bit data sizes are supported by a left shift; no mask is needed, only the appropriate count truncation and instruction.

vmp_psllH (MMX) 8x16-bit Vector

```
psllw  mm0,mm2        ; val << count
psllw  mm1,mm2
```

vmp_psllW (MMX) 4x32-bit Vector

```
pslld  mm0,mm2        ; val << count
pslld  mm1,mm2
```

Pseudo Vec (PowerPC)

The following is an example needed for the PowerPC processor without AltiVec, as it does not support SIMD shifting and is extremely similar to the X86 sample. Only the instruction and registers differ.

vmp_psllB (PowerPC) SLL 16x8-bit by n:{0...7}

Listing 7- 3: vmp_ppc\chap07\prot\psllPPC.cpp

```
unsigned int register a0, a1, a2, a3, eMsk, rTbl;

__asm {
  clrlwi r5,r5,29     // Clear upper 29 bits (0...27)
  lwz  rTbl,rrMaskBD

  slwi eMsk,r5,2      // x4
  add  rTbl,rTbl,eMsk // index into mask table
  lwz  eMsk,0(rTbl)

  lwz  a0,0(r4)       // Read A  Bits {0...31}
  lwz  a1,4(r4)
  lwz  a2,8(r4)
  lwz  a3,12(r4)      // Read     "    {96...127}

  and  a0,a0,eMsk     // AND mask
  and  a1,a1,eMsk
  and  a2,a2,eMsk
  and  a3,a3,eMsk

  slw  a0,a0,r5       // Shift left
  slw  a1,a1,r5
  slw  a2,a2,r5
```

```
  slw  a3,a3,r5

  stw  a0,0(r3)       // Write D Bits {0...31}
  stw  a1,4(r3)
  stw  a2,8(r3)
  stw  a3,12(r3)      //       "       {96...127}
}
```

vmp_psllH (PowerPC) SLL 8x16-bit by n:{0...15}

For the 16-bit sizes, only the number of bits cleared as well as the masking table will be modified. All the other supporting code is identical.

```
clrlwi r5,r5,28  // Clear upper 28 bits thus (0...27)
lwz  rTbl,rrMaskHD
```

vmp_psllW (PowerPC) SLL 4x32-bit by n:{0...31}

With the 32-bit form, it is only a matter of the shift, as no mask is required, so it is virtually the same code.

The AltiVec instruction set with its 128-bit registers vastly accelerates code and minimizes data manipulation. Compare the previous PowerPC assembly to the following AltiVec library call. As was discussed earlier, you will not be burdened with the excess baggage necessary for non-aligned support.

The AltiVec has an interesting feature. Each element requires its own shift value, which allows multiple shifting to occur in parallel but with different amounts. This is non-standard, so in our cross-platform function code we need to use a table lookup to allow all shifts to be uniform. The tables are too unwieldy to display here, but viewing just their first and last entries should give you the idea. These are also located on the companion CD.

```
vector unsigned char V8Shift[] = {
  { 0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0 },
                :
  { 7,7,7,7,7,7,7,7, 7,7,7,7,7,7,7,7 }};

vector unsigned short V16Shift[] = {
  { 0,0,0,0,0,0,0,0 },
            :
  { 15,15,15,15,15,15,15,15 }};

vector unsigned int V32Shift[] = {
  { 0,0,0,0 },
       :
  { 31,31,31,31 }};
```

vmp_psllB (AltiVec) Aligned — 16x8-bit by n:{0...7}

Listing 7-4: vmp_ppc\chap07\prot\PRotAltivec.cpp

```
void vmp_psllB( uint8*pbD, uint8 *pbA, uint count)
{
  ASSERT_PTR16(pbA);
  ASSERT_PTR16(pbD);

  count &= (8-1);  // Clip count to data bit size-1
  *(vector unsigned char *)pbD =
          vec_sl( (*(vector unsigned char *)pbA),
                  V8Shift[count] );
}
```

vmp_psllH (AltiVec) Aligned — 8x16-bit by n:{0...15}

Listing 7-5: vmp_ppc\chap07\prot\PRotAltivec.cpp

```
void vmp_psllH( uint16*phD, uint16 *phA, uint count)
{
  ASSERT_PTR16(phA);
  ASSERT_PTR16(phD);

  count &= (16-1);  // Clip count to data bit size-1
  *(vector unsigned short *)pbD =
          vec_sl( (*(vector unsigned short *)pbA),
                  V16Shift[count] );
}
```

vmp_psllW (AltiVec) Aligned — 4x32-bit by n:{0...31}

Listing 7-6: vmp_ppc\chap07\prot\PRotAltivec.cpp

```
void vmp_psllW( uint32*phD, uint32 *phA, uint count)
{
  ASSERT_PTR16(pwA);
  ASSERT_PTR16(pwD);

  count &= (32-1);  // Clip count to data bit size-1
  *(vector unsigned int *)pbD =
          vec_sl( (*(vector unsigned int *)pbA),
                  V32Shift[count] );
}
```

Pseudo Vec (MMI)

The shell used by shifting and rotation is not much different than that of the Boolean operations. If a fixed amount of shifting is needed for a half-word or word, an immediate shift is no problem.

vmp_psll1W (MMI) 4x32-bit Vector by 1 (or Immediate)

Listing 7-7: vmp_mips\chap07\prot\psllMMI.s

```
lq    t1, 0(a1)          // pwA Load 4x32bit
LDELAY                   // NOP - Load Delay Slot

psllw t0,t1,1            // D = A << 1

sq    t0, 0(a0)          // pwD Save 4x32bit result
```

Replacing the *psllw* instruction with *psllh* allows 16 bits to be shifted instead of 32 bits. The value of *one* can also be replaced with a larger amount for a higher number of bits to be shifted.

vmp_psll1B (MMI) 16x8-bit Vector by 1 (or Immediate)

Shifting an 8-bit byte, on the other hand, is more complex, as it is not really supported. For a shift left by one bit, a simple trick of using addition can be utilized:

```
paddb t0,t1,t1        // D = A << 1
```

...merely adding it to itself ($A+A = 2A$). A higher shift count would entail either a series of additions or something a bit more complex.

vmp_psllB (MMI) 16x8-bit Vector by n:{0...7}

The instruction *psllvw*, which shifts 32 bits left by a variable amount has an unusual method of shifting data. It only shifts the even words bits {31...0} and {95...64) and sign extends the result. This is not very friendly when trying to simulate other unsupported instructions. Therefore, the even words are shifted, and then the odd and even are shuffled and shifted again and blended together. A mask is also used to protect the odd and even bits from contaminating each other. In the code comments, notice the movement of words from field to field in preparation for the shifting operations. Then notice the masking operation.

Listing 7-8: vmp_mips\chap07\prot\psllMMI.s

```
lq    t1, 0(a1)       // {15...0} pbA
and   t2,a2,8-1       // 00000111b Clip count to (8-1)

pcpyld t2,t2,t2       // {0 count 0 count}

psllvw t0,t1,t2       // {# 2 # 0}  A << count

pexcw  t1,t1          // {3 1 2 0}
pexcw  t0,t0          // {# # 2 0}
```

```
pextuw t1,t1,t1       // {3 3 1 1}
la     t3,llMaskBO    // Left 16x8bit mask base

psllvw t1,t1,t2       // {# 3 # 1}  D = A << count

sll    t2,t2,4        // x16 Table index
pexcw  t1,t1          // {# # 3 1}
add    t3,t3,t2       // Adjust for index

pextlw t0,t1,t0       // {3 2 1 0}
lq     t3,0(t3)       // Left 16x8bit mask

pand   t0,t0,t3       // Strip right (unwanted) bits

sq     t0, 0(a0)      // pbD
```

The 16-bit half-word data size is not supported either, but a slight alteration of the previous 8-bit code sample by changing it to 16 bit and using *llMaskHO* instead of *llMaskBO* will achieve the desired result.

vmp_psllW (MMI) 4x32-bit Vector by n:{0...31}

As mentioned earlier, only the even 32-bit words are affected, and so the data must be swizzled to process the odd words. Also, note that the masks are no longer needed, as the data size is the needed 32 bits!

Listing 7-9: vmp_mips\chap07\prot\psllMMI.s

```
lq     t1, 0(a1)      // {3 2 1 0} pwA
and    t2,a2,32-1     // 00011111b Clip count to (32-1)
pcpyld t2,t2,t2       // {0 count 0 count}

psllvw t0,t1,t2       // {# 2 # 0}  D = A << count

pexcw  t1,t1          // {3 1 2 0}
pexcw  t0,t0          // {# # 2 0}
pextuw t1,t1,t1       // {3 3 1 1}

psllvw t1,t1,t2       // {# 3 # 1}  D = A << count

pexcw  t1,t1          // {# # 3 1}
pextlw t0,t1,t0       // {3 2 1 0}

sq     t0, 0(a0)      // pwD
```

For specific information, see a MIPS C790, PS2 Linux Kit, or PS2 devTool manual.

Parallel Shift (Logical) Right

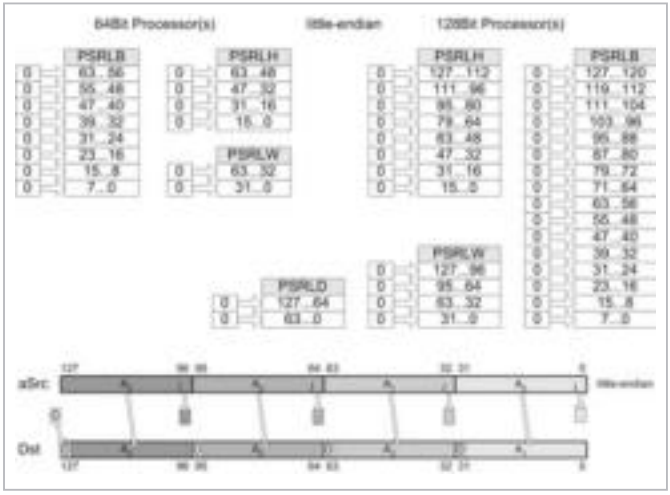


Figure 7-2: Miscellaneous examples of data types being logical shifted to the right by one bit

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vsr(b/h/w) *Dst*, *aSrc*, *count* *vD=vec_sr(vA, vB)*	[Un]signed	128
MMX	psrl(w/d/q) *mmDst*, *count(#/mm/m64)*	[Un]signed	64
SSE2	psrl(w/d/q) *xmmDst*, *count(#/xmm/m128)*	[Un]signed	128
MMI	psrl(h/w) *Dst*, *aSrc*, *count(#)*	[Un]signed	128
	psrlvw *Dst*, *aSrc*, *bSrc*	[Un]signed	128

These Multimedia and SIMD Extension instructions are parallel operations that logically right shift each of the data bit blocks by a *count* of bits. Depending upon the processor and instruction, block sizes of 8, 16, 32, or 64 bits can be shifted by the specified *count*.

This is a divisor (2^n) of unsigned values by shifting a 0 into the MSB of each packed value of the source *aSrc (xmmSrc)* causing all bits to be shifted to the right and the LSB of each packed value being lost and the result stored in the destination *Dst (xmmDst)*. There is typically no carry flag to save the bit. If the *count* indicating the number of bits to shift is more than (packed bits – 1) — 7 for (**B**ytes), 15 for (**H**alfwords), 31 for (**W**ords), or 63 for (**D**ouble words) — the destination will

be typically set to a value of zero. This is most effective when used in an integer math function where multiple unsigned numbers need to be divided by 2^n concurrently.

Pseudo Vec

This C code simulating the functionality is almost identical to the instruction Parallel Shift (Logical) Left, previously discussed in this chapter, except in this case, a different mask and a shift to the right is used. This should look very similar to you, as it was previously reviewed in SLL; only the bold areas should be different.

Packed Shift Right Logical 16x8-bit by n:{0...7}

Listing 7-10: \chap07\prot\PRot.cpp

```
void vmp_psrlB( uint8 * pbD, uint8 * pbA, uint count )
{
  uint32 msk, *pD, *pA;

  pD=(uint32*) pvD;
  pA=(uint32*) pvA;

  count &= (8-1);        // Clip count to data bit size-1
  msk = llMaskBD[count];

  *(pD+0) = (*(pA+0) & msk) >> count;
  *(pD+1) = (*(pA+1) & msk) >> count;
  *(pD+2) = (*(pA+2) & msk) >> count;
  *(pD+3) = (*(pA+3) & msk) >> count;
}
```

So with that in mind, note the following similarities between left and right logical shifting for the various processor instructions.

Table 7-3: Instruction substitution table to convert a previous SLL (Shift Left Logical) instruction into an SRL (Shift Right Logical), as well as masks and their complements

Instructions		Masks	
SLL	**SRL**	**SLL**	**SRL**
psllq	psrlq	rrMaskBQ	llMaskBQ
psllh	psrlh		
psllw	psrlw		
pslld	psrld	llMaskBO	rrMaskBO
psllvw	psrlvw	llMaskHO	rrMaskHO

A shift logical would use a mask with a one's complement to that used by the right shift, and the direction of the shift would be inverted as well.

Parallel Shift (Arithmetic) Right

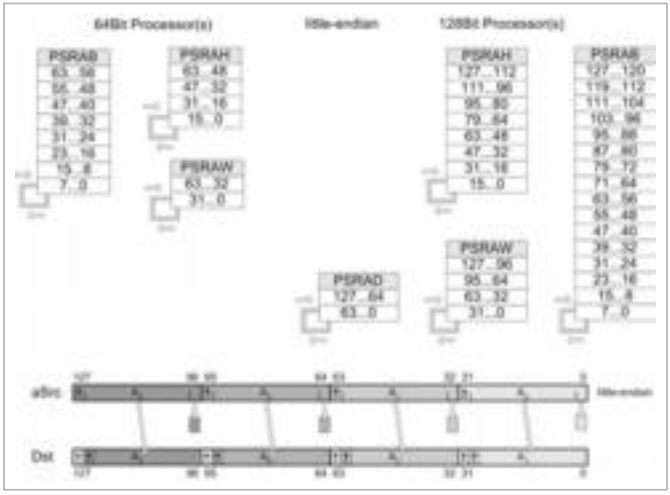


Figure 7-3: Miscellaneous examples of data types being arithmetically shifted to the right by one bit

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vsra(b/h/w) *Dst*, *aSrc*, *count* *vD=vec_sra(vA, vB)*	Signed	128
MMX	psra(w/d/q) *mmDst*, *count(#/mm/m64)*	Signed	64
SSE2	psra(w/d/q) *xmmDst*, *count(#/xmm/m128)*	Signed	128
MMI	psra(h/w) *Dst*, *aSrc*, *count(#)*	[Un]signed	128
MMI	psravw *Dst*, *aSrc*, *bSrc*	[Un]signed	128

These Multimedia and SIMD Extension instructions are parallel operations that arithmetically right shift each of the data bit blocks by a *count* of bits. Depending on the processor and the instruction, block sizes of 8, 16, 32, or 64 bits can be shifted by the specified *count*.

This is a divisor (2^n) by shifting the MSB, a "sticky" bit, of each signed packed value of the source *aSrc (xmmSrc)*, causing all bits to be shifted to the right and the LSB of each packed value being lost and the result stored in the destination *Dst (xmmDst)*. There is typically no carry flag to save the bit. If the *count* indicating the number of bits to shift is more than (packed bits – 1) — 7 for (**B**ytes), 15 for (**H**alf-words), 31 for (**W**ords), or 63 for (**D**ouble words) — then the destination will be typically set to the same value as the MSB, a value of zero. This is most

effective when used in an integer math function where multiple signed numbers need to be divided by 2^n concurrently.

Each data block essentially has a "sticky MSB," which is great for use in the generating of masks. The MSB is also the sign bit, which indicates a negative value when set. When shifted to the right, it leaves a trail of the same bit. Examining the following two samples, the first positive and the second negative, note the leftmost bits during a shift.

```
00000000000000001010010110100101b 00000a5a5h (42405)
00000000000000001010010110100101b 0000052d2h (21202)
00000000000000000101001011010010b 000002969h (10601)
00000000000000000010100101101001b 0000014b4h (5300)

10000000000000001010010110100101b 08000a5a5h (–2147441243)
11000000000000001010010110100101b 0c00052d2h (–1073720622)
11100000000000000101001011010010b 0e000a5a5h (–536860311)
11110000000000000010100101101001b 0f000a5a5h (–268430156)
```

The shifting of the MSB by the width of the packed bits less one will generate a mask of all ones if the value is negative and all zeros if the number is positive. In a similar method, by using the logical right shift and shifting the size of the data width less one, all bits will contain a zero and the LSB will contain a one if it is a negative number and zero if positive {1, 0}.

```
msk = ( val > 32 ) 0x00000000 : 0xFFFFFFFF;

val –= (32+1);              // ...33 (+)   ,  32...31... (–)
msk = ((int)val) >> 31;     // 0x00000000  0xFFFFFFFF
```

If inverse logic is required, then using a logical NOT would invert the bit mask. An alternative method of subtracting by one would return a {0,–1}, effectively obtaining the same solution. The interesting thing is one utilizes positive and one negative logic.

```
msk = ( val > 32 ) 0xFFFFFFFF : 0x00000000;

val –= (32+1);              // ...33 (+)   ,  32...31... (–)
msk = ((int)val) >> 31;     // 0x00000000  0xFFFFFFFF
msk ^= 0xFFFFFFFF;          // msk = NOT msk
```

From a scalar point of view, this is a very powerful instruction. When used in a parallel process, each set of packed bits is handled separately.

Pseudo Vec

Pseudo vectors do not work well with this instruction due to the nature of the sticky bit. The technique demonstrated in a previous section of this chapter related to left shift discussed emulated vector processing by shifting followed by a logical AND, but it does not work with this instruction. Even though a logical AND can be used here, the question arises: Which bit is placed into those vacated bit positions, a zero or a one? Each pseudo MSB that contains that bit gets shifted downward, but only the true MSB is sticky. Note that in the following 32-bit example, the MSB sticks and gets shifted for each set of 8-bit packed bits, but the signed bit (sb) merely gets shifted along. The underlined bits are merely garbage bits from the adjacent set of packed bits that are carried along by the occurring shifts. Those can be masked out with the AND to place a zero in those bits, and a logical OR can place a one in those bits.

```
    ↓ MSB
    10000000 00000000 10100101 10100101b   08000a5a5h
    11000000 00000000 01010010 11010010b   0c00052d2h
    11100000 00000000 00101001 01101001b   0e000a5a5h
    ↑ ↑ ↑sticky ↑sb      ↑sb      ↑sb

AND 11111111 00011111 00011111 00011111b   01F1F1F1Fh Mask
OR  00000000 11100000 11100000 11100000b   0E0E0E0E0h
```

However, although the most significant packed bits are correct, the others are wrong, so how do they get corrected? How does the significant bit of each packed bit select the correct bit pattern? There lies the problem. By the time a series of loop code is executed to resolve it, one might as well just handle each set of packed bits as a scalar. Thus, pseudo vector processing does not work well in this case! But the following are examples of how to do it anyway. The following 4x8-bit parallel arithmetic shift uses C and a series of logical and arithmetic shifting and masking.

I refer to the following method of 4x8-bit arithmetic shift as the ratchet method (socket wrench tool; tighten, swing, repeat). It is a matter of using the normal arithmetic shift by shifting the data element into position, performing the result, and then shifting it back out. At this point, you are probably wondering why you should bother with the overhead due to the shift and unshift that has to occur. But what about those processors that do not have a smaller data size and handle only that of an integer? In those particular cases, the smaller data size would have to be converted to the larger sized integer, arithmetically shifted, and then converted back to a smaller data type.

Packed Shift Right Arithmetic 4x8-bit by n:{0...7}

```
int32 psraBD( int32 val, uint count )
{
    count &= (8–1);    // Clip count to data bit size –1

 return (((uint32)((val<<24) >> count)) >> 24)
     | ((((uint32)((val<<16) >> count)) >> 16) & 0x0000FF00)
     | ((((uint32)((val<< 8) >> count)) >>  8) & 0x00FF0000)
     | ((val >> count) & 0xFF000000);
}
```

Packed Shift Right Arithmetic 16x8-bit by n:{0...7}

Although the previous code is portable, it is extremely bulky, cumbersome, and slow. This gets exaggerated as in the following 128-bit form, which is very ugly!

Listing 7-11: \chap07\prot\PRot.cpp

```
void vmp_psraB( int8 * pbD, int8 * pbA, uint count )
{
  int32 val, *pA;
  uint32 *pD;

  ASSERT_PTR4(pbA);
  ASSERT_PTR4(pbD);

  pA = (int32*)pbA;
  pD = (uint32*)pbD;

  count &= (8-1);     // Clip count to data bit size -1

  val = *(pA+0);
  *(pD+0) = (((uint32)((val<<24) >> count)) >> 24)
  | ((((uint32)((val<<16) >> count)) >> 16) & 0x0000FF00)
  | ((((uint32)((val<< 8) >> count)) >>  8) & 0x00FF0000)
  | ((val >> count) & 0xFF000000);

  val = *(pA+1);
  *(pD+1) = (((uint32)((val<<24) >> count)) >> 24)
  | ((((uint32)((val<<16) >> count)) >> 16) & 0x0000FF00)
  | ((((uint32)((val<< 8) >> count)) >>  8) & 0x00FF0000)
  | ((val >> count) & 0xFF000000);

  val = *(pA+2);
  *(pD+2) = (((uint32)((val<<24) >> count)) >> 24)
  | ((((uint32)((val<<16) >> count)) >> 16) & 0x0000FF00)
  | ((((uint32)((val<< 8) >> count)) >>  8) & 0x00FF0000)
  | ((val >> count) & 0xFF000000);

  val = *(pA+3);
  *(pD+3) = (((uint32)((val<<24) >> count)) >> 24)
  | ((((uint32)((val<<16) >> count)) >> 16) & 0x0000FF00)
  | ((((uint32)((val<< 8) >> count)) >>  8) & 0x00FF0000)
  | ((val >> count) & 0xFF000000);
}
```

See, I told you it was ugly! But as the bit count of a data element gets larger, the C code gets a little simpler.

Packed Shift Right Arithmetic 8x16-bit by n:{0...15}

Listing 7-12: \chap07\prot\PRot.cpp

```
void vmp_psraH( int16 * phD, int16 * phA, uint count )
{
  int32 val, *pA;
  uint32 *pD;

  ASSERT_PTR4(phA);
  ASSERT_PTR4(phD);

  pA = (int32*)phA;
  pD = (uint32*)phD;

  count &= (16-1);    // Clip count to data bit size -1

  val = *(pA+0);
  *(pD+0) = (((uint32)((val<<16) >> count)) >> 16)
          | ((val >> count) & 0xFFFF0000);
  val = *(pA+1);
  *(pD+1) = (((uint32)((val<<16) >> count)) >> 16)
          | ((val >> count) & 0xFFFF0000);
  val = *(pA+2);
  *(pD+2) = (((uint32)((val<<16) >> count)) >> 16)
          | ((val >> count) & 0xFFFF0000);
  val = *(pA+3);
  *(pD+3) = (((uint32)((val<<16) >> count)) >> 16)
          | ((val >> count) & 0xFFFF0000);
}
```

Packed Shift Right Arithmetic 4x32-bit by n:{0...31}

Finally, there is a normal data element size of 32 bits.

Listing 7-13: \chap07\prot\PRot.cpp

```
void vmp_psraW( int32 * pwD, int32 * pwA, uint count )
{
  ASSERT_PTR4(pwA);
  ASSERT_PTR4(pwD);

  count &= (32-1);    // Clip count to data bit size -1

  *(pwD+0) = (*(pwA+0)) >> count;
  *(pwD+1) = (*(pwA+1)) >> count;
  *(pwD+2) = (*(pwA+2)) >> count;
  *(pwD+3) = (*(pwA+3)) >> count;
}
```

Pseudo Vec (X86)

This gets simpler when just working with assembly. Note that it is not optimized to maintain readability. By taking advantage of the SRA of 16 bit, the 8 bit can be handled by a shift adjustment of the lower 8 bits and then bit blending the two 8-bit halves together.

vmp_psraB (MMX) 16x8-bit Vector

Listing 7-14: vmp_x86\chap07\prot\psraX86M.asm

```
mov   eax,pbA
mov   edx,pbD
mov   ecx,dword ptr count  ; # of bits to shift
and   ecx,8-1              ; Clip count to 0...7 bits
movd  mm4,ecx

movq  mm0,[eax+0]          ; Read data
movq  mm1,[eax+8]
movq  mm2,mm0
movq  mm3,mm1

psllw mm2,8       ; Shift lower byte into sra position
psllw mm3,8
psraw mm0,mm4     ; SRA upper byte(s)
psraw mm1,mm4
psraw mm2,mm4     ; SRA lower byte(s)
psraw mm3,mm4
psrlw mm2,8       ; Shift lower byte(s) back
psrlw mm3,8

pand  mm0,lMaskHQ ; Clear lower debris bits
pand  mm1,lMaskHQ
por   mm0,mm2     ; Blend upper/lower byte(s)
por   mm1,mm3

movq  [edx+0],mm0 ; Write data
movq  [edx+8],mm1
```

vmp_psraH (MMX) 8x16-bit Vector

Now it becomes easy, as MMX supports 16- and 32-bit packed SAR. The following are just like the shift (left/right) logical.

Listing 7-15: vmp_x86\chap07\prot\psraX86M.asm

```
mov   ecx,dword ptr count  ; # of bits to shift
mov   edx,pwD
mov   eax,pwA
and   ecx,16-1             ; Clip count to 0...7 bits

movq  mm0,[eax+0]          ; Read data
movq  mm1,[eax+8]
movd  mm2,ecx
```

```
psraw mm0,mm2              ; val >> count (0...15)
psraw mm1,mm2

movq  [edx+0],mm0          ; Write data
movq  [edx+8],mm1
```

vmp_psraW (MMX) 4x32-bit Vector

For 32 bit, merely substitute *psrad* for *psraw.*

```
psrad mm0,mm2              ; val >> count (0...31)
psrad mm1,mm2
```

Because of the overhead associated with performing a packed 8-bit SAR on an X86, seriously consider bumping up to a 16- or 32-bit data element. If the extra memory is not a concern, it will pay for itself in speed!

Pseudo Vec (PowerPC)

The PowerPC has similar problems to that of the X86 processor, as each data element must be shifted into position, SAR, shifted back out to its original position, and then blended for the final result. Check out the companion CD for more information.

For the AltiVec assembly, there is full support of SRA for 8, 16, and 32 bit, so just substitute the appropriate function in the SLL code.

SLL	SRL	SRA
vec_sl	vec_sr	vec_sra

Pseudo Vec (MIPS)

The packed arithmetic shift, whether by a fixed amount or variable amount, is not much fun on the MIPS processor, except in the case of the natural 32-bit word shown in the following section.

vmp_psra1W (MMI) 4x32-bit Vector by 1 (or Immediate)

Listing 7-16: vmp_mips\chap07\prot\psraMMI.s

```
lq     t1, 0(a1)       // pwA
LDELAY                 // NOP - Load Delay Slot

psraw  t0,t1,1         // D = A >> 1

sq     t0, 0(a0)       // pwD
```

vmp_psraW (MMI) 4x32-bit Vector by n:{0...31}

Just like the other processors, a real arithmetic shift is taken advantage of. However, when a variable shift is used, the data not only is shifted into position, but must also work in even/odd pairs due to its functionality, thus making it a bit more complicated.

Listing 7-17: vmp_mips\chap07\prot\psraMMI.s

```
lq     t1, 0(a1)      // {3 2 1 0} pwA
and    t2,a2,32-1     // 00011111b Clip count to (32-1)
pcpyld t2,t2,t2       // {0 count 0 count}

psravw t0,t1,t2       // {# 2 # 0}  D = A >> count

pexcw  t1,t1          // {3 1 2 0}
pexcw  t0,t0          // {# # 2 0}
pextuw t1,t1,t1       // {3 3 1 1}

psravw t1,t1,t2       // {# 3 # 1}  D = A >> count

pexcw  t1,t1          // {# # 3 1}
pextlw t0,t1,t0       // {3 2 1 0}

sq     t0, 0(a0)      // pwD
```

Okay, I admit it! This is the same shell as the function vmp_psllw, but I am not trying to burn pages by replicating code. It is here for comparison with the following variable shifted code sample.

vmp_psraH (MMI) 8x16-bit Vector by n:{0...15}

This is not too bad, but it is a bit of work to maintain the sticky sign bit. Due to only the even 32-bit data elements being shifted, twice the shifting work must be done for 16-bit data.

Listing 7-18: vmp_mips\chap07\prot\psraMMI.s

```
lq     t1, 0(a1)      // {7...0} phA
and    t2,a2,16-1     // 00001111b Clip count to (16-1)

pcpyld t2,t2,t2       // {0 count 0 count}
       // [7 6 * 5 4 * 3 2 * 1 0]
psllw  t4,t1,16       // {6_ 4_ 2_ 0_}

psravw t0,t1,t2       // {# 5   # 1}  A >> count
psravw t5,t4,t2       // {# 4   # 0}

pexcw  t1,t1          // {7,6  3,2  5,4  1,0}
pexcw  t0,t0          // {#  #  5  1}
pexcw  t4,t4          // {6_ 2_ 4_ 0_}
pexcw  t5,t5          // {#  #  4  0}
pextuw t1,t1,t1       // {7,6  7,6  3,2  3,2}
pextuw t4,t4,t4       // {6#   6#   2#   2#}
```

```
psravw t1,t1,t2       // {# 7  # 3}  A >> count
psravw t4,t4,t2       // {# 6  # 2}  A >> count

pexcw  t1,t1          // {#  #  7  3}
pexcw  t4,t4          // {#  #  6  2}
pextlw t0,t1,t0       // {7# 5# 3# 1#}
pextlw t5,t4,t5       // {6# 4# 2# 0#}
psrlw  t0,t0,16       // {7 5 3 1}
psrlw  t5,t5,16       // {6 4 2 0}
pinteh t0,t0,t5       // {7 6 5 4 3 2 1 0}

sq     t0, 0(a0)      // phD
```

vmp_psraB (MMI) 16x8-bit Vector by n:{0...7}

I personally find this particular function to be horrific, but if the variable functionality is needed, then it needs to be done. But I strongly suggest staying away from variable 8-bit arithmetic shifting! Use an immediate value or a larger data type for the data instead, if at all possible. Keep in mind that only two 32-bit arithmetic shifts occur in parallel; thus, this needs to be done eight times to obtain all 16 shifts. Ouch!

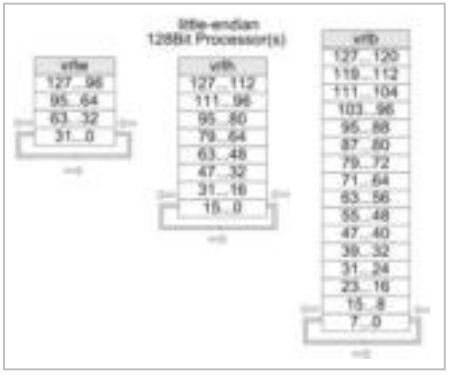
Listing 7-19: vmp_mips\chap07\prot\psraMMI.s

```
lq     t1, 0(a1)      // {15...0} pbA
and    t2,a2,8-1      // 00000111b Clip count to (8-1)

pcpyld t2,t2,t2       // {0 count 0 count}

         //t1=           FEDC [BA98] 7654 [3210]
psravw t3,t1,t2       // {####  B###  ####  3###}

psllw  t4,t1,8        // {EDC_  A98_  654_  210_}
psravw t4,t4,t2       // {####  A###  ####  2###}

psllw  t5,t1,16       // {DC__  98__  54__  10__}
psravw t5,t5,t2       // {####  9###  ####  1###}

psllw  t6,t1,24       // {C___  8___  4___  0___}
psravw t6,t6,t2       // {####  8###  ####  0###}

pextub t7,t3,t4       // {## ##  ## ##  BA ##  ## ##}
pextub t8,t5,t6       // {## ##  ## ##  98 ##  ## ##}
pextlb t3,t3,t4       // {## ##  ## ##  32 ##  ## ##}
pextlb t5,t5,t6       // {## ##  ## ##  10 ##  ## ##}
pextlh t7,t7,t8       // {BA 98  ## ##  ## ##  ## ##}
pextlh t3,t3,t5       // {32 10  ## ##  ## ##  ## ##]
pcpyud t9,t3,t7       // {BA98####   3210####}

    // Second Bank    t1= [FEDC] BA98 [7654] 3210

pexcw  t1,t1          // {FEDC  7654  BA98  3210}
pextuw t1,t1,t1       // {FEDC  FEDC  7654  7654}
psravw t3,t1,t2       // {####  F###  ####  7###}
```

```
psllw  t4,t1,8        // {EDC_  EDC_  654_  654_}
psravw t4,t4,t2       // {####  E###  ####  6###}

psllw  t5,t1,16       // {DC__  DC__  54__  54__}
psravw t5,t5,t2       // {####  D###  ####  5###}

psllw  t6,t1,24       // {C___  C___  4___  4___}
psravw t6,t6,t2       // {####  C###  ####  4###}

pextub t7,t3,t4       // {#,# #,# #,# #,# F,E #,# #,# #,#}
pextub t8,t5,t6       // {#,# #,# #,# #,# D,C #,# #,# #,#}
pextlb t3,t3,t4       // {#,# #,# #,# #,# 7,6 #,# #,# #,#}
pextlb t5,t5,t6       // {#,# #,# #,# #,# 5,4 #,# #,# #,#}
pextlh t7,t7,t8       // {FE DC  ## ##  ## ##  ## ##}
pextlh t3,t3,t5       // {76 54  ## ##  ## ##  ## ##]
pcpyud t8,t3,t7       // {FEDC####   7654####}
pexcw  t9,t9          // {BA98  3210  ####  ####}
pexcw  t8,t8          // {FEDC  7654  ####  ####}
pextuw t0,t8,t9       // {FEDC  BA98  7654  3210}

sq     t0, 0(a0)     // pbD
```

For specific information, see a MIPS C790, PS2 Linux Kit, or PS2 devTool manual.

Rotate Left (or N-Right)



AltiVec	*MMX*	*SSE*	*SSE2*	*3DNow*	*3DMX+*	*MIPS*	*MMI*

AltiVec vrl(b/h/w) *Dst*, *aSrc*, *count* [Un]signed 128
vD=vec_rl(vA, vB)

This Multimedia Extension instruction is a parallel operation that logically rotates each of the data bit blocks by a *count* of bits. Depending on

the processor and the instruction, block sizes of 8, 16, or 32 bits can be shifted by the specified *count*.

This is a multiplier (2^n) by shifting the MSB into the LSB of the destination *aSrc* causing all bits to be shifted to the left and then back into the MSB of the result destination *Dst*. The *count* is typically masked with the (packed bits – 1) — 7 for (**B**ytes), 15 for (**H**alf-words), or 31 for (**W**ords) — to only allow a rotation of (packed bits – 1).

Pseudo Vec

Rotating data is a process of shifting up the lower bits, shifting down the upper bits, masking, and blending by the specified count.

Packed Rotate Left 16x8-bit by n:{0...7}

Listing 7-20: \chap07\prot\PRot.cpp

```
void vmp_prolB( uint8 * pbD, uint8 * pbA, uint count )
{
  uint iCnt;
  uint32 rmsk, lmsk;

  ASSERT_PTR4(pbA);
  ASSERT_PTR4(pbD);

  count &= (8-1);        // Clip count to data bit size -1
  if ( count )
    {
      iCnt = 8-count;
      rmsk = rrMaskBD[count];  // 7f
      lmsk = llMaskBD[iCnt];   // 80

      *(((uint32*)pbD)+0) =
                  ((*(((uint32*)pbA)+0) & rmsk) << count)
                | ((*(((uint32*)pbA)+0) & lmsk) >> iCnt);
      *(((uint32*)pbD)+1) =
                  ((*(((uint32*)pbA)+1) & rmsk) << count)
                | ((*(((uint32*)pbA)+1) & lmsk) >> iCnt);
      *(((uint32*)pbD)+2) =
                  ((*(((uint32*)pbA)+2) & rmsk) << count)
                | ((*(((uint32*)pbA)+2) & lmsk) >> iCnt);
      *(((uint32*)pbD)+3) =
                  ((*(((uint32*)pbA)+3) & rmsk) << count)
                | ((*(((uint32*)pbA)+3) & lmsk) >> iCnt);
    }
  else
    {
      *(((uint32*)pbD)+0) = *(((uint32*)pbA)+0);
      *(((uint32*)pbD)+1) = *(((uint32*)pbA)+1);
      *(((uint32*)pbD)+2) = *(((uint32*)pbA)+2);
      *(((uint32*)pbD)+3) = *(((uint32*)pbA)+3);
    }
}
```

See the CD for other data sizes.

Hint: To rotate right, rotate left by the number of packed bits in the data size less the desired right rotation (DataWidth – *count*).

```
valC = vmp_prol(valA, count);    // rotate left
valE = vmp_prol(valA, 32-count); // rotate right
```

The simpler arithmetic and logical shift instructions are common to most processors, while the more robust rotation instructions are less common but needed by multiple algorithms. A solution would be to use a combination of left and right logical shifts in conjunction with bit masking logic to simulate the same solution.

Pseudo Vec (X86)

There is a problem, however, when dealing with the left or right rotation of eight 8-bit values. Note that the X86 does not support 8-bit packed shifts with MMX instructions, so there is a need for a creative solution. In this left rotation, any data shifted will affect adjacent packed data and thus corrupt it. The solution is to generate a unique mask for both the left and right shifts of each byte and allow the data to be blended for the simulated rotation solution. The same masking tables used for left and right logical shifting are utilized.

Packed rotation is not supported by any of the X86 processors but can be emulated with the use of multiple instructions simulating the desired result, thus functionally becomes a pseudo vector. MMX and SSE can only handle 64 bit, but the SSE2 can handle 128 bit. The following snippet is a 64-bit parallel rotate left and right solution for two 32-bit values. Note that the X86 supports 32-bit packed shifts with MMX instructions, so as data is being shifted, it is automatically replaced with zeros. The results only need to be logical OR'd together to achieve the simulated rotation.

vmp_prolB (MMX) 16x8-bit Vector

Listing 7-21: vmp_x86\chap07\prot\prolX86M.asm

```
mov   ecx,dword ptr count  ; # of bits to shift
mov   edx,pbD
mov   eax,pbA
and   ecx,8-1              ; Clip count to 0...7 bits

movq  mm0,[eax+0]          ; Read data
movq  mm1,[eax+8]
mov   eax,8
movd  mm4,ecx              ; count
```

```
sub   eax,ecx              ; iCnt=8-count

movq  mm2,mm0
movq  mm3,mm1
movd  mm5,eax              ; iCnt

pand  mm0,llMaskBQ[eax*8]  ; val&80
pand  mm1,llMaskBQ[eax*8]
pand  mm2,rrMaskBQ[ecx*8]  ; val&7f
pand  mm3,rrMaskBQ[ecx*8]

psrlq mm0,mm5              ; (val & 80) >> iCnt
psrlq mm1,mm5
psllq mm2,mm4              ; (val & 7f) << count
psllq mm3,mm4

por   mm0,mm2
por   mm1,mm3

movq  [edx+0],mm0          ; Write data
movq  [edx+8],mm1
```

vmp_prolH (MMX) 16x8-bit Vector

The same logic can be applied to the 16-bit data elements by substituting *16* for *8*, *psrlw* for *psrlq*, and *psllw* for *psllq*.

vmp_prolW (MMX) 4x32-bit Vector

The same logic can be applied to the 32-bit data elements by substituting *32* for *8*, *psrld* for *psrlq*, and *pslld* for *psllq*.

Pseudo Vec (PowerPC)

The PowerPC instruction set has a 32-bit rotation, but the 128-bit parallel rotation is only available under AltiVec, so it has to be simulated similar to that of the X86 through a process of shifting, masking, and blending.

vmp_prolB (PowerPC) 16x8-bit Vector

Listing 7-22: vmp_ppc\chap07\prot\prolPPC.cpp

```
  unsigned int register a0, a1, a2, a3, b0, b1, b2, b3, lMsk, rMsk, iCnt;

__asm {
  li     iCnt,8
  clrlwi r5,r5,29      // Clear upper 29 bits so {0...28}
  lwz    b0,rrMaskBD
  lwz    b1,llMaskBD+(8*4)

  slwi   lMsk,r5,2     // x4 32bit table ref.
```

```
  sub    iCnt,iCnt,r5  // iCnt=8-count

  add    rMsk,b0,lMsk  // +(count*4)
  sub    lMsk,b1,lMsk  // -(count*4)

  lwz    rMsk,0(rMsk)  // right mask
  lwz    lMsk,0(lMsk)  // left mask

  lwz    a0,0(r4)
  lwz    a1,4(r4)
  lwz    a2,8(r4)
  lwz    a3,12(r4)

  and    b0,a0,rMsk    // (*(((uint32*)phA)+0) & rmsk)
  and    a0,a0,lMsk    // (*(((uint32*)phA)+0) & lmsk)
  and    b1,a1,rMsk
  and    a1,a1,lMsk
  and    b2,a2,rMsk
  and    a2,a2,lMsk
  and    b3,a3,rMsk
  and    a3,a3,lMsk

  srw    a0,a0,iCnt
  slw    b0,b0,r5
  srw    a1,a1,iCnt
  slw    b1,b1,r5
  srw    a2,a2,iCnt
  slw    b2,b2,r5
  srw    a3,a3,iCnt
  slw    b3,b3,r5

  or     a0,a0,b0      // Blend bits
  or     a1,a1,b1
  or     a2,a2,b2
  or     a3,a3,b3

  stw    a0,0(r3)
  stw    a1,4(r3)
  stw    a2,8(r3)
  stw    a3,12(r3)
}
```

vmp_prolH (PowerPC) 8x16-bit Vector

This function uses a function almost identical to the 8-bit version, except with the modification of the following values.

Listing 7-23: vmp_ppc\chap07\prot\prolPPC.cpp

```
  li     iCnt,16
  clrlwi r5,r5,28      // Clear upper 28 bits so (0...27)
  lwz    b0,rrMaskHD
  lwz    b1,llMaskHD+(8*4)
```

vmp_prolW (PowerPC) 4x32-bit Vector

Since this is in the natural 32-bit integer form, the function is fairly simple. The rotation is handled as a standard 32-bit rotation.

Listing 7-24: vmp_ppc\chap07\prot\prolPPC.cpp

```
  lwz   a0,0(r4)
  lwz   a1,4(r4)
  lwz   a2,8(r4)
  lwz   a3,12(r4)

  rlwnm a0,a0,r5,0,31   // rotate and set ALL bits
  rlwnm a1,a1,r5,0,31
  rlwnm a2,a2,r5,0,31
  rlwnm a3,a3,r5,0,31

  stw   a0,0(r3)
  stw   a1,4(r3)
  stw   a2,8(r3)
  stw   a3,12(r3)
```

vmp_prolB (AltiVec) 16x8-bit Vector — Aligned

For AltiVec, it is merely a call to the standard library.

Listing 7-25: vmp_ppc\chap07\prot\PRotAltivec.cpp

```
void vmp_prolB( uint8 * pbD, uint8 * pbA, uint count )
{
  ASSERT_PTR16(pbA);
  ASSERT_PTR16(pbD);

  count &= (8-1);     // Clip count to data bit size -1

  *(vector unsigned char *)pbD =
       vec_rl( (*(vector unsigned char *)pbA),
               V8Shift[count] );
}
```

Pseudo Vec (MIPS)

The packed left rotation is one of those operations that needs to be simulated on a MIPS processor. It has various data shuffling but only on a data-word basis and not on a bit basis. So in essence, the instruction must be simulated using left and right shifts with data bit blending.

vmp_prol1W (MMI) 4x32-bit Vector by 1 (or Immediate)

Notice that since the data size is 32 bit (merely a left and right shift which shifts zero bits into position), a logical OR is utilized to achieve the desired result.

Listing 7-26: vmp_mips\chap07\prot\prolMMI.s

```
lq     t1, 0(a1)      // pwA
LDELAY                // NOP - Load Delay Slot

psrlw  t0,t1,31       // (A >> 31)
psllw  t1,t1,1        // (A << 1)
por    t0,t0,t1

sq     t0, 0(a0)      // pwD
```

vmp_prol1H (MMI) 8x16-bit Vector by 1 (or Immediate)

A rotation of a 16-bit value is just as simple; merely substitute *psrlh* for *psrlw* and adjust the data size.

```
psrlh  t0,t1,15       // (A >> 15)
psllh  t1,t1,1        // (A << 1)
```

vmp_prol1B (MMI) 16x8-bit Vector by 1 (or Immediate)

By using 8-bit masking, an immediate amount of shifting can be easily handled.

Listing 7-27: vmp_mips\chap07\prot\prolMMI.s

```
la     t2,llMaskB0+(16*7)  // 80
la     t3,rrMaskB0+(16*1)  // 7F
lq     t1, 0(a1)           // phA
lq     t2, 0(t2)           // get 8080...8080 mask
lq     t3, 0(t3)           // get 7F7F...7F7F mask

pand   t0,t1,t2            // 10000000
pand   t1,t1,t3            // 01111111
psrlh  t0,t0,7             //        (A >> 7)
psllh  t1,t1,1             // (A << 1)
por    t0,t0,t1

sq     t0, 0(a0)           // phD
```

vmp_prolW (MMI) 4x32-bit Vector by n:[0...31}

When it has to be simulated, sometimes simulating packed instructions using other packed instructions is impractical. As such, it pays off to use general-purpose registers in conjunction with lower bit count data and generic code instead. Only the 32-bit data size is shown here, as the 16- and 8-bit data sizes require even more complicated code. For more information on those, check out the companion CD.

This first code snippet is that of using generic code:

Listing 7-28: vmp_mips\chap07\prot\prolMMI.s

```
#if 01
  subu   t6,zero,a2
  lw     t1, 0(a1)      // {0} pwA
  lw     t3, 4(a1)      // {1} pwA

  srlv   t0,t1,t6       // -count >>
  sllv   t1,t1,a2       //        << count
  srlv   t2,t3,t6       // -count >>
  sllv   t3,t3,a2       //        << count
  or     t0,t0,t1
  or     t2,t2,t3

  sw     t0, 0(a0)      // pwD
  sw     t2, 4(a0)      // pwD

  lw     t1, 8(a1)      // {2} pwA
  lw     t3,12(a1)      // {3} pwA

  srlv   t0,t1,t6       // -count >>
  sllv   t1,t1,a2       //        << count
  srlv   t2,t3,t6       // -count >>
  sllv   t3,t3,a2       //        << count
  or     t0,t0,t1
  or     t2,t2,t3

  sw     t0, 8(a0)      // pwD
  sw     t2,12(a0)      // pwD
```

This second code snippet uses parallel instruction to simulate a parallel rotation. You will notice the extra length compared to the previous generic code:

```
#else

  lq     t1, 0(a1)      // {3 2 1 0} pwA
  li     t3,32
  and    t2,a2,32-1     // 00011111b Clip count to (32-1)
  sub    t3,t3,t2       // 32-count
  pcpyld t2,t2,t2       // {0 count 0 count}
  pcpyld t3,t3,t3       // {0 32-count 0 32-count}

  psllvw t0,t1,t2       // {# 2 # 0}  A << count
```

```
    psrlvw t4,t1,t3       // {# 2 # 0}  32-count >> A

    pexcw  t1,t1          // {3 1 2 0}
    pexcw  t0,t0          // {# # 2 0} *left bits*
    pexcw  t4,t4          // {# # 2 0} *right bits*
    pextuw t1,t1,t1       // {3 3 1 1}

    psrlvw t5,t1,t3       // {# 3 # 1}  32-count >> A
    psllvw t1,t1,t2       // {# 3 # 1}  A << count

    pexcw  t5,t5          // {# # 3 1} *right bits*
    pexcw  t1,t1          // {# # 3 1} *left bits*
    pextlw t4,t5,t4       // {3 2 1 0} right bits
    pextlw t0,t1,t0       // {3 2 1 0} left bits

        // Re-blend

    sll    t3,t3,4        // x16 Table Index (right)
    sll    t2,t2,4        // x16 Table Index (left)
    la     t1,llMaskW0    // Left 4x32bit mask base
    la     t5,rrMaskW0    // Right
    add    t1,t2          // Adjust for index
    add    t5,t3
    lq     t1,0(t1)       // left 4x32bit mask
    lq     t5,0(t5)       // right

    pand   t0,t0,t1       // retain left bits
    pand   t4,t4,t5       // retain right bits
    por    t0,t0,t4       // blend bits

    sq     t0, 0(a0)      // pwD
#endif
```

So from this observation, it would probably be a good idea to avoid data rotation if possible!

For specific information, see a MIPS C790, PS2 Linux Kit, or PS2 devTool manual.

Secure Hash Algorithm (SHA-1)

Chapter 4, "Vector Methodologies," gave a simple example of organizing a function for vector processing. This problem is toward the more difficult end of the spectrum for a vector solution. An algorithm used primarily in networking but also in video game communications and in some copy protection schemes is the "Secure Hash Standard" (FIPS PUB 180-1), instituted by the National Institute of Standards and Technology. This tends to be a very slow algorithm that uses a 160-bit (20-byte) Message Digest Seed, and by processing a 512-bit (64-byte) block of data generates a new 160-bit Message Digest. Each sequential data block is continuously processed, generating a new 160-bit

Message Digest at the conclusion of each and used as the seed for the next. For more detailed information, search the Internet and see the references section of this book.

The basic algorithm will only be looked at from a superficial point of view, and thus only functional blocks will be utilized. There are basically four algorithmic equations, of which the first equation is used 16 consecutive times, the second four times, the third 20, the fourth 20, and again the third 20 times for a total of 80 calculations.

```
MACRO: EQ#(n, r, s, t, u, v )
EQ0: r+=rol(u,5)+ fn0(n)+val+(((v^t)&s)^t);      s=ror(s,2); //0...15
EQ1: r+=rol(u,5)+ fn1(n)+val+(((v^t)&s)^t);      s=ror(s,2); //16...19
EQ2: r+=rol(u,5)+ fn1(n)+val+(s^v^t);            s=ror(s,2); //20...39, 60...79
EQ3: r+=rol(u,5)+ fn1(n)+val+(((s|v)&t)|(s&v));  s=ror(s,2); //40...59
```

These equations appear extremely similar and thus a good candidate for vector processing, so let's examine them closer. One does not just sit down and start writing vector code. A plan is needed first!

The *rol* and *ror* is a rotation of the 32-bit value left and right by the specified amount.

For the 80 calculations, *n* specifies the index {0...79}.

val is an immediate value. The initial 160-bit message seed is broken up into five 32-bit blocks.

```
     A          B          C          D          E
0x67452301, 0xEFCDAB89, 0x98BADCFE, 0x10325476, xC3D2E1F0
```

Each of those {A,B,C,D,E} message seeds is passed as selected arguments {r,s,t,u,v} to the equations. Note that each equation actually modifies two of these five values.

Finally, the $fn_0()$ and $fn_1()$ access the 512-bit (32 bit x 16) memory. The function $fn_0()$ merely reads adjacent 32-bit data pairs and does an endian conversion to big endian on little endian machines. The function $fn_1()$ is more complicated, as memory is accessed as two sets of four selected 32-bit values, performs a parallel Boolean exclusive OR, and then writes two 32-bit values back to memory.

The following is an examination of just the first equation "EQ0" a little closer with the first two sets of values for index zero and one.

```
MACRO: EQ0(n, r, s, t, u, v )   (0,  E,B,D,A,C)   (1, D,A,C,E,B)
EQ0: r+=rol(u,5)   + fn0(n)+val+(((v^t)&s)^t);     s=ror(s,2);
     e0=e+rol(a,5) + fn0(0)+val+(((c   ^d)&b)^d);  b30=ror(b,2);
     d0=d+rol(e0,5)+ fn0(1)+val+(((b30^c)&a)^c);   a30=ror(a,2);
```

The problem is that these equations are dependency based, as each accepts six values of which two are modified each time. The good news is that two are modified every other equation. Figure 7-4 demonstrates

the initial six steps of processing the first equation. In the first step the initial values of A_0 and B_0 are modified and passed to the next function block along with the previous values of C_0 through F_0. The second step uses the new values A_1 and B_1 along with the previous values and then modifies C_0 and D_0. The process continues 16 times, and then the equation is altered and the steps continue.

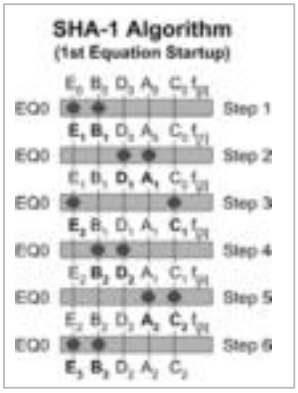


Figure 7-4: Mapping for first SHA-1 equation and pairing of solutions for each step

This does not work too well for handling a single message digest using individual 32-bit calculations within full vector calculations because of the dependencies of the results of the previous steps needed by the following steps. Now step out of the box. Examining this same problem from a different perspective, there are actually two solutions.

One would be to handle multiple message digests in parallel so that two equations use 64-bit data or four equations use 128-bit data processing. This is unfortunately not a truly viable solution, unless it is used in a server that has a need to handle multiple messages simultaneously, and the SHA-1 algorithm would then be considered to be a service.

An alternative would be that of interlacing the steps, such as in Table 7-4. The key to this problem is that "every other time" the same value is being altered. If you look closely, you will note that a value is calculated every other time, so this would be a hint that the equations can be handled in pairs. The only other item is that the pairs have to be skewed so that they are not handled exactly at the same time. Since only two equations can be handled simultaneously, a full 128-bit vector does not work efficiently, but a 64-bit semi-vector does, sort of!

```
e0=e+rol(a,5) +fn0(0)+val+(((c  ^d)&b)^d); b30=ror(b,2);
d0=d+rol(e0,5)+fn0(1)+val+(((b30^c)&a)^c); a30=ror(a,2);
```

Table 7-4: Equation evaluation steps needed to resolve the single pair of solutions

	{1,D,A,C,E,B}	**{0,E,B,D,A,C}**
rol(n,5)		a_5=rol(a,5)
rol(n,30)	a_{30}=rol(a,30)	b_{30}=rol(b,30)
⊕	d_0=(b_{30}⊕c)	e_0=(d⊕c)
^	d_0=(d0^a)	e_o=(e_0^b)
	b=b_{30}	**a**=a_{30}
⊕	d_0=(d_0⊕c)	e_0=(e_0⊕d)
+	d_0=d_0+fn_0(1)	e_0=e_0+fn_0(0)
+	d_0=d_0+val	e_0=e_0+val
+	d_0=d_0+0	e_0=e_0+a_5
+	d_0=d_0+d	e_0=e_0+e
rol(e_0,5)	e_{0_5}=rol(e_0,5)	
	d=d_0+e_{0_5}	**e**=e_0+0

Examining the above single step table, you should notice a little preparation and a little wrap-up, but the middle section uses exactly the same operations. Perfect for vector processing, right?

Well, not exactly! If you recall, it was mentioned that only two equations could be handled simultaneously and no more due to dependencies. As demonstrated in Table 7-5, just handling three parallel equations is fragmentary at best, due to data depending upon the results of a previous equation; thus only two can be realistically handled.

Table 7-5: Attempt to resolve three or more parallel equation evaluations with no success

{2,C,E,B,D,A}	**{1,D,A,C,E,B}**	**{0,E,B,D,A,C}**
		a_5=rol(a,5)
	a_{30}=rol(a,30)	b_{30}=rol(b,30)
	d_0=(b_{30}⊕c)	e_0=(d⊕c)
	d_0=(d_0^a)	e_o=(e_0^b)
	b=b_{30}	**a**=a_{30}
c_0=(b⊕a)	d_0=(d_0⊕c)	e_0=(e_0⊕d)
c_1=fn_0(2)	d_0=d_0+fn_0(1)	e_0=e_0+fn_0(0)
c_1=c_1+val	d_0=d_0+val	e_0=e_0+val
	d_0=d_0+0	e_0=e_0+a_5
	d_0=d_0+d	e_0=e_0+e
	e_{0_5}=rol(e_0,5)	
	d=d_0+e_{0_5}	**e**=e_0+0
d_{0_5}=rol(d,30)		
e_{30}=rol(e,30)		
c_0=(c_0^e_0)		

Handling two equations with 32-bit values is problematic for processors such as an X86 with only MMX capabilities. The individual 32-bit values could be shifted into upper and lower 32-bit data blocks, but that is a lot of data manipulation to gain the parallelism. The function call to access the eight 32-bit memory values, $fn_0()$, could be used to camouflage the register dependency stalls, but it just is not enough.

The best solution is to use 32-bit registers and processor pipelining to your advantage. That is, d_0 and e_0 would in essence be alternately calculated. That would remove register dependency stalls, for example, preventing consecutive read writes to the register containing the e_0 value. The only requirement is that nine or ten 32-bit registers would be needed to be most effective.

So you see, this is the kind of code where vector processing does not pay off, but, if you can prove me wrong, send me your sample code. I would love to see it!

Do not let this example discourage you. Vectorizing code is merely a matter of planning, flow charting, testing for viability, and doing time trials to see if the vector code actually paid off or is in fact slower!

Exercises

1. Given:

   0xB83DE7820

   With a 32-bit data element size, what is the result of a logical right shifting of this data by 34 bits? With an arithmetic right shift? With a logical left shift?

2. Based upon what this chapter discusses about data rotation, what would be the smallest (most efficient) data size to use for rotation of packed data for each of the three processors {X86, MIPS, PowerPC} and for each of their super-set instruction sets?

Chapter 8
Vector Addition and Subtraction

At this point, the focus is turning to the floating-point and integer addition and subtraction of numbers in parallel. With the general-purpose instructions of a processor, normal calculations of addition and subtraction take place one at a time. They can be pipelined so that multiple integer calculations can occur simultaneously, but when performing large numbers of similar type calculations there is a bottleneck of calculation time over processor time. By using vector calculations, multiple like calculations can be performed simultaneously. The only trick here is to remember the key phrase: "multiple like calculations."

If, for example, four pairs of 32-bit words are being calculated simultaneously, such as in the following addition:

47	53	38	87
+23	+74	+39	+16
70	127	77	103

...or the subtraction:

47	53	38	87
–23	–74	–39	–16
24	–21	–1	71

...the point is that the calculations all need to use the same operator. There is an exception, but it is too early to discuss it. There are workarounds, such as if only a couple of expressions need a calculated adjustment while others do not, then adding or subtracting a zero would keep their result neutral. Remember the algebraic law of additive identity from Chapter 4, "Vector Methodologies" (n+0 = 0+n = n).

It is in essence a wasted calculation, but its use as a placeholder helps make SIMD instructions easier to use.

Algebraic Law:

Additive Inverse	a – b = a + (–b)

The other little item to remember is that subtraction is merely the addition of a value's additive inverse:

$$a–b = a+(–b)$$

The samples on the CD are actually three different types of examples: a standard single data element solution, a 3D value which is typically an {XYZ} value, or a 4D value {XYZW}. Integer or fixed point is important, but in terms of fast 3D processing, single-precision floating-point is of more interest.

CD Workbench Files: /Bench/*architecture*/chap08/*project*/*platform*

	architecture	Add/Sub	*project*	*platform*
PowerPC	/vmp_ppc/	**Float**	**/fas/**	/mac9cw
X86	/vmp_x86/	**3D Float**	**/vas3d/**	/vc6
MIPS	/vmp_mips/	**4vec Float**	**/qvas3d/**	/vc.net
		Integer	**/pas/**	/devTool

Vector Floating-Point Addition

$d_{(0...n-1)} = a_{(0...n-1)} + b_{(0...n-1)}$ n={ 4, 2}

AltiVec	*MMX*	**SSE**	**SSE2**	**3DNow**	*3DMX+*	**MIPS**	*MMI*

AltiVec	vaddfp *Dst*, *aSrc*, *bSrc* *vD=vec_add(vA, vB)*	Single-Precision	128
3DNow	pfadd *mmDst, mmSrc(mm/mm64)*	Single-Precision	64
SSE	addps *xmmDst, xmmSrc(xmm/m128)*	Single-Precision	128
SSE2	addpd *xmmDst, xmmSrc(xmm/m128)*	Double-Precision	128
MIPS V	add.ps *Dst, aSrc, bSrc*	Single-Precision	64

This vector instruction is a parallel operation that uses an adder on each of the source floating-point blocks *aSrc (xmmSrc)* and *bSrc (xmmDst)* and stores the result in the destination *Dst (xmmDst)*.

The instructions may be labeled as packed, parallel, or vector, but each block of floating-point bits is, in reality, isolated from each other.

The following are 64- and 128-bit single- and double-precision summation samples.

64-bit single-precision floating-point:

63...32	31...0
120.885	–86.479
+–120.622	+30.239
0.263	–56.240

128-bit single-precision floating-point:

127...96	95...64	63...32	31...0
56.929	–20.193	120.885	–86.479
+–124.783	+–49.245	+–120.622	+30.239
–67.854	–69.438	0.263	–56.240

128-bit double-precision floating-point:

127...64	63...0
–75.499	57.480
+124.073	+–50.753
48.574	6.727

Vector Floating-Point Addition with Scalar

$d_{(0...n-1)} = a_{(0...n-1)} + b \quad n=\{4, 2\}$

AltiVec	*MMX*	**SSE**	**SSE2**	*3DNow*	*3DMX+*	*MIPS*	*MMI*

SSE addss *xmmDst, xmmSrc(xmm/m32*) Single-Precision 128
SSE2 addsd *xmmDst, xmmSrc(xmm/m64)* Double-Precision 128

This vector instruction is a scalar operation that uses an adder with the source scalar *xmmSrc* and the source floating-point value in the least significant block within *xmmDst* and stores the result in the destination *xmmDst*. The upper float elements are unaffected.

The instructions may be labeled as packed, parallel, or vector, but each block of floating-point bits is, in reality, isolated from the others.

Notice in the following 128-bit single- and double-precision scalar summation samples, only the lowest set of bits in bold are affected, while the other bits remain unchanged.

128-bit single-precision floating-point:

127...96	95...64	63...32	31...0
56.929	–20.193	120.885	–86.479
+ 0.0	+ 0.0	+ 0.0	+**30.239**
56.929	–20.193	120.885	–**56.240**

128-bit double-precision floating-point:

127...64	63...0
–75.499	57.480
+ 0.0	+–**50.753**
–75.499	**6.727**

Vector Floating-Point Subtraction

$d_{(0...n-1)} = a_{(0...n-1)} - b_{(0...n-1)}$ n={ 4, 2}

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vsubfp *Dst, aSrc, bSrc* *vD=vec_sub(vA,vB)*	Single-Precision	128
3DNow	pfsub *mmDst, mmSrc(mm/m64)*	Single-Precision	64
SSE	subps *xmmDst, xmmSrc(xmm/m128)*	Single-Precision	128
SSE2	subpd *xmmDst, xmmSrc(xmm/m128)*	Double-Precision	128
MIPS V	sub.ps *Dst, aSrc, bSrc*	Single-Precision	64

This vector instruction is a parallel operation that subtracts each of the source floating-point blocks *bSrc (xmmSrc)* from *aSrc (xmmDst)* with the result stored in the destination *Dst (xmmDst)*.

Note: Be careful here, as the register and operator ordering is as follows:

$xmmDst_{(31...0)} = xmmDst_{(31...0)} - xmmSrc_{(31...0)}$ D–=A

$Dst_{(31...0)} = bSrc_{(31...0)} - aSrc_{(31...0)}$ D=B–A

This is contrary to normal, where *aSrc* is associated with *xmmSrc* and *bSrc* is associated with *xmmDst*, and not the other way around as in this particular case!

The instructions may be labeled as packed, parallel, or vector, but each block of floating-point bits is, in reality, isolated from each other.

64-bit single-precision floating-point:

63...32	31...0
–98.854	124.264
–50.315	–33.952
–48.539	158.216

128-bit single-precision floating-point:

127...96	95...64	63...32	31...0
–64.185	108.856	–98.854	124.264
+–114.223	– –117.045	–50.315	–33.952
–178.408	225.901	–48.539	158.216

128-bit double-precision floating-point:

127...64	63...0
–48.043	127.277
– –106.051	– –77.288
58.008	204.565

$d_{(0...n-1)} = -a_{(0...n-1)}$

vmp_VecNeg

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	**MIPS**	MMI

MIPS V neg.ps *Dst, aSrc* Single-Precision 64

A vector can also be negated by setting its inverse.

```
void vmp_VecNeg(vmp3DVector *vD, vmp3DVector *vA)
{
  vD->x = -(vA->x);
  vD->y = -(vA->y);
  vD->z = -(vA->z);
}
```

Vector Floating-Point Subtraction with Scalar

$d_{(0...n-1)} = a_{(0...n-1)} - b$ n={ 4, 2}

AltiVec	MMX	**SSE**	**SSE2**	3DNow	3DMX+	MIPS	MMI

SSE subss *xmmDst, xmmSrc(xmm/m32*) Single-Precision 128
SSE2 subsd *xmmDst, xmmSrc(xmm/m64)* Double-Precision 128

This vector instruction is a scalar operation that subtracts the least-significant source floating-point block of *xmmSrc* from the same block in *xmmDst* and stores the result in the destination *xmmDst*. The upper float elements are unaffected.

Notice in the following 128-bit single- and double-precision scalar subtraction samples, only the lowest set of bits in bold are affected, while the other bits remain unchanged.

127...96	95...64	63...32	31...0
–64.185	108.856	–98.854	124.264
– 0.0	– 0.0	– 0.0	**–33.952**
–64.185	108.856	–98.854	**158.216**

127...64	63...0
–48.043	127.277
–0.0	– **–77.288**
–48.043	**204.565**

Pseudo Vec

Since a subtraction is merely the summation of the additive inverse, merely negate the second operand and use a floating-point scalar addition. An alternative solution is to use the algebraic law of additive identity with the non-scalar values by setting them on the second operand to zero and using the standard floating-point subtraction. This can be done with a combination of a logical AND, XOR of the sign bit, and then a summation.

127...96	95...64	63...32	31...0
–64.185	108.856	–98.854	124.264
+ 0.0	+ 0.0	+ 0.0	+(–(33.952))
–64.185	108.856	–98.854	158.216

Vector Floating-Point Reverse Subtraction

$d_{(0...1)} = b_{(0...1)} - a_{(0...1)}$

AltiVec	MMX	SSE	SSE2	**3DNow**	3DMX+	MIPS	MMI

3DNow pfsubr *mmDst, mmSrc(mm/m64)* Single-Precision 64

This vector instruction is a parallel operation that subtracts each of the source floating-point blocks, *mmDst* from *mmSrc*, and stores the result in the destination *mmDst*.

$$mmDst_{(31...0)} = mmSrc_{(31...0)} - mmDst_{(31...0)}$$
$$mmDst_{(63...32)} = mmSrc_{(63...32)} - mmDst_{(63...32)}$$

The instructions may be labeled as packed, parallel, or vector, but each block of floating-point bits is, in reality, isolated from the others.

A typical subtraction uses an equation similar to {a=a–b}, but what happens if the equation {a=b–a} is needed instead? This instruction solves that situation by limiting any special handling needed to exchange values between registers, such as the following:

```
c[0]=a[0];          c[1]=a[1];

a[0]=b[0];          a[1]=b[1];

a[0]=a[0] – c[0];   a[1]=a[1] – c[1];

b[0]=c[0];          b[1]=c[1];
```

or

```
exchange(a, b)     A=b     B=a ← a=a  b=b
A = A – B          b=b–a   a=a
exchange(a, b)     a=(b–a) b=a ← b=(b–a)  a=a
```

Vector Addition and Subtraction (Single-Precision)

The addition and subtraction of vectors is a relatively simple matter for vector math instructions to handle — even in the case of single vectors.

Pseudo Vec

By now you should be very aware that you should be using assertions in your code, such as ASSERT_PTR4 for normal pointers and ASSERT_PTR16 for pointers to vectors properly aligned in memory, so I will try not to bore you with it anymore in print. You should also by now be aware of the penalties for dealing with out-of-alignment memory, especially when dealing with the MIPS or AltiVec instruction sets. The same principals of aligning memory to use the instructions as discussed earlier need to be in place, so you will not be burdened with it in print from this point forward. You will find the items mentioned and more used in

the sample code on the CD; keep it in mind when writing your own code. There will also be a limitation of the use of the term "const" to help make the printed code less wordy and more clear.

You will find that for purposes of cross-platform compatibility, these functions return no arguments. They are instead written as procedures where the first argument points to a buffer in which the result is stored. This is not written to make your life confusing. It is written this way because of one particular processor: the X86. Due to its MMX versus FPU usage, an *emms* instruction must be called to reset that functionality as a clean slate, so only one of them can be used at a time. By not returning a value such as a float or array of floats, the risk that the programmer might accidentally try to use the returned value while in the wrong mode is minimized. In this way, the vmp_SIMDEntry() and vmp_SIMDExit() procedure calls are made to throw the gate to change mode of operation. Of course, if you are working with a non-X86 processor, these SIMD gates are stubbed to nothingness and do not affect you. Confused? Bored? Do not care about X86? Fine, let us continue! Since most of you will be focused upon float and not integer or fixed-point vector math, that will be the focus, but the principals are the same.

The simple addition and subtraction of a single (scalar) float has been included here as a reference.

Single-Precision Float Addition

Listing 8-1: \chap08\fas\Fas.cpp

```
void vmp_FAdd( float *pfD, float fA, float fB )
{
  *pfD = fA + fB;
}
```

Single-Precision Float Subtraction

Listing 8-2: \chap08\fas\Fas.cpp

```
void vmp_FSub( float *pfD, float fA, float fB )
{
  *pfD = fA – fB;
}
```

This is simple scalar addition and subtraction using single-precision floats. Now view the addition of two vectors containing a three-cell {XYZ} float.

Single-Precision Vector Float Addition

Listing 8-3: \chap08\vas3d\Vas3D.cpp

```
void vmp_VecAdd(vmp3DVector * const pvD,
          const vmp3DVector * const pvA,
          const vmp3DVector * const pvB)
{
  pvD->x = pvA->x + pvB->x;
  pvD->y = pvA->y + pvB->y;
  pvD->z = pvA->z + pvB->z;
}
```

Single-Precision Vector Float Subtraction

Listing 8-4: \chap08\vas3d\Vas3D.cpp

```
void vmp_VecSub(vmp3DVector * const pvD,
          const vmp3DVector * const pvA,
          const vmp3DVector * const pvB)
{
  pvD->x = pvA->x - pvB->x;
  pvD->y = pvA->y - pvB->y;
  pvD->z = pvA->z - pvB->z;
}
```

Now view the addition and subtraction of two vectors containing a four-cell (quad) {XYZW} single-precision float. For the sample cross-platform libraries, there is a differentiation between a Vec being a standard 3D tri-elemental value and a QVec being a full four-quad float vector. The Vec is more oriented to the AoS (Array of Structures) approach and the QVec would work best in an SoA (Structure of Arrays) that was discussed in Chapter 4, "Vector Methodologies."

Single-Precision Quad Vector Float Addition

Listing 8-5: \chap08\qvas3d\QVas3D.cpp

```
void vmp_QVecAdd(vmp3DQVector * const pvD,
           const vmp3DQVector * const pvA,
           const vmp3DQVector * const pvB)
{
  pvD->x = pvA->x + pvB->x;
  pvD->y = pvA->y + pvB->y;
  pvD->z = pvA->z + pvB->z;
  pvD->w = pvA->w + pvB->w;
}
```

Single-Precision Quad Vector Float Subtraction

Listing 8-6: \chap08\qvas3d\QVas3D.cpp

```
void vmp_QVecSub(vmp3DQVector * const pvD,
           const vmp3DQVector * const pvA,
           const vmp3DQVector * const pvB)
{
  pvD->x = pvA->x - pvB->x;
  pvD->y = pvA->y - pvB->y;
  pvD->z = pvA->z - pvB->z;
  pvD->w = pvA->w - pvB->w;
}
```

Pseudo Vec (X86)

Now examine these functions more closely using X86 Assembly. As MMX does not support floating-point, only 3DNow! and SSE can be utilized. 3DNow! supports 64 bit, so two loads and two stores must be handled simultaneously, but it is a simple matter of adding the two pairs of floats to each other. Keep in mind that three floats {XYZ} are being used, and the fourth element {W} is being ignored!

```
mov   eax,vA    ; Vector A
mov   ebx,vB    ; Vector B
mov   edx,vD    ; Vector Destination
```

vmp_VecAdd (3DNow!)

Listing 8-7: vmp_x86\chap08\vas3d\Vas3DX86M.asm

```
movq  mm0,[eax]                ;vA.xy  {Ay Ax}
movq  mm2,[ebx]                ;vB.xy  {By Bx}
movd  mm1,(vmp3DVector PTR [eax]).z   ; {0 Az}
movd  mm3,(vmp3DVector PTR [ebx]).z   ; {0 Bz}
pfadd mm0,mm2                  ; {Ay+By Ax+Bx}
pfadd mm1,mm3                  ;  {0+0  Az+Bz}
movq  [edx],mm0                ; {Ay+By Ax+Bx}
movd  (vmp3DVector PTR [edx]).z,mm1 ;{0 Az+Bz}
```

vmp_VecSub (3DNow!)

For subtraction, the functions are virtually identical to that of the addition, except for the exchanging of *pfsub* for *pfadd*.

Listing 8-8: vmp_x86\chap08\vas3d\Vas3DX86M.asm

```
movq  mm0,[eax]                ;vA.xy  {Ay Ax}
movq  mm2,[ebx]                ;vB.xy  {By Bx}
movd  mm1,(vmp3DVector PTR [eax]).z   ; {0 Az}
movd  mm3,(vmp3DVector PTR [ebx]).z   ; {0 Bz}
pfsub mm0,mm2                  ; {Ay-By Ax-Bx}
```

```
pfsub mm1,mm3                    ;  {0-0  Az-Bz}
movq  [edx],mm0                  ; {Ay-By Ax-Bx}
movd  (vmp3DVector PTR [edx]).z,mm1 ;{0 Az-Bz}
```

vmp_QVecAdd (3DNow!)

A quad vector access is not much different. Instead of loading a single float for each vector, a double float pair is loaded instead (*movq* instead of *movd*).

Listing 8-9: vmp_x86\chap08\vas3d\Vas3DX86M.asm

```
movq  mm0,[eax+0]              ;vA.xy   {Ay Ax}
movq  mm2,[ebx+0]              ;vB.xy   {By Bx}
movq  mm1,[eax+8]              ;vA.zw   {Aw Az}
movq  mm3,[ebx+8]              ;vB.zw   {Bw Bz}
pfadd mm0,mm2                  ;  {Ay+By Ax+Bx}
pfadd mm1,mm3                  ;  {Aw+Bw Az+Bz}
movq  [edx+0],mm0              ;  {Ay+By Ax+Bx}
movq  [edx+8],mm1              ;  {Aw+Bw Az+Bz}
```

vmp_VecAdd (SSE) Unaligned

The SSE processor in the following code snippet can load 128 bits at a time, so the entire 96-bit vector can be loaded at once, including an extra 32 bits. This introduces a problem of contamination when the 96-bit value is written to memory as 128 bits. The solution is to read those destination bits, preserve the upper 32 bits, and write the newly merged 128 bits. Keep in mind efficient memory organization and memory tail padding previously discussed in Chapter 6, "Bit Mangling." Data can be misaligned or aligned, but 128-bit alignment would be preferable.

Okay, I lied a wee bit earlier. You need to know about two SSE instructions: *movaps* and *movups*.

- *movaps* — For use in aligned memory access of single-precision floating-point values
- *movups* — For use in unaligned memory access of single-precision floating-point values

One other item that should be brought to light is the special handling required by vectors versus quad vectors. As previously discussed in Chapter 6, "Bit Mangling," the vector is three single-precision floats 96 bits in size, but when accessed as a vector, 128 bits are accessed simultaneously. This means that those extra 32 bits must be preserved and not destroyed. Also, the data contained within it must not be expected to be a float but to be garbage data to that particular expression and valid data

to another expression. Thus, it must be treated as such. Therefore, the easiest method is to clear and then restore those bits. The following declarations work nicely as masks for bit blending just for that purpose.

```
himsk32 DWORD 000000000h, 000000000h, 000000000h,
              0FFFFFFFFh      ; Save upper 32bits
lomsk96 DWORD 0FFFFFFFFh, 0FFFFFFFFh, 0FFFFFFFFh,
              000000000h      ; Save lower 96bits
```

Also note that if bits are being preserved with a mask, then others are being cleared to zero. Of course, it depends upon the endian type byte ordering of the platform, but for X86 it is as listed!

Listing 8-10: vmp_x86\chap08\vas3d\Vas3DX86M.asm

```
movups xmm2,[edx]          ;vD.xyzw {Dw Dz Dy Dx}
movups xmm0,[ebx]          ;vB.xyzw {Bw Bz By Bx}
movups xmm1,[eax]          ;vA.xyzw {Aw Az Ay Ax}
andps  xmm2,OWORD PTR himsk32    ; {Dw  0  0  0}
addps  xmm0,xmm1           ;  {Aw+Bw Az+Bz Ay+By Ax+Bx}
andps  xmm0,OWORD PTR lomsk96 ; { 0 Az+Bz Ay+By Ax+Bx}
orps   xmm0,xmm2               ; {Dw Az+Bz Ay+By Ax+Bx}
movups [edx],xmm0              ; {Dw  Dz    Dy    Dx}
```

vmp_VecAdd (SSE) Aligned

By replacing the *movups* marked in bold above with *movaps*, the data must be properly aligned or an exception will occur, but the application will run more smoothly. This is where two versions of the function would work out nicely. One is when data alignment is unknown and the other is when alignment is guaranteed!

Listing 8-11: vmp_x86\chap08\vas3d\Vas3DX86M.asm

```
movaps xmm2,[edx]          ;vD.xyzw {Dw Dz Dy Dx}
movaps xmm0,[ebx]          ;vB.xyzw {Bw Bz By Bx}
movaps xmm1,[eax]          ;vA.xyzw {Aw Az Ay Ax}
andps  xmm2,OWORD PTR himsk32    ; {Dw  0  0  0}
addps  xmm0,xmm1           ;  {Aw+Bw Az+Bz Ay+By Ax+Bx}
andps  xmm0,OWORD PTR lomsk96 ; { 0 Az+Bz Ay+By Ax+Bx}
orps   xmm0,xmm2               ; {Dw Az+Bz Ay+By Ax+Bx}
movaps [edx],xmm0              ; {Dw  Dz    Dy    Dx}
```

See, the code looks almost identical, so from this point forward, the printed material will only show the aligned code using *movaps*!

vmp_QVecAdd (SSE) Aligned

For quad vectors, it is even easier, as there is no masking of the fourth float {W} — just read, evaluate, and write! Of course, the function

should have the instructions arranged for purposes of optimization, but here they are left in a readable form.

Listing 8-12: vmp_x86\chap08\qvas3d\QVas3DX86M.asm

```
movaps xmm1,[ebx] ; {Bw Bz By Bx}
movaps xmm0,[eax] ; {Aw Az Ay Ax}
addps  xmm0,xmm1  ; {Aw+Bw Az+Bz Ay+By Ax+Bx}
movaps [edx],xmm0 ; { Dw    Dz    Dy    Dx}
```

Pseudo Vec (PowerPC)

A drawback of vector processing with three-float elements instead of four is the extra baggage necessary to handle the missing fourth element! Another problem is that the three-float tri-vector {XYZ} has to be transferred to another working vector, and a valid fourth float {W} has to be inserted. That original fourth value in memory may not be a legal formatted float, and so a placeholder needs to be put in its place. During the writing of the result, the fourth value needs to be protected from any possible change. This is a nice built-in feature of the MIPS VU coprocessor of the PS2, but unfortunately for this processor, the 12 bytes representing the three floats {XYZ} have to be written back, protecting the original fourth float {W}. The following section attempts to illustrate this.

vmp_VecAdd (AltiVec) Unaligned

Listing 8-13: vmp_ppc\chap08\vas3d\Vas3DAltivec.cpp

```
vector float vD, vA, vB;

((vmp3DQVector *) &vA)->x = pvA->x;
((vmp3DQVector *) &vA)->y = pvA->y;
((vmp3DQVector *) &vA)->z = pvA->z;

((vmp3DQVector *) &vB)->x = pvB->x;
((vmp3DQVector *) &vB)->y = pvB->y;
((vmp3DQVector *) &vB)->z = pvB->z;

vD = vec_add( vA, vB );  //  Dxyzw=Axyzw+Bxyzw

pvD->x = ((vmp3DQVector *) &vD)->x;
pvD->y = ((vmp3DQVector *) &vD)->y;
pvD->z = ((vmp3DQVector *) &vD)->z;
                         //  Dw is thrown away
```

The aligned parallel quad vector float addition is more efficient, thus no extra packaging of data is required.

vmp_QVecAdd (AltiVec) Aligned

Listing 8-14: vmp_ppc\chap08\qvas3d\QVas3DAltivec.cpp

```
void vmp_QVecAdd(vmp3DQVector * const pvD,
           const vmp3DQVector * const pvA,
           const vmp3DQVector * const pvB)
{
  *(vector float *)pvD =
         vec_add( (*(vector float *)pvA),
                  (*(vector float *)pvB) );
}
```

vmp_QVecSub (AltiVec) Aligned

This is just like QVecAdd, only it needs *vec_sub* substituted for *vec_add*.

Pseudo Vec (MIPS)

The MIPS processor family is similar to the X86 processor family in the sense that there are so many flavors of the processor. MIPS I through MIPS V and the Application Specific Extensions (ASE) MIPS16, SmartMIPS, MIPS-3D, and MDMX come in the MIPS32 and MIPS64 architectures. With all of these, there are multiple methods for handling vector math.

vmp_VecAdd (MIPS V) Aligned

In this first example, a MIPS64 architecture using the MIPS V instruction set handles a vector by using the paired single-precision floating-point for the first element pair {XY} and a scalar addition {W}.

Listing 8-15: vmp_mips\chap08\vas3d\Vas3DMips.cpp

```
  ldc1   $f4, 0(a1)     ; {A1 A0} Mem to PS_FP
  ldc1   $f6, 0(a2)     ; {B1 B0}
  lwc1   $f5, 8(a1)     ; {   A2}
  lwc1   $f7, 8(a2)     ; {   B2}

  add.ps $f4, $f4, $f6  ; {A1+B1 A0+B0}
  add.s  $f5, $f5, $f7  ; {      A2+B2}

  sdc1   $f4, 0(a0)     ; {D1 D0}
  swc1   $f5, 8(a0)     ; {   D2}
```

vmp_QVecAdd (MIPS V) Aligned

For quad vectors, two paired single-precision floating-point additions are utilized.

Listing 8-16: vmp_mips\chap08\qvas3d\QVas3DMips.cpp

```
ldc1   $f4, 0(a1)     ; {A1 A0} Mem to PS_SPFP
ldc1   $f6, 0(a2)     ; {B1 B0}
ldc1   $f5, 8(a1)     ; {A3 A2}
ldc1   $f7, 8(a2)     ; {B3 B2}

add.ps $f4, $f4, $f6  ; {A1+B1 A0+B0}
add.ps $f5, $f5, $f7  ; {A3+B3 A2+B2}

sdc1   $f4, 0(a0)     ; {D1 D0}
sdc1   $f5, 8(a0)     ; {D3 D2}
```

vmp_QVecSub (MIPS V) Aligned

A subtraction merely replaces the paired addition instruction *add.ps* with its equivalent subtraction instruction *sub.ps*.

```
sub.ps $f4, $f4, $f6  ; {A1–B1 A0–B0}
sub.ps $f5, $f5, $f7  ; {A3–B3 A2–B2}
```

vmp_VecAdd (VU0) Aligned

The VU coprocessor is a bit more proprietary. For three field vectors, remember that the coprocessor COP2 within the PS2 has data element swizzle capability, and once that is understood, the rest is easy.

Review the section "vmp_FUNCTION (MIPS — VU0 Assembly) Aligned" in Chapter 3.

For specific information, see your PS2 Linux Kit or devTool manual.

Vector Scalar Addition and Subtraction

The scalar addition and subtraction of vectors is also a relatively simple matter for vector math instructions to handle. Scalar math appears in one of two forms — either a single element processed within each vector, or one element is swizzle, shuffle, splat (see Chapter 5, "Vector Data Conversion") into each element position and applied to the other source vector. When this type of instruction is not supported by a processor, the trick is to replicate the scalar so it appears as a second vector.

Single-Precision Quad Vector Float Scalar Addition

Listing 8-17: \chap08\vas3d\Vas3D.cpp

```
void vmp_VecAddScalar(vmp3DVector * const pvD,
   const vmp3DVector * const pvA, float fScalar)
{
  pvD->x = pvA->x + fScalar;
  pvD->y = pvA->y + fScalar;
  pvD->z = pvA->z + fScalar;
}
```

Single-Precision Quad Vector Float Scalar Subtraction

Listing 8-18: \chap08\qvas3d\QVas3D.cpp

```
void vmp_VecSubScalar(vmp3DVector * const pvD,
   const vmp3DVector * const pvA, float fScalar)
{
  pvD->x = pvA->x – fScalar;
  pvD->y = pvA->y – fScalar;
  pvD->z = pvA->z – fScalar;
}
```

Did that look strangely familiar? The big question is, how do we replicate a scalar to look like a vector, since there tends not to be mirrored scalar math on processors? Typically, a processor will interpret a scalar calculation as the lowest (first) float being evaluated with a single scalar float. This is fine and dandy, but there are frequent times when a scalar needs to be replicated and summed to each element of a vector. So the next question is, how do we do that?

With the 3DNow! instruction set, it is easy. Since the processor is really a 64-bit half vector, the data is merely unpacked into the upper and lower 32 bits.

```
movd      mm2,fScalar   ; fScalar {0 s}
punpckldq mm2,mm2       ; fScalar {s s}
```

Then it is just used twice — once with the upper 64 bits and once with the lower 64 bits.

```
pfadd     mm0,mm2       ;       {Ay+s  Ax+s}
pfadd     mm1,mm2       ;       {Aw+s  Az+s}
```

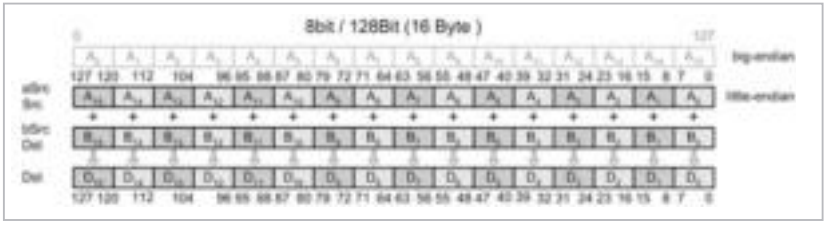
With the SSE instruction set, it is almost as easy. The data is shuffled into all 32-bit floats.

```
movss     xmm1,fScalar          ; {0 0 0 s}
shufps    xmm1,xmm1,00000000b   ; {s s s s}
```

Now the scalar is the same as the vector. Any questions?

```
addps     xmm0,xmm1     ; {Aw+s Az+s Ay+s Ax+s}
```

Vector Integer Addition



$d_{(0...n-1)} = a_{(0...n-1)} + b_{(0...n-1)}$ n={16, 8, 4, 2}

AltiVec	**MMX**	*SSE*	**SSE2**	*3DNow*	*3DMX+*	*MIPS*	**MMI**

AltiVec	vaddu(b/h/w)m *Dst, aSrc, bSrc* *vD=vec_add(vA, vB)*	[Un]signed	128
MMX	padd(b/w/d/q) *mmDst, mmSrc(mm/m64)*	[Un]signed	64
SSE2	padd(b/w/d/q) *xmmDst, xmmSrc(xmm/m128)*	[Un]signed	128
MMI	padd(b/h/w) *Dst, aSrc, bSrc*	[Un]signed	128

This vector instruction is a parallel operation that uses an adder on each of the source bit blocks *aSrc (xmmSrc)* and *bSrc (xmmDst)* and stores the result in the destination *Dst (xmmDst)*.

The instructions may be labeled as packed, parallel, or vector, but each block of bits is, in reality, isolated from the others. The following is a 32-bit example consisting of four unsigned 8-bit values:

31...24	**23...16**	**15...8**	**7...0**	**31...0**
95	231	131	187	0x5F E7 83 BB
+85	+37	+103	+11	+0x55 25 67 0B
(0xB4) 180	(0x0C) 12	(0xEA) 234	(0xC6) 198	0xB4 0C EA C6

...and four signed 8-bit values:

31...24	23...16	15...8	7...0	31...0
95	–25	–125	–69	0x5F E7 83 BB
+85	+37	+103	+11	+0x55 25 67 0B
(0xB4) 180	(0x0C) 12	(0xEA) –22	(0xC6) –58	0xB4 0C EA C6

Regardless of the decimal representation of unsigned or signed, the hex values of the two examples remained the same, thus the reason for these being [Un]signed and sign neutral.

Notice in the following additions of 7-bit signed values that with the limit range of –64...63, the worst case of negative and positive limit values results with no overflow.

```
  11000000b  C0 (–64)      00111111b  3F (63)
+ 11000000b  C0 (–64)    + 00111111b  3F (63)
  80000000b  80 (–128)     11111110b  7E (126)
```

Of course, positive and negative signed values could also be intermixed without an overflow. For a 7-bit unsigned value, 0...127, there would be no overflow.

```
  11000000b  C0 (–64)      01111111b  7F (127)
+ 00111111b  3F ( 63)    + 01111111b  7F (127)
  11111111b  FF  (–1)      11111110b  FE (254)
```

The eighth unused bit is, in reality, used as a buffer preventing any overflow to a ninth bit.

Pseudo Vec

Most processors that support SIMD also support packed addition. If your processor does not, unsigned addition can be emulated, but the bit width used by both values must be at least one less than the size that can be contained by both values. Similar to what was discussed previously, two values of 7 bits or less can be summed to generate a value that would fit into 8 bits without a carry into an adjacent set of packed bits. A negative, on the other hand, is a little trickier. A summation of the MSB of the eighth bit would generate a carry if both numbers were negative.

For an averaging of two values, with rounding, one would use the formula (a+1+b)/2. If you note the result of each of the sample limit calculations above, there is room to add a positive value of one to each of them without causing an overflow before the division.

7F + 7F = FE + 1 = FF

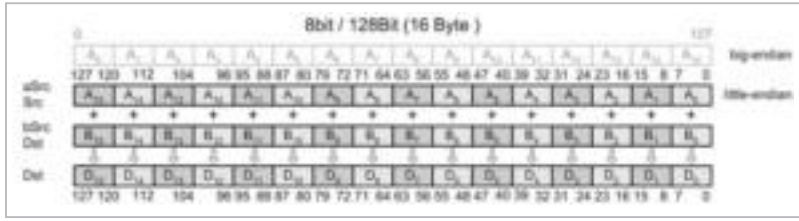
A logical right shift would be used for the unsigned values. Keeping all of this in mind, including the verification that the limits are never exceeded, multiple summations can occur simultaneously with no carry to affect an adjacent block of bits.

A sample use of this instruction would be for the translation of a set of points by displacing their coordinates. For example, in a video game, a Sasquatch-creature object is designed with its origin {center of gravity; anchor point} at the point under its feet where it makes contact with the ground while standing on its rear legs. It is given coordinate {0,0,0}. All polygonal mesh coordinates that make up the legs, torso, and head are relative to that point. When the character is placed within a scene, a copy of its coordinates is copied to a render list, but all the coordinates are translated — that is, adjusted in three-dimensional {X,Y,Z} space.

The following is a list of the corner vertices of a cube, as well as those same coordinates translated so that its origin is set on a surface at coordinate {24,0,9}.

Cube	Translated
{–1,0,–1}	{23,0,8}
{–1,0,1}	{23,0,10}
{1,0,1}	{25,0,10}
{1,0,–1}	{25,0,8}
{–1,2,–1}	{23,2,8}
{–1,2,1}	{23,2,10}
{1,2,1}	{25,2,10}
{1,2,–1}	{25,2,8}

Vector Integer Addition with Saturation



$d_{(0...n-1)}=Max(Min(a_{(0...n-1)}+b_{(0...n-1)}, HI\_LIMIT), LO\_LIMIT)_{n=\{16, 8, 4, 2\}}$

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

AltiVec	vaddu(b/h/w)s *Dst, aSrc, bSrc*	Unsigned	128
	vadds(b/h/w)s *Dst, aSrc, bSrc*	Signed	
	vD=vec_adds(vA, vB)		
MMX	paddus(b/w) *mmDst, mmSrc (mm/m64)*	Unsigned	64
	padds(b/w) *mmDst, mmSrc(mm/m64)*	Signed	64
SSE2	paddus(b/w) *xmmDst, xmmSrc(xmm/m128)*		
		Unsigned	128
	padds(b/w) *xmmDst, xmmSrc(xmm/m128)*	Signed	
MMI	paddu(b/h/w) *Dst, aSrc, bSrc*	Unsigned	128
	padds(b/h/w) *Dst, aSrc, bSrc*	Signed	128

This vector instruction is a parallel operation that uses an adder on each of the source bit block registers *aSrc (xmmSrc)* and *bSrc (xmmDst)* and stores the result in the destination *Dst (xmmDst)* using saturation logic to prevent any possible wraparound.

Each calculation limits the value to the extents of the related data type so that if the limit is exceeded, it is clipped inclusively to that limit. This is handled differently, whether it is signed or unsigned, as they both use different limit values. Effectively, the result of the summation is compared to the upper limit with a Min expression and compared to the lower limit with a Max equation. Notice in the previous section that when two signed 8-bit values of 0x7F (127) are summed, a value of 0xFE (254) results but is clipped to the maximum value of 0x7f (127). The same applies if, for example, two values of 0x80 (–128) are summed resulting in –256 but clipped to the minimum value of 0x80 (–128). Check out Chapter 10, "Special Functions," regarding the functionality of Min and Max.

The instructions may be labeled as packed, parallel, or vector, but each block of bits is, in reality, isolated from the others.

31...24	23...16	15...8	7...0	31...0
95	120	–125	–69	0x5F 78 83 BB
+85	+37	+ –32	+11	+0x55 25 E0 0B
(0x7F) 127	**(0x7F) 127**	**(0x80) –128**	(0xC6) –58	0x7F 7F 80 C6

A sample use of this instruction would be for sound mixing where two sound waves are mixed into a single wave for output. The saturation point keeps the amplitude of the wave from wrapping from a positive or high level into a negative or low one, thus creating a pulse-encoded harmonic and distortion.

For saturation, the limits are different for the data size, as well as for signed and unsigned.

	8 Bit	16 Bit
signed	–128...127	–32768...32767
unsigned	0...255	0...65535

Unsigned 8-bit addition saturation @ 255 using code branching:

```
int v;
v = (v>255) ? 255 : v;
```

...using branchless coding:

```
v = (uint8)(0xff & (v | ((0xff–v)>>31)));
```

...unsigned 16-bit addition w/saturation @ 65535:

```
*pD++ = (uint16)(0xffff & (v  | ((0xffff–v) >> 31)));
```

With branchless coding, there is no misprediction for branching, so code tends to run faster by using bit masking/blending logic. With unsigned addition, the result will either increase or remain the same. Since data is being unpacked from 8 or 16 bit to an integer, there is no need to worry about an overflow.

When subtraction or signed values are involved, an upper and lower limit needs to be tested for. In the following 8-bit min/max, first adjust the lower limit to a threshold of zero by adding 128 so that a –128 (the saturation low) becomes zero and anything lower generates a negative number, so if too low, the number becomes negative and a mask of 1s is generated. The upper limit is adjusted by subtracting that same factor, thus adjusting it to that of an unsigned number and treating it as such.

```
v =  128 + ((int)*pA++) + ((int)*pB++);
v &= (–1^(v>>31));                                    // max(0,n)
*pD++ = (int8)(–128 + (0xff & (v | ((0xff–v) >> 31)))); // min(255,n)
```

The 16-bit value with its min/max {–32768, 32767} is effectively the same!

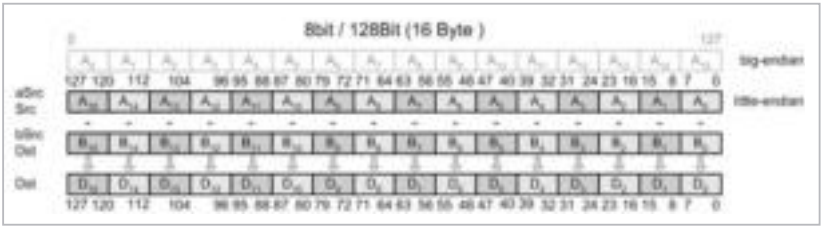
```
v =  32768 + ((int)*pA++) + ((int)*pB++);
v &= (–1^(v>>31));     // (v<0) ? 0 : v;
*pD++ = (uint16)(–32768 + (0xffff & (v | ((0xffff–v) >> 31)))); //65536
```

A little confusing? Well, it can be! Just remember to tag it with a comment so it can be recognized immediately, and you will have the best of both worlds!

```
// Max(v,65535),                 or     // Max( Min(v, –32768), 32767)
```

Warning: Watch the overhead; sometimes double limit min and max branchless is more costly in CPU time than actually using a branch!

Vector Integer Subtraction



$d_{(0...n-1)} = a_{(0...n-1)} - b_{(0...n-1)}$ n={16, 8, 4, 2}

AltiVec	**MMX**	*SSE*	**SSE2**	*3DNow*	*3DMX+*	*MIPS*	**MMI**

AltiVec vsubu(b/h/w)m *Dst, aSrc, bSrc* [Un]signed 128
vsubs(b/h/w)m *Dst, aSrc, bSr*
vD=vec_sub(vA, vB)

MMX psub(b/w/d/q) *mmDst, mmSrc (mm/m64)*
[Un]signed 64

SSE2 psub(b/w/d/q) *xmmDst, xmmSrc(xmm/m128)*
[Un]signed 128

MMI psub(b/h/w) *Dst, aSrc, bSrc* [Un]signed 128

This vector instruction is a parallel operation that subtracts each of the source bit blocks *bSrc (xmmSrc)* from *aSrc (xmmDst)* and stores the result in the destination *Dst (xmmDst)*.

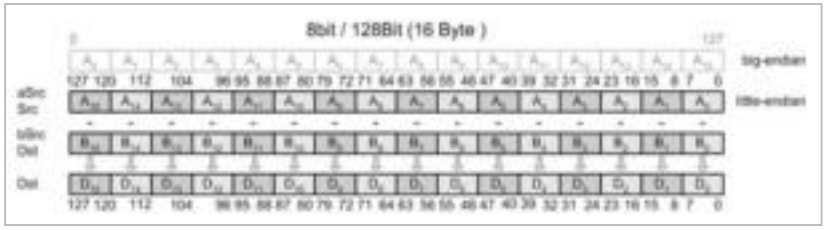
Note: Be careful here, as the register and operator ordering is as follows:

$$xmmDst_{(31...0)} = xmmDst_{(31...0)} - xmmSrc_{(31...0)} \quad D=-A$$
$$Dst_{(31...0)} = aSrc_{(31...0)} - bSrc_{(31...0)} \quad D=B-A$$

...and is contrary to normal, where *aSrc* is associated with *xmmSrc* and *bSrc* is associated with *xmmDst* and not the other way around, as in this particular case!

31...24	23...16	15...8	7...0	31...0
–126	91	–56	–96	0x82 5B C8 A0
– 12	–122	– –57	–114	–0x0C 7A C7 72
(0x76) 118	(0xE1) –31	(0x0) 1	**(0x2E) 46**	0x76 E1 01 2E

Vector Integer Subtraction with Saturation



$d_{(0...n-1)}=\text{Max}(\text{Min}(a_{(0...n-1)}-b_{(0...n-1)}, \text{HI\_LIMIT}), \text{LO\_LIMIT})$ $n=\{16, 8, 4, 2\}$

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	**MMI**

AltiVec	vsubu(b/h/w)s *Dst*, *aSrc, bSrc*	Unsigned	128
	vsubs(b/h/w)s *Dst*, *aSrc, bSrc*	Signed	
	vD=vec_subs(vA,vB)		
MMX	psubus(b/w) *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
	psubs(b/w) *mmDst*, *mmSrc (mm/m64)*	Signed	
SSE2	psubus(b/w) *xmmDst*, *xmmSrc(xmm/m128)*	Unsigned	128
	psubs(b/w) *xmmDst*, *xmmSrc(xmm/m128)*	Signed	
MMI	psubu(b/h/w) *Dst, aSrc, bSrc*	Unsigned	128
	psubs(b/h/w) *Dst, aSrc, bSrc*	Signed	128

This vector instruction is a parallel operation that subtracts each of the source bit blocks *bSrc (xmmSrc)* from *aSrc (xmmDst)* and stores the result in the destination *Dst (xmmDst)*.

Note: Be careful here, as the register and operator ordering is as follows:

$$\begin{aligned} \text{xmmDst}_{(31\ldots0)} &= \text{xmmDst}_{(31\ldots0)} - \text{xmmSrc}_{(31\ldots0)} && \text{D=-A} \\ \text{Dst}_{(31\ldots0)} &= \text{bSrc}_{(31\ldots0)} - \text{aSrc}_{(31\ldots0)} && \text{D=B-A} \end{aligned}$$

...and is contrary to normal, where *aSrc* is associated with *xmmSrc* and *bSrc* is associated with *xmmDst* and not the other way.

31...24	23...16	15...8	7...0	31...0
–126	91	–56	–96	0x82 5B C8 A0
– 12	–122	– –57	–114	–0x0C 7A C7 72
(0x80) –128	(0xE1) –31	(0x01) 1	**(0x80) –128**	0x76 E1 01 80

Vector Addition and Subtraction (Fixed Point)

For most of the number crunching in your games or tools, you will most likely use single-precision floating-point. For AI and other high-precision calculations, you may wish to use the higher precision double-precision, but it unfortunately only exists in scalar form for some of the processors covered by this book, except in the case of the SSE2, so functionality must be emulated in a sequential fashion whenever possible. Even with the higher precision, there is still a bit of an accuracy problem.

An alternative would be to use integer calculations in a fixed-point format of zero or more places. If the data size is large enough to contain the number, then there is no precision loss!

Pseudo Vec

These can get pretty verbose, since for fixed-point (integer) addition there would be support for 8-, 16-, and 32-bit data elements within a 128-bit vector, and these would be signed and unsigned, with and without saturation. The interesting thing about adding signed and unsigned numbers, other than the carry or borrow, is the resulting value will be exactly the same so the same equation can be used. For signed data, call with type casting or a macro:

```
#define paddB( D, A, B ) \
                        vmp_paddB((uint8*)(D), (uint8*)(A), (uint8*)(B))
```

For subtraction, just substitute a negative sign (–) for the positive sign (+) in the following examples.

16x8-bit Signed/Unsigned Addition

Listing 8-19: \chap08\pas\Pas.cpp

```
void vmp_paddB( uint8 *pbD,
                uint8 *pbA, uint8 *pbB )
{
  *(pbD+0) = *(pbA+0) + *(pbB+0);
  *(pbD+1) = *(pbA+1) + *(pbB+1);
           :
  *(pbD+14) = *(pbA+14) + *(pbB+14);
  *(pbD+15) = *(pbA+15) + *(pbB+15);
}
```

8x16-bit Signed/Unsigned Addition

Listing 8-20: \chap08\pas\Pas.cpp

```
void vmp_paddH( uint16 *phD,
                uint16 *phA, uint16 *phB )
{
  *(phD+0) = *(phA+0) + *(phB+0);
  *(phD+1) = *(phA+1) + *(phB+1);
           :
           :
  *(phD+6) = *(phA+6) + *(phB+6);
  *(phD+7) = *(phA+7) + *(phB+7);
}
```

4x32-bit Signed/Unsigned Addition

Listing 8-21: \chap08\pas\Pas.cpp

```
void vmp_paddW( uint32 *pwD,
                uint32 *pwA, uint32 *pwB )
{
  *(pwD+0) = *(pwA+0) + *(pwB+0);
  *(pwD+1) = *(pwA+1) + *(pwB+1);
  *(pwD+2) = *(pwA+2) + *(pwB+2);
  *(pwD+3) = *(pwA+3) + *(pwB+3);
}
```

16x8-bit Unsigned Addition with Saturation (0...255)

Listing 8-22: \chap08\pas\Pas.cpp

```
void vmp_paddusB( uint8 *pbD,
                  uint8 *pbA, uint8 *pbB )
{
  int v, cnt;
  uint8 *pD, *pA, *pB;

  pD = (uint8*)pbD;
  pA = (uint8*)pbA;
  pB = (uint8*)pbB;

  cnt = 16;
  do {
    v =  (int)(((uint)*pA++) + ((uint)*pB++));
    *pD++ = (uint8)(0xff & (v|((0xff-v)>>31)));
  } while (--cnt);
}
```

Pseudo Vec (X86)

Now examine these functions more closely using X86 Assembly. MMX and SSE2 have the biggest payoff, as 3DNow! and SSE are primarily for floating-point support. SSE2 is a top-of-the-line processor and not necessarily owned by the consumer (although the price is dropping), so that leaves the MMX as the more viable instruction set for integer support.

```
mov  ebx,pbB        ; Vector B
mov  eax,pbA        ; Vector A
mov  edx,pbD        ; Vector Destination
```

vmp_paddB (MMX) 16x8-bit

The following is a 16 x 8-bit addition, but substituting *psubb* for *paddb* will transform it into a subtraction.

Listing 8-23: vmp_x86\chap08\pas\PAddX86M.asm

```
movq  mm0,[ebx+0]   ; Read B Data {B7...B0}
movq  mm1,[ebx+8]   ;             {BF...B8}
movq  mm2,[eax+0]   ; Read A Data {A7...A0}
movq  mm3,[eax+8]   ;             {AF...A8}

paddb mm0,mm2       ; lower 64bits {A7+B7 ... A0+B0}
paddb mm1,mm3       ; upper 64bits {AF+BF ... A8+B8}

movq  [edx+0],mm0
movq  [edx+8],mm1
```

vmp_paddB (SSE2) 16x8-bit

For SSE, it is essentially the same function wrapper with mind aligned memory *movdqa* versus non-aligned memory *movdqu*.

Listing 8-24: vmp_x86\chap08\pas\PAddX86M.asm

```
movdqa xmm0,[ebx]   ; Read B Data {BF...B0}
movdqa xmm1,[eax]   ; Read A Data {AF...A0}
paddb  xmm0,xmm1    ; {vA+vB} 128bits {AF+BF ... A0+B0}
movdqa [edx],xmm0   ; Write D Data
```

vmp_paddH (MMX) 8x16-bit

Substitute *paddw* for *paddb* for addition and *psubw* for subtraction.

vmp_paddW (MMX) 4x32-bit

Substitute *paddd* for *paddb* for addition, and *psubd* for subtraction.

Confused? Check out the sample code on the CD.

Pseudo Vec (PowerPC)

These are some straightforward AltiVec library calls.

vmp_paddB (AltiVec) 16x8-bit Aligned

Listing 8-25: vmp_ppc\chap08\pas\PasAltivec.cpp

```
*(vector unsigned char *)pbD =
     vec_add( (*(vector unsigned char *)pbA),
              (*(vector unsigned char *)pbB) );
```

vmp_psubH (AltiVec) 8x16-bit Aligned

Listing 8-26: vmp_ppc\chap08\pas\PasAltivec.cpp

```
*(vector signed short *)phD =
     vec_subs( (*(vector signed short *)phA),
               (*(vector signed short *)phB) );
```

Pseudo Vec (MIPS)

This is where things get interesting. The VU coprocessor with VU0 and VU1 are both primarily floating-point vector processors, but they do have some 16-bit scalar capability and no 128-bit integer support. For 128-bit vector-based integer math, the enhanced MIPS processor with multimedia instructions has to be used instead.

vmp_paddB (MMI) 16x8-bit Aligned Memory

When dealing with aligned memory, this is a snap.

Listing 8-27: vmp_mips\chap08\pas\PAddMMI.s

```
lq     t1, 0(a1)         // pbA
lq     t2, 0(a2)         // pbB
nop                      // NOP - Load Delay Slot
paddb  t0, t1, t2
sq     t0, 0(a0)         // pbD
```

vmp_paddB (MMI) 16x8-bit Unaligned Memory

As usual, dealing with unaligned memory gets very complicated. As you may recall, the unaligned memory must be loaded in left and right halves 64 bits at a time and then blended together for the 128-bit value.

Listing 8-28: vmp_mips\chap08\pas\PAddMMI.s

```
ldl    t1, 7(a1)          // pbA  {63...0}
ldr    t1, 0(a1)
ldl    t3, 15(a1)         //     {127...64}
ldr    t3, 8(a1)

ldl    t2, 7(a2)          // pbB  {63...0}
ldr    t2, 0(a2)
ldl    t4, 15(a2)         //     {127...64}
ldr    t4, 8(a2)

pcpyld t1, t3, t1         // A  {127...0} Blended
pcpyld t2, t4, t2         // B  {127...0} Blended

paddb  t0, t1, t2         // D  {127...0}

pcpyud t1, t0, t0         // D  {63...0  127...64}

sdl    t0, 7(a0)          // pbD {63...0}
sdr    t0, 0(a0)
sdl    t1, 15(a0)         //    {127...64}
sdr    t1, 8(a0)
```

For other additions and subtractions, merely substitute {*paddsb, paddub, paddh, paddsh, padduh, paddw, psubb, psubsb, psubub, psubh, psubsh, psubuh, psubw*} for the instruction *paddb*, while using the same function wrapper.

For specific information, see a MIPS C790, PS2 Linux Kit, or devTool manual.

Exercises

1. Using only Boolean logic, how could two numbers be summed?
2. If your processor had no instructions for parallel subtraction, how would you find the difference of two numbers?
3. Invert the sign of the even-numbered elements of a signed 8-bit byte, 16-bit half-word, 32-bit word of a 128-bit data value using: a) Pseudo Vector C code, b) MMX, c) SSE2, d)AltiVec, and e) MIPS Multi-Media.
4. Same as exercise 3 but use odd-numbered elements.
5. Invert the sign of all the elements of four packed single-precision floating-point values.

6. You have been given a 4096-byte audio sample consisting of left and right channel components with a PCM (Pulse Coded Modulation) of unsigned 16 bit with 0x8000 as the baseline.

```
unsigned short leftStereo[1024], rightStereo[1024];
signed char Mono[???];
```

   a. How many bytes is the mixed sample?
   b. Write a mixer function to sum the two channels from stereo into mono and convert to a signed 8-bit sample. Choose your favorite processor.

Project

You now have enough information to write an SHA-1 algorithm discussed in Chapter 7 for your favorite processor. Write one!

Hint: Write the function code in C first!

Chapter 9
Vector Multiplication and Division

There are multitudes of variations of multiplicative mnemonic manipulations. It seems that almost every processor supports a slightly different scheme involving different integer word sizes, floating-point precision types, methods of rounding, saturations, etc. Fundamentally, despite its variations, it is very similar to and uses the same methodologies as the addition detailed in the last chapter.

Notice that the integer multiplication on the left in the following diagram requires more bits to contain the results of the operation, and thus different methods have been implemented to reduce that value back to its component size. The results of the floating-point multiplication on the right follow the rules of the IEEE-754 standard for binary floating-point arithmetic. The result of a multiplication is stored with no increase in data containment size, but there is a penalty as a loss of precision.

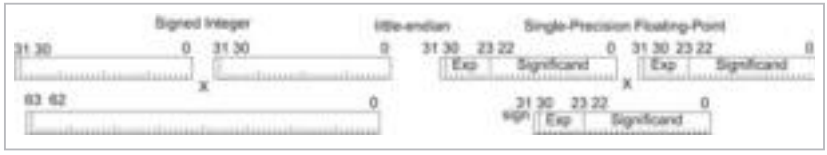


Figure 9-1: A visual representation that the product of two like-size integers requires double the space of one of the source integers. The product of two floating-point values stores nicely into the same size floating-point value.

Table 9-1:This table is an example of an integer and the loss of precision resulting from an equivalent floating-point operation.

Integer	Floating-Point	Expanded FP
3287565	3.28757e+006	3287570.0
* 593	593.0	* 593.0
1949526045	1.94953e+009	1949530000.0

CD Workbench Files: /Bench/*architecture*/chap09/*project*/*platform*

	architecture	Mul/Div	*project*	*platform*
PowerPC	/vmp_ppc/	**Float**	**/fmd/**	/mac9cw
X86	/vmp_x86/	**3D Float**	**/vmd3d/**	/vc6
MIPS	/vmp_mips	**4vec Float**	**/qvmd3d/**	/vc.net
		Integer	**/pmd/**	/devTool

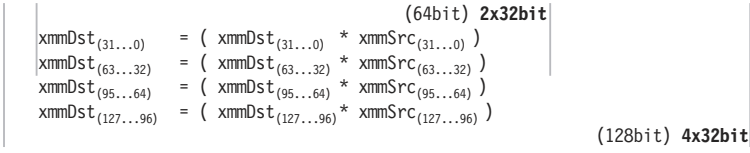
Floating-Point Multiplication

AltiVec	*MMX*	**SSE**	**SSE2**	**3DNow**	*3DMX+*	**MIPS**	*MMI*

NxSP-FP Multiplication

3DNow	pfmul *mmDst*, *mmSrc*	Single-Precision	64
SSE	mulps *xmmDst*, *xmmSrc(xmm/m128)*	Single-Precision	128
MIPS V	mul.ps *Dst, aSrc, bSrc*	Single-Precision	64

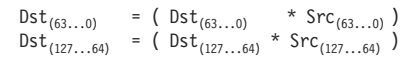
This SIMD instruction uses a 64-bit (128-bit) data path, and so two (four) operations occur in parallel. The product is calculated for each of the Real32 single-precision floating-point elements of the multiplicand *xmmDst* and the multiplier *xmmSrc* for each block, and the result is stored in each of the original Real32 of the destination *xmmDst*.



(Semi-Vector) DP-FP Multiplication

SSE2	mulpd *xmmDst*, *xmmSrc(xmm/m128)*	Double-Precision	128

This vector instruction uses a 128-bit data path, and so two operations occur in parallel. The product is calculated for each of the Real64 (double-precision floating-point) pairs of the multiplicand *xmmDst* and the multiplier *xmmSrc* for each block, and the result is stored in each of the original Real64 of the destination *xmmDst*.



SP-FP Scalar Multiplication

SSE mulss *xmmDst*, *scalar(xmm/m32)* Single-Precision 128

This vector instruction uses a 128-bit data path and only the first Real32 (single-precision floating-point) source *scalar* multiplier and the multiplicand *xmmDst*, and the result is stored in the original Real32 of the destination *xmmDst*.

```
Dst(31...0)    = ( Dst(31...0) * Src(31...0) )
Dst(127...32)  = remains unchanged.
```

DP-FP Scalar Multiplication

SSE2 mulsd *xmmDst*, *scalar(xmm/m64)* Double-Precision 128

This vector instruction uses a 128-bit data path and only the first Real64 (double-precision floating-point) source *scalar* multiplier and the multiplicand *xmmDst,* and the result is stored in the original Real64 of the destination *xmmDst*.

```
Dst(63...0)    = ( Dst(63...0) * Src(63...0) )
Dst(127...64)  = remains unchanged.
```

NxSP-FP Multiplication — Add

AltiVec vmaddfp *Dst, aSrc, bSrc, cSrc* Single-Precision 128
vD = vec_madd(vA, vB, vC)

MIPS V madd.ps *Dst, cSrc, aSrc, bSrc* Single-Precision 64

```
Dst =    aSrc *    bSrc    +    cSrc
```

This vector operation calculates the Real32 (single-precision floating-point) products of the source four multiplicand *aSrc* and the multiplier *bSrc* for each bit block. The result is then summed with *cSrc* and stored in each Real32 element of the related destination *Dst*.

```
Dst(31...0)    = ( aSrc(31...0)   * bSrc(31...0) )   + cSrc(31...0)
Dst(63...32)   = ( aSrc(63...32)  * bSrc(63...32) )  + cSrc(63...32)
Dst(95...64)   = ( aSrc(95...64)  * bSrc(95...64) )  + cSrc(95...64)
Dst(127...96)  = ( aSrc(127...96) * bSrc(127...96) ) + cSrc(127...96)
```

There is no single-precision floating-point multiply on AltiVec, but it can be emulated with the summation of a vector zero.

```
static vector float vZero = (vector float) (0.0, 0.0, 0.0, 0.0);

Dst = aSrc * bSrc + 0
```

SP-FP Multiplication — Subtract with Rounding

AltiVec vnmsubfp *Dst, aSrc, bSrc, cSrc* Single-Precision 128
vD = vec_nmsub(vA, vB, vC)

This vector operation calculates the Real32 (single-precision floating-point) products of the source four multiplicand *aSrc* and the multiplier *bSrc* for each bit block. The result is then subtracted by *cSrc* and rounded, and the value is stored in the related Real32 destination *Dst*.

$$
\begin{aligned}
Dst_{(31\ldots0)} &= (aSrc_{(31\ldots0)} * bSrc_{(31\ldots0)}) - cSrc_{(31\ldots0)} \\
Dst_{(63\ldots32)} &= (aSrc_{(63\ldots32)} * bSrc_{(63\ldots32)}) - cSrc_{(63\ldots32)} \\
Dst_{(95\ldots64)} &= (aSrc_{(95\ldots64)} * bSrc_{(95\ldots64)}) - cSrc_{(95\ldots64)} \\
Dst_{(127\ldots96)} &= (aSrc_{(127\ldots96)} * bSrc_{(127\ldots96)}) - cSrc_{(127\ldots96)}
\end{aligned}
$$

Vector (Float) Multiplication — Add

Vector floating-point multiplication is one of the mathematical equations that you will tend to use the most in your video games whether as a tri or quad float vector.

Pseudo Vec

The multiplication of vectors is similar to the addition of vectors.

Single-Precision Float Multiplication

Listing 9-1: \chap09\fmd\Fmd.cpp

```
void vmp_FMul( float *pfD, float fA, float fB )
{
  *pfD = fA * fB;
}
```

Single-Precision Vector Float Multiplication

Listing 9-2: \chap09\vmd3d\Vmd3D.cpp

```
void vmp_VecMul(vmp3DVector * const pvD,
          const vmp3DVector * const pvA,
          const vmp3DVector * const pvB)
{
  pvD->x = pvA->x * pvB->x;
  pvD->y = pvA->y * pvB->y;
  pvD->z = pvA->z * pvB->z;
}
```

Single-Precision Quad Vector Float Multiplication

Listing 9-3: \chap09\qvmd3d\QVmd3D.cpp

```
void vmp_QVecMul(vmp3DQVector * const pvD,
           const vmp3DQVector * const pvA,
           const vmp3DQVector * const pvB)
{
  pvD->x = pvA->x * pvB->x;
  pvD->y = pvA->y * pvB->y;
  pvD->z = pvA->z * pvB->z;
  pvD->w = pvA->w * pvB->w;
}
```

Single-Precision Quad Vector Float Multiplication — Add

The multiplication-add (*madd*) is merely a multiplication followed by a summation. Some processors, such as AltiVec, do not support just a stand-alone multiplication. Some support only the multiplication-add. Some support both. But it is much more efficient to call a madd type instruction when appropriate, instead of separately.

Listing 9-4: \chap09\qvmd3d\QVmd3D.cpp

```
void vmp_QVecMAdd(vmp3DQVector * const pvD,
            const vmp3DQVector * const pvA,
            const vmp3DQVector * const pvB)
{
  pvD->x = pvA->x * pvB->x + pvC->x;
  pvD->y = pvA->y * pvB->y + pvC->y;
  pvD->z = pvA->z * pvB->z + pvC->z;
  pvD->w = pvA->w * pvB->w + pvC->w;
}
```

Pseudo Vec (X86)

Now examine these functions more closely using X86 assembly. 3DNow! supports 64-bit data, so two loads must be handled simultaneously as well as two stores, but it is a simple matter of adding the two pairs of floats to each other.

```
mov   eax,vA    ; Vector A
mov   ebx,vB    ; Vector B
mov   edx,vD    ; Vector Destination
```

vmp_VecMul (3DNow!)

Listing 9-5: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
movq  mm0,[ebx]                ;vB.xy  {By Bx}
movq  mm1,[eax]                ;vA.xy  {Ay Ax}
movd  mm2,(vmp3DVector PTR [ebx]).z  ; {0  Bz}
movd  mm3,(vmp3DVector PTR [eax]).z  ; {0  Az}
pfmul mm1,mm0                        ; {AyBy AxBx}
pfmul mm3,mm2                        ; { 0   AzBz}
movq  [edx],mm1                      ; {AyBy AxBx}
movd  (vmp3DVector PTR [edx]).z,mm3  ; {     AzBz}
```

As you may have noticed, the vector only loaded one float instead of two, set the second to zero, calculated the product, and then only wrote the three values back to memory.

vmp_QVecMul (3DNow!)

Listing 9-6: vmp_x86\chap09\qvmd3d\QVmd3DX86M.asm

```
movq  mm0,[ebx+0]             ;vB.xy   {By Bx}
movq  mm1,[eax+0]             ;vA.xy   {Ay Ax}
movq  mm2,[ebx+8]             ;vB.zw   {Bw Bz}
movq  mm3,[eax+8]             ;vA.zw   {Aw Az}
pfmul mm1,mm0                 ;    {AyBy AxBx}
pfmul mm3,mm2                 ;    {AwBw AzBz}
movq  [edx+0],mm1             ;    {AyBy AxBx}
movq  [edx+8],mm3             ;    {AwBw AzBz}
```

vmp_QVecMAdd (3DNow!)

For *madd*, the addition needs to be handled separately.

Listing 9-7: vmp_x86\chap09\qvmd3d\QVmd3DX86M.asm

```
mov   ecx,vC                  ; Vector C
movq  mm0,[ebx+0]             ;vB.xy   {By Bx}
movq  mm1,[eax+0]             ;vA.xy   {Ay Ax}
movq  mm4,[ecx+0]             ;vC.xy   {Cy Cx}
movq  mm2,[ebx+8]             ;vB.zw   {Bw Bz}
movq  mm3,[eax+8]             ;vA.zw   {Aw Az}
movq  mm5,[ecx+8]             ;vC.zw   {Cw Cz}

pfmul mm1,mm0                 ; {AyBy AxBx}
pfmul mm3,mm2                 ; {AwBw AzBz}
pfadd mm1,mm4                 ;{AyBy+Cy AxBx+Cx}
pfadd mm3,mm5                 ;{AwBw+Cw AzBz+Cz}

movq  [edx+0],mm1             ;{AyBy+Cy AxBx+Cx}
movq  [edx+8],mm3             ;{AwBw+Cw AzBz+Cz}
```

vmp_VecMul (SSE)

The SSE processor in the following code snippet can load 128 bits at a time so the entire 96-bit vector can be loaded at once including an extra 32 bits. This introduces a problem of contamination when the 96-bit value is written to memory as 128 bits. The solution is to read those destination bits, preserve the upper 32 bits through bit masking and blending, and write the newly merged 128 bits. Keep in mind efficient memory organization and memory tail padding discussed earlier. Data can be misaligned or aligned, but 128-bit alignment is preferable. Only aligned memory will be discussed, but for SSE, keep in mind the use of *movups* instead of *movaps* when memory alignment cannot be guaranteed.

vmp_VecMul (SSE) Aligned

Listing 9-8: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
movaps xmm2,[edx]              ;vD.###w {Dw #  #  #}
movaps xmm1,[ebx]              ;vB.xyz# {# Bz By Bx}
movaps xmm0,[eax]              ;vA.xyz# {# Az Ay Ax}
andps  xmm1,OWORD PTR lomsk96  ;        {0 Az Ay Ax}
andps  xmm2,OWORD PTR himsk32  ;        {Dw 0  0  0}
mulps  xmm0,xmm1               ; {## AzBz AyBy AxBx}
andps  xmm0,OWORD PTR lomsk96  ; limit –0
orps   xmm0,xmm2               ; {Dw AzBz AyBy AxBx}
movaps [edx],xmm0              ; {Dw AzBz AyBy AxBx}
```

vmp_QVecMul (SSE) Aligned

Listing 9-9: vmp_x86\chap09\qvmd3d\QVmd3DX86M.asm

```
movaps xmm1,[ebx]              ; vB.xyzw {Bw Bz By Bx}
movaps xmm0,[eax]              ; vA.xyzw {Aw Az Ay Ax}
mulps  xmm0,xmm1               ; {AwBw AzBz AyBy AxBx}
movaps [edx],xmm0              ; {AwBw AzBz AyBy AxBx}
```

vmp_QVecMAdd (SSE) Aligned

For *madd*, the summation is an appended instruction as compared to the previous vector multiplication.

Listing 9-10: vmp_x86\chap09\qvmd3d\QVmd3DX86M.asm

```
movaps xmm0,[eax]              ; vA.xyzw {Aw Az Ay Ax}
movaps xmm1,[ebx]              ; vB.xyzw {Bw Bz By Bx}
movaps xmm2,[ecx]              ; vC.xyzw {Cw Cz Cy Cx}
mulps  xmm0,xmm1               ; {AwBw AzBz AyBy AxBx}
addps  xmm0,xmm2               ; {AwBw+Cw ... AxBx+Cx}
movaps [edx],xmm0              ; {AwBw+Cw ... AxBx+Cx}
```

Pseudo Vec (PowerPC)

static vector float vZero = (vector float) (0.0, 0.0, 0.0, 0.0);

vmp_VecMul (AltiVec) Aligned

This is an emulated multiplication, as AltiVec only supports a multiply-add. So to resolve the addition, a summation with zero is utilized, since a*b+0 = a*b.

Listing 9-11: vmp_ppc\chap09\vmd3d\Vmd3DAltivec.cpp

```
vector float vD, vA, vB;

((vmp3DQVector *) &vA)->x = pvA->x;
((vmp3DQVector *) &vA)->y = pvA->y;
((vmp3DQVector *) &vA)->z = pvA->z;

((vmp3DQVector *) &vB)->x = pvB->x;
((vmp3DQVector *) &vB)->y = pvB->y;
((vmp3DQVector *) &vB)->z = pvB->z;

vD = vec_madd( vA, vB, vZero );  // vmaddfp

pvD->x = ((vmp3DQVector *) &vD)->x;
pvD->y = ((vmp3DQVector *) &vD)->y;
pvD->z = ((vmp3DQVector *) &vD)->z;
```

Here we have a quandary. The fourth element {W} is a case of garbage in, garbage out (GIGO), but since the data is aligned, do we keep code smaller by using the unaligned vector multiply, or do we go for speed by using the direct 128-bit reads?

Listing 9-12: vmp_ppc\chap09\qvmd3d\QVmd3DAltivec.cpp

```
vD = vec_madd(
            (*(vector float *)pvA),
            (*(vector float *)pvB), vZero );

pvD->x = ((vmp3DQVector *) &vD)->x;
pvD->y = ((vmp3DQVector *) &vD)->y;
pvD->z = ((vmp3DQVector *) &vD)->z;
```

When a fourth element exists, it is a very simple operation. The extra visible verbiage is merely casting to convert the cross-platform generic type definitions to that of the AltiVec library.

vmp_QVecMul (AltiVec) Aligned

Listing 9-13: vmp_ppc\chap09\qvmd3d\QVmd3DAltivec.cpp

```
*(vector float *)pvD = vec_madd(
           (*(vector float *)pvA),
           (*(vector float *)pvB), vZero );
```

vmp_QVecMAdd (AltiVec) Aligned

Does the following look familiar? Only the summation parameter needs to be modified from zero to an actual value.

Listing 9-14: vmp_ppc\chap09\qvmd3d\QVmd3DAltivec.cpp

```
*(vector float *)pvD = vec_madd(   (*(vector float *)pvA),
                                   (*(vector float *)pvB),
                                   (*(vector float *)pvC) );
```

Pseudo Vec (MIPS)

For the embedded MIPS programmers, the single-precision floating-point is handled by the MIPS V-paired product instruction *mul.ps*.

vmp_VecMul (MIPS V) Aligned

The MIPS64 architecture using the MIPS V instruction set handles a vector by using the paired single-precision floating-point product calculation for the first element pair {XY} and a scalar product {Z} for the third element.

Listing 9-15: vmp_mips\chap09\vmd3D\Vmd3DMips.s

```
ldc1   $f4, 0(a1)     // {Ay Ax} Mem to PS_FP
ldc1   $f6, 0(a2)     // {By Bx}
lwc1   $f5, 8(a1)     // {   Az}
lwc1   $f7, 8(a2)     // {   Bz}

mul.ps $f4, $f4, $f6  // {AyBy AxBx}
mul.s  $f5, $f5, $f7  // {     AzBz}

sdc1   $f4, 0(a0)     // {Dy Dx}
swc1   $f5, 8(a0)     // {   Dz}
```

vmp_QVecMul (MIPS V) Aligned

For quad vectors, two 64-bit paired single-precision floating-point products ({XY} and {ZW}) are utilized.

Listing 9-16: vmp_mips\chap09\qvmd3D\QVmd3DMips.s

```
  ldc1   $f4, 0(a1)     // {Ay Ax} Mem to PS_FP
  ldc1   $f6, 0(a2)     // {By Bx}
  ldc1   $f5, 8(a1)     // {Aw Az}
  ldc1   $f7, 8(a2)     // {Bw Bz}

  mul.ps $f4, $f4, $f6  // {AyBy AxBx}
  mul.ps $f5, $f5, $f7  // {AwBw AzBz}

  sdc1   $f4, 0(a0)     // {Dy Dx}
  sdc1   $f5, 8(a0)     // {Dw Dz}
```

vmp_QVecMAdd (MIPS V) Aligned

Note the sum value of C_{xyzw} is the first source argument and not the last, as with most processors!

Listing 9-17: vmp_mips\chap09\qvmd3D\QVmd3DMips.s

```
  ldc1   $f4, 0(a1)     // {Ay Ax} Mem to PS_FP
  ldc1   $f6, 0(a2)     // {By Bx}
  ldc1   $f5, 8(a1)     // {Aw Az}
  ldc1   $f7, 8(a2)     // {Bw Bz}
  ldc1   $f8, 0(a3)     // {Cy Cx}
  ldc1   $f9, 8(a3)     // {Cw Cz}
;          D    C    A    B
  madd.ps $f4, $f8, $f4, $f6  // {AyBy+Cy AxBx+Cx}
  madd.ps $f5, $f9, $f5, $f7  // {AwBw+Cw AzBz+Cz}

  sdc1   $f4, 0(a0)     // {Dy Dx}
  sdc1   $f5, 8(a0)     // {Dw Dz}
```

vmp_QVecMul (VU0) Aligned

For specific information, see your PS2 Linux Kit or devTool manual.

Vector Scalar Multiplication

The scalar multiplication of vectors is also a relatively simple matter for vector math instructions to handle, just like the scalar addition and subtraction of vectors. The trick is to replicate the scalar so it appears like a second vector.

Pseudo Vec

Single-Precision Vector Float Multiplication with Scalar

This function multiplies a scalar with each element of a vector. A scalar has multiple uses, but the primary is in the use of "scaling" a vector. A scalar of one would result in the same size. Two would double the length of the vector, etc.

Listing 9-18: \chap09\vmd3d\Vmd3D.cpp

```
void vmp_VecScale(vmp3DVector * const pvD,
            const vmp3DVector * const pvA,
            float fScalar)
{
  pvD->x = pvA->x * fScalar;
  pvD->y = pvA->y * fScalar;
  pvD->z = pvA->z * fScalar;
}
```

Single-Precision Quad Vector Float Multiplication with Scalar

Listing 9-19: \chap09\qvmd3d\QVmd3D.cpp

```
void vmp_QVecScale(vmp3DQVector * const pvD,
             const vmp3DQVector * const pvA,
             float fScalar )
{
  pvD->x = pvA->x * fScalar;
  pvD->y = pvA->y * fScalar;
  pvD->z = pvA->z * fScalar;
  pvD->w = pvA->w * fScalar;
}
```

Pseudo Vec (X86)

```
  mov     eax,vA            ; Vector A
  mov     edx,vD            ; Vector Destination
```

vmp_VecScale (3DNow!)

The 32-bit scalar is unpacked into a pair and then treated similar to the vector multiplication of two vectors.

Listing 9-20: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
  movd    mm0,fScalar               ; fScalar {0 s}
  punpckldq mm0,mm0                 ;         {s s}
  movq    mm1,[eax]                 ;vA.xy  {Ay Ax}
  movd    mm2,(vmp3DVector PTR [eax]).z  ;   {0  Az}
```

```
pfmul   mm1,mm0                         ; {Ays Axs}
pfmul   mm2,mm0                         ; {Os  Azs}
movq    [edx],mm1                       ; {Ays Axs}
movd   (vmp3DVector PTR [edx]).z,mm2    ;   {  Azs}
```

vmp_VecScale (SSE) Aligned

The SSE version of the code is changed from a 64-bit load to a 128-bit load, but the principals remain the same.

Listing 9-21: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
pxor   xmm1,xmm1                    ; {0 0 0 0}
movss  xmm1,fScalar                 ; {0 0 0 s}
movaps xmm2,[edx]                   ; {Dw # # #}
movaps xmm0,[eax]                   ;vA.xyz#  {# Az Ay Ax}
shufps xmm1,xmm1,11000000b          ; 3 0 0 0 {0  s  s  s}
andps  xmm2,OWORD PTR himsk32       ; {Dw 0  0  0}

mulps  xmm0,xmm1                    ; {#  Azs Ays Axs}
andps  xmm0,OWORD PTR lomsk96       ; {0  Azs Ays Axs}
orps   xmm0,xmm2                    ; {Dw Azs Ays Axs}
movaps [edx],xmm0                   ; {Dw Azs Ays Axs}
```

vmp_QVecScale (SSE) Aligned

Listing 9-22: vmp_x86\chap09\qvmd3d\QVmd3DX86M.asm

```
movss  xmm1,fScalar                ;         {0 0 0 s}
movaps xmm0,[eax]                  ;vA.xyzw {Aw Az Ay Ax}
shufps xmm1,xmm1,00000000b         ; 0 0 0 0 {s s s s}
mulps  xmm0,xmm1                   ; {Aws Azs Ays Axs}
movaps [edx],xmm0                  ; {Aws Azs Ays Axs}
```

Pseudo Vec (PowerPC)

vmp_QVecScale (AltiVec) Aligned

As mentioned earlier, the multiply-add with zero is necessary for this operation.

Listing 9-23: vmp_ppc\chap09\vmd3d\Vmd3DAltivec.cpp

```
vector float vB;

((vmp3DQVector *) &vB)->x = fScalar;
((vmp3DQVector *) &vB)->y = fScalar;
((vmp3DQVector *) &vB)->z = fScalar;
((vmp3DQVector *) &vB)->w = fScalar;

*(vector float *)pvD = vec_madd(
          (*(vector float *)pvA), vB, vZero );
```

Pseudo Vec (MIPS)

vmp_QVecScale (MIPS V) Aligned

Note that this is written using the *pll* instruction for a little endian implementation.

Listing 9-24: vmp_mips\chap09\qvmd3D\QVmd3DMips.s

```
lwc1    $f8, 0(a2)    // fScalar
ldc1    $f4, 0(a1)    // {Ay Ax}
ldc1    $f5, 8(a1)    // {Aw Az}

pll.ps  $f8, $f8, $f8 // {   s    s} Replicate Scalar
mul.ps  $f4, $f4, $f8 // {Ay*s Ax*s}
mul.ps  $f5, $f5, $f8 // {Aw*s Az*s}

sdc1    $f4, 0(a0)    // {Dy Dx}
sdc1    $f5, 8(a0)    // {Dw Dz}
```

Graphics 101

Dot Product

A dot product, also known as an *inner product* of two vectors, is the summation of the results of the product for each of their {xyz} elements, resulting in a scalar. Not to oversimplify it, but this scalar is equal to 0 if the angle made up by the two vectors are perpendicular (=90º), positive if the angle is acute (<90º), and negative if the angle is obtuse (>90º).

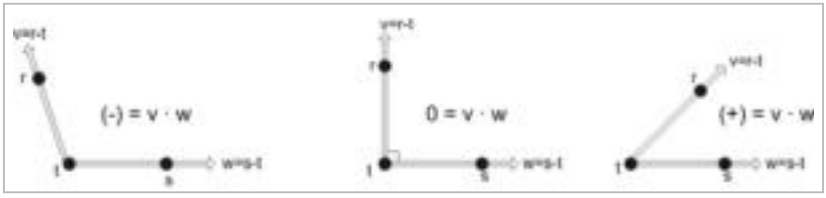


Figure 9-2: Dot product (inner product). A positive number is an acute angle, zero is perpendicular, and negative is an obtuse angle.

$v = \{v_1, v_2, v_3\}$ and $w = \{w_1, w_2, w_3\}$

These are vectors that produce a scalar defined by $v \cdot w$ when their products are combined. The dot product is represented by the following equation:

$$v \cdot w = v_1w_1 + v_2w_2 + v_3w_3$$

The equation resolves to the following simplified form:

$$D = A_xB_x + A_yB_y + A_zB_z \qquad D = Ax*Bx + Ay*By + Az*Bz$$

Pseudo Vec

Single-Precision Dot Product

Listing 9-25: \chap09\vmd3d\Vmd3D.cpp

```
void vmp_DotProduct(float * const pfD,
        const vmp3DVector * const pvA,
        const vmp3DVector * const pvB)
{
  *pfD = pvA->x * pvB->x
       + pvA->y * pvB->y
       + pvA->z * pvB->z;
}
```

This is one of my favorite equations because it does not slice, dice, or chop, but it culls, illuminizes, simplifies, and cosineizes, (not a real word, but you know what I mean). It is the Sledge-O-Matic!!! Well, it's not quite comedian Gallagher's watermelon disintegration kitchen utensil, but it does do many things, and so it is just as useful.

Note in Figure 9-2 that if the resulting scalar value is positive (+), the vectors are pointing in the same general direction. If it's zero (0), they are perpendicular from each other, and if negative (–), they are pointed in opposite directions.

Before explaining further, it should be pointed out that to keep 3D graphic algorithms as simple as possible, the three vertices for each polygon should all be ordered in the same direction. For example, by using the left-hand rule and keeping all the vertices of a visible face in a clockwise direction, such as in the following figure, back face culling will result. If all visible face surfaces use this same orientation and if the vertices occur in a counterclockwise direction, they are back faced and thus pointing away and need not be drawn, which saves render time.

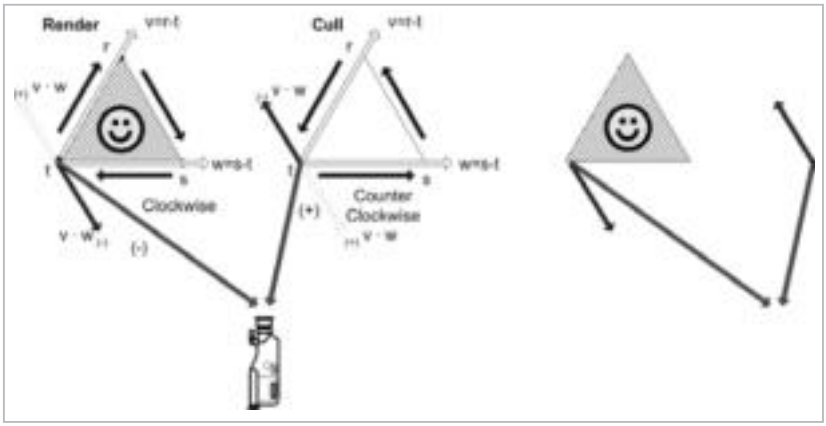


Figure 9-3: This is a face culling mechansim where if the angle between the camera and the perpendicular to the face plane is obtuse, the face is pointed away from the camera and can be culled.

On the contrary, if polygons are arranged in a counterclockwise orientation, the inverse occurs, where a positive value is drawn and a negative value is culled. Keep in mind, however, that most software algorithms keep things in a clockwise orientation.

By calculating the dot product of the normal vector of the polygon with a vector between one of the polygon's vertices and the camera, it can be determined if the polygon is back facing and in need of culling. A resulting positive value indicates that the face is pointed away, hence back facing, and can be culled and not rendered; a negative value is a face oriented toward the camera and thus visible.

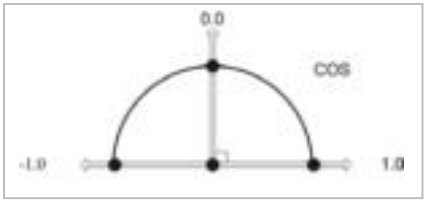


Figure 9-4: Cosine of two intersecting lines

Another use for the dot product equation is that it is also the cosine of the angle. This will be discussed in detail later in Chapter 11, "A Wee Bit o' Trig," but a quick note here is that the cosine is returned by dividing the dot product by the product of the magnitudes of the two vectors. Note that v and w are vectors and $|v|$ and $|w|$ are their magnitudes.

$$\text{Cos}\,\theta = \frac{A_xB_x + A_yB_y + A_zB_z}{\sqrt{(A_x^2 + A_y^2 + A_z^2)} \times \sqrt{(B_x^2 + B_y^2 + B_z^2)}} = \frac{v \cdot w}{|v| \times |w|}$$

Here we use standard trigonometric formulas, such as:

$$1 = Cos^2 + Sin^2$$

and sine and other trigonometric results can be calculated.

So the good stuff is yet to come!

Pseudo Vec (X86)

vmp_DotProduct (3DNow!)

The 3DNow! instruction set uses the 64-bit MMX registers, but 64-bit memory alignment cannot be guaranteed. In this case, it is typically better to handle memory access as individual 32-bit floats and then unpack into 64-bit pairs, process, and save individually as 32 bit. The instruction *pfacc* is unique, as it allows the high/low 32 bits to be summed with each other, within each of the vectors.

Listing 9-26: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
mov    ebx,vB                          ; Vector B
mov    eax,vA                          ; Vector A
mov    edx,vD               ; Vector Destination

movd   mm0,(vmp3DVector PTR [ebx]).z ;   {0 Bz}
movd   mm1,(vmp3DVector PTR [eax]).z ;   {0 Az}
movq   mm2,[ebx]                     ;  {By Bx}
movq   mm3,[eax]                     ;  {Ay Ax}
pfmul  mm0,mm1                      ; {00 BzAz}
pfmul  mm2,mm3              ;       {ByAy BxAx}
pfacc  mm2,mm2           ; {ByAy+BxAx ByAy+BxAx}
pfadd  mm0,mm2   ; {ByAy+BxAx+0 ByAy+BxAx+BzAz}
movd   [edx],mm0        ; Save {ByAy+BxAx+BzAz}
```

vmp_DotProduct (SSE) Aligned

The SSE instruction loads the 96-bit vector value using a 128-bit XMM register. The operation entails the multiplication of the {xyz} pairs from both vectors. The data is swizzled to allow scalar additions, and then the 32-bit single-precision float scalar result is written to memory.

Listing 9-27: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
movaps xmm1,[ebx]         ;vB.xyz# {# Bz By Bx}
movaps xmm0,[eax]         ;vA.xyz# {# Az Ay Ax}
mulps  xmm0,xmm1        ; {A#B# AzBz AyBy AxBx}
movaps xmm1,xmm0
movaps xmm2,xmm0
unpckhps xmm0,xmm0          ; {A#B# A#B# AzBz AzBz}
shufps xmm1,xmm1,11100001b ; {A#B# AzBz AxBx AyBy}
addss  xmm2,xmm0       ; {A#B# AzBz AxBx AzBz+AxBx}
```

```
addss  xmm2,xmm1      ; {A#B# AzBz AxBx AzBz+AxBx+AyBy}
movss  [edx],xmm2              ; Save {AzBz+AxBx+AyBy}
```

Pseudo Vec (PowerPC)

vmp_DotProduct (AltiVec) Unaligned

The AltiVec instruction set uses the 128-bit registers, which must be memory aligned for the correct data to be accessed. In the case where memory alignment cannot be guaranteed, each field {x,y,z} must be individually aligned and then loaded into the AltiVec register for processing.

This function is the start of where things get interesting. A dot product is based upon a vector and not a quad vector, but realistically we need to have two versions because they both still deal with only three floats, and the fourth is ignored. But in a vector, the fourth float cannot be guaranteed to be a float for the initial multiplication, so then it either needs to be treated very similar to that of unaligned memory or have extra code to support a faster version.

Listing 9-28: vmp_ppc\chap09\vmd3d\Vmd3DAltivec.cpp

```
vector float vD, vA, vB;

((vmp3DQVector *) &vA)->x = pvA->x;
((vmp3DQVector *) &vA)->y = pvA->y;
((vmp3DQVector *) &vA)->z = pvA->z;

((vmp3DQVector *) &vB)->x = pvB->x;
((vmp3DQVector *) &vB)->y = pvB->y;
((vmp3DQVector *) &vB)->z = pvB->z;

vD = vec_madd( vA, vB, vZero );

*pfD = ((vmp3DVector *) &vD)->x
     + ((vmp3DVector *) &vD)->y
     + ((vmp3DVector *) &vD)->z;
```

The summation is pretty much straightforward, as only the first three floats are summed. Loading is much easier in the case where memory is guaranteed to be aligned, and the fourth elements (of both vectors) that are not needed for the solution are known to be floats.

vmp_DotProduct (AltiVec) Aligned

Listing 9-29: vmp_ppc\chap09\vmd3d\Vmd3DAltivec.cpp

```
  vector float vD;

  vD = vec_madd( (*(vector float *)pvA),
                 (*(vector float *)pvB), vZero );

 *pfD = ((vmp3DQVector *) &vD)->x    // x+y+z
      + ((vmp3DQVector *) &vD)->y
      + ((vmp3DQVector *) &vD)->z;
```

Keep this in mind when organizing your database files for use in your mathematical calculations.

Pseudo Vec (MIPS)

For embedded processors, product summing of upper/lower elements is a lot easier with the MIPS-3D add-reduction (*addr*) instruction.

vmp_DotProduct (MIPS V and MIPS-3D) Aligned

Listing 9-30: vmp_mips\chap09\vmd3D\Vmd3DMips.s

```
  ldc1    $f4, 0(a1)    // {Ay Ax} Mem to PS_FP
  ldc1    $f6, 0(a2)    // {By Bx}
  lwc1    $f5, 8(a1)    // {   Az}
  lwc1    $f7, 8(a2)    // {   Bz}

  mul.ps  $f4, $f4, $f6 // {AyBy AxBx}
  mul.s   $f5, $f5, $f7 // {     AzBz}
  addr.ps $f4, $f4, $f4 // {AyBy+AxBx AyBy+AxBx}
  add.s   $f5, $f5, $f4 // {     AzBz+AyBy+AxBx}

  swc1    $f5, 0(a0)
```

Graphics 101

Cross Product

A cross product (also known as the *outer product*) of two vectors is a third vector perpendicular to the plane of the two original vectors. The two vectors define two sides of a polygon face, and their cross product points away from that face.

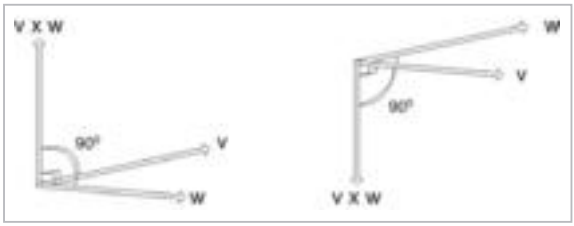


Figure 9-5: Cross product (outer product) — the perpendicular to the two vectors v and w

$v = \{v_1, v_2, v_3\}$ and $w = \{w_1, w_2, w_3\}$ are vectors of a plane denoted by matrix R^3. The cross product is represented by the following equation. Note that the standard basis vectors are i=(1,0,0) j=(0,1,0), k=(0,0,1).

$$v \times w = (v_2w_3 - v_3w_2)\mathbf{i} - (v_1w_3 - v_3w_1)\mathbf{j} + (v_1w_2 - v_2w_1)\mathbf{k}$$

$$\det \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v_1 & v_2 & v_3 \\ w_1 & w_2 & w_3 \end{bmatrix} \quad \text{thus} \quad \begin{bmatrix} v_2w_3 - v_3w_2 \\ v_3w_1 - v_1w_3 \\ v_1w_2 - v_2w_1 \end{bmatrix}$$

The equation resolves to the following simplified form:

$D_x = A_yB_z - A_zB_y$ Dx = Ay*Bz – Az*By;
$D_y = A_zB_x - A_xB_z$ Dy = Az*Bx – Ax*Bz;
$D_z = A_xB_y - A_yB_x$ Dz = Ax*By – Ay*Bx;

Note the following simple vector structure is actually 12 bytes, which will pose a data alignment problem for SIMD operations.

Listing 9-31: \inc???\vmp3D.h

```
typedef struct
{
   float x;
   float y;
   float z;
} vmp3DVector;
```

One method is to use individual single-precision floating-point calculations with which you may already be familiar. With this in mind, examine the following simple C language function to implement it. Note the use of the temporary floats *x, y* to prevent the resulting solutions of each field {x,y,z} from affecting either source *pvA* or *pvB* when the destination *pvD* is also a source.

Listing 9-32: \chap09\vmd3d\Vmd3D.cpp

```
void vmp_CrossProduct( vmp3DVector* const pvD,
   const vmp3DVector* pvA, const vmp3DVector* pvB)
{
 float x, y;
       x = pvA->y * pvB->z - pvA->z * pvB->y;
```

```
      y = pvA->z * pvB->x - pvA->x * pvB->z;
  pvD->z = pvA->x * pvB->y - pvA->y * pvB->x;
  pvD->x = x;
  pvD->y = y;
}
```

vmp_CrossProduct (3DNow!)

The 3DNow! instruction set uses the 64-bit MMX registers, but 64-bit memory alignment cannot be guaranteed. In this case, it is typically better to handle memory access as individual 32-bit floats and unpack into 64-bit pairs, process, and save individually as 32 bit. This one is kind of big, so there are extra blank lines to help separate the various logic stages. It is not optimized to make it more readable.

Listing 9-33: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
mov    ebx,vB                          ; Vector B
mov    eax,vA                          ; Vector A
mov    edx,vD               ; Vector Destination

movd mm0,(vmp3DVector PTR [ebx]).x  ;vB.x {0 Bx}
movd mm1,(vmp3DVector PTR [ebx]).y  ;vB.y {0 By}
movd mm2,(vmp3DVector PTR [ebx]).z  ;vB.z {0 Bz}
movd mm3,(vmp3DVector PTR [eax]).x  ;vA.x {0 Ax}
movd mm4,(vmp3DVector PTR [eax]).y  ;vA.y {0 Ay}
movd mm5,(vmp3DVector PTR [eax]).z  ;vA.z {0 Az}

pfmul mm4,mm0                    ;vB.xy {0 AyBx}
punpckldq mm0,mm1                ;       {By Bx}

movd  mm1,(vmp3DVector PTR [eax]).y ;vA.y { Ay}
movd  mm6,(vmp3DVector PTR [ebx]).y ;vB.y { By}

punpckldq mm2,mm2                ;       {Bz Bz}
punpckldq mm3,mm1                ;       {Ay Ax}
punpckldq mm5,mm5                ;       {Az Az}

pfmul mm2,mm3                ; vA.xy {BzAy BzAx}
pfmul mm5,mm0                ; vB.xy {AzBy AzBx}
pfmul mm6,mm3                 ;vA.xy  {0Ay ByAx}

movq  mm7,mm2                ;       {BzAy BzAx}
pfsub mm2,mm5           ; {BzAy-AzBy  BzAx-AzBx}

psrlq mm2,32         ;x@           {0 BzAy-AzBy}
pfsub mm5,mm7        ;y@  {AzBy-BzAy  AzBx-BzAx}
pfsub mm6,mm4        ;z@         {0-0 ByAx-AyBx}

movd  (vmp3DVector PTR [edx]).x,mm2 ;x=AyBz-AzBy
movd  (vmp3DVector PTR [edx]).y,mm5 ;y=AzBx-AxBz
movd  (vmp3DVector PTR [edx]).z,mm6 ;z=AxBy-AyBx
```

If you examine it closely, you will notice the operations performed within each block and how they correlate to the generic C code that was provided. For the visually obscure optimized version, check out the CD.

vmp_CrossProduct (SSE) Aligned

The SSE instruction set uses the 128-bit XMM registers, not forgetting to use *movups* instead of *movaps* for unaligned memory. This function has also been unoptimized to make it more readable.

Listing 9-34: vmp_x86\chap09\vmd3d\Vmd3DX86M.asm

```
movaps xmm1,[ebx]          ;vB.xyz# {# Bz By Bx}
movaps xmm0,[eax]          ;vA.xyz# {# Az Ay Ax}
;  Crop the 4th (w) field
andps  xmm1,OWORD PTR lomsk96    ;  {0 Bz By Bx}
andps  xmm0,OWORD PTR lomsk96    ;  {0 Az Ay Ax}

movaps xmm5,xmm1
movaps xmm6,xmm0

shufps xmm1,xmm1,11010010b ;  3 1 0 2 {0 By Bx Bz}
shufps xmm0,xmm0,11001001b ;  3 0 2 1 {0 Ax Az Ay}
shufps xmm6,xmm6,11010010b ;  3 1 0 2 {0 Ay Ax Az}
shufps xmm5,xmm5,11001001b ;  3 0 2 1 {0 Bx Bz By}

movaps xmm2,[edx]       ; Get Destination {Dw # # #}
mulps  xmm1,xmm0
mulps  xmm5,xmm6
andps  xmm2,OWORD PTR himsk32    ;     {Dw 0 0 0}
subps  xmm1,xmm5                 ;     { 0 z y x}
orps   xmm1,xmm2                 ;     [Dw z y x}
movups [edx],xmm1             ;vD.wxyz {Dw z y x}
```

vmp_CrossProduct (AltiVec) Unaligned

The AltiVec instruction set uses the 128-bit registers, which must be memory aligned for the correct data to be accessed. In the case where memory alignment cannot be guaranteed, then each field {xyz} must be individually aligned and loaded into the AltiVec registers for processing. Similar to what had to be done for the dot product, the fourth field {w} is not used in this calculation and therefore cannot be guaranteed. So rather than have an algorithm for unaligned memory and another for vectors, one function can handle both. If you have the memory to spare, then I would recommend going ahead and building a fourth element cropping version using the quad vector version.

Listing 9-35: vmp_ppc\chap09\vmd3d\Vmd3DAltivec.cpp

```
static vector float vZero = (vector float) (0.0, 0.0, 0.0, 0.0);

vector float vD, vA, vB;

// Note: the XYZ fileds are pre-crossed as part of the equation.

  ((vmp3DQVector *) &vA)->x = pvA->z;
  ((vmp3DQVector *) &vA)->y = pvA->x;
  ((vmp3DQVector *) &vA)->z = pvA->y;

  ((vmp3DQVector *) &vB)->x = pvB->y;
  ((vmp3DQVector *) &vB)->y = pvB->z;
  ((vmp3DQVector *) &vB)->z = pvB->x;

  // Dx =-(AyBz-AzBy)   Dy =-(AzBx-AxBz)   Dz =-(AxBy-AyBx)

  vD = vec_madd( vA, vB, vZero );
  vD = vec_sub( vZero, vD );

  ((vmp3DQVector *) &vA)->x = pvA->y;
  ((vmp3DQVector *) &vA)->y = pvA->z;
  ((vmp3DQVector *) &vA)->z = pvA->x;

  ((vmp3DQVector *) &vB)->x = pvB->z;
  ((vmp3DQVector *) &vB)->y = pvB->x;
  ((vmp3DQVector *) &vB)->z = pvB->y;

  vD = vec_madd( vA, vB, vD );

  pvD->x = ((vmp3DQVector *) &vD)->x;
  pvD->y = ((vmp3DQVector *) &vD)->y;
  pvD->z = ((vmp3DQVector *) &vD)->z;
```

Vector Floating-Point Division

$$\text{Divisor} = \frac{\text{Quotient}}{\text{Dividend}}$$

$$\text{Remainder}$$

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

In the previous chapter, it was discussed that a difference is the summation of a term and the inverse of a second term using the additive inverse algebraic law. A division is also a play on an equation transformation: a multiplication of the dividend by the reciprocal of the divisor.

$$D = A \div B = \frac{A}{1} \div \frac{B}{1} = \frac{A}{1} \times \frac{1}{B} = \frac{A}{B}$$

Some instruction sets, such as 3DNow! and AltiVec, do not directly support floating-point division but do support the product of a reciprocal.

(Vector) SP-FP Division

SSE divps *xmmDst*, *xmmSrc(xmm/m128)* Single-Precision 128

This vector instruction uses a 128-bit data path, and so four operations occur in parallel. The result is calculated for each of the source Real32 (single-precision floating-point) quads of the quotient *xmmDst* and the divisor *xmmSrc* of each block, and the result is stored in each of the original Real32 of the destination *xmmDst*.

$$\begin{aligned}
Dst_{(31\ldots0)} &= (Dst_{(31\ldots0)} / Src_{(31\ldots0)}) \\
Dst_{(63\ldots32)} &= (Dst_{(63\ldots32)} / Src_{(63\ldots32)}) \\
Dst_{(95\ldots64)} &= (Dst_{(95\ldots64)} / Src_{(95\ldots64)}) \\
Dst_{(127\ldots96)} &= (Dst_{(127\ldots96)} / Src_{(127\ldots96)})
\end{aligned}$$

(Semi-Vector) DP-FP Division

SSE2 divpd *xmmDst*, *xmmSrc(xmm/m128)* Double-Precision 128

This vector instruction uses a 128-bit data path, and so two operations occur in parallel. The result is calculated for each of the source Real64 (double-precision floating-point) pairs of the quotient *xmmDst* and the divisor *xmmSrc* of each block, and the result is stored in each of the original Real64 of the destination *xmmDst*.

$$\begin{aligned}
Dst_{(63\ldots0)} &= (Dst_{(63\ldots0)} / Src_{(63\ldots0)}) \\
Dst_{(127\ldots64)} &= (Dst_{(127\ldots64)} / Src_{(127\ldots64)})
\end{aligned}$$

SP-FP Scalar Division

SSE divss *xmmDst*, *scalar(xmm/m32)* Single-Precision 128

This scalar instruction uses a 128-bit data path, but only the first Real32 (single-precision floating-point) source *scalar* divisor and the quotient *xmmDst*. The result is stored in the lower 32 bits of the destination *xmmDst*, leaving the upper 96 bits unaffected.

$$\begin{aligned}
Dst_{(31\ldots0)} &= (Dst_{(31\ldots0)} / Src_{(31\ldots0)}) \\
Dst_{(127\ldots32)} &= \text{remains unchanged.}
\end{aligned}$$

DP-FP Scalar Division

SSE2 divsd *xmmDst*, *scalar(xmm/m64)* Double-Precision 128

This scalar instruction uses a 128-bit data path and only the first Real64 (double-precision floating-point) source *scalar* divisor and the quotient *xmmDst*. The result is stored in the original lower 64 bits of the destination *xmmDst*, leaving the upper 64 bits unaffected.

$Dst_{(63...0)} = (Dst_{(63...0)} / Src_{(63...0)})$
$Dst_{(127...64)} =$ remains unchanged.

SP-FP Reciprocal (14 bit)

AltiVec	vrefp *Dst*, *aSrc* *vD = vec_re(vA)*	Single-Precision	128
3DNow!	pfrcp mmDst, *scalar(mm/m32)*	Single-Precision	32/64
SSE	rcpps *xmmDst*, *(xmm/m128)*	Single-Precision	128
MIPS IV	recip *Dst, aSrc*	Single-Precision	32
MIPS-3D	recip1.s *Dst, aSrc*		32
	recip1.ps *Dst, aSrc*		64
	recip2.d *Dst, aSrc*	Double-Precision	64

This 3DNow! scalar instruction uses a 64-bit data path and only the first Real32 (single-precision floating-point) source *scalar* divisor, produces the 14-bit reciprocal, and stores the result in both the lower 32 bits and upper 32 bits of the destination *mmDst*.

$Dst_{(63...32)} = Dst_{(31...0)} = (1.0 / Src_{(31...0)})$

To convert the result to a division, follow up the instruction with the multiplication instruction *pfmul*. This instruction would be considered a low-precision division.

Hint: Fast or slow algorithm? Of course, fast! Why would anyone even bother calling a slow algorithm or keep a slow algorithm in memory unused? In reality, this title is misleading. It really means:

Fast — Quicker algorithm, but less accurate response

Slow — Not slow, just the standard algorithm with best precision possible for the supported data size

Picking and choosing an appropriate algorithm is just another level of code optimization. In a video game, the standard algorithm would be used for rendering of the display or other precision-required event, but the fast (quick) algorithm would be used for quick processing, such as pre-culling of polygons, quick distances between sprites, etc.

```
movd  mm3,fB                ; {0    B}
movd  mm0,fA                ; {0    A}
mov   edx,pfD               ; float Destination
```

vmp_FDiv (3DNow!) Fast Float Division 14-bit Precision

A division, whether it has a 1/x or a/b orientation, is time consuming. Whenever possible, a multiplication of a reciprocal value should be used instead, but if that is not possible, then the next logical method would be that of a choice between an imprecise and quick or a more accurate but slower calculation. The following code is for a simple, 14-bit accuracy scalar division $D=A\div B$ supported by the 3DNow! instruction set.

Note that the code has the fast precision set down to 0.001f to accommodate SSE, but 0.0001f works for 3DNow! estimation.

Listing 9-36: vmp_x86\chap09\fmd\FmdX86M.asm

```
; Calculate reciprocal of source B then mult A
pfrcp mm1,mm3                   ; {1/B 1/B}
pfmul mm0,mm1                   ; {# A*(1/B)}
movd  [edx],mm0                 ; A/B
```

SP-FP Reciprocal (2 Stage) (24 Bit)

A fast version of the previous instruction would entail taking advantage of the two stage vector instructions *pfrcpit1* and *pfrcpit2*, in conjunction with the result of the reciprocal instruction *pfrcp*, to achieve a higher 24-bit precision. It uses a variation of the Newton-Raphson reciprocal square approximation.

This is an error-correcting scheme to infinitely reduce the error, but typically only a single pass is used — not to simplify it, but this typically involves calculating the product of the estimated square root, finding the difference from the original number, and adjusting by that ratio.

- First stage for 24-bit reciprocal:

3DNow! pfrcpit1 *mmDst*, *scalar(mm/m32)* Single-Precision 64

- Second stage for 24-bit reciprocal and/or square root:

3DNow! pfrcpit2 *mmDst*, *scalar(mm/m32)* Single-Precision 64

vmp_FDiv (3DNow!) Standard Float Division 24-bit Precision

The following is the same as the previous scalar division algorithm but is coded for 24-bit precision. Note the addition of the *pfrcpit1* and *pfrcpit2* instructions. Note the following code is not optimized to make it more readable. Check the CD for the optimized version.

Listing 9-37: vmp_x86\chap09\fmd\FmdX86M.asm

```
; First, calculate 14-bit accuracy
pfrcp    mm1,mm3                  ; {1/B 1/B}
; Second calculate 1/sqrt() accurate to 24 bits
pfrcpit1 mm3,mm1                  ; {1st step}
pfrcpit2 mm3,mm1      ; 24 bits    ; {2nd step}

movd     mm0,fA                   ; {0 A}
pfmul    mm0,mm3                  ; {# A*(1/B)}
movd    [edx],mm0                 ; A/B
```

vmp_FDiv (SSE) Standard Float Division 24-bit Precision

The SSE version merely reads the floats as scalars and then divides them as scalars and stores the scalar result.

Listing 9-38: vmp_x86\chap09\fmd\FmdX86M.asm

```
movss   xmm1,fB              ;B     {0 0 0 B}
movss   xmm0,fA              ;A     {0 0 0 A}
mov     eax,pfD              ; Float Destination
divss   xmm0,xmm1            ;     {0 0 0 A/B}
movss   [edx],xmm0           ;            A/B
```

Pseudo Vec (PowerPC)

For AltiVec and PowerPC, use the generic code version, as AltiVec does not support a scalar division, or preload the single-precision value, process using vector instructions, and save only the scalar result.

Pseudo Vec (MIPS)

MIPS-3D	recip2.s *Dst, aSrc*	Single-Precision	32
	recip2.ps *Dst, aSrc*		64
	recip2.d *Dst, aSrc*	Double-Precision	64

vmp_FDiv (MIPS IV) Fast Float Division

Listing 9-39

```
mfc1     $f4, a1       // A fScalar (FPU  CPU)
mfc1     $f5, a2       // B fScalar (FPU  CPU)

recip.s $f5, $f5       // {1/B}

mul.s   $f4, $f4, $f5  // {A*(1/B)}
sdc1     $f4, 0(a0)    // {Dy Dx}
```

vmp_FDiv (MIPS-3D) Standard Float Division 24-bit Precision

Replace the previous *recip.s* with the following code for higher precision.

```
recip1.s $f6, $f5              // {1/B}
recip2.s $f7, $f6, $f5
madd.s   $f5, $f6, $f6, $f7    // 24 bit {1/B}
```

Pseudo Vec

For the vector and quad vector operations, it is not much different. The scalar in essence becomes replicated into all the denominator fields, and then the product of the reciprocals (division) takes place.

Single-Precision Vector Float Scalar Division

Listing 9-40: \chap09\fmd\Fmd.cpp

```
void vmp_VecDiv(vmp3DVector * const pvD,
          const vmp3DVector * const pvA,
          float fScalar)
{
  pvD->x = pvA->x / fScalar;
  pvD->y = pvA->y / fScalar;
  pvD->z = pvA->z / fScalar;
}
```

Pseudo Vec (X86)

Now examine these functions more closely using X86 assembly. As MMX does not support floating-point, only 3DNow! and SSE can be utilized. 3DNow! supports 64 bit, so two loads must be handled simultaneously. The functionality is in essence a reciprocal of the scalar that is calculated and mirrored into each of the denominator positions, and

the product is calculated with the original vector and the result stored. These examples are all quad vectors and special consideration must be taken when dealing with three-float vectors to preserve the {W} float element.

```
movd mm2,fScalar       ;  {0  s}
mov  eax,vA            ; Vector A
mov  edx,vD            ; Vector Destination
```

vmp_QVecDiv (3DNow!) Fast Quad Float Division 14-bit Precision

Listing 9-41: vmp_x86\chap09\qvmd3d\QVmd3DX86.asm

```
pfrcp mm2,mm2          ; {1/s 1/s}  14 bit
movq  mm0,[eax+0]      ; vA.xy  {Ay Ax}
movq  mm1,[eax+8]      ; vA.zw  {Aw Az}
pfmul mm0,mm2          ; {Ay*1/s Ax*1/s}
pfmul mm1,mm2          ; {Aw*1/s Az*1/s}
movq  [edx+0],mm0      ;     {Ay/s Ax/s}
movq  [edx+8],mm1      ;     {Aw/s Az/s}
```

vmp_QVecDiv (3DNow!) Standard Quad Float Division 24-bit Precision

The following code is unoptimized to make it more readable. Notice that in the standard precision, the second and third stage reciprocal instructions are used.

Listing 9-42: vmp_x86\chap09\qvmd3d\QVmd3DX86.asm

```
pfrcp     mm3,mm2          ;   {1/s 1/s}  14 bit
punpckldq mm2,mm2          ;   { s   s}
pfrcpit1  mm2,mm3          ;   {1/s 1/s}
pfrcpit2  mm2,mm3

movq      mm0,[eax+0]      ;vA.xy {Ay Ax}
movq      mm1,[eax+8]      ;vA.zw {Aw Az}

pfmul     mm0,mm2          ;     {Ay*1/s Ax*1/s}
pfmul     mm1,mm2          ;     {Aw*1/s Az*1/s}
movq      [edx+0],mm0      ;     {Ay/s Ax/s}
movq      [edx+8],mm1      ;     {Aw/s Az/s}
```

vmp_QVecDiv (SSE) Standard Quad Float Division 24-bit Precision

Listing 9-43: vmp_x86\chap09\qvmd3d\QVmd3DX86M.asm

```
movaps xmm0,[eax]       ;vA.xyzw {Aw Az Ay Ax}
movss  xmm1,fScalar        ;         {0 0 0 s}
```

```
shufps xmm1,xmm1,00000000b ; 0 0 0 0 {s s s s}
divps  xmm0,xmm1           ; {Aw/s Az/s Ay/s Ax/s}
movaps [edx],xmm0          ; {Aw/s Az/s Ay/s Ax/s}
```

It is fairly simple. Similar to a scalar multiplication, the scalar is merely distributed to each of the elements of the denominator, and then the division takes place. (Have you read this enough yet?)

Pseudo Vec (PowerPC)

With the AltiVec instruction set, care must be taken because their fast version of code has an accuracy of 1/4096, in essence a sloppy amount of accuracy for the sake of speed. Use this instruction with care! For a more accurate calculation, use the slower generic code sample.

vmp_QVecDiv (AltiVec) Fast Quad Float Division 14-bit Precision

Listing 9-44: vmp_ppc\chap09\qvmd3d\QVmd3DAltivec.cpp

```
vector float vB;

((vmp3DQVector *) &vB)->w = fScalar;
((vmp3DQVector *) &vB)->x = fScalar;
((vmp3DQVector *) &vB)->y = fScalar;
((vmp3DQVector *) &vB)->z = fScalar;

vB = vec_re( vB );    // 1/n
*(vector float *)pvD = vec_madd(
         (*(vector float *)pvA), vB, vZero );
```

Pseudo Vec (MIPS)

vmp_QVecDiv (MIPS V and MIPS-3D) Quad Float Division 14-bit Precision

Listing 9-45: vmp_mips\chap09\qvmd3D\QVmd3DMips.s

```
mfc1     $f8, a2       // fScalar (FPU←CPU)
ldc1     $f4, 0(a1)    // {Ay Ax}
ldc1     $f5, 8(a1)    // {Aw Az}

recip1.s $f6, $f8      // {1/s}

pll.ps   $f8, $f8, $f8 // {1/s 1/s} Replicate Scalar
mul.ps   $f4, $f4, $f8 // {Ay/s Ax/s}
mul.ps   $f5, $f5, $f8 // {Aw/s Az/s}

sdc1     $f4, 0(a0)    // {Dy Dx}
sdc1     $f5, 8(a0)    // {Dw Dz}
```

vmp_QVecDiv (MIPS V and MIPS-3D) Quad Float Division 24-bit Precision

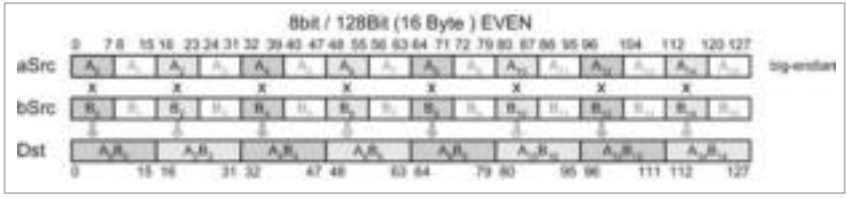
Listing 9-46: vmp_mips\chap09\qvmd3D\QVmd3DMips.s

```
mfc1     $f8, a2       // fScalar (FPU←CPU)
ldc1     $f4, 0(a1)    // {Ay Ax}
ldc1     $f5, 8(a1)    // {Aw Az}

// Note: Could use recip.s if MIPS-3D is not available!
recip1.s $f6, $f8      // {1/s}
recip2.s $f7, $f6, $f8
madd.s   $f8, $f6, $f6, $f7 // 24bit {1/s}

pll.ps   $f8, $f8, $f8 // {1/s 1/s} Replicate Scalar
mul.ps   $f4, $f4, $f8 // {Ay*1/s Ax*1/s}
mul.ps   $f5, $f5, $f8 // {Aw*1/s Az*1/s}

sdc1     $f4, 0(a0)    // {Dy Dx}
sdc1     $f5, 8(a0)    // {Dw Dz}
```

Packed {8/16/32} Bit Integer Multiplication

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

8x8-bit Multiply Even



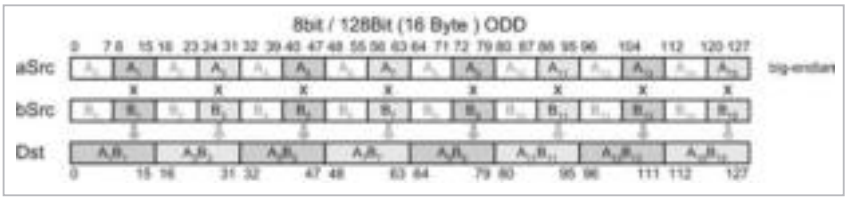
AltiVec	vmuleub *Dst, aSrc, bSrc*	Unsigned	128
	vmulesb *Dst, aSrc, bSrc*	Signed	128
	vD = vec_mule(vA, vB)		

This vector operation calculates the product of the eight even-numbered bytes of the multiplicand *aSrc* shaded in the illustration above and the multiplier *bSrc* for each 8-bit block, also shaded, and stores the result in each of the eight 16-bit destination *Dst* blocks.

$$Dst_{(15\ldots0)} = (aSrc_{(7\ldots0)} * bSrc_{(7\ldots0)})$$
$$Dst_{(31\ldots16)} = (aSrc_{(23\ldots16)} * bSrc_{(23\ldots16)})$$

$Dst_{(47...32)} = (aSrc_{(39...32)} * bSrc_{(39...32)})$
$Dst_{(63...48)} = (aSrc_{(55...48)} * bSrc_{(55...48)})$
$Dst_{(79...64)} = (aSrc_{(71...64)} * bSrc_{(71...64)})$
$Dst_{(95...80)} = (aSrc_{(87...80)} * bSrc_{(87...80)})$
$Dst_{(111...96)} = (aSrc_{(103...96)} * bSrc_{(103...96)})$
$Dst_{(127...112)} = (aSrc_{(119...112)} * bSrc_{(119...112)})$

8x8-bit Multiply Odd

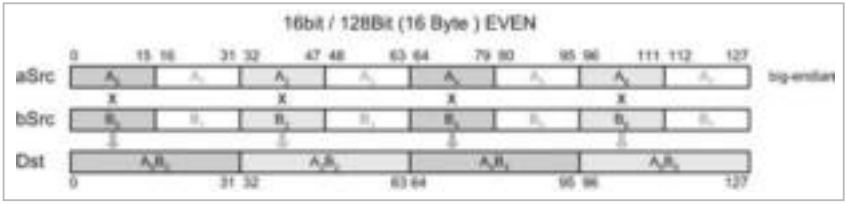


AltiVec vmuloub *Dst, aSrc, bSrc* Unsigned 128
vmulosb *Dst, aSrc, bSrc* Signed 128
vD = vec_mulo(vA, vB)

This vector operation calculates the product of the eight odd-numbered bytes of the multiplicand *aSrc* shaded in the illustration above and the multiplier *bSrc* for each 8-bit block, also shaded, and stores the result in each of the eight odd 16-bit destination *Dst* blocks.

$Dst_{(15...0)} = (aSrc_{(15...8)} * bSrc_{(15...8)})$
$Dst_{(31...16)} = (aSrc_{(31...24)} * bSrc_{(31...24)})$
$Dst_{(47...32)} = (aSrc_{(47...40)} * bSrc_{(47...40)})$
$Dst_{(63...48)} = (aSrc_{(63...56)} * bSrc_{(63...56)})$
$Dst_{(79...64)} = (aSrc_{(79...72)} * bSrc_{(79...72)})$
$Dst_{(95...80)} = (aSrc_{(95...88)} * bSrc_{(95...88)})$
$Dst_{(111...96)} = (aSrc_{(111...104)} * bSrc_{(111...104)})$
$Dst_{(127...112)} = (aSrc_{(127...120)} * bSrc_{(127...120)})$

4x16-bit Multiply Even

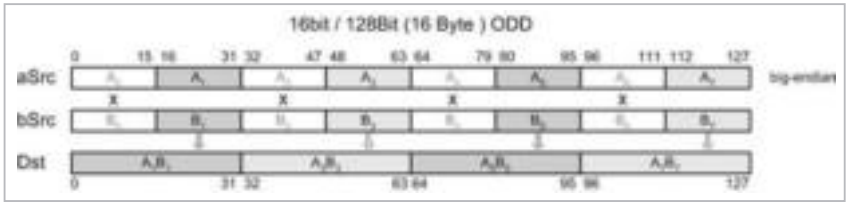


AltiVec vmuleuh *Dst, aSrc, bSrc* Unsigned 128
vmulesh *Dst, aSrc, bSrc* Signed
vD = vec_mule(vA, vB)

This vector operation calculates the product of the four even-numbered 16-bit half-words of the multiplicand *aSrc* shaded in the previous illustration and the multiplier *bSrc* for each 16-bit block, also shaded, and stores the result in each of the four even 32-bit destination *Dst* word blocks.

$$
\begin{aligned}
Dst_{(31\ldots0)} &= (aSrc_{(15\ldots0)} * bSrc_{(15\ldots0)}) \\
Dst_{(63\ldots32)} &= (aSrc_{(47\ldots32)} * bSrc_{(47\ldots32)}) \\
Dst_{(95\ldots64)} &= (aSrc_{(79\ldots64)} * bSrc_{(79\ldots64)}) \\
Dst_{(127\ldots96)} &= (aSrc_{(111\ldots96)} * bSrc_{(111\ldots96)})
\end{aligned}
$$

4x16-bit Multiply Odd

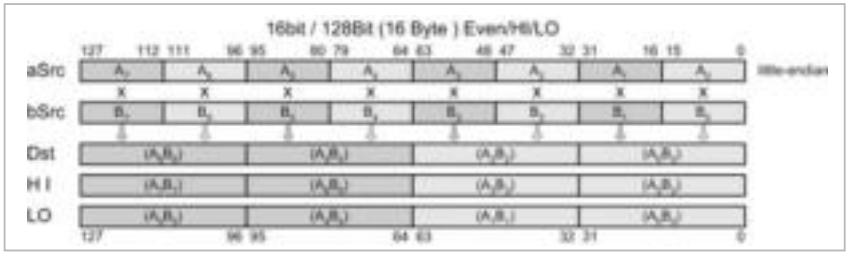


AltiVec	vmulouh *Dst, aSrc, bSrc*	Unsigned	128
	vmulosh *Dst, aSrc, bSrc*	Signed	
	vD = vec_mulo(vA, vB)		

This vector operation calculates the product of the four odd-numbered half-words of the multiplicand *aSrc* shaded in the illustration above and the multiplier *bSrc* for each 16-bit block, also shaded. It stores the result in each of the four 32-bit destination *Dst* blocks.

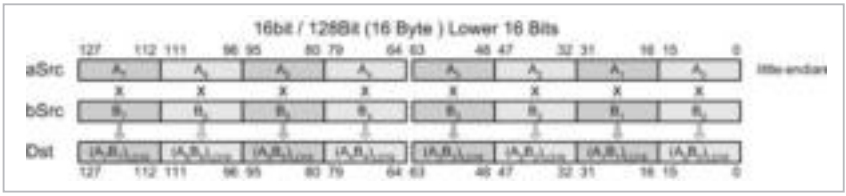
$$
\begin{aligned}
Dst_{(31\ldots0)} &= (aSrc_{(31\ldots16)} * bSrc_{(31\ldots16)}) \\
Dst_{(63\ldots32)} &= (aSrc_{(63\ldots48)} * bSrc_{(63\ldots48)}) \\
Dst_{(95\ldots64)} &= (aSrc_{(104\ldots88)} * bSrc_{(104\ldots88)}) \\
Dst_{(127\ldots96)} &= (aSrc_{(127\ldots112)} * bSrc_{(127\ldots112)})
\end{aligned}
$$

8x16-bit Parallel Multiply Half-Word



MMI	pmulth *Dst*, *aSrc, bSrc*	Signed	128

Nx16-Bit Parallel Multiplication (Lower)



MMX	pmullw *mmDst*, *mmSrc(mm/m64)*	[Un]signed	64
MMX+	pmullw *mmDst*, *mmSrc(mm/m64)*	[Un]signed	64
SSE2	pmullw *xmmDst*, *xmmSrc(xmm/m128)*	[Un]signed	128

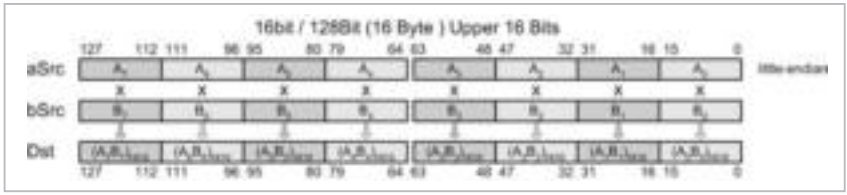
These vector instructions use a 64- (128)-bit data path, so four (eight) parallel operations occur in parallel. The product is calculated using each of the 16-bit half-words of the multiplicand *mmSrc (xmmSrc)* and the 16-bit half-words of the multiplier *mmDst (xmmDst)* for each 16-bit block. It stores the lower 16 bits of each of the results in the original 16-bit half-words of the destination *mmDst (xmmDst)*.

$$
\begin{aligned}
Dst_{(15\ldots0)} &= LOWER16(\ Dst_{(15\ldots0)} * Src_{(15\ldots0)}\ ) \quad \text{(64bit)}\\
Dst_{(31\ldots16)} &= LOWER16(\ Dst_{(31\ldots16)} * Src_{(31\ldots16)}\ ) \quad \textbf{4x16bit}\\
Dst_{(47\ldots32)} &= LOWER16(\ Dst_{(47\ldots32)} * Src_{(47\ldots32)}\ )\\
Dst_{(63\ldots48)} &= LOWER16(\ Dst_{(63\ldots48)} * Src_{(63\ldots48)}\ )\\
Dst_{(79\ldots64)} &= LOWER16(\ Dst_{(79\ldots64)} * Src_{(79\ldots64)}\ ) \quad \text{(128bit)}\\
Dst_{(95\ldots80)} &= LOWER16(\ Dst_{(95\ldots80)} * Src_{(95\ldots80)}\ ) \quad \textbf{8x16bit}\\
Dst_{(111\ldots96)} &= LOWER16(\ Dst_{(111\ldots96)} * Src_{(111\ldots96)}\ )\\
Dst_{(127\ldots112)} &= LOWER16(\ Dst_{(127\ldots112)} * Src_{(127\ldots112)}\ )
\end{aligned}
$$

D63...D48	D47...D32	D31...D16	D15...D0
5678h	5678h	5678h	5678h
x0012h	x0023h	x0034h	x0056h
00061470h	000bd268h	00119060h	001d0c50h
1470h	d268h	9060h	0c50h
1470h	0d268h	9060h	0c50h

Hint: When multiplying two integers of *n* bit size together, regardless of their values being signed or unsigned, the lower *n* bits of the base result will be the same, so this instruction is considered sign neutral, thus [un]signed.

Nx16-bit Parallel Multiplication (Upper)



MMX	pmulhuw *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
	pmulhw *mmDst*, *mmSrc(mm/m64)*	Signed	
MMX+	pmulhuw *mmDst, mmSrc(mm/m64)*	Unsigned	64
SSE2	pmulhuw *xmmDst*, *xmmSrc(xmm/m128)*	Unsigned	128
	pmulhw *xmmDst*, *xmmSrc(xmm/m128)*	Signed	

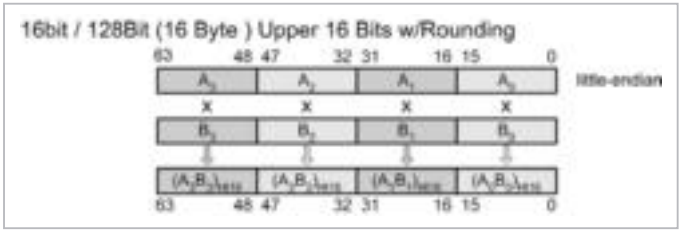
These vector instructions use a 64- (128)-bit data path, so four (eight) parallel operations occur in parallel. The product is calculated using each of the 16-bit half-words of the multiplicand *mmSrc (xmmSrc)* and the 16-bit half-word of the multiplier *mmDst (xmmDst)* for each 16-bit block. It stores the upper 16 bits of each of the results in the original 16-bit half-words of the destination *mmDst (xmmDst)*.

$Dst_{(15\ldots0)} = UPPER16(Dst_{(15\ldots0)} * Src_{(15\ldots0)})$ (64bit) **4x16bit**
$Dst_{(31\ldots16)} = UPPER16(Dst_{(31\ldots16)} * Src_{(31\ldots16)})$
$Dst_{(47\ldots32)} = UPPER16(Dst_{(47\ldots32)} * Src_{(47\ldots32)})$
$Dst_{(63\ldots48)} = UPPER16(Dst_{(63\ldots48)} * Src_{(63\ldots48)})$
$Dst_{(79\ldots64)} = UPPER16(Dst_{(79\ldots64)} * Src_{(79\ldots64)})$ (128bit) **8x16bit**
$Dst_{(95\ldots80)} = UPPER16(Dst_{(95\ldots80)} * Src_{(95\ldots80)})$
$Dst_{(111\ldots96)} = UPPER16(Dst_{(111\ldots96)} * Src_{(111\ldots96)})$
$Dst_{(127\ldots112)} = UPPER16(Dst_{(127\ldots112)} * Src_{(127\ldots112)})$

63...48	47...32	31...16	15...0	63...0
8374	373	9382	2043	20B6 0175 24A6 07FB
*54	*38	*5	*7	*0036 0026 0005 0007
452196	14174	46910	14301	E664 375E B73E 37DD

D63...D48	D47...D32	D31...D16	D15...D0
5678h	5678h	5678h	5678h
x0012h	x0023h	x0034h	x0056h
00061470h	000bd268h	00119060h	001d0c50h
0006 h	000b h	0011 h	001d h
0006h	000bh	0011h	001dh

Signed 4x16-bit Multiplication with Rounding (Upper)



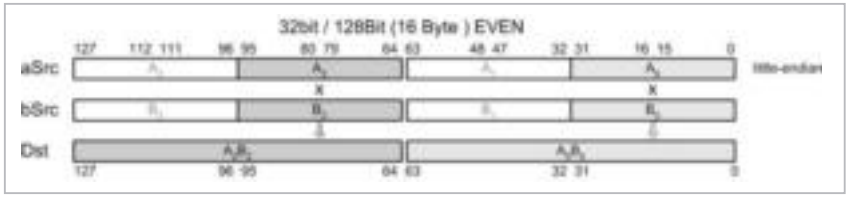
3DNow! pmulhrw *mmDst*, *mmSrc(mm/m64)* Signed 64

This vector instruction uses a 64-bit data path, and so four parallel operations occur in parallel. The product is calculated using each of the unsigned 16-bit half-words of the multiplicand *mmDst* and the 16-bit half-word of the multiplier *mmSrc* for each 16-bit block and summing 00008000 hex to the 32-bit product. The resulting upper 16 bits is stored in the destination *mmDst (xmmDst)*.

$Dst_{(15...0)} = UPPER16((\ Dst_{(15...0)} * Src_{(15...0)}\ ) + 0x8000\ )$
$Dst_{(31...16)} = UPPER16((\ Dst_{(31...16)} * Src_{(31...16)}\ ) + 0x8000\ )$
$Dst_{(47...32)} = UPPER16((\ Dst_{(47...32)} * Src_{(47...32)}\ ) + 0x8000\ )$
$Dst_{(63...48)} = UPPER16((\ Dst_{(63...48)} * Src_{(63...48)}\ ) + 0x8000\ )$

D63...D48	D47...D32	D31...D16	D15...D0
5678h	5678h	5678h	5678h
x0012h	x0023h	x0034h	x0056h
00061470h	000bd268h	00119060h	001d0c50h
+00008000h	+00008000h	+00008000h	+00008000h
00069470h	000c5268h	00121060h	001d8c50h
0006 h	000c h	0012 h	001d h
0006h	000bh	0012h	001dh

Unsigned Nx32-bit Multiply Even



SSE2	pmuludq *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
	pmuludq *xmmDst*, *xmmSrc(xmm/m128)*	Unsigned	128
MMI	pmultw Ds*t*, *aSrc, bSrc*	Signed	128
	pmultuw Ds*t*, *aSrc, bSrc*	Unsigned	128

This vector instruction calculates the product of the (even) lower 32 bits of each set of 64 (128) bits of the multiplicand *aSrc (xmmSrc)* and the lower 32 bits of the multiplier *bSrc (xmmDst)* for each 64- (128)-bit block. It stores each full 64-bit (128-bit) integer result in the destination *Dst (xmmDst)*.

$$Dst_{(63\ldots0)} = Dst_{(31\ldots0)} * Src_{(31\ldots0)} \quad \text{(64bit) 1x32bit}$$

$$Dst_{(127\ldots64)} = Dst_{(95\ldots64)} * Src_{(95\ldots64)} \quad \text{(128bit) 2x32bit}$$

In the following 64-bit table, the upper 32 bits {63...32} of *mmSrc* and *mmDst* are ignored. Note that this is not a SIMD reference but included for reference.

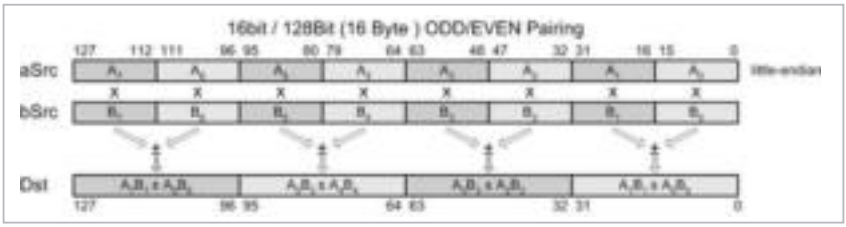
31...0	63...0
3287565	0000000000322A0D
* 593	*0000000000000251
1949526045	000000007433681D

In the following 128-bit table, the upper odd pairs of 32 bits {127...96, 63...32} of *xmmDst* and *xmmSrc* are ignored.

95...64	31...0	127...0
85490485	3287565	00000000051 87B350000000000322A0D
* 9394	* 593	* 00000000000024B2000000000000000251
803097616090	1949526045	000000BAFC591EDA000000007433681D
(0xBAFC591EDA)	(0x7433681D)	

Integer Multiplication and Addition/ Subtraction

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI



Signed Nx16-bit Parallel Multiplication and Addition

MMX	pmaddwd *mmDst*, *mmSrc(mm/m64)*	Signed	64
SSE2	pmaddwd *xmmDst*, *xmmSrc(xmm/m128)*	Signed	128
MMI	phmadh *Dst*, *aSrc, bSrc*	Signed	128

Signed Nx16-bit Parallel Multiplication and Subtraction

MMI	phmsbh *Dst*, *aSrc, bSrc*	Signed	128

This vector instruction calculates the 32-bit signed products of two pairs of two 16-bit multiplicand *aSrc(xmmSrc)* and the multiplier *bSrc(xmmDst)* for each bit block. The first and second 32-bit products are summed and stored in the lower 32 bits of the *Dst(xmmDst)*. The third and fourth 32-bit products are summed and stored in the next 32 bits of the destination *Dst(xmmDst)*. The same is repeated for the upper 64 bits for the 128-bit data model.

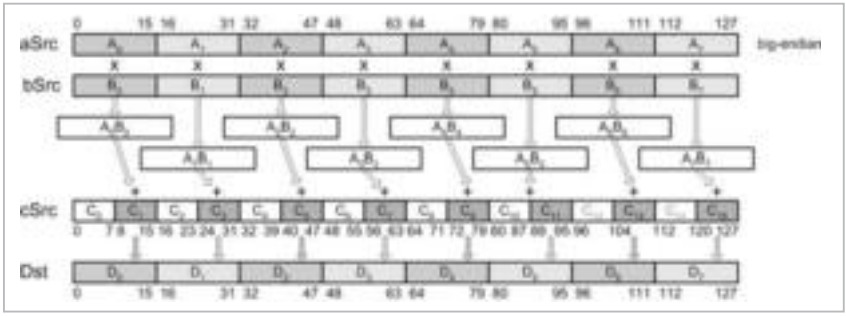
Warning: Each 32-bit result can wrap due to overflow if the two pairings are all set to hex 8000 (–32768). Note the inversion of the sign to the overflow.

```
(0x8000*0x8000) + (0x8000*0x8000) = 2(0x40000000) = x80000000
(–32768*–32768) + (–32768*–32768) = 2147483648  –2147483648
```

$$
\begin{aligned}
Dst_{(31\ldots0)} &= (Dst_{(15\ldots0)} * Src_{(15\ldots0)}) + (Dst_{(31\ldots16)} * Src_{(31\ldots16)}) \\
Dst_{(63\ldots32)} &= (Dst_{(47\ldots32)} * Src_{(47\ldots32)}) + (Dst_{(63\ldots48)} * Src_{(63\ldots48)}) \quad (64bit) \\
Dst_{(95\ldots64)} &= (Dst_{(79\ldots64)} * Src_{(79\ldots64)}) + (Dst_{(95\ldots80)} * Src_{(95\ldots80)}) \\
Dst_{(127\ldots96)} &= (Dst_{(111\ldots96)} * Src_{(111\ldots96)}) + (Dst_{(127\ldots112)} * Src_{(127\ldots112)}) \quad (128bit)
\end{aligned}
$$

D63...D48	D47...D32	D31...D16	D15...D0
9DF6h	5A63h	18E7h	93C3h
BD8Bh	x205Bh	x1B38h	x8459h
74F46292h	0B6C8131h	02A5CF88h	4C63EAC6h
74F46292h + 0B6C8131h		02A5CF88h + 4C63EAC6h	
8060E3C3h		4F09BA4Eh	

[Un]signed 8x16-bit Multiplication then Add



AltiVec vmladduhm *Dst*, *aSrc*, *bSrc*, *cSrc* Sign Neutral 128

vD = vec_mladd(vA, vB, vC)

This vector instruction uses a 128-bit data path and eight signless operations in parallel. The product is calculated for each of the 16-bit octals of the multiplicand *aSrc* and the 16-bit octals of the multiplier *bSrc* for each 16-bit block, then adds each of the 16-bit octals of *cSrc*, and stores the lower 16 bits of each of the solutions in the bits of the destination *Dst*.

$Dst_{(15...0)}$ = LO16(($aSrc_{(15...0)}$ * $bSrc_{(15...0)}$) + $cSrc_{(15...0)}$)
$Dst_{(31...16)}$ = LO16(($aSrc_{(31...16)}$ * $bSrc_{(31...16)}$) + $cSrc_{(31...16)}$)
$Dst_{(47...32)}$ = LO16(($aSrc_{(47...32)}$ * $bSrc_{(47...32)}$) + $cSrc_{(47...32)}$)
$Dst_{(63...48)}$ = LO16(($aSrc_{(63...48)}$ * $bSrc_{(63...48)}$) + $cSrc_{(63...48)}$)
$Dst_{(79...64)}$ = LO16(($aSrc_{(79...64)}$ * $bSrc_{(79...64)}$) + $cSrc_{(79...64)}$)
$Dst_{(95...80)}$ = LO16(($aSrc_{(95...80)}$ * $bSrc_{(95...80)}$) + $cSrc_{(95...80)}$)
$Dst_{(111...96)}$ = LO16(($aSrc_{(111...96)}$ * $bSrc_{(111...96)}$) + $cSrc_{(111...96)}$)
$Dst_{(127...112)}$ = LO16(($aSrc_{(127...112)}$ * $bSrc_{(127...112)}$) + $cSrc_{(127...112)}$)

Signed 8x16-bit Multiply then Add with Saturation

AltiVec vmhaddshs *Dst*, *aSrc*, *bSrc*, *cSrc* Sign Neutral 128

This vector instruction uses a 128-bit data path and eight signless operations in parallel. They calculate the product of each of the 16-bit octals of the multiplicand *aSrc* and the 16-bit octals of the multiplier *bSrc* for each 16-bit block, then add each of the 16-bit octals of *cSrc* to each of the upper 17 bits of the product, and store the lower 16 bits of each of the solutions in the bits of the destination *Dst*.

$$
\begin{aligned}
Dst_{(15\ldots0)} &= SAT(UP17(aSrc_{(15\ldots0)} * bSrc_{(15\ldots0)}) + cSrc_{(15\ldots0)})\\
Dst_{(31\ldots16)} &= SAT(UP17(aSrc_{(31\ldots16)} * bSrc_{(31\ldots16)}) + cSrc_{(31\ldots16)})\\
Dst_{(47\ldots32)} &= SAT(UP17(aSrc_{(47\ldots32)} * bSrc_{(47\ldots32)}) + cSrc_{(47\ldots32)})\\
Dst_{(63\ldots48)} &= SAT(UP17(aSrc_{(63\ldots48)} * bSrc_{(63\ldots48)}) + cSrc_{(63\ldots48)})\\
Dst_{(79\ldots64)} &= SAT(UP17(aSrc_{(79\ldots64)} * bSrc_{(79\ldots64)}) + cSrc_{(79\ldots64)})\\
Dst_{(95\ldots80)} &= SAT(UP17(aSrc_{(95\ldots80)} * bSrc_{(95\ldots80)}) + cSrc_{(95\ldots80)})\\
Dst_{(111\ldots96)} &= SAT(UP17(aSrc_{(111\ldots96)} * bSrc_{(111\ldots96)}) + cSrc_{(111\ldots96)})\\
Dst_{(127\ldots112)} &= SAT(UP17(aSrc_{(127\ldots112)} * bSrc_{(127\ldots112)}) + cSrc_{(127\ldots112)})
\end{aligned}
$$

Signed 8x16-bit Multiply Round then Add with Saturation

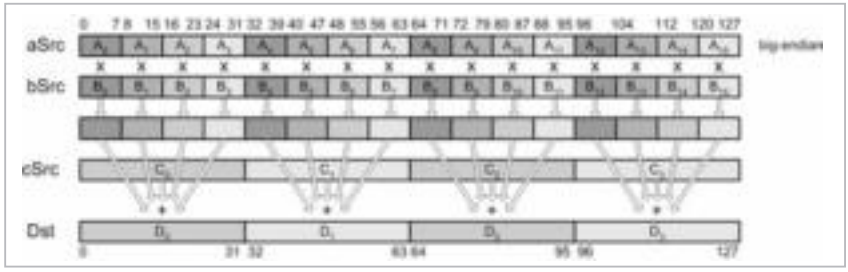
AltiVec vmhraddshs *Dst*, *aSrc*, *bSrc*, *cSrc* Signed 128

This vector instruction uses a 128-bit data path and eight signless operations in parallel. The product is calculated for each of the 16-bit octals of the multiplicand *aSrc* and the 16-bit octals of the multiplier *bSrc* for each 16-bit block, then each of the 16-bit octals of *cSrc* is added to each of the upper 17 bits of the product. The lower 16 bits of each of the solutions is stored in the bits of the destination *Dst*.

$$
\begin{aligned}
Dst_{(15\ldots0)} &= SAT(UP17(aSrc_{(15\ldots0)} * bSrc_{(15\ldots0)} + 0x4000) + cSrc_{(15\ldots0)})\\
Dst_{(31\ldots16)} &= SAT(UP17(aSrc_{(31\ldots16)} * bSrc_{(31\ldots16)} + 0x4000) + cSrc_{(31\ldots16)})\\
Dst_{(47\ldots32)} &= SAT(UP17(aSrc_{(47\ldots32)} * bSrc_{(47\ldots32)} + 0x4000) + cSrc_{(47\ldots32)})\\
Dst_{(63\ldots48)} &= SAT(UP17(aSrc_{(63\ldots48)} * bSrc_{(63\ldots48)} + 0x4000) + cSrc_{(63\ldots48)})\\
Dst_{(79\ldots64)} &= SAT(UP17(aSrc_{(79\ldots64)} * bSrc_{(79\ldots64)} + 0x4000) + cSrc_{(79\ldots64)})\\
Dst_{(95\ldots80)} &= SAT(UP17(aSrc_{(95\ldots80)} * bSrc_{(95\ldots80)} + 0x4000) + cSrc_{(95\ldots80)})\\
Dst_{(111\ldots96)} &= SAT(UP17(aSrc_{(111\ldots96)} * bSrc_{(111\ldots96)} + 0x4000) + cSrc_{(111\ldots96)})\\
Dst_{(127\ldots112)} &= SAT(UP17(aSrc_{(127\ldots112)} * bSrc_{(127\ldots112)} + 0x4000) + cSrc_{(127\ldots112)})
\end{aligned}
$$

Integer Multiplication and Summation-Addition

16x8-bit Multiply then Quad 32-bit Sum

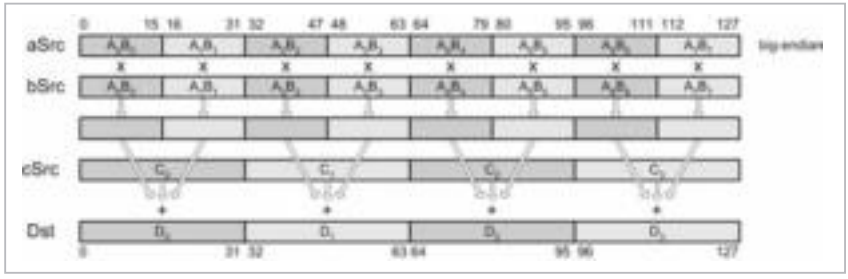


AltiVec	vmsumubm D*st, aSrc, bSrc, cSrc*	Unsigned	128
	vmsumsbm *Dst, aSrc, bSrc, cSrc*	Signed	
	vD = vec_msum(vA, vB, vC)		

This vector operation calculates the 16 8-bit integer products from the 16 multiplicand *aSrc* and the multiplier *bSrc* for each 8-bit block. Those are grouped in quads, and the results are then summed with *cSrc* and stored in the related 32-bit destination *Dst*.

$$
\begin{aligned}
Dst_{(31\ldots0)} &= (aSrc_{(7\ldots0)} * bSrc_{(7\ldots0)}) + (aSrc_{(15\ldots8)} * bSrc_{(15\ldots8)}) \\
&\quad + cSrc_{(31\ldots0)} + (aSrc_{(23\ldots16)} * bSrc_{(23\ldots16)}) + (aSrc_{(31\ldots24)} * bSrc_{(31\ldots24)}) \\
Dst_{(63\ldots32)} &= (aSrc_{(39\ldots32)} * bSrc_{(39\ldots32)}) + (aSrc_{(47\ldots40)} * bSrc_{(47\ldots40)}) \\
&\quad + cSrc_{(63\ldots32)} + (aSrc_{(55\ldots48)} * bSrc_{(55\ldots48)}) + (aSrc_{(63\ldots56)} * bSrc_{(63\ldots56)}) \\
Dst_{(95\ldots64)} &= (aSrc_{(71\ldots64)} * bSrc_{(71\ldots64)}) + (aSrc_{(79\ldots72)} * bSrc_{(79\ldots72)}) \\
&\quad + cSrc_{(95\ldots64)} + (aSrc_{(87\ldots80)} * bSrc_{(87\ldots80)}) + (aSrc_{(95\ldots88)} * bSrc_{(95\ldots88)}) \\
Dst_{(127\ldots96)} &= (aSrc_{(103\ldots96)} * bSrc_{(103\ldots96)}) + (aSrc_{(111\ldots104)} * bSrc_{(111\ldots104)}) \\
&\quad + cSrc_{(127\ldots96)} + (aSrc_{(119\ldots112)} * bSrc_{(119\ldots112)}) + (aSrc_{(127\ldots120)} * bSrc_{(127\ldots120)})
\end{aligned}
$$

8x16-bit Multiply then Quad 32-bit Sum



AltiVec	vmsumuhm *Dst, aSrc, bSrc, cSrc* vmsumshm *Dst, aSrc, bSrc, cSrc* *vD = vec_msum(vA, vB, vC)*	Unsigned Signed	128

This vector operation calculates the four 32-bit integer products from the 16 multiplicand *aSrc* and the multiplier *bSrc* for each 8-bit block. Each are in groupings of four, and the results are then summed with *cSrc* and stored in the related 32-bit destination *Dst*.

$$
\begin{aligned}
Dst_{(31\ldots0)} &= (aS_{(15\ldots0)} * bS_{(15\ldots0)}) + (aS_{(31\ldots16)} * bS_{(31\ldots16)}) + cS_{(31\ldots0)} \\
Dst_{(63\ldots32)} &= (aS_{(47\ldots32)} * bS_{(47\ldots32)}) + (aS_{(63\ldots48)} * bS_{(63\ldots48)}) + cS_{(63\ldots32)} \\
Dst_{(95\ldots64)} &= (aS_{(79\ldots64)} * bS_{(79\ldots64)}) + (aS_{(95\ldots80)} * bS_{(95\ldots80)}) + cS_{(95\ldots64)} \\
Dst_{(127\ldots96)} &= (aS_{(111\ldots96)} * bS_{(111\ldots96)}) + (aS_{(127\ldots112)} * bS_{(127\ldots112)}) + cS_{(127\ldots96)}
\end{aligned}
$$

8x16-bit Multiply then Quad 32-bit Sum with Saturation

AltiVec	vmsumuhs *Dst, aSrc, bSrc, cSrc* vmsumshs *Dst, aSrc, bSrc, cSrc* *vD = vec_msum(vA, vB, vC)*	Unsigned Signed	128

This vector operation calculates the four 32-bit integer products from the 16 multiplicand *aSrc* and the multiplier *bSrc* for each 8-bit block. Each are in groupings of four, and the results are then summed with *cSrc* and stored in the related 32-bit destination *Dst*.

$$
\begin{aligned}
Dst_{(31\ldots0)} &= SAT((aS_{(15\ldots0)} * bS_{(15\ldots0)}) + (aS_{(31\ldots16)} * bS_{(31\ldots16)}) + cS_{(31\ldots0)}) \\
Dst_{(63\ldots32)} &= SAT((aS_{(47\ldots32)} * bS_{(47\ldots32)}) + (aS_{(63\ldots48)} * bS_{(63\ldots48)}) + cS_{(63\ldots32)}) \\
Dst_{(95\ldots64)} &= SAT((aS_{(79\ldots64)} * bS_{(79\ldots64)}) + (aS_{(95\ldots80)} * bS_{(95\ldots80)}) + cS_{(95\ldots64)}) \\
Dst_{(127\ldots96)} &= SAT((aS_{(111\ldots96)} * bS_{(111\ldots96)}) + (aS_{(127\ldots112)} * bS_{(127\ldots112)}) + cS_{(127\ldots96)})
\end{aligned}
$$

Vector (Integer) Multiplication and Add

Packed integer multiplication is one of the mathematical equations that you will tend to use in your video games either as fixed-point or parallel integer processing. This works out nicely when it is necessary to increase the magnitude of a series of integers. The problem here, which comes up as fixed-point multiplication, is not like floating-point multiplication. In floating-point, there is a precision loss with each calculation since a numerical value is stored in an exponential form. With fixed-point there is no precision loss, which is great, but this leads to another problem. When two integers are used in a summation, the most significant bits are carried into an additional (n+1) bit. With a multiplication of two integers, the resulting storage required is (n+n=2n)

bits. This poses a problem of how to deal with the resulting solution. Since the data size increases, there are multiple solutions to contain the result of the calculation:

- Store upper bits.
- Store lower bits.
- Store upper/lower bits into two vectors.
- Store even *n*-bit elements into 2*n*-bit elements.
- Store odd *n*-bit elements into 2*n*-bit elements.

You had a preview of this when you first encountered all the flavors of the packed integer multiplications. Each one, in reality, has a useful functional solution. I felt it wise to implement a variety of combinations as, depending on the application, some different algorithms would work better than others. These are typically used with a signed or unsigned data type. An 8 bit x 8 bit is effectively not a very large number to warrant a multiplication, and so it is not typically supported by the processor in a packed form, but 16 bit x 16 bit is supported. The case of 32 bit x 32 bit on the other hand is not supported very well since only the lower 32 bits of the 64-bit solution would be preserved.

Pseudo Vec

Packed [Un]signed Multiplication Low (16 Bit) = 8x16-bit

Here's an interesting characteristic of the product of two signed or unsigned integers. The lower set of bits for either the signed or unsigned multiplications are exactly the same! The following is such a case of two half-word (16-bit) integer vectors.

```
D = LOW16( A × B )  [Un]signed
```

Listing 9-47: \chap09\pmd\Pmd.cpp

```
void vmp_pmulluH( uint16 * phD, uint16 * phA, uint16 * phB )
{
 *(phD+0)=(uint16)((((uint)*(phA+0))*((uint)*(phB+0)))&0x0000ffff);
 *(phD+1)=(uint16)((((uint)*(phA+1))*((uint)*(phB+1)))&0x0000ffff);
 *(phD+2)=(uint16)((((uint)*(phA+2))*((uint)*(phB+2)))&0x0000ffff);
 *(phD+3)=(uint16)((((uint)*(phA+3))*((uint)*(phB+3)))&0x0000ffff);
 *(phD+4)=(uint16)((((uint)*(phA+4))*((uint)*(phB+4)))&0x0000ffff);
 *(phD+5)=(uint16)((((uint)*(phA+5))*((uint)*(phB+5)))&0x0000ffff);
 *(phD+6)=(uint16)((((uint)*(phA+6))*((uint)*(phB+6)))&0x0000ffff);
 *(phD+7)=(uint16)((((uint)*(phA+7))*((uint)*(phB+7)))&0x0000ffff);
}
```

Packed Signed Multiplication High (16-bit) = 8x16-bit

```
D = UPPER16( A × B )  signed
```

Listing 9-48: \chap09\pmd\Pmd.cpp

```
void vmp_pmulhH( int16 * phD, int16 * phA, int16 * phB )
{
  *(phD+0) = (int16)((((int)*(phA+0)) * ((int)*(phB+0))) >> 16);
  *(phD+1) = (int16)((((int)*(phA+1)) * ((int)*(phB+1))) >> 16);
  *(phD+2) = (int16)((((int)*(phA+2)) * ((int)*(phB+2))) >> 16);
  *(phD+3) = (int16)((((int)*(phA+3)) * ((int)*(phB+3))) >> 16);
  *(phD+4) = (int16)((((int)*(phA+4)) * ((int)*(phB+4))) >> 16);
  *(phD+5) = (int16)((((int)*(phA+5)) * ((int)*(phB+5))) >> 16);
  *(phD+6) = (int16)((((int)*(phA+6)) * ((int)*(phB+6))) >> 16);
  *(phD+7) = (int16)((((int)*(phA+7)) * ((int)*(phB+7))) >> 16);
}
```

For unsigned integers, substitute *uint16* for *int16* and *uint* for *int*.

Packed Unsigned Multiplication High (16-bit) = 8x16-bit

```
D = UPPER16( A × B )  unsigned
```

Listing 9-49: \chap09\pmd\Pmd.cpp

```
void vmp_pmulhuH( uint16 * phD, uint16 * phA, uint16 * phB )
{
  *(phD+0) = (uint16)((((uint)*(phA+0)) * ((uint)*(phB+0))) >> 16);
  *(phD+1) = (uint16)((((uint)*(phA+1)) * ((uint)*(phB+1))) >> 16);
  *(phD+2) = (uint16)((((uint)*(phA+2)) * ((uint)*(phB+2))) >> 16);
  *(phD+3) = (uint16)((((uint)*(phA+3)) * ((uint)*(phB+3))) >> 16);
  *(phD+4) = (uint16)((((uint)*(phA+4)) * ((uint)*(phB+4))) >> 16);
  *(phD+5) = (uint16)((((uint)*(phA+5)) * ((uint)*(phB+5))) >> 16);
  *(phD+6) = (uint16)((((uint)*(phA+6)) * ((uint)*(phB+6))) >> 16);
  *(phD+7) = (uint16)((((uint)*(phA+7)) * ((uint)*(phB+7))) >> 16);
}
```

Pseudo Vec (X86)

Here's an interesting thing about the SSE instruction set. It is primarily designed for floating-point operations. The MMX instruction set handles most of the packed integer processing, except the case of the unsigned 16-bit multiplication. This was not resolved until a later release of extensions for the MMX by AMD and the SSE by Intel, but this is only 64 bits. Intel came back with the SSE2 instruction set with a complete set of packed instructions for supporting 128 bits.

```
mov   ebx,phB   ; Vector B
mov   eax,phA   ; Vector A
mov   edx,phD   ; Vector Destination
```

vmp_pmulluH (MMX) [Un]signed Multiplication Low (16-bit) = 8x16-bit

The instruction *pmullw* is the [un]signed multiplication storing the lower 16 bits, *pmulhw* is the signed multiplication storing the upper 16 bits, and finally *pmulhuw* is the unsigned multiplication storing the upper 16 bits. These are designed for use with the 64-bit MMX register and are used in pairs for the following sample.

Listing 9-50: vmp_x86\chap09\pmd\PMulX86M.asm

```
movq   mm0,[ebx+0] ; Read B Data {3...0}
movq   mm1,[ebx+8] ;             {7...4}
movq   mm2,[eax+0] ; Read A Data {3...0}
movq   mm3,[eax+8] ;             {7...4}

pmullw mm0,mm2
pmullw mm1,mm3

movq   [edx+0],mm0 ; Write D ??? 16bits   {3...0}
movq   [edx+8],mm1 ;   (upper or lower)   {7...4}
```

vmp_pmulhH (MMX) Signed Multiplication High (16-bit) = 8x16-bit

```
pmulhw mm0,mm2     ; SIGNED HIGH   {3...0}
pmulhw mm1,mm3     ;               {7...4}
```

vmp_pmulhuH (MMX+, SSE) Unsigned Multiplication High (16-bit) = 8x16-bit

```
pmulhuw mm0,mm2    ; UNSIGNED HIGH {3...0}
pmulhuw mm1,mm3    ;               {7...4}
```

vmp_pmulluH (SSE2) [Un]signed Multiplication Low (16-bit) = 8x16-bit

With the release of the SSE2 instruction set, Intel handles 128 bits simultaneously using the XMM registers. The instruction *pmullw* is the [un]signed multiplication storing the lower 16 bits, *pmulhw* is the signed multiplication storing the upper 16 bits, and finally *pmulhuw* is the unsigned multiplication storing the upper 16 bits. Do not forget that unaligned memory would require *movdqu* instead of *movdqa*.

Listing 9-51: vmp_x86\chap09\pmd\PMulX86M.asm

```
movdqa xmm0,[ebx]  ; Read B Data  {7...0}
movdqa xmm1,[eax]  ; Read A Data  {7...0}
pmullw xmm0,xmm1
movdqa [edx],xmm0  ; Write D lower 16bits {7...0}
```

vmp_pmulhH (SSE2) Signed Multiplication High (16-bit) = 8x16-bit

```
pmulhw xmm0,xmm1   ; SIGNED HIGH   {7...0}
```

vmp_pmulhuH (SSE2) Unsigned Multiplication High (16-bit) = 8x16-bit

```
pmulhuw xmm0,xmm1  ; UNSIGNED HIGH {7...0}
```

Pseudo Vec (MIPS)

The implementation of integer products on the MIPS processor is a bit more complicated, as only 16-bit and some 32-bit instructions are supported. Eight-bit, however, is not. To obtain 8-bit products, data must be expanded from 8 bit to 16 bit, processed, and then packed back into the expected result.

8-bit Products

The following is generic MIPS code for preparing packed 8-bit signed values for 16-bit packed product calculations:

```
lq     t1, 0(a1)      // pbA {A15 ... A0}
lq     t3, 0(a2)      // pbB {B15 ... B0}

// Convert 16x8 bit to 2x 8x16 bit
pextlb t0, t1, t1     // {A7  A7  ... A0 A0}
pextub t1, t1, t1     // {A15 A15 ... A8 A8}
pextlb t2, t3, t3     // {B7  B7  ... B0 B0}
pextub t3, t3, t3     // {B15 B15 ... B8 B8}

// Shift upper 8 bits of each 16 bit arithmetically to extend the sign bit
into those upper 8 bits, resulting in a 16-bit signed value!

psrah  t0,t0,8        //  {±A7  ... ±A0}
psrah  t1,t1,8        //  {±A15 ... ±A8}
psrah  t2,t2,8        //  {±B7  ... ±B0}
psrah  t3,t3,8        //  {±B15 ... ±B8}
```

If the 8-bit values were unsigned, then a logical right shift would be used instead, extending a leading zero bit!

```
psrlh  t0,t0,8        //  {A7  ... A0}
psrlh  t1,t1,8        //  {A15 ... A8}
psrlh  t2,t2,8        //  {B7  ... B0}
psrlh  t3,t3,8        //  {B15 ... B8}
```

Now the signed or unsigned processing can take place upon the 16-bit values. Upon the completion of all processing, the data needs to be converted back into the desired result through bit shifting and bit blending.

vmp_pmulluH (MMI) [Un]signed Multiplication Low (16-bit) = 8x16-bit

In the following code, two special-purpose registers, HI and LO, are taken to advantage. The results of the instruction *pmulth* results in the distribution of the elements as indicated. The *pmfhi* and *pmflo* instructions transfer that result from those special registers to the 128-bit general-purpose ones.

Listing 9-52: vmp_mips\chap09\pmd\PMulMMI.s

```
lq     t1, 0(a1)      // phA
lq     t2, 0(a2)      // phB
LDELAY                // nop - Load Delay Slot

pmulth t0,t1,t2       // Dst = {D6 D4 D2 D0}
pmfhi  t1             // HI  = {D7 D6 D3 D2}
pmflo  t2             // LO  = {D5 D4 D1 D0}

 ; Insert code here for upper 16-bit extraction

ppach  t1,t1,t2       // {D7 D6 D3 D2 D5 D4 D1 D0}
pexcw  t0,t1          // {D7 D6 D5_D4 D3_D2 D1 D0}

sq     t0, 0(a0)      // phD
```

vmp_pmulhH (MMI) Signed Multiplication High (16-bit) = 8x16-bit

The previous code extracts the lower 16 bits, so by inserting the following code, the upper-signed 16 bits are shifted into the lower 16 bits and extracted.

```
psrlw  t1,t1,16       // {_D7 _D6 _D3 _D2}
psrlw  t2,t2,16       // {_D5 _D4 _D1 _D0}
```

Pseudo Vec

Packed Signed Multiplication Hi/Lo (2x16-bit) = 8x16-bit

Now things get even more interesting. This is essentially a combination of both the upper and lower 16 bits, similar to that above but both upper and lower results are written to memory as two separate vectors.

Listing 9-53: \chap09\pmd\Pmd.cpp

```
void vmp_pmulhlH( int16 * phHi, int16 * phLo,
                  int16 * phA,  int16 * phB )
{
  int hilo[8];

  hilo[0] = ((int)*(phA+0)) * ((int)*(phB+0));
  hilo[1] = ((int)*(phA+1)) * ((int)*(phB+1));
  hilo[2] = ((int)*(phA+2)) * ((int)*(phB+2));
  hilo[3] = ((int)*(phA+3)) * ((int)*(phB+3));
  hilo[4] = ((int)*(phA+4)) * ((int)*(phB+4));
  hilo[5] = ((int)*(phA+5)) * ((int)*(phB+5));
  hilo[6] = ((int)*(phA+6)) * ((int)*(phB+6));
  hilo[7] = ((int)*(phA+7)) * ((int)*(phB+7));

  *(phHi+0) = (int16)(hilo[0] >> 16);   // Upper 16 bits
  *(phHi+1) = (int16)(hilo[1] >> 16);
  *(phHi+2) = (int16)(hilo[2] >> 16);
  *(phHi+3) = (int16)(hilo[3] >> 16);
  *(phHi+4) = (int16)(hilo[4] >> 16);
  *(phHi+5) = (int16)(hilo[5] >> 16);
  *(phHi+6) = (int16)(hilo[6] >> 16);
  *(phHi+7) = (int16)(hilo[7] >> 16);

  *(phLo+0) = (int16)(hilo[0] & 0x0000ffff);  // Lower 16
  *(phLo+1) = (int16)(hilo[1] & 0x0000ffff);
  *(phLo+2) = (int16)(hilo[2] & 0x0000ffff);
  *(phLo+3) = (int16)(hilo[3] & 0x0000ffff);
  *(phLo+4) = (int16)(hilo[4] & 0x0000ffff);
  *(phLo+5) = (int16)(hilo[5] & 0x0000ffff);
  *(phLo+6) = (int16)(hilo[6] & 0x0000ffff);
  *(phLo+7) = (int16)(hilo[7] & 0x0000ffff);
}
```

Packed Unsigned Multiplication Hi/Lo (2x16-bit) = 8x16-bit

For unsigned in the previous sample, replace all occurrences of *int* with *uint* and *int16* with *uint16*.

Pseudo Vec (X86)

Just as in the handling of upper and lower 16 bits in the previous code, the implementation is very similar.

```
mov    ebx,phB   ; Vector B
mov    eax,phA   ; Vector A
mov    edx,phHi  ; Vector Dest. Upper 16 bits
mov    ecx,phLo  ; Vector Dest. Lower 16 bits
```

vmp_pmulhlH (MMX) Signed Multiplication High/Low (2x16-bit) = 8x16-bit

Listing 9-54: vmp_x86\chap09\pmd\PMulX86M.asm

```
movq   mm0,[ebx+0] ; Read B data {3...0}
movq   mm1,[ebx+8] ;             {7...4}
movq   mm2,[eax+0] ; Read A data {3...0}
movq   mm3,[eax+8] ;             {7...4}
movq   mm4,mm0
movq   mm5,mm1

pmulhw mm0,mm2     ;      upper 16 bits  {3...0}
pmulhw mm1,mm3     ;                     {7...4}
pmullw mm2,mm4     ;      lower 16 bits  {3...0}
pmullw mm3,mm5     ;                     {7...4}

movq   [edx+0],mm0 ; Write Upper 16 bits {3...0}
movq   [edx+8],mm1 ;                     {7...4}
movq   [ecx+0],mm2 ; Write Lower 16 bits {3...0}
movq   [ecx+8],mm3 ;                     {7...4}
```

vmp_pmulhluH (MMX, MMX+, SSE) Unsigned Multiplication High/Low (2x16-bit) = 8x16-bit

For the unsigned version, merely replace *pmulhw* with *pmulhuw*. Note that the same code sample is used and slightly altered as well as whether it was the signed or unsigned version; the lower 16 bits are both handled by *pmullw*, as explained previously.

```
pmulhuw mm0,mm2    ;      upper 16bits  {3...0}
pmulhuw mm1,mm3    ;                    {7...4}
pmullw  mm2,mm4    ;      lower 16bits  {3...0}
pmullw  mm3,mm5    ;                    {7...4}
```

Pseudo Vec (PowerPC)

For AltiVec, the only concern is to keep in mind the endian orientation and the different source versus destination data width types. With that in mind, the even/odd multiplications initiate the pattern of the data within the elements, and it is merely a matter of shuffling the patterns into position. This is only one possible solution for this equation.

Please pay attention to the change in data size, as the even and odd products are handled. This uses odd and even multiplication a wee bit early!

vmp_pmulhlH (AltiVec) Signed Multiplication High/Low (2x16-bit) = 8x16-bit

Listing 9-55: vmp_ppc\chap09\PMulAltivec.cpp

```
vector signed short vhLo, vhHi, vhA,  vhB;
vector signed int   vwE,  vwO,  vwHi, vwLo;

vwE = vec_mule( vhA, vhB );  // [e0 e2 e4 e6] (AB)0,2,4,6  Up/Lo
vwO = vec_mulo( vhA, vhB );  // [o1 o3 o5 o7] (AB)1,3,5,7  Up/Lo
vwHi = vec_mergeh(vwE, vwO); // [e0 o1 e2 o3]  Up/Lo
vwlo = vec_mergel(vwE, vwO); // [e4 o5  e6 o7]  Up/Lo

vhLo = vec_pack(vwHi, vwLo); // [Le0 Lo1 Le2 Lo3 Le4 Lo5  Le6 Lo7]

vwhi = vec_sr(vwHi, V32Shift[16]); // [_Ue0 _Uo1 _Ue2 _Uo3]
vwlo = vec_sr(vwLo, V32Shift[16]); // [_Ue4 _Uo5  _Ue6 _Uo7]

vhHi = vec_pack(vwHi, vwLo); // [Ue0 Uo1 Ue2 Uo3 Ue4 Uo5 Ue6 Uo7]
```

vmp_pmulhlH (AltiVec) Unsigned Multiplication High/Low (2x16-bit) = 8x16-bit

The only difference between signed and unsigned versions is the sign of the data type definitions. The function macros take care of the rest.

```
vector unsigned short vhLo, vhHi, vhA,  vhB;
vector unsigned int   vwE,  vwO,  vwHi, vwLo;
```

Pseudo Vec (MIPS)

As noted earlier, the *pmulth* instruction distributes the full 32-bit signed results into the specified register and two special registers. As such, the data needs to have its upper and lower 16-bit halves separated and then each of them grouped back into sequential order as two groups. Since the specified register receives the even-addressed elements, we need to collate the LO and HI to extract the odd elements into a 128-bit general-purpose register. This is done with the *pmfhl* instruction with the *uw* feature set (*pmfhl.uw*).

vmp_pmulhlH (MMI) Signed Multiplication High/Low (2x16-bit) = 8x16-bit

Listing 9-56: vmp_mips\chap09\pmd\PMulMMI.s

```
lq     t1, 0(a2)      // phA
lq     t2, 0(a3)      // phB
LDELAY                // nop - Load Delay Slot

pmulth   t1,t1,t2     // t1 = {D6 D4 D2 D0} even
```

```
pmfhl.uw t2           // t2 = {D7 D5 D3 D1} odd

pinteh   t0,t2,t1     // Interlace lower 16bits of D7...D0

psrlw    t2,t2,16     // {_D7 _D5 _D3 _D1}
psrlw    t1,t1,16     // {_D6 _D4 _D2 _D0}
pinteh   t1,t2,t1     // Interlace upper 16bits of D7...D0

sq       t0, 0(a1)    // phLo {D7...D0} Lower 16bits
sq       t1, 0(a0)    // phHi {D7...D0} Upper 16bits
```

Pseudo Vec

Are we having fun yet? Finally we have come to the odd and even. That is where the odd fields or the even fields of the 16-bit numbers are selectively multiplied and stored using the wider 32-bit data path. The even fields are the 0, 2nd, 4th, and 6th fields. The odd fields are the 1st, 3rd, 5th, and 7th fields. Note the source 16 bit and destination 32 bit! As before, for unsigned versions, merely replace *int* with *uint* and *int16* with a *uint16* data type.

Packed Signed Multiplication 32-bit Even = 8x16-bit

$D_{even} = A \times B$

Listing 9-57: \chap09\pmd\Pmd.cpp

```
void vmp_pmulleH( int32 * pwD, int16 * phA, int16 * phB )
{
  *(pwD+0) = ((int)*(phA+0)) * ((int)*(phB+0));
  *(pwD+1) = ((int)*(phA+2)) * ((int)*(phB+2));
  *(pwD+2) = ((int)*(phA+4)) * ((int)*(phB+4));
  *(pwD+3) = ((int)*(phA+6)) * ((int)*(phB+6));
}
```

Packed Signed Multiplication 32-bit Odd = 8x16-bit

$D_{odd} = A \times B$

Listing 9-58: \chap09\pmd\Pmd.cpp

```
void vmp_pmulloH( int32 * pwD, int16 * phA, int16 * phB )
{
  *(pwD+0) = ((int)*(phA+1)) * ((int)*(phB+1));
  *(pwD+1) = ((int)*(phA+3)) * ((int)*(phB+3));
  *(pwD+2) = ((int)*(phA+5)) * ((int)*(phB+5));
  *(pwD+3) = ((int)*(phA+7)) * ((int)*(phB+7));
}
```

Pseudo Vec (X86)

These are a little different than before, but they are actually simpler. They are merely the interlaced multiplication, and they store to a different data size, the result of the product — simple!

```
mov   ebx,phB   ; Vector B  (16-bit half-word)
mov   eax,phA   ; Vector A
mov   edx,pwD   ;           (32-bit word)
```

vmp_pmulleH (MMX) Packed Signed Multiplication 32-bit Even = 8x16-bit

Listing 9-59: vmp_x86\chap09\pmd\PMulX86M.asm

```
movq      mm0,[ebx+0] ; B3 B2 B1 B0 - Read B
movq      mm3,[ebx+8] ; B7 B6 B5 B4
movq      mm2,[eax+0] ; A3 A2 A1 A0 - Read A
movq      mm1,[eax+8] ; A7 A6 A5 A4
movq      mm4,mm0
movq      mm5,mm2
punpcklwd mm0,mm3     ; B5 B1 B4 B0
punpckhwd mm4,mm3     ; B7 B3 B6 B2
punpcklwd mm2,mm1     ; A5 A1 A4 A0
punpckhwd mm5,mm1     ; A7 A3 A6 A2

punpcklwd mm0,mm4     ; B6 B4 B2 B0
punpcklwd mm2,mm5     ; A6 A4 A2 A0

movq      mm1,mm0
pmullw    mm0,mm2     ; A6*B6 A4*B4 A2*B2 A0*B0 lower
pmulhw    mm2,mm1     ; A6*B6 A4*B4 A2*B2 A0*B0 upper
movq      mm1,mm0
punpcklwd mm0,mm2     ; H_a2b2 l_a2b2 h_a0b0 l_a0b0
punpckhwd mm1,mm2     ; h_a6b6 l_a6b6 h_a4b4 l_a4b4
movq      [edx+0],mm0 ; Write 4 32bit
movq      [edx+8],mm1
```

vmp_pmulloH (MMX) Packed Signed Multiplication 32-bit Odd = 8x16-bit

This one is also very simple. By substituting the following two lines of code using the *punpckhwd* instruction for the isolated lines using the *punpcklwd* instruction, the odd instructions will be selected. The rest of the code is identical.

```
punpckhwd mm0,mm4     ; B7 B5 B3 B1
punpckhwd mm2,mm5     ; A7 A5 A3 A1
```

vmp_pmulleH (MMX, MMX+, SSE) Unsigned 32-bit Even = 8x16-bit
vmp_pmulluoH (MMX, MMX+, SSE) Unsigned 32-bit Odd = 8x16-bit

For the unsigned versions, merely replace *pmulhw* with the *pmulhuw* instruction.

vmp_pmulleH (SSE2) Packed Signed Multiplication 32-bit Even = 8x16-bit

Listing 9-60: vmp_x86\chap09\pmd\PMulX86M.asm

```
movdqa xmm0,[eax]  ; A7 A6 A5 A4 A3 A2 A1 A0
movdqa xmm1,[ebx]  ; B7 B6 B5 B4 B3 B2 B1 B0
movdqa xmm2,xmm0

pmullw xmm0,xmm1   ;  A7*B7 A6*B6 ... A1*B1 Lo
pmulhw xmm1,xmm2   ;  A7*B7 A6*B6 ... A1*B1 Up

pand   xmm0,lmask  ;  0  l_A6B6 ...   0  l_A0B0
pslld  xmm1,16     ;h_A6B6  0    ... h_A0B0 0
por    xmm0,xmm1
movdqa [edx],xmm0  ; Write D 4x32bits  {even}
```

vmp_pmulloH (SSE2) Packed Signed Multiplication 32-bit Odd = 8x16-bit

For odd elements there only needs to be the insertion of the packed shift right instruction to shift the odd fields into the position previously occupied by the even elements.

```
pmulhw xmm1,xmm2   ;  A7*B7 A6*B6 ... A1*B1 Up
psrld  xmm0,16     ;  0  l_A7B7 ...  0  l_A1B1
```

vmp_pmulleH (MMX, MMX2, SSE) Unsigned 32-bit Even = 8x16-bit
vmp_pmulluoH (MMX, MMX2, SSE) Unsigned 32-bit Odd = 8x16-bit

As before, the unsigned versions merely replace *pmulhw* with the *pmulhuw* instruction.

Pseudo Vec (PowerPC)

Listing 9-61: vmp_ppc\chap09\pmd\PmdAltivec.cpp

```
vmp_pmulleB (AltiVec) Packed Signed Mul. 32-bit Even = 16x8 bit
*(vector signed short *)pwD = vec_mule( (*(vector signed char *)phA),
                                        (*(vector signed char *)phB) );

vmp_pmullueB (AltiVec) Packed Unsigned Mul. 32-bit Even = 16x8 bit
*(vector unsigned short *)pwD = vec_mule(
       (*(vector unsigned char *)phA), (*(vector unsigned char *)phB) );
vmp_pmulleH (AltiVec) Packed Signed Mul. 32-bit Even = 8x16 bit
*(vector signed int *)pwD = vec_mule( (*(vector signed short *)phA),
                                      (*(vector signed short *)phB) );

vmp_pmullueH (AltiVec) Packed Unsigned Mul. 32-bit Even = 8x16 bit
*(vector unsigned int *)pwD = vec_mule( (*(vector unsigned short *)phA),
                                        (*(vector unsigned short *)phB) );

vmp_pmulloB (AltiVec) Packed Signed Mul. 32-bit Odd = 16x8 bit
*(vector signed short *)pwD = vec_mulo( (*(vector signed char *)phA),
                                        (*(vector signed char *)phB) );

vmp_pmulluoB (AltiVec) Packed Unsigned Mul. 32-bit Odd = 16x8 bit
*(vector unsigned short *)pwD = vec_mulo( (*(vector unsigned char *)phA),
                                          (*(vector unsigned char *)phB) );

vmp_pmulloH (AltiVec) Packed Signed Mul. 32-bit Odd = 8x16 bit
*(vector signed int *)pwD = vec_mulo( (*(vector signed short *)phA),
                                      (*(vector signed short *)phB) );

vmp_pmulluoH (AltiVec) Packed Unsigned Mul. 32-bit Odd = 8x16 bit
*(vector unsigned int *)pwD = vec_mulo( (*(vector unsigned short *)phA),
                                        (*(vector unsigned short *)phB) );
```

Pseudo Vec (MIPS)

vmp_pmulleH (MMI) Packed Signed Multiplication 32-bit Even = 8x16-bit

The *pmulth* instruction is pretty handy because the destination register receives the 32-bit even elements.

Listing 9-62: vmp_mips\chap09\pmd\PMulMMMI.s

```
  lq       t1, 0(a1)    // phA
  lq       t2, 0(a2)    // phB
  LDELAY                // nop - Load Delay Slot

  pmulth   t0,t1,t2     // t0  = {D6 D4 D2 D0}
                        // HI  = {D7 D6 D3 D2}
                        // LO  = {D5 D4 D1 D0}
  sq       t0, 0(a0)    // pwD
```

vmp_pmulloH (MMI) Packed Signed Multiplication 32-bit Odd = 8x16-bit

For odd elements, the results in the special-purpose registers HI and LO need to be collated to a 128-bit general-purpose register. So the *pmulth* instruction is followed by:

```
pmfhl.uw t0          // t0 -  {D7 D5 D3 D1}
```

For additional MIPS product functionality, check out the CD or the web sites listed in the references section in the back of the book.

Exercises

1. What is an inner product?
2. A cross product is known by another name. What is it?
3. What happens to a vector if a negative scalar is applied as a product?
4. a) What is the solution for A × B + C × D, if A=2, B=5, C=3, and D=4?
   b) Using A = B = C = D = 0x80000000, calculate the result with and without saturation.
5. What is the equation for a dot product?
6. Given the two vertices v:{–8, 4, –6, 4} and w:{8, 2, –6, 8}, resolve:
   a) $v+w$ b) vw c) $v \bullet w$ d) $v \times w$

Chapter 10
Special Functions

This chapter discusses functionality not related to basic mathematical operators but not quite at the level of trigonometric operations.

CD Workbench Files: /Bench/*architecture*/chap10/*project*/*platform*

	architecture	Special	*project*	*platform*
PowerPC	/vmp_ppc/	**Float**	**/fsf3d/**	/mac9cw
X86	/vmp_x86/	**3D Float**	**/vsf3d/**	/vc6
MIPS	/vmp_mips/	**4vec Float**	**/qvsf3d/**	/vc.net
				/devTool

Min — Minimum

AltiVec	*MMX*	SSE	SSE2	3DNow	3DMX+	*MIPS*	MMI
16 x 8 bit 8 x 16 bit 4 x 32 bit 4 x SP		8 x 8 bit 4 x 16 bit 4 x SP 1 x SPs	16 x 8 bit 8 x 16 bit 2 x DP 1 x DPs	2 x SP	8 x 8 bit 4 x 16 bit		8 x 16 bit 4 x 32 bit

The simplified form of this parallel instruction individually compares the integer or floating-point source arguments and returns the minimum value result in the destination.

```
vD[] = ( vA[] < vB[] ) ? vA[] : vB[];  // an element
```

Pseudo Vec

The previous C equation is a branching equation, which can cause a processor misprediction whether the branch is taken or not. A scalar operation could be done with branchless code, such as the following:

```
__inline MIN( int p, int q )
{             // r=(p < q) ? p : q;
   r = (p-q) >> INT_MAX_BITS;          // (-)=0xFFFFFFFF (+)=0x00000000
   return (p & r) | (q & (r^-1));      // keep lower of p or q
}
```

The two values *p* and *q* are being compared so that the retained value is the smaller one. When subtracting the two values (*p–q*) a negative value is generated if *p* is less than *q*. The sign bit is then arithmetically shifted to the right the size of the data word, which would be a 31-bit shift, thus latching a mask of all 1's. If ($p \geq q$) then (*p–q*) is positive, thus the sign bit of zero would be latched, generating a mask of all zeros. By bit blending with the mask and its inverse, the resulting value will be retained. For processors that do not support this instruction, it can be replicated in parallel using a packed arithmetic shift right or with a packed compare, if they are supported.

NxSP-FP Minimum

AltiVec	vminfp *Dst, aSrc, bSrc* *vD = vec_min(vA, vB)*	Single-Precision	128
3DNow	pfmin *mmDst*, *mmSrc(mm/m64)*	Single-Precision	64
SSE	minps *xmmDst*, *xmmSrc(xmm/m128)*	Single-Precision	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the two (four) individual single-precision floating-point source bit blocks *aSrc (xmmDst)* and *bSrc (xmmSrc)* with the minimum value result being stored in the destination *Dst (xmmDst)*.

```
                                                    (64 bit) 2 x 32 bit
Dst(31...0)   = ( aSrc(31...0)   < bSrc(31...0) )  ? aSrc(31...0)   : bSrc(31...0)
Dst(63...32)  = ( aSrc(63...32)  < bSrc(63...32) ) ? aSrc(63...32)  : bSrc(63...32)
Dst(95...64)  = ( aSrc(95...64)  < bSrc(95...64) ) ? aSrc(95...64)  : bSrc(95...64)
Dst(127...96) = ( aSrc(127...96) < bSrc(127...96) ) ? aSrc(127...96) : bSrc(127...96)
                                                    (128 bit) 4 x 32 bit
```

1xSP-FP Scalar Minimum

SSE	minss x*mmDst*, x*mmSrc(xmm/m32)*	Single-Precision	128

This SIMD instruction is a 128-bit scalar operation that compares only the lowest bit block containing the scalar single-precision floating-point *xmmDst* and *xmmSrc* with the minimum value result being stored in the lowest bit block at destination *xmmDst*; the remaining floating-point bit blocks are left intact.

```
Dst(31...0) = ( aSrc(31...0) < bSrc(31...0) ) ? aSrc(31...0) : bSrc(31...0)
Dst(127...32) Remain the same.
```

1xDP-FP Scalar Minimum

SSE2 minsd x*mmDst*, x*mmSrc(xmm/m64)* Double-Precision 128

This SIMD instruction is a 128-bit scalar operation that compares only the lowest bit block containing the scalar double-precision floating-point *xmmDst* and *xmmSrc* with the minimum value result being stored in the lowest bit block at destination *xmmDst*; the remaining floating-point bit blocks are left intact.

$Dst_{(63...0)} = (aSrc_{(63...0)} < bSrc_{(63...0)}) ? aSrc_{(63...0)} : bSrc_{(63...0)}$
$Dst_{(127...64)}$ Remain the same.

Nx8-bit Minimum Integer

AltiVec	vminub *Dst*, *aSrc*, *bSrc*	Unsigned	128
	vminsb *Dst*, *aSrc, bSrc*	Signed	
	vD = vec_min(vA, vB)		
MMX+	pminub *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
SSE	pminub *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
SSE2	pminub *xmmDst*, *xmmSrc(xmm/m128)*	Unsigned	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the eight (16) individual 8-bit source integer bit blocks *aSrc (xmmDst)* and *bSrc (xmmSrc)* with the minimum value result being stored in the destination *Dst (xmmDst)*.

(64 bit) **8 x 8 bit**

$Dst_{(7...0)} = (aSrc_{(7...0)} < bSrc_{(7...0)}) ? aSrc_{(7...0)} : bSrc_{(7...0)}$
$Dst_{(15...8)} = (aSrc_{(15...8)} < bSrc_{(15...8)}) ? aSrc_{(15...8)} : bSrc_{(15...8)}$
$Dst_{(23...16)} = (aSrc_{(23...16)} < bSrc_{(23...16)}) ? aSrc_{(23...16)} : bSrc_{(23...16)}$
$Dst_{(31...24)} = (aSrc_{(31...24)} < bSrc_{(31...24)}) ? aSrc_{(31...24)} : bSrc_{(31...24)}$
$Dst_{(39...32)} = (aSrc_{(39...32)} < bSrc_{(39...32)}) ? aSrc_{(39...32)} : bSrc_{(39...32)}$
$Dst_{(47...40)} = (aSrc_{(47...40)} < bSrc_{(47...40)}) ? aSrc_{(47...40)} : bSrc_{(47...40)}$
$Dst_{(55...48)} = (aSrc_{(55...48)} < bSrc_{(55...48)}) ? aSrc_{(55...48)} : bSrc_{(55...48)}$
$Dst_{(63...56)} = (aSrc_{(63...56)} < bSrc_{(63...56)}) ? aSrc_{(63...56)} : bSrc_{(63...56)}$
$Dst_{(71...64)} = (aSrc_{(71...64)} < bSrc_{(71...64)}) ? aSrc_{(71...64)} : bSrc_{(71...64)}$
$Dst_{(79...72)} = (aSrc_{(79...72)} < bSrc_{(79...72)}) ? aSrc_{(79...72)} : bSrc_{(79...72)}$
$Dst_{(87...80)} = (aSrc_{(87...80)} < bSrc_{(87...80)}) ? aSrc_{(87...80)} : bSrc_{(87...80)}$
$Dst_{(95...88)} = (aSrc_{(95...88)} < bSrc_{(95...88)}) ? aSrc_{(95...88)} : bSrc_{(95...88)}$
$Dst_{(103...96)} = (aSrc_{(103...96)} < bSrc_{(103...96)}) ? aSrc_{(103...96)} : bSrc_{(103...96)}$
$Dst_{(111...104)} = (aSrc_{(111...104)} < bSrc_{(111...104)}) ? aSrc_{(111...104)} : bSrc_{(111...104)}$
$Dst_{(119...112)} = (aSrc_{(119...112)} < bSrc_{(119...112)}) ? aSrc_{(119...112)} : bSrc_{(119...112)}$
$Dst_{(127...120)} = (aSrc_{(127...120)} < bSrc_{(127...120)}) ? aSrc_{(127...120)} : bSrc_{(127...120)}$

(128 bit) **16 x 8 bit**

Nx16-bit Integer Minimum

AltiVec	vminuh *Dst, aSrc, bSrc*	Unsigned	128
	vminsh *Dst, aSrc, bSrc*	Signed	
	vD = vec_min(vA, vB)		
SSE	pminsw *mmDst*, *mmSrc(mm/m64)*	Signed	64
SSE2	pminsw x*mmDst*, x*mmSrc(xmm/m128)*	Signed	128
MMI	pminh *Dst*, a*Src, bSrc*	Signed	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the four (eight) individual 16-bit source integer bit blocks *aSrc (xmmDst)* and *bSrc (xmmSrc)* with the minimum value result being stored in the destination *Dst (xmmDst)*.

```
                                                                (64 bit) 4 x 16 bit
Dst(15...0)   = ( aSrc(15...0)   < bSrc(15...0) )   ? aSrc(15...0)   : bSrc(15...0)
Dst(31...16)  = ( aSrc(31...16)  < bSrc(31...16) )  ? aSrc(31...16)  : bSrc(31...16)
Dst(47...32)  = ( aSrc(47...32)  < bSrc(47...32) )  ? aSrc(47...32)  : bSrc(47...32)
Dst(63...48)  = ( aSrc(63...48)  < bSrc(63...48) )  ? aSrc(63...48)  : bSrc(63...48)
Dst(79...64)  = ( aSrc(79...64)  < bSrc(79...64) )  ? aSrc(79...64)  : bSrc(79...64)
Dst(95...80)  = ( aSrc(95...80)  < bSrc(95...80) )  ? aSrc(95...80)  : bSrc(95...80)
Dst(111...96) = ( aSrc(111...96) < bSrc(111...96) ) ? aSrc(111...96) : bSrc(111...96)
Dst(127...112)= ( aSrc(127...112)< bSrc(127...112) )? aSrc(127...112): bSrc(127...112)
                                                                (128 bit) 8 x 16 bit
```

4x32-bit Integer Minimum

AltiVec	vminuw *Dst, aSrc, bSrc*	Unsigned	128
	vminsw *Dst, aSrc, bSrc*	Signed	
	vD = vec_min(vA, vB)		
MMI	pminw *Dst*, a*Src, bSrc*	Signed	128

This SIMD instruction is a 128-bit parallel operation that compares the four individual 32-bit source integer bit blocks *aSrc* and *bSrc* with the minimum value result being stored in the destination *Dst*.

```
Dst(31...0)   = ( aSrc(31...0)   < bSrc(31...0) )   ? aSrc(31...0)   : bSrc(31...0)
Dst(63...32)  = ( aSrc(63...32)  < bSrc(63...32) )  ? aSrc(63...32)  : bSrc(63...32)
Dst(95...64)  = ( aSrc(95...64)  < bSrc(95...64) )  ? aSrc(95...64)  : bSrc(95...64)
Dst(127...96) = ( aSrc(127...96) < bSrc(127...96) ) ? aSrc(127...96) : bSrc(127...96)
```

Max — Maximum

AltiVec	*MMX*	**SSE**	**SSE2**	**3DNow**	*3DMX+*	*MIPS*	**MMI**
16 x 8 bit 8 x 16 bit 4 x 32 bit 4 x SP		8 x 8 bit 4 x 16 bit 4 x SP 1 x SPs	16 x 8 bit 8 x 16 bit 2 x DP 1 x DPs	2 x SP	8 x 8 bit 4 x 16 bit		8 x 16 bit 4 x 32 bit

The simplified form of this parallel instruction individually compares the integer or floating-point source arguments and returns the maximum value result in the destination.

```
vD[] = ( vA[] > vB[] ) ? vA[] : vB[];  // an element
```

NxSP-FP Maximum

AltiVec	vmaxfp *Dst, aSrc, bSrc* *vD = vec_max(vA, vB)*	Single-Precision	128
3DNow	pfmax *mmDst*, *mmSrc(mm/m64)*	Single-Precision	64
SSE	maxps *xmmDst*, *xmmSrc(xmm/m128)*	Single-Precision	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the two (four) individual single-precision floating-point source bit blocks *aSrc (xmmDst)* and *bSrc (xmmSrc)* with the maximum value result being stored in the destination *Dst (xmmDst)*.

```
                                                        (64bit) 2x32bit
Dst(31...0)   = ( aSrc(31...0)   > bSrc(31...0) )   ? aSrc(31...0)   : bSrc(31...0)
Dst(63...32)  = ( aSrc(63...32)  > bSrc(63...32) )  ? aSrc(63...32)  : bSrc(63...32)
Dst(95...64)  = ( aSrc(95...64)  > bSrc(95...64) )  ? aSrc(95...64)  : bSrc(95...64)
Dst(127...96) = ( aSrc(127...96) > bSrc(127...96) ) ? aSrc(127...96) : bSrc(127...96)
                                                        (128bit) 4x32bit
```

1xSP-FP Scalar Maximum

SSE	maxss x*mmDst*, x*mmSrc(xmm/m32)*	Single-Precision	128

This SIMD instruction is a 128-bit scalar operation that compares only the lowest bit block containing the scalar single-precision floating-point *xmmDst* and *xmmSrc* with the maximum value result being stored in the lowest element at destination *xmmDst*; the remaining floating-point bit blocks are left intact.

```
Dst(31...0)   = ( aSrc(31...0)   > bSrc(31...0) ) ? aSrc(31...0)   : bSrc(31...0)
Dst(127...32) Remain the same.
```

1xDP-FP Scalar Maximum

SSE2	maxsd x*mmDst*, x*mmSrc(xmm/m64)*	Double-Precision	128

This SIMD instruction is a 128-bit scalar operation that compares only the lowest element containing the scalar double-precision floating-point *xmmDst* and *xmmSrc* with the maximum value result being stored

in the lowest element at destination *xmmDst*; the remaining floating-point bit blocks are left intact.

$Dst_{(63...0)} = (aSrc_{(63...0)} > bSrc_{(63...0)}) ? aSrc_{(63...0)} : bSrc_{(63...0)}$
$Dst_{(127...64)}$ Remain the same.

Nx8-bit Integer Maximum

AltiVec	vmaxub *Dst*, *aSrc, bSrc*	Unsigned	128
	vmaxsb *Dst*, *aSrc, bSrc*	Signed	
	vD = vec_max(vA, vB)		
MMX+	pmaxub *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
SSE	pmaxub *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
SSE2	pmaxub *xmmDst*, *xmmSrc(xmm/m128)*	Unsigned	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the eight (16) individual 8-bit source integer bit blocks *aSrc (xmmDst)* and *bSrc (xmmSrc)* with the maximum value result being stored in the destination *Dst (xmmDst)*.

```
                                                                    (64bit) 8x8bit
Dst(7...0)     = ( aSrc(7...0)     > bSrc(7...0) ) ? aSrc(7...0)     : bSrc(7...0)
Dst(15...8)    = ( aSrc(15...8)    > bSrc(15...8) ) ? aSrc(15...8)   : bSrc(15...8)
Dst(23...16)   = ( aSrc(23...16)   > bSrc(23...16) ) ? aSrc(23...16) : bSrc(23...16)
Dst(31...24)   = ( aSrc(31...24)   > bSrc(31...24) ) ? aSrc(31...24) : bSrc(31...24)
Dst(39...32)   = ( aSrc(39...32)   > bSrc(39...32) ) ? aSrc(39...32) : bSrc(39...32)
Dst(47...40)   = ( aSrc(47...40)   > bSrc(47...40) ) ? aSrc(47...40) : bSrc(47...40)
Dst(55...48)   = ( aSrc(55...48)   > bSrc(55...48) ) ? aSrc(55...48) : bSrc(55...48)
Dst(63...56)   = ( aSrc(63...56)   > bSrc(63...56) ) ? aSrc(63...56) : bSrc(63...56)
Dst(71...64)   = ( aSrc(71...64)   > bSrc(71...64) ) ? aSrc(71...64) : bSrc(71...64)
Dst(79...72)   = ( aSrc(79...72)   > bSrc(79...72) ) ? aSrc(79...72) : bSrc(79...72)
Dst(87...80)   = ( aSrc(87...80)   > bSrc(87...80) ) ? aSrc(87...80) : bSrc(87...80)
Dst(95...88)   = ( aSrc(95...88)   > bSrc(95...88) ) ? aSrc(95...88) : bSrc(95...88)
Dst(103...96)  = ( aSrc(103...96)  > bSrc(103...96) ) ? aSrc(103...96) : bSrc(103...96)
Dst(111...104) = ( aSrc(111...104) > bSrc(111...104) ) ? aSrc(111...104) : bSrc(111...104)
Dst(119...112) = ( aSrc(119...112) > bSrc(119...112) ) ? aSrc(119...112) : bSrc(119...112)
Dst(127...120) = ( aSrc(127...120) > bSrc(127...120) ) ? aSrc(127...120) : bSrc(127...120)
                                                                    (128bit) 16x8bit
```

Nx16-bit Integer Maximum

AltiVec	vmaxuh *Dst, aSrc, bSrc*	Unsigned	128
	vmaxsh *Dst, aSrc, bSrc*	Signed	
	vec_max(vA, vB)		
SSE	pmaxsw *mmDst*, *mmSrc(mm/m64)*	Signed	64
SSE2	pmaxsw x*mmDst*, x*mmSrc(xmm/m128)*	Signed	128
MMI	pmaxh *Dst*, a*Src, bSrc*	Signed	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the four (eight) individual 16-bit source integer bit blocks *aSrc (xmmDst)* and *bSrc (xmmSrc)* with the maximum value result being stored in the destination *Dst (xmmDst)*.

```
                                                                (64bit) 4x16bit
Dst(15...0)     = ( aSrc(15...0)     > bSrc(15...0) )   ? aSrc(15...0)     : bSrc(15...0)
Dst(31...16)    = ( aSrc(31...16)    > bSrc(31...16) )  ? aSrc(31...16)    : bSrc(31...16)
Dst(47...32)    = ( aSrc(47...32)    > bSrc(47...32) )  ? aSrc(47...32)    : bSrc(47...32)
Dst(63...48)    = ( aSrc(63...48)    > bSrc(63...48) )  ? aSrc(63...48)    : bSrc(63...48)
Dst(79...64)    = ( aSrc(79...64)    > bSrc(79...64) )  ? aSrc(79...64)    : bSrc(79...64)
Dst(95...80)    = ( aSrc(95...80)    > bSrc(95...80) )  ? aSrc(95...80)    : bSrc(95...80)
Dst(111...96)   = ( aSrc(111...96)   > bSrc(111...96) ) ? aSrc(111...96)   : bSrc(111...96)
Dst(127...112)  = ( aSrc(127...112)  > bSrc(127...112) )? aSrc(127...112)  : bSrc(127...112)
                                                                (128bit) 8x16bit
```

4x32-bit Integer Maximum

AltiVec	vmaxuw *Dst, aSrc, bSrc*	Unsigned	128
	vmaxsw *Dst, aSrc, bSrc*	Signed	
	vD = vec_max(vA, vB)		
MMI	pmaxw *Dst*, a*Src, bSrc*	Signed	128

This SIMD instruction is a 128-bit parallel operation that compares the four individual 32-bit source integer bit blocks *aSrc* and *bSrc* with the maximum value result being stored in the destination *Dst*.

```
Dst(31...0)   = ( aSrc(31...0)   > bSrc(31...0) )   ? aSrc(31...0)   : bSrc(31...0)
Dst(63...32)  = ( aSrc(63...32)  > bSrc(63...32) )  ? aSrc(63...32)  : bSrc(63...32)
Dst(95...64)  = ( aSrc(95...64)  > bSrc(95...64) )  ? aSrc(95...64)  : bSrc(95...64)
Dst(127...96) = ( aSrc(127...96) > bSrc(127...96) ) ? aSrc(127...96) : bSrc(127...96)
```

Vector Min and Max

As discussed, the minimum and maximum functions have very similar expressions, except for the difference in comparison and thus their value assignment.

Pseudo Vec

Listing 10-1: \chap10\qvsf3d\QVsf3D.cpp

```
void vmp_QVecMin( vmp3DQVector * const pvD,
            const vmp3DQVector * const pvA,
            const vmp3DQVector * const pvB )
{
  pvD->x = (pvA->x < pvB->x) ? pvA->x : pvB->x;
  pvD->y = (pvA->y < pvB->y) ? pvA->y : pvB->y;
  pvD->z = (pvA->z < pvB->z) ? pvA->z : pvB->z;
  pvD->w = (pvA->w < pvB->w) ? pvA->w : pvB->w;
}
```

Listing 10-2: \chap10\qvsf3d\QVsf3D.cpp

```
void vmp_QVecMax( vmp3DQVector * const pvD,
            const vmp3DQVector * const pvA,
            const vmp3DQVector * const pvB )
{
  pvD->x = (pvA->x > pvB->x) ? pvA->x : pvB->x;
  pvD->y = (pvA->y > pvB->y) ? pvA->y : pvB->y;
  pvD->z = (pvA->z > pvB->z) ? pvA->z : pvB->z;
  pvD->w = (pvA->w > pvB->w) ? pvA->w : pvB->w;
}
```

To handle a three-float vector or unaligned data, refer back to the explanations for correcting alignment, as demonstrated in Chapter 8, "Vector Addition and Subtraction."

Now examine these floating-point functions more closely using X86 assembly. As MMX does not support floating-point, only 3DNow! and SSE can be utilized.

Pseudo Vec (X86)

3DNow! supports 64 bit, so two loads as well as saves must be handled simultaneously, but it is a simple matter of adding the two pairs of floats to each other.

```
mov   eax,vA    ; Vector A
mov   ebx,vB    ; Vector B
mov   edx,vD    ; Vector Destination
```

vmp_QVecMin (3DNow!)

The following is very similar to the 3DNow! versions of the addition, subtraction, multiplication, and division code that you examined earlier.

Listing 10-3: vmp_x86\chap10\qvsf3d\QVsf3DX86M.asm

```
movq  mm0,[eax]                ; {Ay   Ax}
movq  mm1,[eax+8]              ; {Aw   Az}
movq  mm2,[ebx]                ; {By   Bx}
movq  mm3,[ebx+8]              ; {Bw   Bz}
pfmin mm0,mm2                  ; {Ay<By Ax<Bx}
pfmin mm1,mm3                  ; {Aw<Bw Az<Bz}
movq  [edx+0],mm0              ; {Miny   Minx}
movq  [edx+8],mm1              ; {Minw   Minz}
```

vmp_QVecMax (3DNow!) SPFP

Max code is also the same by merely replacing *pfmin* with a *pfmax* instruction.

vmp_QVecMin (SSE) SPFP Aligned

The following is very similar to the SSE versions of the addition, subtraction, multiplication, and division code that you examined earlier. This version uses aligned 128-bit source and/or destination data, but by changing *movaps* to *movups*, unaligned data can optimally be used.

Listing 10-4: vmp_x86\chap10\qvsf3d\QVsf3DX86M.asm

```
movaps xmm0,[eax]          ;vA.xyzw {Aw Az Ay Ax}
movaps xmm1,[ebx]          ;vB.xyzw {Bw Bz By Bx}
minps  xmm0,xmm1                  ; {Aw<Bw ... Ax<Bx}
movaps [edx],xmm0                 ; {Minw  ...  Minx}
```

vmp_QVecMax (SSE) SPFP

Max code is also the same by merely replacing *minps* with a *maxps* instruction.

Pseudo Vec (PowerPC)

Using the aligned quad vector data, a min or max function is simply an AltiVec library call.

vmp_QVecMin (AltiVec) SPFP Aligned

Listing 10-5: vmp_ppc\chap10\qvsf3d\QVsf3DAltivec.cpp

```
*(vector float *)pvD = vec_min(
             (*(vector float *)pvA),
             (*(vector float *)pvB) );
```

vmp_QVecMax (AltiVec) SPFP Aligned

Listing 10-6: vmp_ppc\chap10\qvsf3d\QVsf3DAltivec.cpp

```
*(vector float *)pvD = vec_max(
             (*(vector float *)pvA),
             (*(vector float *)pvB) );
```

Pseudo Vec (MIPS)

Similar to the AltiVec, the quad vector min and max are pretty much straightforward. VU coprocessor instructions exist for handling it.

For specific information, see your PS2 Linux Kit or devTool manual.

CMP — Packed Comparison

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI
16 x 8 bit 8 x 16 bit 4 x 32 bit	16 x 8 bit 8 x 16 bit 4 x 32 bit	4 x SP 1 x SPs	16 x 8 bit 8 x 16 bit 4 x 32 bit 4 x DP 1 x DPs	2 x SP			16 x 8 bit 8 x 16 bit 4 x 32 bit

This SIMD instruction is a 64-bit (128-bit) parallel operation that compares the individual {eight 8 bit, four 16 bit, or two 32 bit} ({sixteen 8 bit, eight 16 bit, or four 32 bit}) source fields. If *aSrc (xmmDst)* meets the condition to *bSrc (xmmSrc)*, then all 1's will be set in the destination *Dst (xmmDst)* for that field. If not, all zeros will be set for that field. This is very similar to the min and max functionality discussed earlier in this chapter.

```
vD[] = ( vA[] ? vB[] ) ? –1 : 0;  // an element
```

Packed Compare if Equal to (=)

AltiVec	vcmpeqfp *Dst, aSrc, bSrc*	Single-Precision	128
	vcmpequ{b/h/w} *Dst, aSrc, bSrc* *vD = vec_cmpeq(vA, vB)*	[Un]signed	
MMX	pcmpeq{b/w/d} *mmDst*, *mmSrc*	[Un]signed	64
3DNow	pfcmpeq *mmDst, mmSrc*	Single-Precision	64
SSE	cmpps *xmmDst, xmmSrc,0*	Single-Precision	128
	cmpss *xmmDst, r32, 0*	Single-Precision Scalar	
SSE2	cmppd *xmmDst, xmmSrc,0*	Double-Precision	128
	cmpsd *xmmDst, r32, 0*	Double-Precision Scalar	
	pcmpeq{b/w/d} *xmmDst*, *xmmSrc*	[Un]signed	128
MMI	pceq{b/h/w} *Dst*, *aSrc, bSrc*	[Un]signed	128

Packed Compare if Greater Than or Equal (≥)

AltiVec	*vcmpgefp Dst, aSrc, bSrc* *vD = vec_cmpge(vA, vB)*	Single-Precision	128
3DNow	pfcmpge *mmDst, mmSrc*	Single-Precision	64
SSE	cmpps *xmmDst, xmmSrc,5*	Single-Precision	128
	cmpss *xmmDst, r32, 5*	Single-Precision Scalar	

SSE2	cmppd *xmmDst, xmmSrc,5*	Double-Precision	128
	cmpsd *xmmDst, r32, 5*	Double-Precision Scalar	

Packed Compare if Greater Than (>)

AltiVec	vcmpgtfp *Dst, aSrc, bSrc*	Single-Precision	128
	vcmpgtu{b/h/w} *Dst*, *aSrc, bSrc*	Unsigned	
	vcmpgts{b/h/w} *Dst*, *aSrc, bSrc*	Signed	
	vD = vec_cmpgt(vA, vB)		
MMX	pcmpgt{b/w/d} *mmDst,mmSrc*	Signed	64
3DNow	pfcmpgt *mmDst, mmSrc*	Single-Precision	64
SSE	cmpps *xmmDst, xmmSrc,6*	Single-Precision	128
	cmpss *xmmDst, r32, 6*	Single-Precision Scalar	
SSE2	cmppd *xmmDst, xmmSrc,6*	Double-Precision	128
	cmpsd *xmmDst, r32, 6*	Double-Precision Scalar	
	pcmpgt{b/w/d} *Dst*, *aSrc, bSrc*	Signed	
MMI	pcgt{b/h/w} *Dst*, *aSrc, bSrc*	Signed	128

Note: AltiVec uses vec_cmple() and vec_cmplt(), which are inverses of these listed comparison functions.

Note: SSE and SSE2 use an immediate value to determine the type of condition for single-precision and double-precision values.

```
0   1   2   3    4   5   6   7
=   <       ~ORD         >   ORD
```

Hint: Just as an assurance, the following is a guide so that if you are either too tired to think or do not have the elementary foundations of comparisons down, each comparison and its complement is shown:

```
    <          =          >

LT – GE    EQ – NE    LE – GT
```

Absolute

```
vD = | vA[] |
```

AltiVec	*MMX*	*SSE*	*SSE2*	*3DNow*	*3DMX+*	*MIPS*	**MMI**

Packed N-bit Absolute

MMI	pabs(h/w) *Dst, aSrc*	Signed	128

Absolute value is not really supported by the many processors but can be easily simulated. For floating-point, it is merely a clearing of the sign bit. For packed integers, use the comparison mask or sign bit in a use of masking logic and implement a two's complement to negate.

For a review of two's complement, see Chapter 6, "Bit Mangling."

```
float f;

if (f < 0.0)
{
  f = -f;
}
```

```
*((unsigned int *)&f) &= 0x7FFFFFFF;
```

```
int i;

if (i < 0)
{
  i = -i;
}
```

```
int msk;

msk = i >> 31;      // 0 or -1
i ^= msk;           // Flip bits
i += msk & 1;       // 0 or 1?
```

Averages

Simulated averages were discussed briefly in the Chapter 4, "Vector Methodologies."

```
VD[] = ( vA[] + vB[] + 1 ) >> 1;
```

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

Nx8-bit [Un]signed Integer Average

AltiVec	vavgub *Dst, aSrc, bSrc*	Unsigned	128
	vavgsb *Dst, aSrc, bSrc*	Signed	
	vD = vec_avg(vA, vB)		
3DNow	pavgusb *mmDst, mmSrc(mm/m64)*	Unsigned	64
MMX+	pavgb *mmDst, mmSrc(mm/m64)*	Unsigned	64
SSE	pavgb *mmDst, mmSrc(mm/m64)*	Unsigned	64
SSE2	pavgb *xmmDst, xmmSrc(xmm/m128)*	Unsigned	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that sums the eight (16) individual 8-bit source integer bit blocks *aSrc* (x*mmSrc)* and *bSrc* (x*mmDst),* adds one, and divides by two, returning the lower 8 bits with the result being stored in the destination *Dst* (x*mmDst)*.

```
                                                    (64 bit) 8 x 8 bit
Dst(7...0)     = ( aSrc(7...0)     +1+ bSrc(7...0)     ) >> 1
Dst(15...8)    = ( aSrc(15...8)    +1+ bSrc(15...8)    ) >> 1
Dst(23...16)   = ( aSrc(23...16)   +1+ bSrc(23...16)   ) >> 1
Dst(31...24)   = ( aSrc(31...24)   +1+ bSrc(31...24)   ) >> 1
Dst(39...32)   = ( aSrc(39...32)   +1+ bSrc(39...32)   ) >> 1
Dst(47...40)   = ( aSrc(47...40)   +1+ bSrc(47...40)   ) >> 1
Dst(55...48)   = ( aSrc(55...48)   +1+ bSrc(55...48)   ) >> 1
Dst(63...56)   = ( aSrc(63...56)   +1+ bSrc(63...56)   ) >> 1
Dst(71...64)   = ( aSrc(71...64)   +1+ bSrc(71...64)   ) >> 1
Dst(79...72)   = ( aSrc(79...72)   +1+ bSrc(79...72)   ) >> 1
Dst(87...80)   = ( aSrc(87...80)   +1+ bSrc(87...80)   ) >> 1
Dst(95...88)   = ( aSrc(95...88)   +1+ bSrc(95...88)   ) >> 1
Dst(103...96)  = ( aSrc(103...96)  +1+ bSrc(103...96)  ) >> 1
Dst(111...104) = ( aSrc(111...104) +1+ bSrc(111...104) ) >> 1
Dst(119...112) = ( aSrc(119...112) +1+ bSrc(119...112) ) >> 1
Dst(127...120) = ( aSrc(127...120) +1+ bSrc(127...120) ) >> 1
                                                   (128 bit) 16 x 8 bit
```

Nx16-bit [Un]signed Integer Average

AltiVec	vavguh *Dst, aSrc, bSrc*	Unsigned	128
	vavgsh *Dst, aSrc, bSrc*	Signed	
	vD = vec_avg(vA, vB)		
MMX+	pavgw *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
SSE	pavgw *mmDst, mmSrc(mm/m64)*	Unsigned	64
SSE2	pavgw *xmmDst, xmmSrc(xmm/m128)*	Unsigned	128

This SIMD instruction is a 64-bit (128-bit) parallel operation that sums the four (eight) individual 16-bit source integer bit blocks *aSrc* (x*mmSrc), bSrc*, (x*mmDst)*, adds one, and divides by two, returning the lower 16 bits with the result being stored in the destination *Dst* (x*mmDst)*.

```
                                                    (64 bit) 4 x 16 bit
Dst(15...0)    = ( aSrc(15...0)    +1+ bSrc(15...0)    ) >> 1
Dst(31...16)   = ( aSrc(31...16)   +1+ bSrc(31...16)   ) >> 1
Dst(47...32)   = ( aSrc(47...32)   +1+ bSrc(47...32)   ) >> 1
Dst(63...48)   = ( aSrc(63...48)   +1+ bSrc(63...48)   ) >> 1
Dst(79...64)   = ( aSrc(79...64)   +1+ bSrc(79...64)   ) >> 1
Dst(95...80)   = ( aSrc(95...80)   +1+ bSrc(95...80)   ) >> 1
Dst(111...96)  = ( aSrc(111...96)  +1+ bSrc(111...96)  ) >> 1
Dst(127...112) = ( aSrc(127...112) +1+ bSrc(127...112) ) >> 1
                                                   (128 bit) 8 x 16 bit
```

4x32-bit [Un]signed Integer Average

AltiVec	vavguw *Dst, aSrc, bSrc*	Unsigned	128
	vavgsw *Dst, aSrc, bSrc*	Signed	
	vD = vec_avg(vA, vB)		

This SIMD instruction is a 128-bit parallel operation that sums the four individual 32-bit source integer bit blocks *aSrc* and *bSrc,* adds one, and divides by two, returning the lower 32 bits with the result being stored in the destination *Dst*.

$$Dst_{(31\ldots0)} = (aSrc_{(31\ldots0)} +1+ bSrc_{(31\ldots0)}) >> 1$$
$$Dst_{(63\ldots32)} = (aSrc_{(63\ldots32)} +1+ bSrc_{(63\ldots32)}) >> 1$$
$$Dst_{(95\ldots64)} = (aSrc_{(95\ldots64)} +1+ bSrc_{(95\ldots64)}) >> 1$$
$$Dst_{(127\ldots96)} = (aSrc_{(127\ldots96)} +1+ bSrc_{(127\ldots96)}) >> 1$$

Sum of Absolute Differences

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

The simplified form of this parallel instruction individually calculates the differences of each of the packed bits and then sums the absolute value for all of them, returning the result in the destination.

```
VD[0] =  | vA[0] – vB[0] | + ... + | vA[n–1] – vB[n–1] |;
```

8x8-bit Sum of Absolute Differences

SSE	psadbw *mmDst*, *mmSrc(mm/m64)*	Unsigned	64
MMX+	psadbw *mmDst*, *mmSrc(mm/m64)*	Unsigned	64

```
Dst(15...0) = abs( Src(7...0)   – Dst(7...0) )   + abs( Src(15...8)  – Dst(15...8) )
            + abs( Src(23...16) – Dst(23...16) ) + abs( Src(31...24) – Dst(31...24) )
            + abs( Src(39...32) – Dst(39...32) ) + abs( Src(47...40) – Dst(47...40) )
            + abs( Src(55...48) – Dst(55...48) ) + abs( Src(63...56) – Dst(63...56) )
Dst(63...16) = 0;
```

16x8-bit Sum of Absolute Differences

SSE2	psadbw *xmmDst*, *xmmSrc(xmm/m128)*	Unsigned	128

```
Dst(31...16) = abs( Src(71...64)   – Dst(71...64) )   + abs( Src(79...72)   – Dst(79...72) )
             + abs( Src(87...80)   – Dst(87...80) )   + abs( Src(95...88)   – Dst(95...88) )
             + abs( Src(103...96)  – Dst(103...96) )  + abs( Src(111...104) – Dst(111...104) )
             + abs( Src(119...112) – Dst(119...112) ) + abs( Src(127...120) – Dst(127...120) )
Dst(127...32) = 0
```

SQRT — Square Root

AltiVec	MMX	SSE	SSE2	3DNow	3DMX+	MIPS	MMI

The reciprocal and square root are two mathematical operations that have special functionality with vector processors. The division operation is typically performed by multiplying the reciprocal of the denominator by the numerator. A square root is not always just a square root. Sometimes it is a reciprocal square root. So first, we examine some simple forms of these.

Equation 10-1: Reciprocal

$$y \div x = \frac{y}{1} \bullet \frac{1}{x} = \frac{y}{x} = \frac{y^1}{x^1} = y^1 \bullet x^{-1} = y^1 x^{-1}$$

$$\text{So } \frac{1}{x} = x^{-1}$$

Equation 10-2: Square root

$$\sqrt{x} = x^{\frac{1}{2}} \quad \frac{1}{\sqrt{x}} = x^{-\frac{1}{2}} \quad \text{so } \frac{x}{\sqrt{x}} = x^{1-\frac{1}{2}} = x^{\frac{1}{2}} = \sqrt{x}$$

Another way to remember this is:

$$\frac{x}{\sqrt{x}} = \frac{\sqrt{x} \bullet \sqrt{x}}{\sqrt{x}} = \frac{\sqrt{x} \bullet \cancel{\sqrt{x}}}{\cancel{\sqrt{x}}} = \frac{\sqrt{x}}{1} = \sqrt{x}$$

The simplified form of this parallel instruction individually calculates the square root of each of the packed floating-point values and returns the result in the destination. Some processors support the square root instruction directly, but some processors, such as in the case of the 3DNow! instruction set, actually support it indirectly through instructional stages. Some processors, such as AltiVec, support it as a reciprocal square root.

So now I pose a little problem. We hopefully all know that a negative number should never be passed into a square root because computers go boom, as they have no idea how to deal with an identity (**i**).

$$\sqrt{-x} = i\sqrt{x}$$

With that in mind, what is wrong with a reciprocal square root? Remember your calculus and limits?

$$\lim_{x \to 0^+} \frac{1}{\sum \sqrt{x}}$$

Hint: *x* approaches zero from the right.

Okay, how about this one?

$$\lim_{x \to 0^+} \frac{1}{\sum x}$$

Do you see it now? You cannot divide by zero, as it results in infinity and is mathematically problematic. So what has to be done is to trap for the *x* being too close to zero (as *x* approaches zero) and then substitute the value of one as the solution for the reciprocal square root.

```
y = ( x < 0.0000001 ) ? 1.0 : (1 / sqrt(x));    // Too close to zero
```

It is not perfect, but it is a solution. The number is so close to infinity that the result of its product upon another number is negligible. So in essence, the result is that other number, thus the multiplicative identity comes to mind (1 * n = n). But how do we deal with this in vectors? Well, you just learned the trick in this chapter! Remember the packed comparison? It is just a matter of using masking and bit blending. The AltiVec code will explain this little trick later in this section.

In the case of a reciprocal square root, the square root can be easily achieved by merely multiplying the result by the original *x* value, thus achieving the desired square root. Remember, the square of a square root is the original value!

$$\sqrt{X} = X^{\frac{1}{2}} \qquad \frac{1}{X} = X^{-2} \qquad \frac{1}{\sqrt{X}} = X^{-\frac{1}{2}}$$

$$\sqrt{X} = \frac{X}{1} \times \frac{1}{\sqrt{X}} = x^{1-\frac{1}{2}} = X^{\frac{1}{2}}$$

```
vD[] = √(vA[]);
```

1xSP-FP Scalar Square Root

SSE sqrtss *xmmDst*, *xmmSrc(xmm/m32)* Single-Precision 128

This SIMD instruction is a 128-bit scalar operation that calculates the square root of only the lowest single-precision floating-point element containing the scalar *xmmSrc,* and the result is stored in the lowest single-precision floating-point block at destination *xmmDst*; the remaining bit blocks are left intact.

$Dst_{(31...0)}$ = sqrt($Src_{(31...0)}$)
$Dst_{(127...32)}$ = remains unchanged

```
movss  xmm0,fA       ; {0 0 0  A}
sqrtss xmm0,xmm0     ; {0 0 0 √A}
```

4xSP-FP Square Root

SSE sqrtps *xmmDst*, *xmmSrc(xmm/m128)* Single-Precision 128

This SIMD instruction is a 128-bit parallel operation that calculates the square root of the four single-precision floating-point blocks contained within *xmmSrc* and stores the result in the single-precision floating-point blocks at destination *xmmDst.*

$Dst_{(31...0)}$ = sqrt($Src_{(31...0)}$)
$Dst_{(63...32)}$ = sqrt($Src_{(63...32)}$)
$Dst_{(95...64)}$ = sqrt($Src_{(95...64)}$)
$Dst_{(127...96)}$ = sqrt($Src_{(127...96)}$)

1xDP-FP Scalar Square Root

SSE2 sqrtsd *xmmDst*, *xmmSrc(xmm/m64)* Double-Precision 128

This SIMD instruction is a 128-bit scalar operation that calculates the square root of only the lowest double-precision floating-point block containing the scalar *xmmSrc* and stores the result in the lowest double-precision floating-point block at destination *xmmDst*; the remaining bit blocks are left intact.

$Dst_{(63...0)}$ = sqrt($Src_{(63...0)}$)
$Dst_{(127...64)}$ = remains unchanged

2xDP-FP Square Root

SSE2 sqrtpd *xmmDst*, *xmmSrc(xmm/m128)* Double-Precision 128

This SIMD instruction is a 128-bit parallel operation that calculates the square root of the two double-precision floating-point blocks contained within *xmmSrc* and stores the result in the double-precision floating-point blocks at destination *xmmDst.*

$Dst_{(63\ldots0)} = sqrt(Src_{(63\ldots0)})$
$Dst_{(127\ldots64)} = sqrt(Src_{(127\ldots64)})$

1xSP-FP Scalar Reciprocal Square Root (15 Bit)

3DNow	pfrsqrt *mmDst*, *mmSrc(m32)*	Single-Precision 64
SSE	rsqrtss *xmmDst*, *xmmSrc(m128)*	Single-Precision 32/128
	rsqrtps *xmmDst*, *xmmSrc(m128)*	Single-Precision 128
MIPS IV	rsqrt.s *Dst. aSrc*	Single-Precision 32
MIPS-3D	rsqrt1.s *Dst*, *aSrc*	Single-Precision 32
	rsqrt2.s *Dst, aSrc*	

This SIMD instruction is a 32-bit scalar operation that calculates the square root of only the lowest single-precision floating-point block containing the scalar *mmSrc* and stores the duplicate result in the low and high single-precision floating-point blocks at destination *mmDst.*

$Dst_{(63\ldots32)} = Dst_{(31\ldots0)} = sqrt(Src_{(31\ldots0)})$ $\{\sqrt{A} \quad \sqrt{A}\}$

Pseudo Vec

(Float) Square Root

Listing 10-7: \chap10\fsf\Fsf.cpp

```
void vmp_FSqrt( float * const pfD, float fA )
{
  ASSERT_PTR4(pfD);
  ASSERT_NEG(fA);          // Watch for negative

  *pfD = sqrtf( fA );      // = √A
}
```

Pseudo Vec (X86)

vmp_FSqrt (3DNow!) Fast Float 15-bit Precision

A square root is time consuming and should be omitted whenever possible. If it is indeed needed, then the next logical method would be that of a choice between an imprecise and quick or a more accurate but slower calculation. The following code is for a simple 15-bit accuracy scalar square root D=√A supported by the 3DNow! instruction set.

```
movd  mm0,fA          ; {0    fA}
mov   edx,pfD         ; float Destination
```

Listing 10-8: vmp_x86\chap10\fsf\FsfX86M.asm

```
pfrsqrt mm1,mm0       ; {1/√fA   1/√fA}
pfmul   mm0,mm1       ; {0/√fA  fA/√fA}
                      ; {    0     √fA}
movd    [edx],mm0     ;             √fA
```

SP-FP Square Root (2-stage) (24 Bit)

A fast version of the previous instruction would entail taking advantage of the two-stage vector instructions *pfrsqit1* and *pfrcpit2* in conjunction with the result of the square root instruction *pfrsqrt* to achieve a higher 24-bit precision. It uses a variation of the Newton-Raphson reciprocal square root approximation.

- First stage for 24-bit reciprocal:

3DNow pfrsqit1 *mmDst*, *scalar(mm/m32)* Single-Precision 64

- Second stage for 24-bit reciprocal and/or square root (see reciprocal):

3DNow pfrcpit2 *mmDst*, *scalar(mm/m32)* Single-Precision 64

vmp_FSqrt (3DNow!) Standard Float 24-bit Precision

The following is the same as the previous scalar square root algorithm but is coded for 24-bit precision. Note the addition of the *pfrsqit1* and *pfrcpit2* instructions.

```
mov     edx,pfD    ; (float) Destination
```

Listing 10-9: vmp_x86\chap10\fsf\FsfX86M.asm

```
; First calculate 15-bit accuracy
; Calculate square root upon dot product

pfrsqrt  mm1,mm0                ; {1/√A) 1/√A}

; Second calculate 1/sqrt() accurate to 24 bits

movq     mm2,mm1                ; {1/√A 1/√A}
pfmul    mm1,mm1                ; {1/A  1/A}
pfrsqit1 mm1,mm0                ; 1st step

; Calculate sqrt() = 1/(1/sqrt())  24 bit

pfrcpit2 mm1,mm2                ; 2nd step
pfmul    mm0,mm1          ; {√A √A} {A/√A A/√A}
                          ; {  A         √A}
movd     [edx],mm0        ;              √A
```

vmp_FSqrt (SSE) Float Sqrt 24-bit Precision

For SSE, it is merely a scalar square root instruction.

Listing 10-10: vmp_x86\chap10\fsf\FsfX86M.asm

```
movss    xmm0,fA    ;        {# # # A}
sqrtss   xmm0,xmm0  ; SqRoot {# # # √A}
movss    [edx],xmm0 ; Save square root
```

4xSP-FP Reciprocal Square Root (Estimate)

AltiVec vrsqrtefp *Dst*, *aSrc* Single-Precision 128

vD = vec_rsqrte(vA)

This SIMD instruction calculates the four reciprocal square roots in parallel using estimation logic; thus it has a precision of only 1/4096. It is similar to the method used by 3DNow!, but it is done on this processor as a single stage.

```
vD = vec_rsqrte(vA);    // = 1/√x
```

To get into a normal square root form...

```
vD = vec_madd( vA, vD, vZero ); // √x = x (1/√x)
```

...one merely needs to multiply the reciprocal square root with the original value.

At this point, do you remember what happens if the passed parameter of *x* is equal to zero? As taken for a square root, the result is merely

zero, but then used as the denominator in a reciprocal, we have a little problem of infinity.

$$f(x) = \frac{1}{\sqrt{0}} = \frac{1}{0} = \infty$$

So this infinity has to be dealt with! For this, pretest for closeness to zero and then use a little bit of blending logic:

```
vector float vTiny = (vector float) (0.0000001, 0.0000001,
                                     0.0000001, 0.0000001);
vector float vOne = (vector float) (1.0, 1.0, 1.0, 1.0);

  // Set 1's in position of floats too close to zero
  viM = vec_cmpge(vA, vTiny);

  vD = vec_rsqrte(vA);            // = 1/ √x

  // Correct for infinity due to 1/0 =  1/ √0

  vD = vec_and(vD, viM);          // Preserve #s that are not too small
            // Flip the bit masks (XOR)
  viM = vec_nor(viM, *((vector bool int *)&vZero) ); // 0.0
            // Preserve 1.0s in place of infinity!)
  viM = vec_and(viM, *((vector bool int *)&vOne) );  // 1.0
  vD = vec_or(vD, viM);           // Blended #s and 1.0s
            // Then convert into a square root!
  vD = vec_madd( vA, vD, vZero );// √x = x (1/ √x)
```

Effectively as x approaches zero and goes infinite, the effect upon the original number is negligible, thus the original value is preserved and used as the solution (just a little first-quarter calculus related to limits!).

This is fine and dandy in portions of code that require square roots but do not need accuracy. When accuracy is required, a bit of extra code is needed. This is achieved by utilization of a Newton-Raphson algorithm and needs to be inserted just after the vec_rsqrte() but before the infinity handler and is similar to the following:

```
vector float vHalf = (vector float) (0.5, 0.5, 0.5, 0.5);

    // Increase precision using Newton-Raphson refinement.
  // vD = estimated x

  vT = vec_madd(vD,vD,vZero);     // x' = √x · √x          ≈ x
  vU = vec_madd(vD,vHalf,vZero);  // t1 = √x · 0.5
  vT = vec_nmsub(vA,vT,vOne);     // x" = x · x'  –  1.0
  vD = vec_madd(vT,vU,vD);        // y  = x" · t1  +  √x
```

Pseudo Vec (MIPS)

Depending upon the particular MIPS instruction set being supported, there are two scalar reciprocals available.

vmp_FSqrt (MIPS IV) Fast Float Sqrt (Estimated)

Using the reciprocal square root from MIPS IV:

Listing 10-11

```
mfc1      $f4, a1       // {x}

rsqrt.s   $f5, $f4      // {1/√x}

mul.s     $f4, $f4, $f5 // {x*1/√x}
swc1      $f4, 0(a0)    // {√x}
```

vmp_FSqrt (MIPS-3D) Fast Float Sqrt (Estimated)

Using the reciprocal square root from MIPS-3D with estimated precision:

Listing 10-12

```
mfc1      $f4, a1       // {x}

rsqrt1.s  $f5, $f4      // {1/√x}

mul.s     $f4, $f4, $f5 // {x*1/√x}
swc1      $f4, 0(a0)    // {√x}
```

vmp_FSqrt (MIPS-3D) Float Sqrt 24-bit Precision

Using the reciprocal square root from MIPS-3D with two-step standard 24-bit precision:

Listing 10-13

```
mfc1      $f4, a1       // {x}

rsqrt1.s  $f5, $f4      // {1/√x}
rsqrt2.s  $f6, $f5, $f4
madd.s    $f5, $f5, $f5, $f6 // 24bit {1/√x}

mul.s     $f4, $f4, $f5 // {x*1/√x}
swc1      $f4, 0(a0)    // {√x}
```

Vector Square Root

Are you nuts? Vector square roots? What are you thinking?

Well actually, unless you have a really top-of-the-line supercomputer, I would recommend staying away from vector square roots. Instead, you will typically only need a single square root. If you really need vector-based square roots, remember that your processor can only do one at a time, and your code will have to wait for it to complete before issuing a request to begin the next one. That could take almost forever! Well, not quite, but it is still not a great idea. Also, do not forget about preventing negative numbers from being processed by a square root. That causes exception faults!

Pseudo Vec

Vector Square Root

Listing 10-14: \chap10\vsf3d\Vsf3D.cpp

```
void vmp_VecSqrt(vmp3DVector * const pvD,
           const vmp3DVector * const pvA)
{
  pvD->x = sqrtf( pvA->x );
  pvD->y = sqrtf( pvA->y );
  pvD->z = sqrtf( pvA->z );
}
```

Quad Vector Square Root

Listing 10-15: \chap10\qvsf3d\QVsf3D.cpp

```
void vmp_QVecSqrt(vmp3DQVector * const pvD,
            const vmp3DQVector * const pvA)
{
  pvD->x = sqrtf( pvA->x );
  pvD->y = sqrtf( pvA->y );
  pvD->z = sqrtf( pvA->z );
  pvD->w = sqrtf( pvA->w );
}
```

Similar to an estimated reciprocal for a division, a square root sometimes is available as an estimate as well. Be warned that the estimate square root is faster but has a lower precision. If the lower precision is viable for your application, then investigate using the estimated square root instead.

Pseudo Vec (X86)

The 3DNow! instruction set supports 64 bit, so two loads must be handled simultaneously as well as two saves, but it is a simple matter of adding the two pairs of floats to each other.

```
mov   eax,vA     ; Vector A
mov   edx,vD     ; Vector Destination
```

vmp_QVecSqrt (3DNow!) Fast Quad Float Sqrt 15-bit Precision

Listing 10-16: vmp_x86\chap10\qvsf3d\QVsf3DX86M.asm

```
movq  mm0,[eax+0]              ;vA.xy {Ay   Ax}
movd  mm6,(vmp3DQVector PTR [eax]).y ;{0    Ay}
movq  mm1,[eax+8]              ;vA.zw {Aw   Az}
movd  mm5,(vmp3DQVector PTR [eax]).w ;{0    Aw}

  ; Calculate square root ( pfrsqrt=15-bit accuracy )

pfrsqrt     mm3,mm0       ; {1/ √(Ax)  1/ √(Ax)}
pfrsqrt     mm7,mm6       ; {1/ √(Ay)  1/ √(Ay)}
pfrsqrt     mm2,mm1       ; {1/ √(Az)  1/ √(Az)}
pfrsqrt     mm4,mm5       ; {1/ √(Aw)  1/ √(Aw)}

  ; Calculate 1/sqrt() accurate to 15bits
punpckldq   mm3,mm7       ; {1/ √(Ay)  1/ √(Ax)}
punpckldq   mm2,mm4       ; {1/ √(Aw)  1/ √(Az)}

  ; {Insertion point for 24-bit precision}

pfmul       mm0,mm3       ; {Ay/ √(Ay) Ax/ √(Ax)}
pfmul       mm1,mm2       ; {Aw/ √(Aw) Az/ √(Az)}

movq        [edx+0],mm0   ; { √(Ay) √(Ax)}
movq        [edx+8],mm1   ; { √(Aw) √(Az)}
```

vmp_QVecSqrt (3DNow!) Quad Float Sqrt 24-bit Precision

In the previous code, there is a comment in bold related to insertion for 24-bit precision. By inserting the following code, the higher accuracy will be achieved. It uses the Newton-Raphson reciprocal square approximation.

Listing 10-17: vmp_x86\chap10\qvsf3d\QVsf3DX86M.asm

```
movq        mm6,mm3       ; {1/ √(Ay) 1/ √(Ax)}
movq        mm5,mm2       ; {1/ √(Aw) 1/ √(Az)}
pfmul       mm3,mm3       ; {1/Ay    1/Ax}
pfmul       mm2,mm2       ; {1/Aw    1/Az}
pfrsqit1    mm3,mm0       ; xy  {1st step}
```

```
pfrsqit1    mm2,mm1       ; zw
  ; Calculate √() = 1/(1/ √())  24 bit
pfrcpit2    mm3,mm6       ; xy  {2nd step}
pfrcpit2    mm2,mm5       ; zw
```

vmp_QVecSqrt (SSE) Float Sqrt 24-bit Precision

For SSE, there is a 24-bit precision quad square root. For unaligned memory, substitute *movups* for *movaps*.

Listing 10-18: vmp_x86\chap10\qvsf3d\QVsf3DX86M.asm

```
movaps      xmm0,[eax]   ; { Aw  Az  Ay  Ax}
sqrtps      xmm0,xmm0    ; {√Aw √Az √Ay √Ax}
movaps      [edx],xmm0   ; Save square roots
```

vmp_QVecSqrtFast (SSE) Float Sqrt Approximate

The following is a fast reciprocal square root.

Listing 10-19: vmp_x86\chap10\qvsf3d\QVsf3DX86M.asm

```
movaps      xmm0,[eax]   ;vA.xyzw { Aw  Az  Ay  Ax}
movaps      xmm1,xmm0
movaps      xmm2,xmm0
movaps      xmm3,xmm0

cmpps       xmm2,vTiny,5 ; >=  1's = #'s okay
cmpps       xmm3,vTiny,1 ; <   1's = too close to zero

  ; vD = (1 / sqrt(vA))
rsqrtps     xmm0,xmm0

  ; Correct for infinity due to 1/0 =  1/sqrt(0)

andps       xmm3,ONES  ; Preserve 1.0's for infinite
andps       xmm0,xmm2  ; Preserve #'s too small
orps        xmm0,xmm3  ; Blended #'s and 1.0's
mulps       xmm0,xmm1  ; {Aw/sqrt(Aw) ... Ax/sqrt(Ax)}
movaps      [edx],xmm0 ; Save square roots
```

Pseudo Vec (PowerPC)

Okay, as mentioned earlier in this book, there are always exceptions to the rule. The AltiVec instruction set is one of those, but there is a cost! The precision is extremely poor, but you get a fast quad vector square root. The following code in bold is the basic core functionality, but keep in mind that if there is ever a chance of a zero being passed, then the entire code block is needed! Please note that this code has been discussed earlier in this chapter.

vmp_QVecSqrt (AltiVec) Fast Quad Float Sqrt (1/4096) Precision

Listing 10-20: vmp_ppc\chap10\qvsf3d\QVsf3DAltivec.cpp

```
vector float vD;
vector bool int vIM;

vD = vec_rsqrte(vA);     //  vD =  (1 / √vA)

// Correct for infinity due to 1/0 =  1/√0

vD = vec_and(vD, viM);          // Preserve #s that are not too small
          // Flip the bit masks (XOR)
viM = vec_nor(viM, *((vector bool int *)&vZero) ); // 0.0
          // Preserve 1.0s in place of infinity!
viM = vec_and(viM, *((vector bool int *)&vOne) );  // 1.0
vD = vec_or(vD, viM);           // Blended #s and 1.0s
          // Then convert into a square root!
vD = vec_madd( vA, vD, vZero );// | x = x (1/|x)

*(vector float *)pvD = vec_madd(          // √x = x * (1/√x)
           (*(vector float *)pvA), vD, vZero );
```

vmp_QVecSqrt (AltiVec) Quad Float Sqrt 24-bit Precision

For higher precision, merely insert the following code after the reciprocal square root but before the infinity correction.

Listing 10-21: vmp_ppc\chap10\qvsf3d\QVsf3DAltivec.cpp

```
vector float vHalf = (vector float) (0.5, 0.5, 0.5, 0.5);

   // Increase precision using Newton-Raphson refinement.

  vT = vec_madd(vD,vD,vZero);     // x' = √x · √x        ≈ x
  vU = vec_madd(vD,vHalf,vZero);  // t1 = |x · 0.5
  vT = vec_nmsub(vA,vT,vOne);     // x" = x · x' – 1.0
  vD = vec_madd(vT,vU,vD);        // y  = x" · t1 + √x
```

Pseudo Vec (MIPS)

MIPS-3D	rsqrt1.ps *Dst*, *aSrc* rsqrt2.ps *Dst, aSrc*	Single-Precision	64

vmp_QVecSqrt (MIPS-3D) Fast Quad Float Sqrt Est. Precision

Listing 10-22: vmp_mips\chap10\qvsf3d\QVsf3DMips.cpp

```
ldc1      $f4, 0(a1)    // {Ay Ax}
ldc1      $f5, 8(a1)    // {Aw Az}

rsqrt1.ps $f6, $f4      // {1/ √(Ay) 1/ (Ax)}
rsqrt1.ps $f7, $f5      // {1/ √(Aw) 1/ √(Az)}

mul.ps    $f4, $f4, $f6 // {Ay*1/ √(Ay) Ax*1/ √(Ax)}
mul.ps    $f5, $f5, $f7 // {Aw*1/ √(Aw) Az*1/ √(Az)}

sdc1      $f4, 0(a0)    // { √(Ay) √(Ax)}
sdc1      $f5, 8(a0)    // { √(Aw) √(Az)}
```

vmp_QVecSqrt (MIPS-3D) Quad Float Sqrt 24-bit Precision

Listing 10-23: vmp_mips\chap10\qvsf3d\QVsf3DMips.cpp

```
ldc1     $f4, 0(a1)       // {Ay Ax}
ldc1     $f5, 8(a1)       // {Aw Az}

rsqrt1.ps $f6, $f4        // {1/ √(Ay) 1/ √(Ax)}
rsqrt1.ps $f7, $f5        // {1/ √(Aw) 1/ √(Az)}
rsqrt2.ps $f8, $f6, $f4
rsqrt2.ps $f9, $f7, $f5
madd.s    $f10, $f6, $f6, $f8 // 24bit {1/ √(Ay) 1/ √(Ax)}
madd.s    $f11, $f7, $f7, $f9 // 24bit {1/ √(Aw) 1/ √(Az)}

mul.ps    $f4, $f4, $f10  // {Ay*1/ √(Ay) Ax*1/ √(Ax)}
mul.ps    $f5, $f5, $f11  // {Aw*1/ √(Aw) Az*1/ √(Az)}

sdc1      $f4, 0(a0)      // {(Ay) √(Ax)}
sdc1      $f5, 8(a0)      // {(Aw) √(Az)}
```

Graphics 101

Vector Magnitude (Alias: 3D Pythagorean Theorem)

Ever hear that the shortest distance between two points is typically a straight line? The square of the hypotenuse of a right triangle is equal to the square of each of its two sides, whether it's in 2D or 3D space. The Pythagorean theorem is essentially the distance between two points, in essence the magnitude of their differences.

Hint: Do not use a square root unless you have to!

The first rule of a square root operation is to not use it unless you really have to, as it is a time-intensive mathematical operation. One method typically used for calculating the length of a line between two points, whether it exists in 2D or 3D space, is to use the Pythagorean theorem.

2D Distance

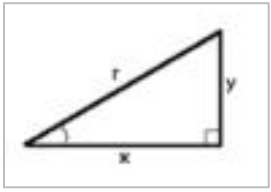


Figure 10-1: 2D right triangle representing a 2D distance

Equation 10-3: 2D distance

$$x^2+y^2=r^2$$

$$r = \sqrt{(x^2 + y^2)}$$

```
Code:  r = sqrt( x*x + y*y );
```

3D Distance

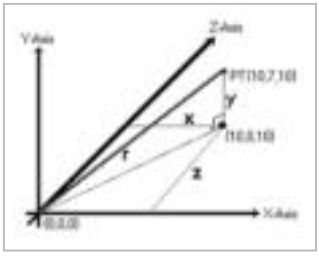


Figure 10-2: Right triangle within a 3D Cartesian coordinate system representing a 3D distance and thus its magnitude

Equation 10-4: 3D distance (magnitude)

$$x^2+y^2+z^2=r^2$$

$$r = \sqrt{(A_x^2 + A_y^2 + A_z^2)}$$

```
Code:  r = sqrt( x*x + y*y + z*z );
```

Mathematical formula:

Pythagorean	$x^2+y^2=r^2$ $r=\sqrt{(x^2+y^2)}$	$x^2+y^2+z^2=r^2$ $r=\sqrt{(x^2+y^2+z^2)}$

2D Distance	$d(P1,P2)= \sqrt{(}\ (x2–x1)^2 +(y2–y1)^2\ )$
3D Distance	$d(P1,P2,P3)= \sqrt{(}\ (x2–x1)^2 +(y2–y1)^2 +(z2–z1)^2)$

$1/x = 1/0 = \infty$

So if the dot product $dp = x^2 + y^2 + z^2$ approaches zero, the value of 1/x gets closer to infinity. Once x becomes zero, the solution becomes undefined ($1/0 = \infty$). When a number is extremely close to infinity and is passed to a square root, the accuracy becomes lost. If we lose precision, we especially lose our sign! (±) So instead, we set 1/0 to a value of one, so y*1=y.

The Pythagorean theorem is essentially the distance between two points. In a terrain following algorithm for creature AI (artificial intelligence), the distance between each of the creatures and the main character would be compared to make an idle, run, or flee determination. The coordinates of each object are known, but their distances would have to be calculated and then compared to each other as part of a solution. If we examine a simplistic equation utilizing *r* to represent the distance between the player and four monsters {mA through mD}:

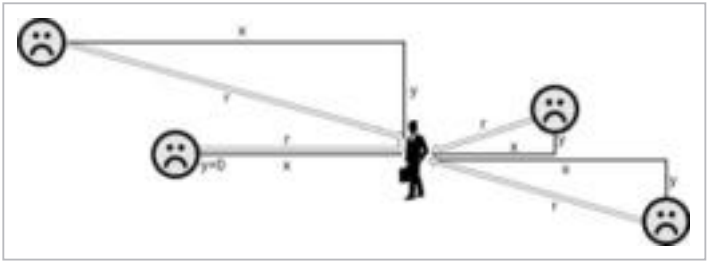


Figure 10-3: Monster-to-player 2D distance calculations

$$\Delta A = MonsterA = \sqrt{((mA_x–pA_x)^2 + (mA_y–pA_y)^2 + (mA_y–pA_y)^2)}$$
$$\Delta B = MonsterB = \sqrt{((mB_x–pA_x)^2 + (mB_y–pA_y)^2 + (mB_y–pA_y)^2)}$$
$$\Delta C = MonsterC = \sqrt{(mC_x–pA_x)^2 + (mC_y–pA_y)^2 + (mC_y–pA_y)^2)}$$
$$\Delta D = MonsterD = \sqrt{((mD_x–pA_x)^2 + (mD_y–pA_y)^2 + (mD_y–pA_y)^2)}$$

If you remember the algebraic law of multiplicative identity, the square root factors out of the equation, as it can be removed from both sides of the equal sign, and the equation will remain in balance.

$$sqrt(\ \Delta A_x^2 + \Delta A_y^2 + \Delta A_z^2\ ) \quad =? \quad sqrt(\ \Delta B_x^2 + \Delta B_y^2 + \Delta B_z^2\ )$$
$$\cancel{sqrt}(\ \Delta A_x^2 + \Delta A_y^2 + \Delta A_z^2\ ) \quad =? \quad \cancel{sqrt}(\ \Delta B_x^2 + \Delta B_y^2 + \Delta B_z^2\ )$$
$$(\ \Delta A_x^2 + \Delta A_y^2 + \Delta A_z^2\ ) \quad =? \quad (\ \Delta B_x^2 + \Delta B_y^2 + \Delta B_z^2\ )$$

Does this look a little similar to the sum of absolute differences operation discussed earlier in this chapter? They are different by the sum of absolutes versus the sum of the squares, but they nevertheless have a similarity. The point is that there is no need to use the square root operation each time in this kind of problem (neat, huh?). It is an old trick and still effective, but here it is for the cases where it is definitely needed.

Now, supposing that it has been discovered that Monster C is the closest monster; take the square root to calculate the distance, and don't forget to use the estimate square root version if accuracy is unnecessary.

Pseudo Vec

Listing 10-24: \chap10\vsf3d\Vsf3D.cpp

```
void vmp_VecMagnitude(float * const pfD,
          const vmp3DVector * const pvA)
{
  *pfD=sqrtf(pvA->x * pvA->x
           + pvA->y * pvA->y
           + pvA->z * pvA->z);
}
```

Pseudo Vec (X86)

The 3DNow! instruction set supports 64 bit, so two loads and/or stores must be handled simultaneously, but the result is a simple matter of adding the two pairs of floats to each other.

```
mov  eax,vA                          ; Vector A
mov  edx,vD                ; Vector Destination
```

vmp_VecMagnitude (3DNow!)

Listing 10-25: vmp_x86\chap10\vsf3d\Vsf3DX86M.asm

```
movq  mm1,[eax]                      ; {Ay   Ax}
movd  mm0,(vmp3DVector PTR [eax]).z ; {0    Az}
pfmul mm1,mm1                       ; {AyAy AxAx}
pfmul mm0,mm0                       ; {0    AzAz}
pfacc mm1,mm1           ; {AyAy+AxAx AyAy+AxAx}
pfadd mm0,mm1    ; {0+AyAy+AxAx AzAz+AyAy+AxAx}

; Ignoring the upper 32 bits    r= (Az)²+(Ay)²+(Ax)²
; Calculate square root

pfrsqrt  mm1,mm0                    ; {# 1/ √r}
; Calculate 1/sqrt() accurate to 24 bits
movq     mm2,mm1                    ; {#  1/ √r}
pfmul    mm1,mm1                    ; {#  1/r}
```

```
pfrsqit1 mm1,mm0                    ; {1st step}
pfrcpit2 mm1,mm2                    ; {2nd step}
pfmul    mm0,mm1
movd     [edx],mm0              ; Save distance
```

vmp_VecMagnitude (SSE) Aligned

Replace *movaps* with *movups* for unaligned memory.

Listing 10-26: vmp_x86\chap10\vsf3d\Vsf3DX86M.asm

```
movaps   xmm0,[eax]                ; {# Az Ay Ax}
mulps    xmm0,xmm0       ; {A#A# AzAz AyAy AxAx}
movaps   xmm1,xmm0
movaps   xmm2,xmm0
unpckhps xmm0,xmm0       ; {A#A# A#A# AzAz AzAz}
shufps   xmm1,xmm1,11100001b ; {A## Az² Ax² Ay²}

addss    xmm2,xmm0           ; {# # # Az²+Ax²}
addss    xmm2,xmm1           ; {# # # Az²+Ax²+Ay²}

; Calculate square root

sqrtss   xmm0,xmm2           ; √(Az²+Ax²+Ay²)
movss    [edx],xmm0          ; Save Scalar
```

Pseudo Vec (PowerPC)

Since the vector and misalignment of code has been discussed earlier, the following is the quad vector implementation of the magnitude. Keep in mind that the magnitude only requires the {XYZ} and not the {W} element of the vector.

vmp_QVecMagnitude (AltiVec) Aligned

Listing 10-27: vmp_ppc\chap10\vsf3d\Vsf3DAltivec.cpp

```
vector float vT;
float f;

vT = vec_madd( (*(vector float *)pvA),  // T=A²+0
               (*(vector float *)pvA), vZero );

f  = ((vmp3DVector *) &vT)->x           // f=Tx+Ty+Tz
   + ((vmp3DVector *) &vT)->y
   + ((vmp3DVector *) &vT)->z;
*pfD = sqrt(f);                         // √f
```

Graphics 101

Vector Normalize

Pseudo Vec

Listing 10-28: \chap10\vsf3d\Vsf3D.cpp

```
void vmp_VecNormalize(vmp3DVector * const pvD,
                const vmp3DVector * const pvA)
{
  float fMag;

  // Magnitude = √(x²+y²+z²)
  fMag = sqrtf(pvA->x*pvA->x + pvA->y*pvA->y
                             + pvA->z*pvA->z);
  if ( fMag < 0.0000001f )
    {                     // too close to zero
      pvD->x = pvA->x;
      pvD->y = pvA->y;
      pvD->z = pvA->z;
    }
  else
    {    // Ick, a division, to obtain a reciprocal!
      fMag = 1.0f / fMag;
      pvD->x = pvA->x * fMag;
      pvD->y = pvA->y * fMag;
      pvD->z = pvA->z * fMag;
    }
}
```

Pseudo Vec (X86)

The 3DNow! processor supports 64 bit, so two loads or two stores must be handled simultaneously, but it is a simple matter of adding the two pairs of floats to each other.

```
mov  eax,vA                          ; Vector A
mov  edx,vD                ; Vector Destination
```

vmp_VecNormalize (3DNow!)

Listing 10-29: vmp_x86\chap10\vsf3d\Vsf3DX86M.asm

```
movq  mm1,[eax]                       ; {Ay Ax}
movd  mm0,(vmp3DVector PTR [eax]).z   ; {0  Az}
movq  mm4,mm1                         ; {Ay Ax}
movq  mm3,mm0                         ; {0  Az}
pfmul mm1,mm1                      ; {AyAy AxAx}
pfmul mm0,mm0                      ; {0    AzAz}
pfacc mm1,mm1           ; {AyAy+AxAx AyAy+AxAx}
pfadd mm0,mm1    ; {0+AyAy+AxAx AzAz+AyAy+AxAx}

  ; Calculate square root (pfrsqrt=15-bit accuracy)
```

```
    ; too close zero ...???  1.0 / 10000.0 ???

  movd  ecx,mm0
  cmp   ecx,FLOAT0001       ; 0.0001
  jl    short zmag          ;just set vD=vA!!!

    ; Not too close to zero, f= AzAz+AyAy+AxAx
    ; for Newton-Raphson 24-bit resolution
  pfrsqrt mm1,mm0         ; {1/ √r      1/ √r}
  movq  mm2,mm1           ; {1/ √r      1/ √r}
  pfmul mm1,mm1           ; {1/r      1/r}
  pfrsqit1 mm1,mm0         ;X2=f(x,x1) {1st step}

    ; *** mm1 = Magnitude ***
    ; Calculate sqrt() = (1/mag)  24bit
  pfrcpit2  mm1,mm2           ; {2nd step}  {# m}
  punpckldq mm1,mm1                   ; {1/m 1/m}
  pfmul  mm4,mm1            ;{Ny Nx}= {Ay/m Ax/m}
  pfmul  mm3,mm1            ;{ 0 Nz}= { 0/m Az/m}

zmag:      ; Save Resulting {x y z} Normals
  movq   [edx+0],mm4                ;    {Ny Nx}
  movd   (vmp3DVector PTR [edx]).z,mm3  ; {0  Nz}
```

vmp_VecNormalize (SSE) Aligned

If the data is unaligned, change the *movaps* instruction to *movups*.

Listing 10-30: vmp_x86\chap10\vsf3d\Vsf3DX86M.asm

```
  movaps xmm0,[eax]                ; {# Az Ay Ax}
  movaps xmm7,[edx]                ; {Dw #  #  #}
  andps  xmm0,lomsk96              ; {0 Az Ay Ax}
  andps  xmm7,himsk32              ; {Dw 0  0  0}
  movaps xmm6,xmm0            ; {0   Az   Ay   Ax}
  mulps  xmm0,xmm0            ; {0 AzAz AyAy AxAx}
  movaps xmm1,xmm0            ; {0 AzAz AyAy AxAx}
  movaps xmm2,xmm0            ; {0 AzAz AyAy AxAx}

  orps   xmm1,ONEHIGH                ; {1 Az² Ay² Ax²}
  shufps xmm1,xmm1,11001001b  ; 3021   {1 Ax² Az² Ay²}
  shufps xmm2,xmm2,11010010b  ; 3102   {0 Ay² Ax² Az²}

  addps  xmm1,xmm0  ; {1+0 Az²+Ax² Ay²+Az² Ax²+Ay²}
  addps  xmm1,xmm2
         ; {1+0 Ay²+Az²+Ax² Ax²+Ay²+Az² Az²+Ax²+Ay²}

    ; Too close zero?

  movss  uflow,xmm1              ;r= Ay²+Az²+Ax²
  cmp    uflow,FLOAT0001         ; 0.0001f
  jl     short zmag              ; set vD=vA!!!

    ; Calculate square root

  sqrtps xmm0,xmm1                ; {1 √r √r √r}
```

```
  divps  xmm6,xmm0                ; {0 Nz Ny Nz}

zmag:
  orps   xmm7,xmm6                ; {Dw Nz Ny Nx}
  movaps [edx],xmm7               ;  Save
```

Pseudo Vec (PowerPC)

vmp_QVecNormalize (AltiVec) Aligned 24-bit Precision

For purposes of understanding, this function is similar to the generic C version of the code, except the square root result of the magnitude becomes the reciprocal of one, is copied into all quad element positions, and then a vector multiplication takes place.

Listing 10-31: vmp_ppc\chap10\qvsf3d\QVsf3DAltivec.cpp

```
  vector float vM;
  float fMag;

  vM = vec_madd( vA, vA, vZero );

  // No float MSUM function!

  fMag = ((vmp3DVector *) &vM)->x
       + ((vmp3DVector *) &vM)->y
       + ((vmp3DVector *) &vM)->z;

  if ( fMag < 0.0000001f )     // If too close to zero
  {
    vD = vA;
  }
  else
  {
    fMag = 1.0f / fMag;
    ((vmp3DQVector *) &vM)->w = fMag;
    ((vmp3DQVector *) &vM)->x = fMag;
    ((vmp3DQVector *) &vM)->y = fMag;
    ((vmp3DQVector *) &vM)->z = fMag;

    vD = vec_madd( vA, vM, vZero );
  }
((vmp3DQVector *)&vD)->w = 0.0f;
```

vmp_QVecNormalize (AltiVec) Aligned Fast (1/4096) Precision

The normalization can be processed faster using the estimated square root instruction. This is very similar to the 24-bit precision version, except the value of the magnitude is splat into all quad element positions, and then the square root equation is done in parallel to get the

15-bit estimation. In this version of the code, a different approach is taken. The magnitude is initially processed as a scalar in the {X} element position of a quad vector. This allows it to be *splat* into all the element positions.

Listing 10-32: vmp_ppc\chap10\qvsf3d\QVsf3DAltivec.cpp

```
  vM = vec_madd( vA, vA, vZero );   // [x*x y*y z*z w*w]

  // No float MSUM function!                    [# # # Mag]

  ((vmp3DVector *)&vM)->x += ((vmp3DVector *) &vM)->y
                           + ((vmp3DVector *) &vM)->z;

  if ( ((vmp3Dvector *)&vM)->x < 0.0000001f )
  {
    vD = vA;                        // Copy original vector
  }
  else
  {
    vM = vec_splat(vM, 0);          // [Mag Mag Mag Mag]
    vM = vec_rsqrte(vM);            // vD[] = 1 / √Mag
    vD = vec_madd( vA, vM, vZero ); // |x = x · 1/√x
  }

((vmp3DQVector *)&vD)->w = 0.0f;    // replace the fourth field
```

Exercises

1. Create a minimum function using compares.
2. Create a maximum function using compares.
3. Create a bounding min/max using compares.
4. If you did not have branching instructions available, how could you write it as a normalized function without masking? Write it.
5. Same as exercise #4 but no masking is available either. Write it. Hint: Algebraic laws of identities.
6. Rewrite the equation $1/(x\sqrt{x})$ into exponential form.
7. What is the magnitude between p{1,7,8} and q{9,4,2}?
8. What is the magnitude between {0,0,0} and each of these points?
   a. {9,15,3}
   b. {–8,–14,6}
   c. {19,–19,–3}
   d. {–2,11,16}
   e. {25,0,–5}

f. {–14,7,6}
g. {16,14,–2}
h. {22,6,–8}
i. {–2,16,19}
j. {–12,11,–2}
k. {–1,–15,–21}
l. {–7,7,18}

The above contains the coordinates of the bad guys. The one good guy player is at coordinate {8,–25,5}. Who is the closest bad guy? Who is the furthest? What are their magnitudes?

9. How would you upgrade the estimated precision version of the code to full 24-bit precision?

Chapter 11
A Wee Bit O'Trig

Trigonometric functions are defined by the measurement of triangles and their component parts of sides and angles. I will not bore you with the part of how the sum of all the interior angles of a triangle add up to 180º and a circle adds up to 360°.

CD Workbench Files: /Bench/*architecture*/chap11/*project*/*platform*

architecture		*project*		*platform*
PowerPC	/vmp_ppc/	**Trig**	**/vtrig3d/**	/mac9cw
X86	/vmp_x86/			/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

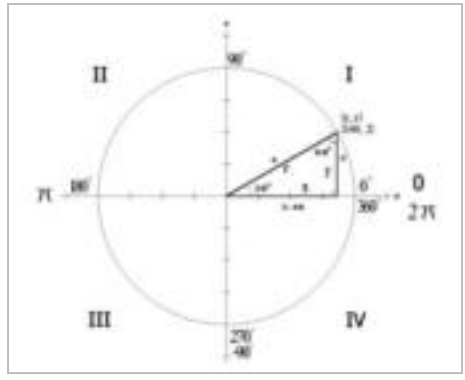


Figure 11-1: 2D geometric circle

This chapter should hopefully be more of a review for you. It is pretty much about standard math library functions with a couple (hopefully new) twists. It is specifically related to the ratios of triangles referred to as trigonometry. Again, if you need more information related to this and other subjects related to math, I recommend *3D Math Primer for Graphics and Game Development* from Wordware Publishing or a visit to your local university bookstore. You can also visit one of the multitude of web sites.

All of the processors discussed in this book only support scalar trigonometric functions using the basic floating-point coprocessor. There is one exception, and that is the vector unit coprocessor in the PS2.

3D Cartesian Coordinate System

The 3D Cartesian coordinate system is similar to what you learned in geometry class for describing points and lines (vectors) in three-dimensional space. Each of the three axes {XYZ} are perpendicular to each other. Three planes are constructed with various combinations of the axes: X-Y, Y-Z, X-Z. Three coordinates, as in the following example {10, 7, 10}, specify a point within 3D space.

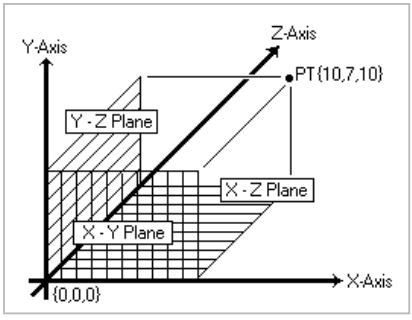


Figure 11-2: 3-dimensional Cartesian coordinate system

A line would be specified by the difference between two points. In the above figure, the two points are {10,7,10} and {0,0,0}, hence

```
{10 – 0, 7 – 0, 10 – 0}.      ΔX=10      ΔY=7      ΔZ=10
```

3D Polar Coordinate System

A 3D polar coordinate system determines a vector by the distance *r* followed by the azimuth angle θ and the elevation angle ψ {*r,* θ*,* ψ}. It is not used in this book but only shown here for reference. The position is defined as the distance *r* from the origin and two angles: an azimuth angle θ from the Z-axis and an elevation angle ψ from the X-axis.

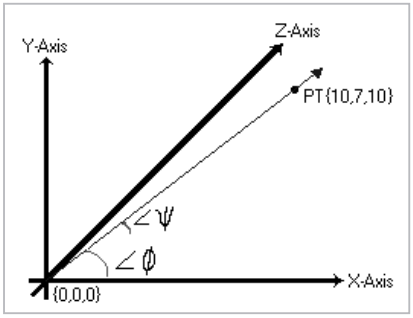


Figure 11-3: 3D polar coordinate system

Analytic Geometry

This chapter is all about angles, edges, and their ratios in conjunction with vectors.

Similar Triangles

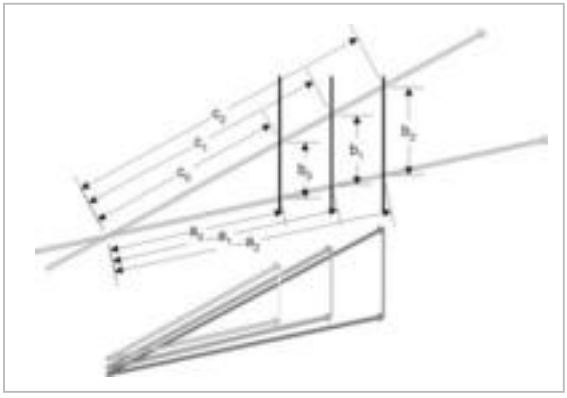


Figure 11-4: If two intersecting lines defining an angle are both intersected by two or more parallel lines, a series of similar overlapping triangles are formed.

Similar Triangle Theorem	Given similar triangles, there is a constant *k* such that: a' = *k*a b'=*k*b c' = *k*c

In essence, corresponding sides are proportional. Similar triangles are ratios of each other, as the ratio of the differences of each edge is the same ratio of all the edges! For example:

Equation 11-1: Ratios of similar triangles

$$\frac{a_0}{b_0} = \frac{a_1}{b_1} = \frac{a_2}{b_2} \qquad \frac{a_0}{c_0} = \frac{a_1}{c_1} = \frac{a_2}{c_2} \qquad \frac{b_0}{c_0} = \frac{b_1}{c_1} = \frac{b_2}{c_2}$$

Using simple ratios: $a_2 = \frac{a_1 c_2}{c_1}$

Unknown edges can be easily calculated with simple algebraic transformations.

Equation of a Straight Line

The equation for the y intercept is one of the following, depending upon which country you come from:

```
y = mx + b
```

```
y = mx + c
```

Which country, you ask? Some countries teach the slope formula with (+c), not (+b) as taught in the United States.

Equation 11-2: Equation of a straight line

$$y = m\,(x - x_1) + b \qquad y - y_1 = m\,(x - x_1) \qquad \frac{y - y_1}{x - x_1} = m$$

Equation of a 2D Circle

With {0, 0} the center of a circle and the origin of the world coordinates in 2D space, and {x, y} a point on that circle, *r* is the radius of that circle.

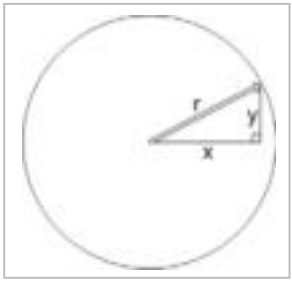


Figure 11-5: Radius of a 2D circle

Equation 11-3: Equation of a 2D circle

$$r^2 = x^2 + y^2$$

If the center of the circle is not at the origin of the world {0, 0} but at coordinates $\{x_0, y_0\}$, then the equation of a circle is:

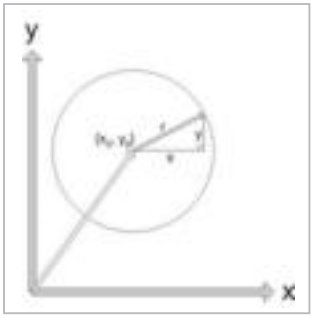


Figure 11-6: Equation of a 2D circle in 2D space

Equation 11-4: 2D circle in 2D space

$$r^2 = (x - x_0)^2 + (y - y_0)^2$$

Sine and Cosine Functions

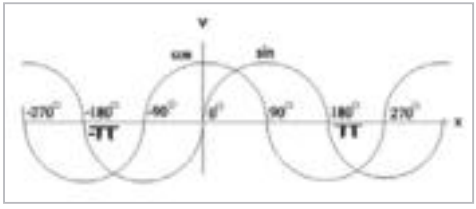


Figure 11-7: Sine-cosine waves

$$\sin\theta = \frac{\text{opposite side}}{\text{hypotenuse}} \qquad \cos\theta = \frac{\text{adjacent side}}{\text{hypotenuse}}$$

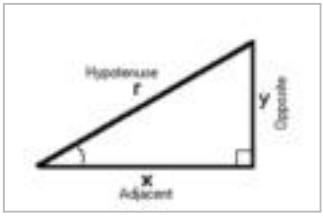


Figure 11-8: Sine and cosine trigonometric relationships

Equation 11-5: Sine and cosine

$$\sin\theta = \frac{y}{r} \qquad y = r\sin\theta \qquad r = \frac{y}{\sin\theta}$$

$$\cos\theta = \frac{x}{r} \qquad x = r\cos\theta \qquad r = \frac{x}{\cos\theta}$$

The standard C math library contains the following functions:

float cos(float x);	double cos(double x);
float sin(float x);	double sin(double x);

You should already be familiar with the fact that the angle in degrees is not passed into those functions but the equivalent value in radians instead. If you recall, π (pi) is equivalent to 180° and 2π to 360°. By using the following macro, an angle in degrees can be converted to radians:

```
#define PI          (3.141592f)
#define DEG2RAD(x)  (((x) * PI) / 180.0F)
```

...and used in the calculations. It can then be converted from radians back to degrees:

```
#define RAD2DEG(x)  (((x) * 180.0f) / PI)
```

...if needed for printing or other purposes!

For a simple 2D rotation, the use is merely one of:

```
x = cos( fRadian );
y = sin( fRadian );
```

There is one thing that has always bothered me about these two functions. When a cosine is needed, a sine is needed as well, and one is in reality 90 degrees out of phase of the other, which means that they share similar math as discussed in the section "Angular Relationships between Trigonometric Functions" later in this chapter.

Equation 11-6: Angular relationship sine to cosine

$$\sin (90° - \theta) = \cos \theta$$

But I find something really interesting. In Intel's wisdom, they not only support the sine and cosine on their FPU (floating-point unit), but they also support the combination sine/cosine, which returns both results. What I find interesting is that very few programmers actually take advantage of it! On top of that, I am upon occasion called in to review other programmers' code, and I quite consistently have to recommend to only call each one once in their matrices. A rotational matrix for an axis will use two identical cosines, a sine, and a negative sine, but why do I keep seeing these functions each called twice? I see code such as the following:

```
Mx[1][2] =  sin(fRadian);
Mx[2][2] =  cos(fRadian);
Mx[1][1] =  cos(fRadian);
Mx[2][1] = -sin(fRadian);
```

...or the just slightly better version of:

```
Mx[1][2] =             sin(fRadian);
Mx[2][2] =  Mx[1][1] = cos(fRadian);
Mx[2][1] =            -sin(fRadian);
```

...and then you wonder why your code runs so slow! Instead, your code should be more like this:

```
Mx[1][2] =               sin(fRadian);
Mx[2][2] =  Mx[1][1] =   cos(fRadian);
Mx[2][1] = -Mx[1][2]; //-sin
```

...or better yet, if you are using an X86 processor, use its combination sine/cosine instruction! I have never seen this one in code reviews the first time I review a programmer's code, but they always cut and paste this recommendation into their next project!

```
sincos(&fSin, &fCos);
Mx[1][2] =              fSin;
Mx[2][2] =  Mx[1][1] =  fCos;
Mx[2][1] = -fSin;
```

The reason will soon become evident!

Pseudo Vec

Similar to the other functions in this book, the sine and cosine functions have been wrapped for extra portability and alternate specialty function replacements.

Listing 11-1: \chap11\ vtrig3d\VTrig3D.cpp

```
void vmp_FSinCos( float * const pfSin,
                  float * const pfCos, float fRads )
{
  *pfSin = sinf(fRads);
  *pfCos = cosf(fRads);
}

void vmp_FSin( float * const pfD, float fRads )
{
  *pfD = sinf( fRads );
}

void vmp_FCos( float * const pfD, float fRads )
{
  *pfD = cosf( fRads );
}
```

Pseudo Vec (X86)

Early Microsoft math libraries did not include this functionality, which is understandable, as the C programming language was born in the land of UNIX and brought forth to the X86 DOS world by other companies and eventually by Microsoft. I believe the absence of a library function for a combination sine/cosine was merely an oversight. It has recently been introduced in third-party libraries and different forms.

AMD-SDK

```
void  _sincos(float, float *);
void  a_sincos(void);              // mm0 -> mm0 (cos|sin)
__m64 _m_sincos(__m64);            // mm0 -> mm0 (cos|sin)
```

3DSMax-SDK

```
inline void SinCos(float angle, float *sine, float *cosine)
```

Geesh! I did a wildcard search on my computer for header files, and this is all that turned up with all the compilers I work with? This is ridiculous! Apparently, most programmers up until now either did not know or did not care! Well, let's fix that right now and, yet again, another reason to buy an extra copy of the book!

vmp_SinCos (X86)

The *fwait* instruction is used to wait for the FPU (floating-point unit) operation to complete. The *fsincos* instruction calculates the sine and cosine simultaneously — slower than calling just one of them, but faster than calling them both consecutively.

Listing 11-2: vmp_x86\chap11\vtrig3d\VTrig3DX86.asm

```
fld   fRads                ; Lad Radians from memory
OFLOW_FIXUP fRads          ; Overflow fixup (optional)

fwait                      ; Wait for FPU to be idle
fsincos                    ; ST(0)=cos ST(1)=sin

mov   eax,pfSin
mov   ecx,pfCos

fwait                      ; Wait for FPU to be idle
                           ; Pop from FPU stack
fstp  (REAL4 PTR [ecx])    ; ST(0)=cos ST(1)=sin
fstp  (REAL4 PTR [eax])    ; sin
```

vmp_SinCos (3DNow!)

AMD has a really nice solution for their 3DNow! functionality. It resolves the sine and cosine simultaneously using the MMX register set. Their algorithm is actually pretty cool, as what they do is ratio π (3.14159) to a value of 4, which in essence makes π radians a base 2 number. Thus, integer math is used in conjunction with parallel floating-point math and MMX registers. Signs are tracked to encode in which quadrant of the circle the angle resides. Since a circle is actually made up of eight quadrants (investigate Bresenham's DDA algorithm for circle plotting), only the modulus 45º angle needs to be resolved.

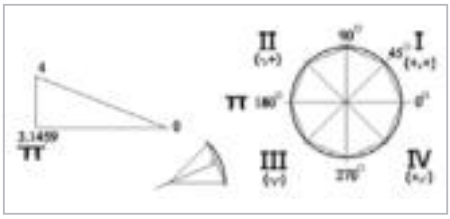


Figure 11-9: The relationship of a circle with its quadrants and associated sizes

I would have loved to include it here, but since this book is "commercial" and AMD's code is licensed only for individual use, I could not legally distribute it here. The code can actually be downloaded from their web site (www.amd.com) and is contained within their Software Development Kit (SDK) – AMD Library Reference.

But all is not lost. Cut and paste it into your own code or, if using Direct3D, it is automatically called if you are using a processor supporting the 3DNow! instruction set. There is also an SSE version. It is a lot more efficient than having to do an FPU/MMX switch, especially as sine and cosine are processed simultaneously in parallel.

Now, I am not telling you to reverse-engineer the DirectX library, as that would be wrong and illegal in the United States due to the Millennium Copyright Act, as well as in violation of your End User License Agreement (EULA), but by setting a breakpoint on the DirectX library function call and upon getting the break, set your view to assembly code and step into the code. You can cut and paste a copy of that code to a text editor, allowing you to edit and analyze how it works. Remember that there is more than one algorithm residing in the library as well.

So now on to other interesting things.

Vector Cosine

A sin and/or cosine function returns a value with a range of {–1...,0,...1}. This book briefly discussed methods of culling polygons in Chapter 9, "Vector Multiplication and Division." You may recall a portion of the following about the use of a dot product:

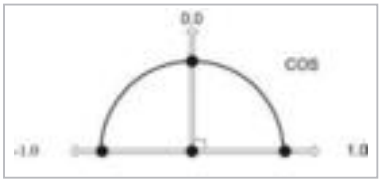


Figure 11-10: Cosine of two intersecting lines

It is the cosine of the angle. So by having two vectors instead of an angle in radians, the cosine can be calculated by dividing the dot product by the product of the magnitudes of the two vectors. Note that v and w are vectors and $|v|$ and $|w|$ are their magnitudes.

Equation 11-7: Cosine angle

$$\text{Cos } \theta = \frac{A_xB_x + A_yB_y + A_zB_z}{\sqrt{(A_x^2 + A_y^2 + A_z^2)} \times \sqrt{(B_x^2 + B_y^2 + B_z^2)}} = \frac{v \cdot w}{|v| \times |w|}$$

Listing 11-3: \chap11\vtrig3d\Bench.cpp

```
void CosAng( float * const pfD, vmp3DQVector *pvA,
                                vmp3DQVector *pvB )
{
  float fDot, fAMag, fBMag;

  ASSERT_PTR16(pvA);
  ASSERT_PTR16(pvB);
  ASSERT_PTR4(pfD);

  vmp_QVecMagnitude(&fAMag, pvA);
  vmp_QVecMagnitude(&fBMag, pvB);
  vmp_QDotProduct(&fDot, pvA, pvB);

    // *pfD = fDot / (fAMag * fBMag);
  vmp_FMul( &fAMag, &fAMag, &fBMag );
  vmp_FDiv( *pfD, &fDot, &fAMag );
}
```

Vertex Lighting

An additional bonus is that the resulting value of the cosine angle can be used for vertex lighting information. Each of the vertices (corners) of a polygon (three for triangles, four for quads) contain position, texture UV, lighting, and other miscellaneous information, depending upon the export tool used to generate the database. The vertex lighting information is precalculated for the corners of each poly and interpolated across each face during renders.

If a graphics library, such as Direct3D, is not used for vertex shading or fogging, the application must handle its own interpolation across the face of each associated polygon.

Depending upon graphic resources, a portion of a texture is typically mapped onto a face but will be flat and tessellated and a pixel stretched to fit. The following image is just such a case. Also, notice the absence of the texture for the trucks (wheels and suspension), and so only a box is in place of those textures.

Figure 11-11: Boxcar model compliments of Ken Mayfield

Vertex lighting information based upon a value whose range is from 0.0 to 1.0 is mapped across that same face, and each pixel is adjusted by that lighting information. This adds more depth (less tessellation) of a scene, and when combined with external lighting information, high quality lighting effects are rendered.

If a polygonal face is perpendicular to a light source, the result is negative, thus full intensity lighting occurs. If the polygon is parallel to that light source (on edge), a zero value would result, thus the absence of light. As explained earlier in this book, if the value is positive, it is back faced and back culled, so no lighting information is needed! As the vertices are normalized, a value ranging from –1.0 through 0.0 would be used for illumination determination.

Tangent and Cotangent Functions

Equation 11-8: Tangent and cotangent

$$\tan\theta = \frac{\text{opposite side}}{\text{adjacent side}} \qquad \cot\theta = \frac{\text{adjacent side}}{\text{opposite side}}$$

$$\tan\theta = \frac{y}{x} \qquad y = x\ \tan\theta \qquad x = \frac{y}{\tan\theta}$$

$$\cot\theta = \frac{x}{y} \qquad x = y\ \cot\theta \qquad y = \frac{x}{\cot\theta}$$

Pseudo Vec

Similar to the other functions in this book, the tangent function has been wrapped for extra portability.

Listing 11-4: \chap11\ vtrig3d\VTrig3D.cpp

```
void vmp_FTan( float * const pfD, float fRads )
{
  *pfD = tanf( fRads );
}
```

Angular Relationships between Trigonometric Functions

Equation 11-9: Angular relationships between trigonometric functions

$$\tan\theta = \frac{\sin\theta}{\cos\theta} \qquad (\cos\theta \neq 0)$$

$$\tan\theta = \frac{1}{\cot\theta} \qquad (\cot\theta \neq 0)$$

$$\cot\theta = \frac{\cos\theta}{\sin\theta} \qquad (\sin\theta \neq 0)$$

Using standard trigonometric formulas, such as:

$$1 = \cos^2 + \sin^2$$

...sine and other trigonometric results can be calculated.

Pythagorean Identity	$\sin^2\theta + \cos^2\theta = 1$

Equation 11-10: Pythagorean identity

$$\sin(2/\pi - \theta) = \sin(90^\circ - \theta) = \cos\theta$$

$$\sin\theta = \pm\sqrt{(1-\cos^2\theta)} \qquad \cos\theta = \pm\sqrt{(1-\sin^2\theta)}$$

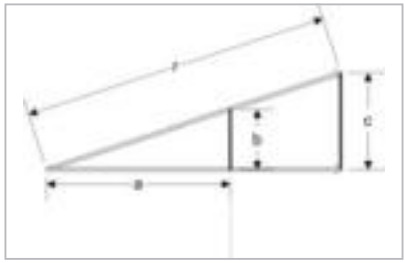
Arc-Sine and Cosine

With a sine or cosine, the angle can be calculated by using the inverse sine ($\sin^{-1}$) or cosine ($\cos^{-1}$). On occasion, it becomes necessary to convert a sine and/or cosine back into an angle in radians or degrees. The following code shows how to do this using the cosine/sine functions as well as a two-argument tangent function.

Pseudo Vec

```
#define PI          (3.141592f)
#define DEG2RAD(x)  (((x) * PI) / 180.0)
#define RAD2DEG(x)  (((x) * 180.0) / PI)

  fRadians = DEG2RAD( fDegrees );

  fCos = cos( fRadians );     // x=
  fSin = sin( fRadians );     // y=
...oops, I meant the function...

  vmp_FsinCos( &fSin, &fCos, fRadians);

  // An arccosine but which hemisphere?

  fAngCS = RAD2DEG( acos( fCos ) );       // {0...180}

  // An arcsine resolves the hemisphere!

  if (asin( fSin ) < 0.0)
  {
    fAngCS = -fAngCS;                     // {-180...180}
  }

     // This is where arctangent-atan2 is really handy, as it accepts the two
     // arguments of the sine and cosine and covers the entire 360 degrees!

  fAngT = RAD2DEG( atan2( fSin, fCos ) ); // {-180...180}
```

Exercises

1. 

If $a = 9$, $b = 5$ and $c=7$, solve for r.

Chapter 12
Matrix Math

For rendering images, typically two primary orientations of a coordinate system are utilized: the left-handed 3D Cartesian coordinate system on the left or the right-handed on the right, as shown in the following figure.

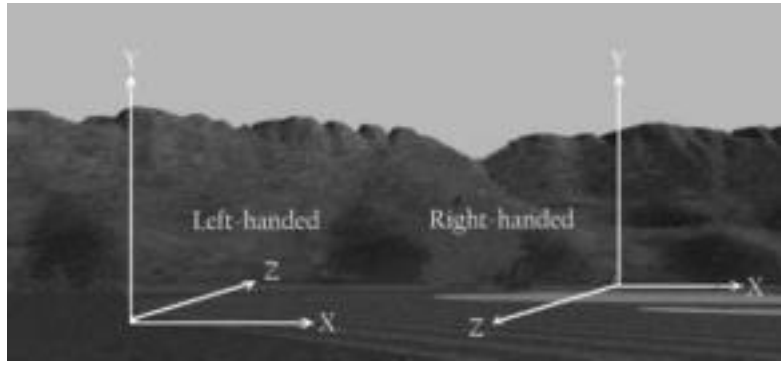


Figure 12-1: Left-handed and right-handed coordinate systems

This is a representation of Euler (pronounced "oiler") angles. The following has been done with all sorts of hand manipulations, etc., but I find the following the easiest non-thinking way to remember. In a left-handed system, place your left arm along the x-axis and the hand at the intersection of this axis; the horizontal hitchhiking thumb easily points in the z+ direction. For a right-handed system, do the same with the right hand, and again the thumb easily points in the direction of the z-axis (unless you are double jointed!)

The left-handed system has the z-axis increase in a positive direction toward the horizon and is what video game system graphic packages, such as Direct3D, typically use. The right-handed system increases the value of the z-axis as it approaches the viewer. This is more along the lines of a high-performance graphics library and standard mathematical conventions. There are variations of these where the

z-axis is the height (elevation) information but this is typically used within some art rendering programs, such as 3D Studio Max.

For purposes of rotation, scaling, and translations of each object within a scene, one of the following two methods is utilized. One of them is the use of the vertex, which has been discussed in some detail in previous chapters, and the other is the use of a quaternion, which will be discussed in Chapter 13. Both of these mechanisms use their own implementation of vectors and matrices for the actual multiple-axis transformations of images and their {XYZ} coordinate information.

CD Workbench Files: /Bench/*architecture*/chap12/*project*/*platform*

	architecture	Matrix	*project*	*platform*
PowerPC	/vmp_ppc/	**Matrix**	**/matrix3d/**	/mac9cw
X86	/vmp_x86/	**D3D Race**	**/matrixRace/**	/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Vectors

A typical 3D coordinate is contained within an {XYZ} 1x3 column vector, but when extended to a fourth element for {XYZW} 1x4 column vector, a one is typically set for the {W} element. As a note, you should remember that a product identity vector contains an {XYZW} of {0,0,0,1}. When working with translations, the fourth row contains translation (displacement) information and the {1} in the {W} element allows it to be processed as part of the solution.

A matrix is used to encapsulate simple to complicated mathematical expressions, which can be applied to a scalar, vector, or another matrix. The product, inverse calculations, and summation expressions can all be combined into a single matrix to help minimize the number of overall calculations needed for resolution. These simple mathematical operations resolve to rotations, scaling, and translations of vectors, just to name a few.

Vector to Vector Summation (v+w)

The summation of two same-sized vertices is simply the scalar of each element of both vertices that are each summed and stored in the same element location of the destination vector.

Equation 12-1: Vector to vector summation ($v+w$)

$$v = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} \quad w = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

$$v + w = \begin{bmatrix} v_1+w_1 & v_2+w_2 & v_3+w_3 & v_4+w_4 \end{bmatrix}$$

The Matrix

A matrix is an array of scalars that are used in mathematical operations. In the case of this book, only two types of matrices are utilized: a 4x4 matrix denoted by ***A***, as illustrated below on the left, and an 1x4 matrix *v* on the right, which is used to represent a vector.

Equation 12-2: 4x4 matrix and 1x4 vector

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad v = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix}$$

Matrices are typically a black box to programmers and engineers, as they typically cut and paste standard matrix algorithms into their code without really understanding how they work. There may be a similarity between the words "matrix" and "magic," but there are no chants and spells here. It is just mathematics!

Equation 12-3: Matrices are arranged into a row column arrangement ($M_{row\ col}$ $A_{row\ col}$). For example, scalar a_{23} is referenced by the second row, third column, as shown here.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Listing 12-1: \inc???\vmp3D.h

```
typedef float  vmp3DMatrix[4][4];
```

A matrix can also be thought of as four quad vectors arranged linearly in memory, such as the following:

```
vmp3DQVector vM[4];
```

vM[0].x=M_{11}	vM[0].y=M_{12}	vM[0].z=M_{13}	vM[0].w=M_{14}
vM[1].x=M_{21}	vM[1].y=M_{22}	vM[1].z=M_{23}	vM[1].w=M_{24}
vM[2].x=M_{31}	vM[2].y=M_{32}	vM[2].z=M_{33}	vM[2].w=M_{34}
vM[3].x=M_{41}	vM[3].y=M_{42}	vM[3].z=M_{43}	vM[3].w=M_{44}

A matrix is arranged linearly in memory, similar to the following array. Note that each element occupies the space of either a float or double depending upon the data type.

```
float M[16];
double M[16];
```

M[0]=M_{11}	M[1]=M_{12}	M[2]=M_{13}	M[3]=M_{14}
M[4]=M_{21}	M[5]=M_{22}	M[6]=M_{23}	M[7]=M_{24}
M[8]=M_{31}	M[9]=M_{32}	M[10]=M_{33}	M[11]=M_{34}
M[12]=M_{41}	M[13]=M_{42}	M[14]=M_{43}	M[15]=M_{44}

Matrix Copy (D=A)

To keep the result of a matrix calculation from contaminating a source matrix if the destination and source are one and the same, a temporary matrix is used to retain the results until the matrix processing has completed; then those results are copied to the destination. A function such as the following or a memcpy (although a memcpy would be inefficient in this particular implementation) is used to copy that matrix. Also, since data is merely copied, the most efficient memory transfer mechanism available for a processor can be utilized, and it does not necessarily need to be floating-point based, as it is merely tasked with the transfer from memory to memory.

Pseudo Vec

Listing 12-2: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixCopy( vmp3DMatrix dMx,
                const vmp3DMatrix aMx )
{
  dMx[0][0]=aMx[0][0];  dMx[0][1]=aMx[0][1];
  dMx[0][2]=aMx[0][2];  dMx[0][3]=aMx[0][3];
  dMx[1][0]=aMx[1][0];  dMx[1][1]=aMx[1][1];
  dMx[1][2]=aMx[1][2];  dMx[1][3]=aMx[1][3];
  dMx[2][0]=aMx[2][0];  dMx[2][1]=aMx[2][1];
  dMx[2][2]=aMx[2][2];  dMx[2][3]=aMx[2][3];
  dMx[3][0]=aMx[3][0];  dMx[3][1]=aMx[3][1];
  dMx[3][2]=aMx[3][2];  dMx[3][3]=aMx[3][3];
}
```

Pseudo Vec (X86)

The code gets more efficient when a processor's X86 pipelining architecture is taken advantage of, such as in the following function. Since pipelining is maximized and dependency stalls are minimized, maximum throughput takes place, and then the XMM 128-bit register works out nicely! As this was implemented with a macro assembler, note the use of the expansion macro *REPT* (repeat) and the *i* to expand a block of code to unroll any loops. Note that it is only lightly optimized to keep the code readable.

vmp_MatrixCopy (X86-Asm)

Listing 12-3: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
  mov    eax,[esp+8]      ; aMx
  mov    edx,[esp+4]      ; dMx
  push   ebx

  mov    ecx,[eax+0]      ; =aMx[0]
  mov    ebx,[eax+4]      ; =aMx[1]

i = 0
REPT     16/2 -1          ; Repeat 7 times
  mov    [edx+0+i],ecx    ;  dMx[0,2,4,6,8,12]=
  mov    ecx,[eax+8+i]    ; =aMx[2,4,6,8,10,12,14]
  mov    [edx+4+i],ebx    ;  dMx[1,3,5,7,9,11,13]=
  mov    ebx,[eax+12+i]   ; =aMx[3,5,7,9,11,13,15]
i = i + 8                 ; 8, 16, 24, 32, 40, 48
ENDM

  mov    [edx+48],ecx     ; dMx[14]=
  mov    [edx+56],ebx     ; dMx[15]=

  pop    ebx
  ret
```

vmp_MatrixCopy (SSE) Aligned

There is really no need for MMX or XMM registers to be utilized, as they are not needed due to the previous pipelined code, unless you wish to reduce code size. So in that case, examine the following SSE code snippet. Note the use of *movaps*, as in this case the matrix is assumed to reside in aligned memory.

Listing 12-4: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
  mov    eax,[esp+8]      ; aMx
  mov    edx,[esp+4]      ; dMx
```

```
movaps xmm0,[eax+0]    ; =aMx[0...3]
movaps xmm1,[eax+16]   ; =aMx[4...7]
movaps xmm2,[eax+32]   ; =aMx[8...11]
movaps xmm3,[eax+48]   ; =aMx[12...15]
movaps [edx+0],xmm0    ;  dMx[0...3]=
movaps [edx+16],xmm1   ;  dMx[4...7]=
movaps [edx+32],xmm2   ;  dMx[8...11]=
movaps [edx+48],xmm3   ;  dMx[12...15]=
ret
```

Of course, *movdqa* (used for aligned integers) could just as easily have been used because we are not processing this data in memory — only moving it!

Pseudo Vec (MIPS)

There are at least two methods to copy matrices, depending upon which source file type is used: assembly (*.S) or inline assembly (*.C) with GNU C. On a PS2 console, the vector base floating-point values are handled by the VU coprocessor, but since no mathematical operations are performed (only memory movement), then only the MMI instructions are needed. Both source file types are presented here.

vmp_MatrixCopy (MMI) Aligned (.s)*

Listing 12-5: vmp_mips\chap12\matrix3d\Matrix3DMMI.s

```
lq   t1, 0(a1)        // aMx[a0...a3]  {a3 a2 a1 a0}
lq   t2, 0(a1)        // aMx[a4...a7]  {a7 a6 a5 a4}
lq   t3, 0(a1)        // aMx[a8...aB]  {aB aA a9 a8}
lq   t4, 0(a1)        // aMx[aC...aF]  {aF aE aD aC}

sq   t1, 0(a0)        // dMx[d0...d3]  {d3 d2 d1 d0}
sq   t2, 0(a0)        // dMx[d4...d7]  {d7 d6 d5 d4}
sq   t3, 0(a0)        // dMx[d8...dB]  {dB dA d9 d8}
sq   t4, 0(a0)        // dMx[dC...dF]  {dF dE dD dC}
```

vmp_MatrixCopy (MMI) Aligned – GNU C (.c)*

Listing 12-6: vmp_mips\chap12\matrix3d\Matrix3DVU0.c

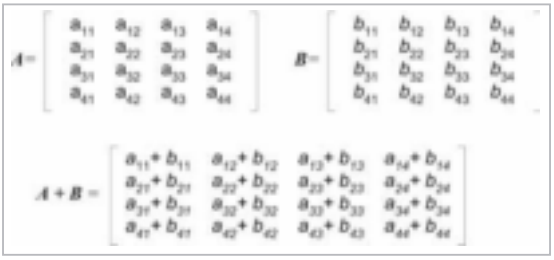
```
lq   $6, 0x00(%1)     // aMx[a0...a3]  {a3 a2 a1 a0}
lq   $7, 0x10(%1)     // aMx[a4...a7]  {a7 a6 a5 a4}
lq   $8, 0x20(%1)     // aMx[a8...aB]  {aB aA a9 a8}
lq   $9, 0x30(%1)     // aMx[aC...aF]  {aF aE aD aC}

sq   $6, 0x10(%0)     // dMx[d0...d3]  {d3 d2 d1 d0}
sq   $7, 0x20(%0)     // dMx[d4...d7]  {d7 d6 d5 d4}
sq   $8, 0x30(%0)     // dMx[d8...dB]  {dB dA d9 d8}
sq   $9, 0x40(%0)     // dMx[dC...dF]  {dF dE dD dC}
```

Matrix Summation (D=A+B)

The summation of two same-sized matrices ($c_{ij}=a_{ij}+b_{ij}$) is extremely easy, as the scalar of both matrices are summed and stored in the same indexed cell location of the same-sized destination matrix.

$$[a_{ij}]+[b_{ij}] = [a_{ij}+b_{ij}]$$

Equation 12-4: Matrix to matrix summation (A + B)



Algebraic Law:

Commutative Law of Addition	a + b = b + a
Commutative Law of Multiplication	ab = ba

In relationship with the algebraic laws, it is both commutative A+B=B+A and associative A+(B+C)=(A+B)+C, as each element is isolated. Thus, no element affects an adjacent element.

Pseudo Vec

Listing 12-7: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixAdd( vmp3DMatrix dMx,
                    const vmp3DMatrix aMx,
                    const vmp3DMatrix bMx )
{
  dMx[0][0] = aMx[0][0] + bMx[0][0];
  dMx[0][1] = aMx[0][1] + bMx[0][1];
  dMx[0][2] = aMx[0][2] + bMx[0][2];
  dMx[0][3] = aMx[0][3] + bMx[0][3];

  dMx[1][0] = aMx[1][0] + bMx[1][0];
  dMx[1][1] = aMx[1][1] + bMx[1][1];
  dMx[1][2] = aMx[1][2] + bMx[1][2];
  dMx[1][3] = aMx[1][3] + bMx[1][3];

  dMx[2][0] = aMx[2][0] + bMx[2][0];
  dMx[2][1] = aMx[2][1] + bMx[2][1];
  dMx[2][2] = aMx[2][2] + bMx[2][2];
  dMx[2][3] = aMx[2][3] + bMx[2][3];
```

```
  dMx[3][0] = aMx[3][0] + bMx[3][0];
  dMx[3][1] = aMx[3][1] + bMx[3][1];
  dMx[3][2] = aMx[3][2] + bMx[3][2];
  dMx[3][3] = aMx[3][3] + bMx[3][3];
}
```

This should easily be recognized as four quad vector summations in parallel. So the code similar to vmp_QVecAdd defined in Chapter 8, "Vector Addition and Subtraction," can be used here.

Scalar Matrix Product (rA)

$r[a_{ij}]$

In a scalar multiplication, a scalar is applied to each element of a matrix and the resulting product is stored in a same-size matrix.

Equation 12-5: Scalar matrix multiplication (*rA*)

$$rA = \begin{bmatrix} ra_{11} & ra_{12} & ra_{13} & ra_{14} \\ ra_{21} & ra_{22} & ra_{23} & ra_{24} \\ ra_{31} & ra_{32} & ra_{33} & ra_{34} \\ ra_{41} & ra_{42} & ra_{43} & ra_{44} \end{bmatrix}$$

Pseudo Vec

Listing 12-8: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixScalar( vmp3DMatrix dMx,
                       const vmp3DMatrix aMx,
                       float r )
{
  dMx[0][0] = aMx[0][0] * r;
  dMx[0][1] = aMx[0][1] * r;
  dMx[0][2] = aMx[0][2] * r;
  dMx[0][3] = aMx[0][3] * r;

  dMx[1][0] = aMx[1][0] * r;
  dMx[1][1] = aMx[1][1] * r;
  dMx[1][2] = aMx[1][2] * r;
  dMx[1][3] = aMx[1][3] * r;

  dMx[2][0] = aMx[2][0] * r;
  dMx[2][1] = aMx[2][1] * r;
  dMx[2][2] = aMx[2][2] * r;
  dMx[2][3] = aMx[2][3] * r;

  dMx[3][0] = aMx[3][0] * r;
  dMx[3][1] = aMx[3][1] * r;
  dMx[3][2] = aMx[3][2] * r;
  dMx[3][3] = aMx[3][3] * r;
}
```

This would use similar assembly code to that of the function vmp_QVecScale defined in Chapter 9, "Vector Multiplication and Division."

Apply Matrix to Vector (Multiplication) (vA)

When a vector is applied to a matrix, the product of each scalar of the vector and each scalar of a column of the matrix is summed, and the total is stored in the destination vector of the same element of the source vector. The first expression uses the highlighted scalars. As a vector has four elements, there are four expressions.

$$w_i = v_1 a_{1i} + v_2 a_{2i} + v_3 a_{3i} + v_4 a_{4i}$$

$$w_1 = v_1 a_{11} + v_2 a_{21} + v_3 a_{31} + v_4 a_{41}$$

Equation 12-6: Apply matrix to vector (multiplication) ($v \square A$)

$$v = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} \qquad A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$$\begin{aligned} w_1 &= v_1 a_{11} + v_2 a_{21} + v_3 a_{31} + v_4 a_{41} \\ w_2 &= v_1 a_{12} + v_2 a_{22} + v_3 a_{32} + v_4 a_{42} \\ w_3 &= v_1 a_{13} + v_2 a_{23} + v_3 a_{33} + v_4 a_{43} \\ w_4 &= v_1 a_{14} + v_2 a_{24} + v_3 a_{34} + v_4 a_{44} \end{aligned}$$

Pseudo Vec

Listing 12-9: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_QVecApplyMatrix( vmp3DQVector *pvD,
                    const vmp3DQVector * const pvA,
                    const vmp3DMatrix   bMx )
{
  vmp3DQVector vD;

  vD.x   = (bMx[0][0] * pvA->x) + (bMx[1][0] * pvA->y)
         + (bMx[2][0] * pvA->z) + (bMx[3][0] * pvA->w);
  vD.y   = (bMx[0][1] * pvA->x) + (bMx[1][1] * pvA->y)
         + (bMx[2][1] * pvA->z) + (bMx[3][1] * pvA->w);
  vD.z   = (bMx[0][2] * pvA->x) + (bMx[1][2] * pvA->y)
         + (bMx[2][2] * pvA->z) + (bMx[3][2] * pvA->w);
  pvD->w = (bMx[0][3] * pvA->x) + (bMx[1][3] * pvA->y)
         + (bMx[2][3] * pvA->z) + (bMx[3][3] * pvA->w);
  pvD->x = vD.x;
  pvD->y = vD.y;
  pvD->z = vD.z;
}
```

Matrix Multiplication (D=AB)

The following demonstrates the product of two 4x4 matrices ($d_{ik}=a_{ij}b_{jk}$). The index *j* represents the Einstein Summation for all indices of *i* and *k*. The first expression uses the highlighted scalars.

$$d_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + a_{i4}b_{4j}$$

$$d_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

Equation 12-7: Matrix to matrix multiplication *(AB)*

Each of the 16 resulting scalars is the product summation of each scalar in a row of matrix A and a scalar from a column of matrix B.

$$\begin{aligned}
d_{11} &= a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}\\
d_{12} &= a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42}\\
d_{13} &= a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} + a_{14}b_{43}\\
d_{14} &= a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} + a_{14}b_{44}
\end{aligned}$$

$$\begin{aligned}
d_{21} &= a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41}\\
d_{22} &= a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42}\\
d_{23} &= a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} + a_{24}b_{43}\\
d_{24} &= a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} + a_{24}b_{44}
\end{aligned}$$

$$\begin{aligned}
d_{31} &= a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} + a_{34}b_{41}\\
d_{32} &= a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42}\\
d_{33} &= a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} + a_{34}b_{43}\\
d_{34} &= a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44}
\end{aligned}$$

$$\begin{aligned}
d_{41} &= a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} + a_{44}b_{41}\\
d_{42} &= a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42}\\
d_{43} &= a_{41}b_{13} + a_{42}b_{23} + a_{43}b_{33} + a_{44}b_{43}\\
d_{44} &= a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44}
\end{aligned}$$

In relationship with the algebraic laws, it is not commutative, but it is associative! Use the representation most familiar to you (mathematicians on the left and C programmers on the right!).

```
AB  BA
```

```
AB != BA
```

That should make all of you happy!

Thus, the ordering of matrix A versus B needs to be considered when performing this operation.

Pseudo Vec

The following C code is an example of where individual floats are processed. This is not very efficient due to the two-dimensional array references involving *u* and *v*. The constants {0,1,2,3} fix the column index and are used in conjunction with row *u* for Matrix A, fix the row in conjunction with *v* for Matrix B, and are used to determine the memory reference to access the float values.

Listing 12-10: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixMul( vmp3DMatrix dMx,  // Dst
                    vmp3DMatrix aMx,  // A
                    vmp3DMatrix bMx ) // B
{
  vmp3DMatrix mx;
  uint u, v;

  for ( u = 0; u < 4; u++ )
  {
    for ( v = 0; v < 4; v++ )
    {
      mx[u][v] = aMx[u][0] * bMx[0][v]
               + aMx[u][1] * bMx[1][v]
               + aMx[u][2] * bMx[2][v]
               + aMx[u][3] * bMx[3][v];
    }
  }

  vmp_MatrixCopy(dMx, mx);
}
```

This following remark is not a vector math technique but actually one of optimization, which easily leads to a method of vectorization. In fact, those constants mentioned earlier are a big clue as to how to vectorize this code. Also the double *u* and *v* loops with double indexing just scream vector pointer math! By incrementing a vector, the same effective address can be realized. The 4x4 matrix is displayed again, only this time an index into an array of floats is used instead to represent each float. In the following parsing dump, the #*# represents the *A* vector element being multiplied with the *B* vector element (*A*B* (index)).

```
M[0][1][2][3]    [4][5][6][7]    [8][9][10][11]    [12][13][14][15]
             #*#      #*#      #*#      #*#
D11= M[0]=   0*0  +   1*4  +   2*8  +  3*12
D12= M[1]=   0*1  +   1*5  +   2*9  +  3*13
D13= M[2]=   0*2  +   1*6  +   2*10 +  3*14
D14= M[3]=   0*3  +   1*7  +   2*11 +  3*15

D21= M[4]=   4*0  +   5*4  +   6*8  +  7*12
D22= M[5]=   4*1  +   5*5  +   6*9  +  7*13
```

```
D23= M[6]=   4*2  +   5*6  +   6*10 +  7*14
D24= M[7]=   4*3  +   5*7  +   6*11 +  7*15

D31= M[8]=   8*0  +   9*4  +  10*8  +  11*12
D32= M[9]=   8*1  +   9*5  +  10*9  +  11*13
D33= M[10]=  8*2  +   9*6  +  10*10 +  11*14
D34= M[11]=  8*3  +   9*7  +  10*11 +  11*15

D41= M[12]= 12*0  +  13*4  +  14*8  +  15*12
D42= M[13]= 12*1  +  13*5  +  14*9  +  15*13
D43= M[14]= 12*2  +  13*6  +  14*10 +  15*14
D44= M[15]= 12*3  +  13*7  +  14*11 +  15*15
```

The blocks are visually separated into blocks of four lines each to represent an equation for one quad vector. If you examine the patterns visible in each block, you will note some similarities. A scalar is replicated within each field. Also, the original four quad vectors denoted in bold are product summed with the replicated scalars for each of the four destination vectors. By translating the dump of arrays to structures, 16 equations come to light with the following as a single example:

$$D[0]_x = A[0]_xB[0]_x + A[0]_yB[1]_x + A[0]_zB[2]_x + A[0]_wB[3]_x$$

If we think about using a simple structure for A and D and advancing it for every four equations:

```
Loop {0...3}
Dx = AxB[0]x + AyB[1]x + AzB[2]x + AwB[3]x
Dy = AxB[0]y + AyB[1]y + AzB[2]y + AwB[3]y
Dz = AxB[0]z + AyB[1]z + AzB[2]z + AwB[3]z
Dw = AxB[0]w + AyB[1]w + AzB[2]w + AwB[3]w
D += 1;   A += 1;
```

This expression is laid out nicely if the processor supports a product-sum instruction. Thus, each element of D is resolved by a single instruction.

$$D_x = A_xB[0]_x + A_yB[1]_x + A_zB[2]_x + A_wB[3]_x$$

```
Dx = AxBx + AyBy + AzBz + AwBw                ; Multiply-Add
```

If the multiplication and sum have to be resolved separately, then the four equations need to be resolved vertically.

```
Loop {0...3}
Dxyzw =          AxB[0]x   AxB[0]y   AxB[0]z   AxB[0]w
Dxyzw = Dxyzw + AyB[1]x   AyB[1]y   AyB[1]z   AyB[1]w
Dxyzw = Dxyzw + AzB[2]x   AzB[2]y   AzB[2]z   AzB[2]w
Dxyzw = Dxyzw + AwB[3]x   AwB[3]y   AwB[3]z   AwB[3]w
D + 1;  A + 1;
```

So with these in mind, the following code shows the code unrolled into a multiply-add supported vector organization.

Listing 12-11: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixMul( vmp3DMatrix dMx,
              const vmp3DMatrix aMx,
              const vmp3DMatrix bMx )
{
  vmp3DMatrix mx;
  uint u;
  vmp3DQVector *pvA, *pvB, *pvD;

  pvA = (vmp3DQVector*)aMx;
  pvB = (vmp3DQVector*)bMx;
  pvD = (vmp3DQVector*)dMx;

  for ( u=0; u < 4; u++ )
  {
    pvD->x = ((pvB+0)->x * pvA->x) + ((pvB+1)->x * pvA->y)
           + ((pvB+2)->x * pvA->z) + ((pvB+3)->x * pvA->w);
    pvD->y = ((pvB+0)->y * pvA->x) + ((pvB+1)->y * pvA->y)
           + ((pvB+2)->y * pvA->z) + ((pvB+3)->y * pvA->w);
    pvD->z = ((pvB+0)->z * pvA->x) + ((pvB+1)->z * pvA->y)
           + ((pvB+2)->z * pvA->z) + ((pvB+3)->z * pvA->w);
    pvD->w = ((pvB+0)->w * pvA->x) + ((pvB+1)->w * pvA->y)
           + ((pvB+2)->w * pvA->z) + ((pvB+3)->w * pvA->w);
    pvA++;
    pvD++;
  }

  vmp_MatrixCopy(dMx, mx);
}
```

Pseudo Vec (X86)

```
mov    eax,[esp+8]       ; aMx
mov    ebx,[esp+12]      ; bMx
mov    edx,[esp+4]       ; dMx
```

vmp_MatrixMul (SSE)

As mentioned earlier, the SSE instruction set does not support a multiplication-add instruction, and so it must be handled with two separate instructions. The following code has been written to solve the previous C example using a vertical orientation, thus the data is handled in parallel. There is no need for a scratch matrix buffer since matrix B is preloaded and a vector row of matrix A is loaded and written as a vector row of matrix D. The code is not optimized and has been rolled into an assembler macro to save printed space.

Listing 12-12: \chap12\matrix3d\Matrix3DX86M.asm

```
movaps xmm4,[ebx+(0*4)];  [b3 b2 b1 b0] wzyx
movaps xmm5,[ebx+(4*4)];  [b7 b6 b5 b4] wzyx
movaps xmm6,[ebx+(8*4)];  [bB bA b9 b8] wzyx
movaps xmm7,[ebx+(12*4)]; [bF bE bD bC] wzyx
```

```
i = 0
REPT   4
  movaps xmm0,[eax+i]          ; [a3 a2 a1 a0] wzyx

  movaps xmm1,xmm0
  movaps xmm2,xmm0
  movaps xmm3,xmm0
  shufps xmm0,xmm0,00000000b   ; [a0 a0 a0 a0]
  shufps xmm1,xmm1,01010101b   ; [a1 a1 a1 a1]
  shufps xmm2,xmm2,10101010b   ; [a2 a2 a2 a2]
  shufps xmm3,xmm3,11111111b   ; [a3 a3 a3 a3]

  mulps  xmm0,xmm4             ; [a0*b3 a0*b2 a0*b1 a0*b0]
  mulps  xmm1,xmm5             ; [a1*b7 a1*b6 a1*b5 a1*b4]
  mulps  xmm2,xmm6             ; [a2*bB a2*bA a2*b9 a2*b8]
  mulps  xmm3,xmm7             ; [a3*bF a3*bE a3*bD a3*bC]

  addps  xmm2,xmm0
  addps  xmm3,xmm1
  addps  xmm3,xmm2
  movaps [edx+i],xmm3          ; [d3 d2 d1 d0] wzyx

i = i + (4*4)
ENDM
```

Pseudo Vec (SH4)

This has been placed here as a courtesy and an honorable mention for the Hitachi SH4 series processors used in embedded systems as well as the now retired Dreamcast video game console. It is pretty much a 32-bit processor, except it has a couple of nifty instructions and a unique use for banks of registers. The processor has thirty-two 32-bit floating-point registers, which are grouped into two banks of 16 each. By loading individual floats into a complete bank and switching banks, they are treated as an XMTRX (register-matrix). A quad vector can be loaded into the alternate bank and processed. The processor is not a vector processor, only a handler of vectors and matrices.

In the case of the *fipr* (Floating-Point Inner Product) instruction:

```
FIPR   FVa,VFb   ; Float Vector Index a,b:{0,4,8,12}
```

...when *a* is not equal to *b*, then it is an inner product (dot product) calculation, and when *a* is equal to *b*, then it is a magnitude (length of vector).

```
(Sum of Squares)        r = a_x^2 + a_y^2 + a_z^2 + a_w^2     when a=b
(Dot Product)           r = a_x b_x + a_y b_y + a_z b_z + a_w b_w     when a<>b
```

In the case of the *ftrv* (Floating-Point Transform Vector) instruction:

```
FTRV   XMTRX,FV#   ; Float vector index #:{0,4,8,12}
```

... each row of a second matrix can be handled simultaneously in a series of one pass each using approximate-value computation processing.

The transform function handles all the multiplication sums for a row of a matrix when each element of the vector is applied by each element of a matrix. When done in four stages, all four vectors of the second matrix are processed. Each element is loaded individually and the instruction can be thought of as a batch processing instruction.

The following is an example function using that processing power:

Listing 12-13: vmp_sh4\chap12\Matrix3DSH4.asm

```
   ; Included here with permission from Hitachi but with some minor
   ; modifications

_vmp_MatrixMulSH4:

  ASSERT_PTR4(r4)       ; pvD
  ASSERT_PTR4(r5)       ; pvA
  ASSERT_PTR4(r6)       ; pvB

  add   #-12,r15        ; Adjust Stack

  mov.l r6,@(8,r15)     ; Save ptr to |B| on local stack
  mov.l r5,@(4,r15)     ;    "        |A|       "
  mov.l r4,@r15         ;    "        |D|       "

  mov.l @(8,r15),r2     ; =|B|

  frchg    ; Set Matrix elements in background bank

  fmov.s @r2+,fr0       ; [b0         ]
  fmov.s @r2+,fr1       ; [   b1      ]
  fmov.s @r2+,fr2       ; [      b2   ]
  fmov.s @r2+,fr3       ; [         b3]

  fmov.s @r2+,fr4       ; [b4         ]
  fmov.s @r2+,fr5       ; [   b5      ]
  fmov.s @r2+,fr6       ; [      b6   ]
  fmov.s @r2+,fr7       ; [         b7]

  fmov.s @r2+,fr8       ; [b8         ]
  fmov.s @r2+,fr9       ; [   b9      ]
  fmov.s @r2+,fr10      ; [      bA   ]
  fmov.s @r2+,fr11      ; [         bB]

  fmov.s @r2+,fr12      ; [bC         ]
  fmov.s @r2+,fr13      ; [   bD      ]
  fmov.s @r2+,fr14      ; [      bE   ]
  fmov.s @r2+,fr15      ; [         bF]
  frchg

  mov.l  @(4,r15),r3    ; =|A|
  mov.l  @r15,r1        ; =|D|
```

```
    ; 1st row             [a0 a1 a2 a3]

  fmov.s @r3+,fr0       ; [a0         ] (Load)
  fmov.s @r3+,fr1       ; [   a1      ]
  fmov.s @r3+,fr2       ; [      a2   ]
  fmov.s @r3+,fr3       ; [         a3]

  ftrv   xmtrx,fv0
  add    #16,r1         ; skip past end of 1st row of |D|

REPT 3                  ; macro - Repeat 3 times
                        ; [ 0  1  2  3] 1st pass
                        ; [ 4  5  6  7] 2nd pass
                        ; [ 8  9  A  B] 3rd pass

  fmov.s fr3,@-r1       ; [         d3] (Save 1st ... 3rd row)
  fmov.s fr2,@-r1       ; [      d2   ]    decrementally
  fmov.s fr1,@-r1       ; [   d1      ]
  fmov.s fr0,@-r1       ; [d0         ]

  fmov.s @r3+,fr0       ; [a0         ] (Load 2nd ... 4th row)
  fmov.s @r3+,fr1       ; [   a1      ]
  fmov.s @r3+,fr2       ; [      a2   ]
  fmov.s @r3+,fr3       ; [         a3]

  ftrv   xmtrx,fv0
  add    #32,r1         ; skip past end of n+2 rows of |D|
ENDM                    ; end macro

  fmov.s fr3,@-r1       ; [         dF] (Save 4th row)
  fmov.s fr2,@-r1       ; [      dE   ] decrementally
  fmov.s fr1,@-r1       ; [   dD      ]
  fmov.s fr0,@-r1       ; [dC         ]

  add    #12,r15        ; Unadjust stack
  rts
  nop                   ; nop = Return Delay Slot
```

The repeat macro was used to help save space!

Matrix Set Identity

An identity matrix (sometimes referred to as *I*) is typically the base foundation of other matrix types. As shown, all scalars are set to zero except for the scalars on the diagonal, which are set to 1s.

Equation 12-8: Identity matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

left & right-handed

This is considered an initialized matrix, as when applied to (multiplied with) a vector, the original vector will result. If this is examined more carefully where matrix *A* is applied to vector *v:*

$$w_1 = v_1a_{11} + v_2a_{21} + v_3a_{31} + v_4a_{41}$$
$$w_2 = v_1a_{12} + v_2a_{22} + v_3a_{32} + v_4a_{42}$$
$$w_3 = v_1a_{13} + v_2a_{23} + v_3a_{33} + v_4a_{43}$$
$$w_4 = v_1a_{14} + v_2a_{24} + v_3a_{34} + v_4a_{44}$$

...and when the identity matrix is substituted for the matrix:

$$w_1 = v_1 = v_1(1) + v_2(0) + v_3(0) + v_4(0)$$
$$w_2 = v_2 = v_1(0) + v_2(1) + v_3(0) + v_4(0)$$
$$w_3 = v_3 = v_1(0) + v_2(0) + v_3(1) + v_4(0)$$
$$w_4 = v_4 = v_1(0) + v_2(0) + v_3(0) + v_4(1)$$

Pseudo Vec

Another way to look at the identity of a matrix is that the result of a product of a matrix and an identity of the same size is a matrix equivalent to the original matrix. Also, do not fear optimizational waste due to the two-dimensional array reference because the const is converted to an offset during compilation.

Listing 12-14: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixSetIdentity( vmp3DMatrix Mx )
{
            Mx[0][1] = Mx[0][2] = Mx[0][3] =
  Mx[1][0] =           Mx[1][2] = Mx[1][3] =
  Mx[2][0] = Mx[2][1] =           Mx[2][3] =
  Mx[3][0] = Mx[3][1] = Mx[3][2] =           0.0f;

  Mx[0][0] =
              Mx[1][1] =
                          Mx[2][2] =
                                      Mx[3][3]=  1.0f;
}
```

Pseudo Vec (X86)

The code gets more efficient when a processor's X86 memory write pipelining architecture is used to its advantage.

vmp_MatrixSetIdentity (X86)

By preloading two separate registers and then writing adjacent bytes, data can be written out at the processor's best speed using all available pipes that support memory write functionality. This is similar to what the C code does but is much more efficient.

Listing 12-15: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
  mov   eax,[esp+4]
  mov   ecx,FLOAT1      ; 1.0f
  mov   edx,FLOAT0      ; 0.0f

  mov   (vmp3DMatrix PTR [eax])._m00,ecx   ;[a0] = 1.0
  mov   (vmp3DMatrix PTR [eax])._m01,edx   ;[a1] = 0.0
  mov   (vmp3DMatrix PTR [eax])._m02,edx   ;[a2]
  mov   (vmp3DMatrix PTR [eax])._m03,edx   ;[a3]

  mov   (vmp3DMatrix PTR [eax])._m10,edx   ;[a4]
  mov   (vmp3DMatrix PTR [eax])._m11,ecx   ;[a5] = 1.0
  mov   (vmp3DMatrix PTR [eax])._m12,edx   ;[a6]
  mov   (vmp3DMatrix PTR [eax])._m13,edx   ;[a7]

  mov   (vmp3DMatrix PTR [eax])._m20,edx   ;[a8]
  mov   (vmp3DMatrix PTR [eax])._m21,edx   ;[a9]
  mov   (vmp3DMatrix PTR [eax])._m22,ecx   ;[aA] = 1.0
  mov   (vmp3DMatrix PTR [eax])._m23,edx   ;[aB]

  mov   (vmp3DMatrix PTR [eax])._m30,edx   ;[aC]
  mov   (vmp3DMatrix PTR [eax])._m31,edx   ;[aD]
  mov   (vmp3DMatrix PTR [eax])._m32,edx   ;[aE]
  mov   (vmp3DMatrix PTR [eax])._m33,ecx   ;[aF] = 1.0
```

vmp_MatrixSetIdentity (MMX)

I strongly recommend not using MMX, unless you are using the 3DNow! instruction set. The problem is that if you are doing any trigonometry or scalar floating-point calculations, an EMMS switch has to be done to change from FPU to MMX mode and then back. This function uses light-weight memory writes, and so well-architected pipelining with 32-bit registers will be equivalent.

Listing 12-16: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
        REAL4  0.0, 0.0, 0.0, 0.0
vec1000 REAL4  1.0, 0.0, 0.0, 0.0

  mov  eax,[esp+4]

  movq  mm2,qword ptr vec1000-4  ; 0 1   [1 0]
  pxor  mm0,mm0                  ; 0 0   [0 0]
  movd  mm1,vec1000              ; 1 0   [0 1]

  movq  (qword ptr [eax+0 ]),mm1 ; 1 0
  movq  (qword ptr [eax+08]),mm0 ;     0 0
  movq  (qword ptr [eax+16]),mm2 ; 0 1
  movq  (qword ptr [eax+24]),mm0 ;     0 0
  movq  (qword ptr [eax+32]),mm0 ; 0 0
  movq  (qword ptr [eax+40]),mm1 ;     1 0
  movq  (qword ptr [eax+48]),mm0 ; 0 0
  movq  (qword ptr [eax+56]),mm2 ;     0 1
```

vmp_MatrixSetIdentity (SSE)

If you are using an SSE-capable processor, then this is the best solution out of many solutions for which only the following has been provided. A single load from memory is shared between two registers with data shuffling and pipelining to minimize stalls.

Listing 12-17: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
                                    ; x y z w   w z y x
movaps xmm0,oword ptr vec1000       ; 1 0 0 0  [0 0 0 1]
mov    eax,[esp+4]

movaps xmm1,xmm0
movaps (oword ptr [eax+0 ]),xmm0
shufps xmm0,xmm0,11110011b          ; 0 1 0 0  [0 0 1 0]
shufps xmm1,xmm1,11001111b          ; 0 0 1 0  [0 1 0 0]
movaps (oword ptr [eax+16]),xmm0
shufps xmm0,xmm0,01111111b          ; 0 0 0 1  [1 0 0 0]
movaps (oword ptr [eax+32]),xmm1
movaps (oword ptr [eax+48]),xmm0
```

Matrix Set Scale

The scaling factor is set by having all zeros in the application matrix and the scale set for {XYZ} set on the diagonal. In a 3D coordinate system, there are two primary methods of display — a left-handed system and a right-handed system. Some matrices, such as those used for scaling, are identical for both coordinate systems. When a lower or upper diagonal is effected, the properties of the matrix are dependent upon the hand orientation.

Equation 12-9: Scaling matrix

$$\begin{bmatrix} x_s & 0 & 0 & 0 \\ 0 & y_s & 0 & 0 \\ 0 & 0 & z_s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

left & right-handed

Note that a scale factor of one is identical to the identity matrix, and thus no change in any of the elemental values will occur. A factor of two doubles the elements in size and a factor of 0.5 halves (shrinks) them. Each {XYZ} coordinate has a separate scale factor, so an object can be scaled at different rates on each of the axes, such that {0.5,1,2} would cause the X element to reduce by half, the Y element to remain the same, and the Z element to double in size. The point is that each vector coordinate is individually scaled.

Pseudo Vec

Listing 12-18: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixSetScaling( vmp3DMatrix Mx,
                           float fSx, float fSy, float fSz )
{
  Mx[0][1] = Mx[0][2] = Mx[0][3] =
Mx[1][0] = Mx[1][2] = Mx[1][3] =
Mx[2][0] = Mx[2][1] = Mx[2][3] =
Mx[3][0] = Mx[3][1] = Mx[3][2] =  0.0f;

  Mx[0][0] = fSx;
             Mx[1][1] = fSy;
                        Mx[2][2] = fSz;
                                   Mx[3][3]=  1.0f;
}
```

Pseudo Vec (X86)

The implementations for the set identity are very similar to this function, except for using the scaling factors instead of the identity of a value of one. The SSE has been shown here for the "out of the box" thinking used for setting up the matrix.

vmp_MatrixSetScaling (SSE)

This one is interesting because with the 4x4 matrix, the scaling sizes are on the diagonals, and there are four elements set to zero between them. Writing an unaligned 128 bits of zero for each of these areas clears all of them with a minimal amount of writes.

Listing 12-19: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
  movaps xmm0,oword ptr vec1000-16      ;[0 0 0 0]

  mov    edx,[esp+4]
  mov    eax,[esp+8]                    ;    fSx
  mov    ebx,[esp+12]                   ;    fSy
  mov    ecx,[esp+16]                   ;    fSz

  ; Four bytes between scale factors so write unaligned!

  movups (oword ptr [edx+(1*4)]),xmm0   ; [ •  1  2  3]
  movups (oword ptr [edx+(6*4)]),xmm0   ; [ 4  •  6  7]
  movups (oword ptr [edx+(11*4)]),xmm0  ; [ 8  9  • 11]
                                        ; [12 13 14  •]

  ; Write individual scaling values

  mov    (dword ptr [edx+(0*4)]),eax    ;[0]=fSx
  mov    (dword ptr [edx+(5*4)]),ebx    ;[5]=fSy
  mov    (dword ptr [edx+(15*4)],FLOAT1 ;[15]= 1.0
  mov    (dword ptr [edx+(10*4)]),ecx   ;[10]=fSz
```

Matrix Set Translation

A translation matrix displaces a vector by translating its position by the amount specified by $t_{\{xyz\}}$.

Equation 12-10: Translation matrix left-handed row versus right-handed column matrix

If there is no translation (adjustment of position), then $t_{\{xyz\}}$ are all set to zero, thus it performs just like an identity matrix. Now if the translation actually has non-zero values, the vector is adjusted and displaced on the {XYZ} axis. Note that the vector really consists of the three coordinates {XYZ} with a fourth tending to be a placeholder. When this matrix is applied to the vector:

$$
\begin{aligned}
w_1 &= v_1 + t_x = & v_1(1) + v_2(0) + v_3(0) + v_4(t_x)\\
w_2 &= v_2 + t_y = & v_1(0) + v_2(1) + v_3(0) + v_4(t_y)\\
w_3 &= v_3 + t_z = & v_1(0) + v_2(0) + v_3(1) + v_4(t_z)\\
w_4 &= v_4 = & v_1(0) + v_2(0) + v_3(0) + v_4(1)
\end{aligned}
$$

...the position is adjusted (displaced) accordingly.

Simplified equation:

```
dx =  x + tx;
dy =  y + ty;
dz =  z + tz;
```

Pseudo Vec

Listing 12-20: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixSetTranslate( vmp3DMatrix Mx,
                  float fTx, float fTy, float fTz )
{
                  Mx[0][1] =   Mx[0][2] =   Mx[0][3] =
  Mx[1][0] =                   Mx[1][2] =   Mx[1][3] =
  Mx[2][0] =     Mx[2][1] =                 Mx[2][3] = 0.0f;
  Mx[3][0] =fTx; Mx[3][1]=fTy; Mx[3][2]=fTz;

  Mx[0][0] =
                  Mx[1][1] =
                               Mx[2][2] =
                                           Mx[3][3] = 1.0f;
}
```

Again, this is similar to the identity matrix, so there is no real need to replicate the code again here with some minor modification. Refer to the sample code on the companion CD for more details.

Matrix Transpose

A transposed matrix A^T is indicated by the superset ^{T}. It is effectively the swap of all elements referenced by row-column with column-row indexing, and vice versa. Effectively, as the row and column are equivalent for the diagonal, those elements are retained as indicated by the gray diagonal.

Equation 12-11: Transpose matrix

Interesting, is it not? Recognize it? The starting matrix on the right is similar to that of AoS (Array of Structures) {XYZW}[4], and the resulting matrix on the left is that of an SoA (Structure of Arrays) {X[4], Y[4], Z[4], W[4]}.

Pseudo Vec

Note that a temporary array needs to be used to save floats, just in case the destination matrix to contain the results of the transpose is the source matrix.

Listing 12-21: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixTranspose( vmp3DMatrix dMx,
                    const vmp3DMatrix aMx )
{
  float f[6];

  dMx[0][0]=aMx[0][0];    dMx[1][1]=aMx[1][1];
  dMx[2][2]=aMx[2][2];    dMx[3][3]=aMx[3][3];

  f[0]=aMx[0][1];    f[1]=aMx[0][2];    f[2]=aMx[0][3];
  f[3]=aMx[1][2];    f[4]=aMx[1][3];    f[5]=aMx[2][3];

  dMx[0][1]=aMx[1][0];    dMx[1][0]=f[0];
  dMx[0][2]=aMx[2][0];    dMx[2][0]=f[1];
  dMx[0][3]=aMx[3][0];    dMx[3][0]=f[2];
  dMx[1][2]=aMx[2][1];    dMx[2][1]=f[3];
  dMx[1][3]=aMx[3][1];    dMx[3][1]=f[4];
  dMx[2][3]=aMx[3][2];    dMx[3][2]=f[5];
}
```

Matrix Inverse (mD = mA^{-1})

An inverse matrix A^{-1} (also referred to as a reciprocal matrix) is indicated by the superset $^{-1}$, and it is effectively the swap of all row-column {XYZ} elements referenced by row-column with column-row indexing, and vice versa. Effectively, as the row and column are equivalent for the diagonal, those elements are retained. The bottom row is set to zero, thus no translation, and the fourth column contains the negative sums indicated by the following expressions:

$$i_1 = -((a_{14}\ a_{11}) + (a_{24}\ a_{21}) + (a_{34}\ a_{31}))$$
$$i_2 = -((a_{14}\ a_{12}) + (a_{24}\ a_{22}) + (a_{34}\ a_{32}))$$
$$i_3 = -((a_{14}\ a_{13}) + (a_{24}\ a_{23}) + (a_{34}\ a_{33}))$$

A sometimes useful expression is that the product of a matrix and its inverse is an identity matrix.

$$AA^{-1} = I$$

Rewriting that equation into the form:

$$A^{-1} = \frac{I}{A}$$

It visualizes the reasoning why this is sometimes referred to as a reciprocal matrix. It should be kept in mind that since this is a square matrix A, it has an inverse iff (if and only if) the determinant of $|A| \neq 0$, thus it is considered non-singular (invertible.)

An equation to remember is an inverse transposed matrix is equal to a transposed inverse matrix.

$$(A^T)^{-1} = (A^{-1})^T$$

Pseudo Vec

Listing 12-22: \chap12\matrix3d\Matrix3D.cpp

```
bool vmp_MatrixInv( vmp3DMatrix dMx,
                    const vmp3DMatrix aMx )
{
  vmp3DMatrix s;
  vmp3DQVector t0, t1, t2, *pv;
  float fDet;
  int j;
  bool bRet;

  vmp_MatrixTransposeGeneric( s, aMx );
                              //     A^T        A
  t0.x = s[2][2] * s[3][3];  //  = A2z*A3w   A2z*A3w
  t0.y = s[2][3] * s[3][2];  //  = A2w*A3z   A3z*A2w
  t0.z = s[2][1] * s[3][3];  //  = A2y*A3w   A1z*A3w
```

```
t0.w = s[2][2] * s[3][1];  //  = A2z*A3y   A2z*A1w

t1.x = s[2][3] * s[3][1];  // += A2w*A3y   A3z*A1w
t1.y = s[2][0] * s[3][3];  // += A2x*A3w   AOz*A3w
t1.z = s[2][3] * s[3][0];  // += A2w*A3x   A3z*AOw
t1.w = s[2][0] * s[3][2];  // += A2x*A3z   AOz*Azw

t2.x = s[2][1] * s[3][2];  // += A2y*A3z   A1z*Azw
t2.y = s[2][2] * s[3][0];  // += A2z*A3x   A2z*AOw
t2.z = s[2][0] * s[3][1];  // += A2x*A3y   AOz*A1w
t2.w = s[2][1] * s[3][0];  // += A2y*A3x   A1z*AOw

dMx[0][0]  = t0.x*s[1][1] + t1.x*s[1][2] + t2.x*s[1][3];
dMx[0][0] -= t0.y*s[1][1] + t0.z*s[1][2] + t0.w*s[1][3];

dMx[0][1]  = t0.y*s[1][0] + t1.y*s[1][2] + t2.y*s[1][3];
dMx[0][1] -= t0.x*s[1][0] + t1.z*s[1][2] + t1.w*s[1][3];

dMx[0][2]  = t0.z*s[1][0] + t1.z*s[1][1] + t2.z*s[1][3];
dMx[0][2] -= t1.x*s[1][0] + t1.y*s[1][1] + t2.w*s[1][3];

dMx[0][3]  = t0.w*s[1][0] + t1.w*s[1][1] + t2.w*s[1][2];
dMx[0][3] -= t2.x*s[1][0] + t2.y*s[1][1] + t2.z*s[1][2];

  // calculate {X_ZW} for first two matrix rows

dMx[1][0]  = t0.y*s[0][1] + t0.z*s[0][2] + t0.w*s[0][3];
dMx[1][0] -= t0.x*s[0][1] + t1.x*s[0][2] + t2.x*s[0][3];

dMx[1][1]  = t0.x*s[0][0] + t1.z*s[0][2] + t1.w*s[0][3];
dMx[1][1] -= t0.y*s[0][0] + t1.y*s[0][2] + t2.y*s[0][3];

dMx[1][2]  = t1.x*s[0][0] + t1.y*s[0][1] + t2.w*s[0][3];
dMx[1][2] -= t0.z*s[0][0] + t1.z*s[0][1] + t2.z*s[0][3];

dMx[1][3]  = t2.x*s[0][0] + t2.y*s[0][1] + t2.z*s[0][2];
dMx[1][3] -= t0.w*s[0][0] + t1.w*s[0][1] + t2.w*s[0][2];

  // calculate XY pairs for last two matrix rows

                         //  A^T    A
t0.x = s[0][2]*s[1][3];  // 0=2 7  A2x*A3y
t0.y = s[0][3]*s[1][2];  // 1=3 6  A2x*A2y
t0.z = s[0][1]*s[1][3];  // 2=1 7  A1x*A3y
t1.x = s[0][3]*s[1][1];  // 3=3 5  A3x*A1y
t2.x = s[0][1]*s[1][2];  // 4=1 6  A1x*A2y
t0.w = s[0][2]*s[1][1];  // 5=2 5  A2x*A1y
t1.y = s[0][0]*s[1][3];  // 6=0 7  AOx*A3y
t1.z = s[0][3]*s[1][0];  // 7=3 4  A3x*AOy
t1.w = s[0][0]*s[1][2];  // 8=0 6  AOx*A2y
t2.y = s[0][2]*s[1][0];  // 9=2 4  A2x*AOy
t2.z = s[0][0]*s[1][1];  //10=0 5  AOx*A1y
t2.w = s[0][1]*s[1][0];  //11=1 4  A1x*AOy

  // calculate {XY_W} for last two matrix rows

dMx[2][0]  = t0.x*s[3][1] + t1.x*s[3][2] + t2.x*s[3][3];
```

```
  dMx[2][0] -= t0.y*s[3][1] + t0.z*s[3][2] + t0.w*s[3][3];

  dMx[2][1]  = t0.y*s[3][0] + t1.y*s[3][2] + t2.y*s[3][3];
  dMx[2][1] -= t0.x*s[3][0] + t1.z*s[3][2] + t1.w*s[3][3];

  dMx[2][2]  = t0.z*s[3][0] + t1.z*s[3][1] + t2.z*s[3][3];
  dMx[2][2] -= t1.x*s[3][0] + t1.y*s[3][1] + t2.w*s[3][3];

  dMx[2][3]  = t0.w*s[3][0] + t1.w*s[3][1] + t2.w*s[3][2];
  dMx[2][3] -= t2.x*s[3][0] + t2.y*s[3][1] + t2.z*s[3][2];

    // calculate {XY_W} for first two matrix rows

  dMx[3][0]  = t0.z*s[2][2] + t0.w*s[2][3] + t0.y*s[2][1];
  dMx[3][0] -= t1.x*s[2][2] + t2.x*s[2][3] + t0.x*s[2][1];

  dMx[3][1]  = t1.w*s[2][3] + t0.x*s[2][0] + t1.z*s[2][2];
  dMx[3][1] -= t2.y*s[2][3] + t0.y*s[2][0] + t1.y*s[2][2];

  dMx[3][2]  = t1.y*s[2][1] + t2.w*s[2][3] + t1.x*s[2][0];
  dMx[3][2] -= t1.z*s[2][1] + t2.z*s[2][3] + t0.z*s[2][0];

  dMx[3][3]  = t2.z*s[2][2] + t2.x*s[2][0] + t2.y*s[2][1];
  dMx[3][3] -= t2.w*s[2][2] + t0.w*s[2][0] + t1.w*s[2][1];

    // calculate determinant
  fDet = s[0][0]*dMx[0][0] + s[0][1]*dMx[0][1]
       + s[0][2]*dMx[0][2] + s[0][3]*dMx[0][3];
  if (0.0f == fDet)
  {
    fDet = 1.0f;
    bRet = false;
  }
  else
  {
    fDet = 1.0f / fDet;
    bRet = true;
  }

  pv = (vmp3DQVector *)dMx;
  j = 4;
  do {
    pv->x *= fDet;
    pv->y *= fDet;
    pv->z *= fDet;
    pv->w *= fDet;
    pv++;
  } while (--j);

  return bRet;
}
```

Note that the DirectX version assigns the determinant to the second passed pointer.

```
D3DXMatrixInverse(D3DXMATRIX *, float *, D3DXMATRIX *);
```

Matrix Rotations

Rotations are much more interesting and more difficult to understand. For example, to rotate the cube in the following diagram, you do not actually rotate on the axis you wish to rotate. Do not worry; this is not nearly as complicated as trying to solve a Rubik's Cube. It just makes a cool prop and another neat office toy! This will be demonstrated later. Now we put into practice the sine of the trigonometry that was covered in the last chapter.

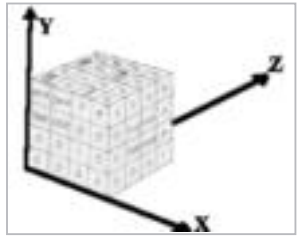


Figure 12-2: Rubik's Cube rotational matrix model

Confusing? It can be! Find yourself a Rubik's Cube. Keep each side as an individual color (do not scramble it). Attach pencils, straws, or swizzle sticks to represent the x, y, and z-axis (this can be done with Legos or Tinker Toys, but the Rubik's Cube is a lot more visual). Now try to rotate the x-axis by only rotating the y- and z-axis. Fun, huh? Ever try to manually control an object in a 3D test jig by controlling the three axes individually?

With matrices, the order of rotations is very important, and there should always be an {XYZ} rotation order. Matrix operations of Euler angles tend to be the primary use of matrices for x-rot, y-rot, and z-rot, but it comes with a price. This will be detailed later in Chapter 13.

Set X Rotation

To rotate on an x-axis, one actually rotates the y and z coordinates, leaving the x coordinate alone. Note the darkened text areas of the following equation; cos and sin are the only differences from an identity matrix. Also note the negative sign on opposite sides of the diagonal, depending on whether the matrix is for a left-handed or right-handed 3D coordinate system.

Equation 12-12: X-axis (left-handed row and right-handed column) rotation matrix. Note that this is nearly identical to an identity matrix, except for the Y and Z row and columns being set to the trigonometric values.

$$
\begin{aligned}
w_x = w_1 = v_1 = \quad & v_1(1) + v_2(0) + v_3(0) + v_4(0) \\
w_y = w_2 = v_2 \cos\theta + v_3 \sin\theta = \quad & v_1(0) + v_2(\cos\theta) + v_3(\sin\theta) + v_4(0) \\
w_z = w_3 = -v_2 \sin\theta + v_3 \cos\theta = \quad & v_1(0) + v_2(-\sin\theta) + v_3(\cos\theta) + v_4(0) \\
w_w = w_4 = v_4 = \quad & v_1(0) + v_2(0) + v_3(0) + v_4(1)
\end{aligned}
$$

Simplified equation:

```
dx =  x;
dy =  y * cos(ang) + z * sin(ang);
dz = -y * sin(ang) + z * cos(ang);
```

Pseudo Vec

To save processing time, the standard C language trigonometric functions sin() and cos() are only used once, and the result of the sin() function is merely negated for the inverse result.

Listing 12-23: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixSetRotateX( vmp3DMatrix Mx, float fRads )
{
  vmp_MatrixSetIdentity( Mx );

  Mx[1][2] =               sinf( fRads );
  Mx[2][2] =  Mx[1][1] =   cosf( fRads );
  Mx[2][1] = -Mx[1][2]; //-sinf
}
```

Pseudo Vec (X86)

The code gets more efficient when the memory write pipelining architecture of the X86 processor is taken advantage of. Also, since this uses the X86 FPU, the combination instruction fsincos (sine and cosine) is used to save processing time. It takes a longer processing time then using just one sine or cosine, but it's faster than actually calling both instructions individually.

vmp_MatrixSetRotateX (X86-FPU Asm)

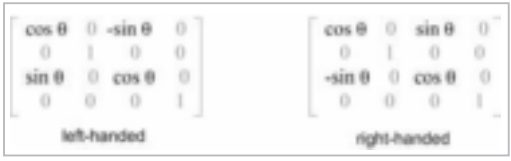
Listing 12-24: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
fld   fRads
OFLOW_FIXUP fRads     ; Overflow fixup (optional)

mov   eax,pMat        ; Matrix
mov   ecx,FLOAT1      ; 1.0f
mov   edx,FLOAT0      ; 0.0f

fsincos                           ;ST(0)=cos ST(1)=sin

mov   (vmp3DMatrix PTR [eax])._m00,ecx   ;1.0
mov   (vmp3DMatrix PTR [eax])._m01,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m02,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m03,edx   ;0.0

fwait
fst   (vmp3DMatrix PTR [eax])._m11  ;ST(0)=cos ST(1)=sin

mov   (vmp3DMatrix PTR [eax])._m10,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m13,edx   ;0.0

fstp  (vmp3DMatrix PTR [eax])._m22  ;     =cos ST(0)=sin

mov   (vmp3DMatrix PTR [eax])._m20,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m23,edx   ;0.0

fst   (vmp3DMatrix PTR [eax])._m12       ;sin

mov   (vmp3DMatrix PTR [eax])._m30,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m31,edx   ;0.0

fchs                                     ;-sin

mov   (vmp3DMatrix PTR [eax])._m32,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m33,ecx   ;1.0
fwait
fstp  (vmp3DMatrix PTR [eax])._m21       ;-sin  ST(0)=sin
```

Set Y Rotation

To rotate on a y-axis, one actually rotates the x and z coordinates, leaving the y alone by not touching the elements of the row and column of y.

Equation 12-13: Y-axis (left-handed row and right-handed column) rotation matrix. Note that this is nearly identical to an identity matrix, except for the X and Z row and columns being set to the trigonometric values.



$$
\begin{aligned}
w_x = w_1 = v_1 \cos\theta + -v_3 \sin\theta = \; & v_1(\cos\theta) + v_2(0) + v_3(-\sin\theta) + v_4(0) \\
w_y = w_2 = v_2 = \; & v_1(0) + v_2(1) + v_3(0) + v_4(0) \\
w_z = w_3 = v_1 \sin\theta + v_3 \cos\theta = \; & v_1(\sin\theta) + v_2(0) + v_3(\cos\theta) + v_4(0) \\
w_w = w_4 = v_4 = \; & v_1(0) + v_2(0) + v_3(0) + v_4(1)
\end{aligned}
$$

Simplified equation:

```
dx = x * cos(ang) + -z * sin(ang);
dy = y;
dz = x * sin(ang) +  z * cos(ang);
```

Pseudo Vec

Listing 12-25: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixSetRotateY( vmp3DMatrix Mx, float fRads )
{
  vmp_MatrixSetIdentity( Mx );

  Mx[2][0] =               sinf( fRads );
  Mx[2][2] =  Mx[0][0] =   cosf( fRads );
  Mx[0][2] = -Mx[2][0]; //-sinf
}
```

Pseudo Vec (X86)

The code gets more efficient when the memory write pipelining architecture of the X86 processor is used to its advantage.

vmp_MatrixSetRotateY (X86-FPU Asm)

Listing 12-26: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
  fld   fRads
  OFLOW_FIXUP fRads     ; Overflow fixup (optional)

  mov   eax,pMat        ; Matrix
  mov   ecx,FLOAT1      ; 1.0f
  mov   edx,FLOAT0      ; 0.0f

  fsincos                           ;ST(0)=cos ST(1)=sin

  mov   (vmp3DMatrix PTR [eax])._m01,edx   ;0.0
  mov   (vmp3DMatrix PTR [eax])._m03,edx   ;0.0
  mov   (vmp3DMatrix PTR [eax])._m10,edx   ;0.0
  mov   (vmp3DMatrix PTR [eax])._m11,ecx   ;1.0

  fwait
  fst   (vmp3DMatrix PTR [eax])._m00  ;ST(0)=cos ST(1)=sin

  mov   (vmp3DMatrix PTR [eax])._m12,edx   ;0.0
  mov   (vmp3DMatrix PTR [eax])._m13,edx   ;0.0

  fstp  (vmp3DMatrix PTR [eax])._m22  ;     =cos ST(0)=sin
```

```
mov   (vmp3DMatrix PTR [eax])._m21,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m23,edx   ;0.0

fst   (vmp3DMatrix PTR [eax])._m20       ;sin

mov   (vmp3DMatrix PTR [eax])._m30,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m31,edx   ;0.0

fchs                                     ;-sin

mov   (vmp3DMatrix PTR [eax])._m32,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m33,ecx   ;1.0

fwait
fstp  (vmp3DMatrix PTR [eax])._m02       ;-sin  ST(0)=sin
```

Set Z Rotation

To rotate on a z-axis, one actually rotates the x and y coordinates, leaving the z coordinate alone by not touching the elements of the row and column of z.

Equation 12-14: Z-axis (left-handed row and right-handed column) rotation matrix. Note that this is nearly identical to an identity matrix, except for the X and Y row and columns being set to the trigonometric values.

$$
\begin{aligned}
w_x = w_1 = v_1 \cos\theta + v_2 \sin\theta &= v_1(\cos\theta) + v_2(\sin\theta) + v_3(0) + v_4(0)\\
w_y = w_2 = -v_1 \sin\theta + v_2 \cos\theta &= v_1(-\sin\theta) + v_2(\cos\theta) + v_3(0) + v_4(0)\\
w_z = w_3 = v_3 = &\quad v_1(0) + v_2(0) + v_3(1) + v_4(0)\\
w_w = w_4 = v_4 = &\quad v_1(0) + v_2(0) + v_3(0) + v_4(1)
\end{aligned}
$$

Simplified equation:

```
dx =  x * cos(ang) + y * sin(ang);
dy = -x * sin(ang) + y * cos(ang);
dz =  z;
```

Pseudo Vec

Listing 12-27: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixSetRotateZ( vmp3DMatrix Mx, float fRads )
{
  vmp_MatrixSetIdentity( Mx );

  Mx[0][1] =               sinf( fRads );
  Mx[1][1] =  Mx[0][0] =   cosf( fRads );
  Mx[1][0] = -Mx[0][1]; //-sinf
}
```

Pseudo Vec (X86)

The code gets more efficient when the memory write pipelining architecture of the X86 processor is used to its advantage.

vmp_MatrixSetRotateZ (X86-FPU Asm)

Listing 12-28: vmp_x86\chap12\matrix3d\Matrix3DX86M.asm

```
fld   fRads
OFLOW_FIXUP fRads     ; Overflow fixup (optional)

mov   eax,pMat        ; Matrix
mov   ecx,FLOAT1      ; 1.0f
mov   edx,FLOAT0      ; 0.0f

fsincos                           ;ST(0)=cos ST(1)=sin

mov   (vmp3DMatrix PTR [eax])._m02,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m03,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m12,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m13,edx   ;0.0

fwait
fst   (vmp3DMatrix PTR [eax])._m00  ;ST(0)=cos ST(1)=sin

mov   (vmp3DMatrix PTR [eax])._m20,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m21,edx   ;0.0

fstp  (vmp3DMatrix PTR [eax])._m11  ;     =cos ST(0)=sin

mov   (vmp3DMatrix PTR [eax])._m22,ecx   ;1.0
mov   (vmp3DMatrix PTR [eax])._m23,edx   ;0.0

fst   (vmp3DMatrix PTR [eax])._m01       ;sin

mov   (vmp3DMatrix PTR [eax])._m30,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m31,edx   ;0.0

fchs                                     ;-sin

mov   (vmp3DMatrix PTR [eax])._m32,edx   ;0.0
mov   (vmp3DMatrix PTR [eax])._m33,ecx   ;1.0

fwait
fstp  (vmp3DMatrix PTR [eax])._m10      ;-sin  ST(0)=sin
```

Matrix to Matrix Rotations

Sometimes it becomes necessary to rotate a matrix on one axis at a time. The following functions are handy for that operation.

Rotate Matrix on X-Axis

Pseudo Vec

Listing 12-29: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixRotateX( vmp3DMatrix dMx,
                  const vmp3DMatrix aMx, float fRads )
{
  vmp3DMatrix mx;

  vmp_MatrixSetRotateX( mx, fRads );
  vmp_MatrixMul( dMx, mx, aMx );
}
```

Rotate Matrix on Y-Axis

Pseudo Vec

Listing 12-30: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixRotateY( vmp3DMatrix dMx,
                  const vmp3DMatrix aMx, float fRads )
{
  vmp3DMatrix mx;

  vmp_MatrixSetRotateY( mx, fRads );
  vmp_MatrixMul( dMx, mx, aMx );
}
```

Rotate Matrix on Z-Axis

Pseudo Vec

Listing 12-31: \chap12\matrix3d\Matrix3D.cpp

```
void vmp_MatrixRotateZ( vmp3DMatrix dMx,
                  const vmp3DMatrix aMx, float fRads )
{
  vmp3DMatrix mx;

  vmp_MatrixSetRotateZ( mx, fRads );
  vmp_MatrixMul( dMx, mx, aMx );
}
```

DirectX Matrix Race

Microsoft has an entire set of three-coordinate {XYZ}, four-coordinate {XYZW}, matrix, and quaternion equations. Each data type function set is not as extensive as what was covered in this book, but as of

DirectX 8.1, the latest and greatest SDK is available for download by the general public at http://msdn.microsoft.com/downloads/.

Look under the category "Graphics and Multimedia — DirectX — DirectX #.# SDK."

In the case of matrices, DirectX has its own data type and function-naming conventions. For example, the following are just a few functions and their Direct3D equivalents:

Functions in This Book	Direct3D Equivalent
vmp_MatrixSetIdentity	D3DXMatrixIdentity
vmp_MatrixSetScaling	D3DXMatrixScaling
vmp_MatrixSetTranslate	D3DXMatrixTranslation
vmp_MatrixSetRotateX	D3DXMatrixRotationX
vmp_MatrixSetRotateY	D3DXMatrixRotationY
vmp_MatrixSetRotateZ	D3DXMatrixRotationZ
vmp_MatrixTranspose	D3DXMatrixTranspose
vmp_MatrixMul	D3DXMatrixMultiply
vmp_MatrixTranslate	D3DXMatrixTranslation

Type casting would be similar to the following to maintain cross-platform compatibility:

```
vmp3DMatrix *pmGWorld;

D3DXMatrixMultiply( (D3DXMATRIX *)pmGWorld,
                    (D3DXMATRIX *)pmGWorld,
                    (D3DXMATRIX *)pmGRotX );
```

With the DirectX SDK, D3DXMATRIXA16 can be utilized instead of D3DXMATRIX, as it uses 16-byte alignment, provided that the Visual C++ compiler is version 7.0 or later or version 6 with a processor pack.

```
#define D3DXMATRIXA16 _ALIGN_16 _D3DXMATRIXA16
```

The big plus is that the Direct3D functions have been optimized for 3DNow! and SSE. Note that 3DNow! Professional uses the MMX register set. There is a benefit and a drawback. The benefit is that the DirectX code was written to handle unaligned memory. The drawback is that the code was written to handle unaligned memory.

Because the SSE code was written for unaligned memory, it is not as fast as it could possibly be. The slower unaligned access *movups* instruction would be used instead of the aligned *movaps* instruction.

The one thing that really bothers me about their manuals is the lack of insistence that a programmer takes to ensure that their data is properly aligned! Admittedly, it does make their library more forgiving.

Hint: Align as much of your data as possible, even if being written for DirectX.

Do not forget the trick I have shown you for aligned vector and matrix stack data.

```
  // enough space + 128-bit alignment padding
float matrixbuf[ 1  *16 +3];
vmp3Dmatrix *pMat;

pMat = (vmp3DMatrix*) ALIGN16((int)matrixbuf);
```

On the companion CD, you will find the matrix race problem. It really is not a race, but a matrix calculation that benchmarks with unaligned and aligned memory for DirectX, generic C (pseudo) code, and this book's unoptimized assembly code. Note that if it was optimized and well tuned, it would be just as fast as (or faster than) the code embedded in the DirectX libraries, but it would be virtually unreadable as the instructions would be shuffled all around for purposes of pipelining, optimization, and throughput, and it would not be visually clear and concise. Optimization issues are a topic for a processor-specific book and not one designed to teach vector processing, such as this one!

vmp_x86\chap12\MatrixRace

Speed is an important point that this book tries to reinforce, but the book was written leaving the issues of optimized assembly mostly up to you. Manufacturers each have their own methods of optimization based upon their unique functionality; thus, you should refer to technical manuals specific to your processor's manufacturer and instruction set.

Exercises

1. Write your own aligned set identity instruction for your favorite processor.
2. Write an unaligned set identity instruction for your favorite processor.
3. Given these two matrices:

$$A=\begin{vmatrix} -8 & -2 & 3 & -3 \\ 6 & 4 & -9 & -2 \\ -3 & -5 & -6 & -5 \\ 5 & -1 & -3 & 4 \end{vmatrix} \qquad B=\begin{vmatrix} 8 & 4 & -7 & -2 \\ 1 & 2 & -3 & -5 \\ -4 & 8 & -6 & 0 \\ -3 & 4 & 9 & 8 \end{vmatrix}$$

   What is the solution to a) A+B? b) AB?
4. How do you convert a left-handed matrix into a right-handed one?

Chapter 13
Quaternion Math

Think of this chapter as a continuation of Chapter 12, "Matrix Math."

CD Workbench Files: /Bench/*architecture*/chap13/*project*/*platform*

	architecture	Matrix	*project*	*platform*
PowerPC	/vmp_ppc/	**Quaternion**	**/quat3d/**	/mac9cw
X86	/vmp_x86/			/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Quaternions

The quaternion? What is that? Oh, this an easy one! It is a four-legged subterranean insect that is found to gnaw on plant roots, pitches woo, and rolls up into a ball when attacked!

Gotcha!

Before we discuss the ins and outs of a quaternion, maybe we should revisit a standard 3D rotation using Euler angles. As you should know, an object would be oriented within 3D space by the specified rotation of its {XYZ} axis rotations. It in essence uses an {XYZ} vector to indicate its position within world coordinate space (translation) and would be oriented (rotated) based upon a set of {XYZ} rotations. Additional information, such as scaling, etc., is utilized as well. Keep in mind that the X, Y, and Z rotations must occur in a precise order every time. But there is a problem, and that is gimbal lock!

This is easier to visualize if we first examine a gimbal, as shown in Figure 13-1. If we remember that Euler angles need to be rotated in a particular order such as {XYZ}, rotating just one axis is not a problem, but when two or three axes are rotated simultaneously, a problem is presented. Keeping in mind that the starting positions of the {XYZ} axis are 90º from each other, such as in a left-handed orientation, first rotate

the x-ring (axis). There is no problem. But then rotating the y-axis, such as the middle ring, by 90º causes a gimbal lock. The same occurs when moving the z-axis. By rotating it 90º as well, all three rings will be locked into the same position. Just two matching rings is an indicator of a gimbal lock!

Figure 13-1: Three-axis gimbal, compliments of Ken Mayfield

Now move one ring, any ring. What are the possible axis angles? Do you see the problem?

If merely using matrices with Euler angles, as discussed in Chapter 12, then the movement of a player needs to be limited. They can be moved (translated) around the playfield with no problem. Rotating them on their y-axis 360° is no problem either. The problem arises when it comes time to tip the character over like a top along its x- or z-axis (crawling, swimming, lying down, running down a steep hill, etc.).

The Euler rotation angles each have an associated {XYZ} coordinate {X=Pitch, Y=Yaw, Z=Roll}. By using a quaternion, this problem is alleviated. Instead of a rotation being stored as separate Euler rotations, it is stored as an {XYZ} vector with a rotation factor.

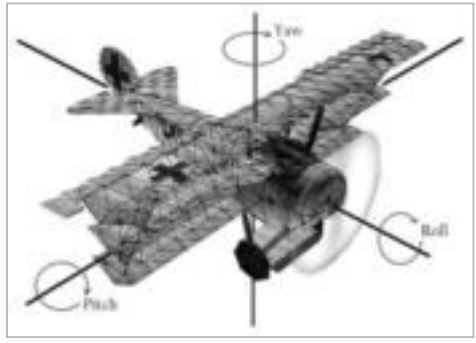


Figure 13-2: Yaw, pitch, roll Euler rotation angles. Fokker Dr I Triplane (Red Baron), compliments of Ken Mayfield

A rotation, however, involves 360°, which presents a problem. Rotational data is stored in radians and a full circle requires 2π to be stored, but a cosine and/or sine can only process up to the equivalent of π (180°), so half the rotation information can be lost. A quaternion resolves this by dividing the angle in radians by two (radians * 0.5). When needed to convert back to a full angle, merely multiply by two! This simple solution helps to condense the data into a form with no data loss.

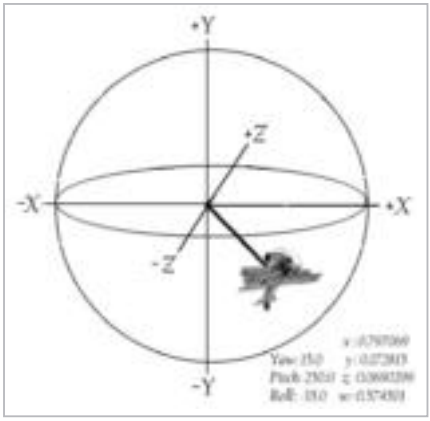


Figure 13-3: Yaw, pitch, roll quaternion rotation. Fokker Dr I Triplane (Red Baron), compliments of Ken Mayfield

A quaternion uses the same storage type as a quad vector, but to prevent accidental mixing of the two, it is better to define its own data type.

Listing 13-1: \inc???\vmp3D.h

```
// Quaternion Vector (single-precision floating-point)

typedef struct              typedef struct
{                           {
  float x;                    float x;
  float y;                    float y;
  float z;                    float z;
  float w;                    float w;
} vmpQuat;                  } vmp3DQVector;
```

A quaternion comes in two forms: a true form, which is the {XYZ} component, and the {W}, which is the rotation.

The second form is a unit vector, which is all four elements with the inclusion of {W}; all are used in the magnitude in calculating a normalized value of one.

An easy way to think of this is that a true quaternion is {XYZ}W, while a unit quaternion is {XYZW}. A unit sphere has a radius of 1.0, a unit triangle has a hypotenuse of 1.0. A unit typically means one!

$$q = w + xi + yj + zk$$

An imaginary is represented by the $i=\sqrt{-1}$.

As such, an identity is defined as {0,0,0,1}. The same rule of identities applies, whereas the product of a quaternion identity is the quaternion.

A quaternion is made up of the imaginaries {*i, j, k*} such that q=w+x***i***+y***j***+z***k***. Two other representations would be q = [*x y z w*] and q = [*w, v*], whereas *w* is a scalar and *v* is the {XYZ} vector. In essence, a quaternion is made up of not one but three imaginaries:

$$ii = -1 \qquad jj = -1 \qquad kk = -1$$

The product of imaginary pairs is similar to that of a cross product of axes in 3D space.

$$i = jk = -kj \qquad j = ki = -ik \qquad k = ij = -ji$$

I find a simple method to remember this is the ordering of imaginary numbers {***i, j, k***}. If it is thought of as a sequence of {***i, j, k, i***,...}, the third imaginary results with a positive sign if the multiplicands are sequential and negative if in reverse order. So ***ij***=***k***, ***jk***=***i***, ***ki***=***j***, but ***kj***=–***i***, ***ji***=–***k***, and ***ik***=–***j***. See, simple! Just remember that the products of imaginaries are not commutative.

Pseudo Vec

There are different implementations of a quaternion library function, depending on left-handed versus right-handed coordinate systems, level of normalization, flavor of the quaternion instantiation, and overall extra complexity needed if branchless code is attempted. The following is an example of such an implementation. DirectX is no exception to this rule, so note the type definition DIRECTX_EQUIV, which indicates a special set of code to match its functionality. DirectX has a set of quaternion functions of its own, but your code may need to mix and match, so keep this in mind.

Some of these functions are exactly the same as those used for quad vectors, such as:

Quaternion	Quad Vector
vmp_QuatCopy()	vmp_QVecCopy()
vmp_QuatAdd()	vmp_QVecAdd
vmp_QuatSub()	vmp_QVecSub
vmp_QuatScale()	vmp_QVecScale

Keep in mind that the calculation of normalized values needs to be followed up by a call to the normalization function to renormalize the result. There are other similar functions to the quad vector functions, except four elements are processed instead of only three.

QuatCopy is identical to a quad vector copy, as those both consist of four packed single-precision floating-point values.

Quaternion Addition

The summation of two quaternions would be as follows:

$$\begin{aligned} q_1 + q_2 &= (w_1 \quad x_1\mathbf{i} \quad y_1\mathbf{j} \quad z_1\mathbf{k}) + (w_2 \quad x_2\mathbf{i} \quad y_2\mathbf{j} \quad z_2\mathbf{k}) \\ &= w_1+w_2 \quad + x_1\mathbf{i} + x_2\mathbf{i} + y_1\mathbf{j} + y_2\mathbf{j} + z_1\mathbf{k} + z_2\mathbf{k} \\ &= (w_1+w_2) + (x_1+x_2)\mathbf{i} + (y_1+y_2)\mathbf{j} + (z_1+z_2)\mathbf{k} \end{aligned}$$

A simpler method to think of this would be as follows:

$$\begin{aligned} q_1 + q_2 &= [w_1 \; v_1] + [w_2 \; v_2] = [w_1+w_2 \quad v_1+v_2] \\ &= [(w_1+w_2) \quad (x_1+x_2 \quad y_1+y_2 \quad z_1+z_2)] \quad = \quad [w_1+w_2 \quad x_1+x_2 \quad y_1+y_2 \quad z_1+z_2] \end{aligned}$$

In its simple elemental form:

$$D_w=A_w+B_w \qquad D_z=A_z+B_z \qquad D_y=A_y+B_y \qquad D_x=A_x+B_x$$

You should hopefully recognize it as the same as a quad vector floating-point addition and the following as a subtraction!

Quaternion Subtraction

The difference of two quaternions would be:

$$q_1 - q_2 = w_1-w_2 + (x_1-x_2)\mathbf{i} + (y_1-y_2)\mathbf{j} + (z_1-z_2)\mathbf{k}$$

In its simple elemental form:

$$D_w=A_w-B_w \qquad D_z=A_z-B_z \qquad D_y=A_y-B_y \qquad D_x=A_x-B_x$$

Now it must be pointed out that there was only so much time to write this book and it is, just as projects are, prone to code bloat. As such, most of the assembly portion of the quaternion section is a rehash of previous chapters, only repackaged in a different way, and so not much assembly code will be presented in this section. Due to constraints on time, there are not many examples on the companion CD, but you have everything you need to build your own. Besides, if I did all the work for you, it would not leave you with much fun programming to do! And reading about something, followed by an implementation of the programming, helps to reinforce what you learned!

Quaternion Dot Product (Inner Product)

$$q = w + xi + yj + zk$$

$$q_1 \bullet q_2 = w_1w_2 + (x_1x_2)\mathbf{i} + (y_1y_2)\mathbf{j} + (z_1z_2)\mathbf{k}$$

$$D = A_wB_w + A_zB_z + A_yB_y + A_xB_x$$

This is very similar to a vector dot product, except it contains the fourth product sum element of {W}.

DirectX: *D3DXQuaternionDot*

Listing 13-2: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatDotProduct(float * const pfD,
     const vmpQuat * const pqA, const vmpQuat * const pqB)
{
  ASSERT_PTR4(pfD);
  ASSERT_PTR4(pqA);
  ASSERT_PTR4(pqB);

  *pfD = (pqA->x * pqB->x + pqA->y * pqB->y
       + pqA->z * pqB->z + pqA->w * pqB->w);
}
```

Pseudo Vec (X86)

```
mov   ebx,vB          ; Vector B
mov   eax,vA          ; Vector A
mov   edx,vD          ; Vector Destination
```

vmp_QuatDotProduct (3DNow!) Aligned

Listing 13-3: vmp_x86\chap13\quat3d\Quat3DX86M.asm

```
movq  mm0,[ebx+8]     ; vB.zw  {Bw Bz}
movq  mm1,[eax+8]     ; vA.zw  {Aw Az}
movq  mm2,[ebx+0]     ; vB.xy  {By Bx}
movq  mm3,[eax+0]     ; vA.xy  {Ay Ax}

pfmul mm0,mm1         ;  vB.z  {BwAw BzAz}
pfmul mm2,mm3         ;        {ByAy BxAx}

pfacc mm0,mm2         ; {ByAy+BxAx BwAw+BzAz}
pfacc mm0,mm0         ; {#  ByAy+BxAx+BwAw+BzAz}

movd  [edx],mm0       ; Save resulting scalar
```

vmp_QuatDotProduct (SSE) Aligned

Listing 13-4: vmp_x86\chap13\quat3d\Quat3DX86M.asm

```
  movaps xmm1,[ebx]           ;vB.xyzw {Bw Bz By Bx}
  movaps xmm0,[eax]           ;vA.xyzw {Aw Az Ay Ax}

  mulps  xmm0,xmm1            ; {AwBw AzBz AyBy AxBx}
  movaps xmm1,xmm0            ; {AwBw AzBz AyBy AxBx}
  movaps xmm2,xmm0            ; {AwBw AzBz AyBy AxBx}

  shufps xmm1,xmm1,01001110b  ; {AyBy AxBx AwBw AzBz}
  addps  xmm1,xmm0
              ; {AyBy+AwBw AxBx+AzBz AwBw+AyBy AzBz+AxBx}
  shufps xmm0,xmm1,11111101b  ; 3 3 3 1
              ; {AyBy+AwBw AxBx+AzBz AzBz+AxBx AwBw+AyBy}

  addss  xmm0,xmm1            ; {# # # AzBz+AxBx+AwBw+AyBy}
  movss  [edx],xmm0           ; Save Scalar
```

Quaternion Magnitude (Length of Vector)

Just like a dot product, this function is similar to a vector magnitude, except it uses the squared sum of the {W} element before the square root.

$$q = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k}$$

$$q_1q_2 = \sqrt{(w_1w_2 + (x_1x_2)\mathbf{i} + (y_1y_2)\mathbf{j} + (z_1z_2)\mathbf{k})}$$
$$q^2 = \sqrt{(w^2 + x^2\mathbf{i} + y^2\mathbf{j} + z^2\mathbf{k})}$$

$$D = \sqrt{(A_wA_w + A_zA_z + A_yA_y + A_xA_x)}$$

DirectX: *D3DXQuaternionLength*

Listing 13-5: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatMagnitude(float * const pfD,
               const vmpQuat * const pqA)
{
  ASSERT_PTR4(pfD);
  ASSERT_PTR4(pqA);

  *pfD = sqrtf(pqA->x * pqA->x + pqA->y * pqA->y
            + pqA->z * pqA->z + pqA->w * pqA->w);
}
```

Pseudo Vec (X86)

```
mov   eax,vA          ; Vector A
mov   edx,vD          ; Vector Destination
```

vmp_QuatMagnitude (3DNow!) Fast Aligned

Note that the following code is fast, as it uses an estimated reciprocal square root. For other "fast" estimated code, review reciprocals in Chapter 9, "Vector Multiplication and Division," and reciprocal square roots in Chapter 10, "Special Functions."

The same principals of coding to support estimated calculations for speed are usable here. This is definitely something you do not learn in other books — quaternions in assembly code, and fast versions as well! Note, however, that as usual, they are not optimized to keep them as readable as possible to make them easier for you to understand.

Listing 13-6: vmp_x86\chap13\quat3d\Quat3DX86M.asm

```
  ; Calculate XYZW sum of squares
movq  mm0,[eax+0]      ;vA.xy   {Ay   Ax}
movq  mm1,[eax+8]      ;vA.zw   {Aw   Az}

pfmul mm0,mm0          ;       {AyAy AxAx}
pfmul mm1,mm1          ;       {AwAw AzAz}

pfacc mm0,mm1          ;    {AwAw+AzAz AyAy+AxAx}
pfacc mm0,mm0          ; {#  AwAw+AzAz+AyAy+AxAx}

  ; mm0=AwAw+AzAz+AyAy+AxAx
; Calculate square root upon sum of squares
pfrsqrt mm1,mm0       ;X0=f(x)=1/√x  {1/√r 1/√r}

  ; Insert Newton-Raphson precision code here!

pfmul   mm0,mm1       ;              {r/√r r/√r}
                      ; r = r/√r
movd  [edx],mm0       ; Save resulting distance float
```

vmp_QuatMagnitude (3DNow!) (Insertion Code)

Listing 13-7: vmp_x86\chap13\quat3d\Quat3DX86M.asm

```
; refine Newton-Raphson reciprocal square approximation
; Calculate 1/√ accurate to 24 bits
movq     mm2,mm1     ;   {#####  1/√(AzAz+AyAy+AxAx)}
pfmul    mm1,mm1     ;X1=f(x0,x0) {# 1/ (AzAz+AyAy+AxAx)}
pfrsqit1 mm1,mm0     ;X2=f(x,x1)     {1st step}
 ; Calculate sqrt() = 1/(1/√)  24 bit
pfrcpit2 mm1,mm2     ; y=f(x2,x0)    {2nd step}
```

vmp_QuatMagnitude (SSE) Aligned

Listing 13-8: vmp_x86\chap13\quat3d\Quat3DX86M.asm

```
  ; Calculate sum of squares
  movaps  xmm0,[eax]              ; vA.xyzw {Aw Az Ay Ax}
  mulps   xmm0,xmm0               ; {AwAw AzAz AyAy AxAx}
  movaps  xmm1,xmm0               ; {AwAw AzAz AyAy AxAx}
  shufps  xmm0,xmm0,01001110b     ; {AyAy AxAx AwAw AzAz}
  addps   xmm0,xmm1
               ; {AyAy+AwAw AxAx+AzAz AwAw+AyAy AzAz+AxAx}
  movaps  xmm1,xmm0
  shufps  xmm0,xmm1,10110001b ; 2 3 0 1
               ; {AxAx+AzAz AyAy+AwAw AzAz+AxAx AwAw+AyAy}

  addss   xmm0,xmm1          ; { # # # AzAz+AxAx+AwAw+AyAy}
```

At this point things get really different between normal precision...

```
  ;  Calculate square root upon sum of squares  xmm0
  sqrtss  xmm0,xmm0               ; SqRoot
  movss   [edx],xmm0              ; Save
```

...and faster low precision!

```
  ;  Calculate square root upon sum of squares
  movaps   xmm2,xmm0
  movaps   xmm3,xmm0
  movaps   xmm1,xmm0

  cmpss    xmm2,vTiny,5  ; >=  1's = #'s Okay
  cmpss    xmm3,vTiny,1  ; <   1's = too close to zero

  rsqrtss  xmm0,xmm0     ; vD = (1/√vA)

 ; Correct for infinity due to 1/0 =  1/sqrt(0)
  andps    xmm3,vOne   ; Preserve 1.0's (in pos. that were ∞ !)
  andps    xmm0,xmm2   ; Preserve #'s that are not too small
  orps     xmm0,xmm3   ; Blended #'s and 1.0's
  mulss    xmm0,xmm1   ; {  ... Ax*1/sqrt(Ax)}
  movss    [edx],xmm0  ; Save
```

But first see how to normalize a quaternion.

Quaternion Normalization

Note that most of the quaternion functions are dependent upon the normalization of a number. That is the dividing of each element by the magnitude of the quaternion. There is only one little condition to watch out for and handle — the divide by zero! But remember the calculus that was discussed in reciprocals, where as x approaches zero the result goes infinite, so the effect on the original value is negligible and the solution is the original value.

The same principals of assembly code are applied here, so only the C code is shown!

$$q = w + xi + yj + zk$$

$$r = \sqrt{(w^2 + x^2i + y^2j + z^2k)}$$

$$norm(q) = \frac{w}{r} + \frac{xi}{r} + \frac{yj}{r} + \frac{zk}{r}$$

$$norm(q) = q / r$$

Okay, here is where I tend to get a little confused! If you examine quaternion code by others, they typically have:

$$r=\sqrt{(x^2 + y^2 + z^2 + w^2)}$$

$$D_x=x/r \quad D_y=y/r \quad D_z=z/r \quad D_w=w/r$$

But wait! What happens if q={0,0,0,0}? Will divide by zero not exist? That cannot possibly happen because a quaternion of no length {0,0,0} will have a rotation of one — the quaternion identity {0,0,0,1}. Well, let's assume that a normalized quaternion is subtracted from itself; that would leave a quaternion of {0,0,0,0}. Using those other algorithms, that would lead to a problem of the divide by zero, as well as not being normalized. The solution is that if the value is too close to zero, then assigning a one to the rotation would make no difference and resolve the normalization issue. One other item is that if the value is very close to being finite and negative, the value would be off a bit. So by setting the {W} element to one but with the same sign as the original {W}, the value will be properly normalized.

DirectX: *D3DXQuaternionNormalize*

Listing 13-9: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatNormalize(vmpQuat * const pqD,
                       const vmpQuat * const pqA)
{
  float fLen;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);

  fLen = vmp_QuatDotProduct( pqA, pqA );   // Sum of squares

  if (SINGLE_PRECISION <= fLen)  // Not too close to zero?
  {
    fLen = 1.0f / sqrtf(fLen);   // Magnitude
    pqD->x = pqA->x * fLen;
    pqD->y = pqA->y * fLen;
    pqD->z = pqA->z * fLen;
```

```
    pqD->w = pqA->w * fLen;
  }
  else   // TOO Close to zero (infinity) so minimal effect!
  {
    pqD->x = pqA->x;
    pqD->y = pqA->y;
    pqD->z = pqA->z;
    pqD->w = 1.0f;           // Force to 1 for normalization

#if 01  // Clone sign bit!
    *(uint32*)&(pqD->w) |=
                          0x80000000 & *(uint32 *)&(pqA->w);
#else
    if (0.0f > pqA->w) pqD->w = -1.0f;  // Need ± 1
#endif

  }
}
```

Pseudo Vec

IsQuatEqual (Is Quaternion Equal)

This function compares two quaternions. There is another little tidbit about a quaternion that you should know: A quaternion can have two different {XYZ} vector components to represent the same value. If you recall your geometry, a vector points in one direction based upon its {XYZ} values, but a two's complement (negative) of each of those values indicates the direction from where the vector came. So the vector {X,Y,Z} pointing in an equal but opposite direction would be {–X,–Y,–Z}. Here comes the zinger: The positive value of {W} is related to a clockwise rotation of the axis represented by the vector. A negative value actually indicates the vector is in reverse.

The quaternion {X,Y,Z,θ} is equivalent to {–X,–Y,–Z,–θ}.

It is also a complement of {X,Y,Z,–θ} and {–X,–Y,–Z,θ}, as they point in opposite directions!

Listing 13-10: \chap13\quat3d\Quat3D.cpp

```
bool vmp_IsQuatEqual( const vmpQuat * const pqA,
                  const vmpQuat * const pqB, float fPrec )
{
  ASSERT_PTR4(pqA);
  ASSERT_PTR4(pqB);

  if (   !vmp_IsFEqual(pqA->x, pqB->x, fPrec)
      || !vmp_IsFEqual(pqA->y, pqB->y, fPrec)
      || !vmp_IsFEqual(pqA->z, pqB->z, fPrec)
      || !vmp_IsFEqual(pqA->w, pqB->w, fPrec))
  {
```

```
    // No match (maybe vector XYZ pointed in opposite direction
    // with opposite rotation W?).

    if (   !vmp_IsFEqual(-pqA->x, pqB->x, fPrec)
        || !vmp_IsFEqual(-pqA->y, pqB->y, fPrec)
        || !vmp_IsFEqual(-pqA->z, pqB->z, fPrec)
        || !vmp_IsFEqual(-pqA->w, pqB->w, fPrec))
    {
      return false;
    }
  }

  return true;
}
```

Quaternion Conjugate ($D=\overline{A}$)

A conjugate is merely the inverse of a vector. As such, the rotation is unaffected. The inverse of a vector can result from negating the sign of the rotation {W}, but it is preferred to keep the rotation in a positive form. So instead, only each axis of the vector is inverted. It is assumed that the rotation is already positive.

$$q = w + xi + yj + zk$$

$$\bar{q} = w - xi - yj - zk$$

$$D_w=A_w \qquad D_z=-A_z \qquad D_y=-A_y \qquad D_x=-A_x$$

DirectX: *D3DXQuaternionConjugate*

Listing 13-11: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatConjugate(vmpQuat * const pqD,
                       const vmpQuat * const pqA)
{
  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);

  pqD->x = -pqA->x;
  pqD->y = -pqA->y;
  pqD->z = -pqA->z;
  pqD->w =  pqA->w;
}
```

Pseudo Vector (X86)

This is very simple and mentioned here as a reminder to change the sign of a floating-point number to merely exclusive OR the sign bit.

```
          ;XYZW {-          -          -          +       }
conjugate DWORD 080000000h,080000000h,080000000h,000000000h

  movaps  xmm0,[eax]             ;vA.xyzw {Aw  Az  Ay  Ax}
  xorps   xmm0,oword ptr conjugate
  movaps  [edx],xmm0             ;        {Aw -Az -Ay -Ax}
```

Quaternion Inverse ($D=A^{-1}$)

$$q = w + xi + yj + zk$$

$$q^{-1} = \frac{\bar{q}}{norm(q)} = \frac{w - xi - yj - zk}{w^2 + x^2 + y^2 + z^2}$$

DirectX: *D3DXQuaternionInverse*

Listing 13-12: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatInv(vmpQuat * const pqD,
           const vmpQuat * const pqA)
{
  float fLen;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);

  fLen = vmp_QuatDotProduct( pqA, pqA );   // Sum of squares

  if (0.0f != fLen)
  {
    fLen = 1.0f / fLen;
    pqD->x = -pqA->x * fLen;
    pqD->y = -pqA->y * fLen;
    pqD->z = -pqA->z * fLen;
    pqD->w =  pqA->w * fLen;
  }
  else  // Divide by zero is too close to infinity so minimal effect!
  {
    pqD->x = -pqA->x;
    pqD->y = -pqA->y;
    pqD->z = -pqA->z;

#ifndef DIRECTX_EQUIV      // If NOT DirectX
    pqD->w =  pqA->w;
#else                      // DIRECTX equivalent
    pqD->w = (0.0f > pqA->w) -pqD->w : fLen;
#endif
  }
}
```

Quaternion Multiplication (D=AB)

Please note the algebraic commutative laws do not apply here!

AB BA AB != BA

$$q_1q_2 = (w_1\ \ x_1\mathbf{i}\ \ y_1\mathbf{j}\ \ z_1\mathbf{k})\ (w_2\ \ x_2\mathbf{i}\ \ y_2\mathbf{j}\ \ z_2\mathbf{k})$$

$$\begin{aligned}
&= w_1w_2 + w_1x_2\mathbf{i} + w_1y_2\mathbf{j} + w_1z_2\mathbf{k} \\
&+ x_1w_2\mathbf{i} + x_1x_2\mathbf{i}^2 + x_1y_2\mathbf{ij} + x_1z_2\mathbf{ik} \\
&+ y_1w_2\mathbf{j} + y_1x_2\mathbf{ji} + y_1y_2\mathbf{j}^2 + y_1z_2\mathbf{jk} \\
&+ z_1w_2\mathbf{k} + z_1x_2\mathbf{ki} + z_1y_2\mathbf{kj} + z_1z_2\mathbf{k}^2
\end{aligned}$$

Remember that the square of an imaginary is –1, and the product of two different imaginaries is a positive third imaginary if the two products are sequential; otherwise, negative: ***ij=k***, ***ji=–k***, ***jk=i***, ***kj=–i***, ***ki=j***, ***ik=–j***.

$$\begin{aligned}
&= w_1w_2 + w_1x_2\mathbf{i} + w_1y_2\mathbf{j} + w_1z_2\mathbf{k} \\
&+ x_1w_2\mathbf{i} - x_1x_2 + x_1y_2\mathbf{k} - x_1z_2\mathbf{j} \\
&+ y_1w_2\mathbf{j} - y_1x_2\mathbf{k} - y_1y_2 + y_1z_2\mathbf{i} \\
&+ z_1w_2\mathbf{k} + z_1x_2\mathbf{j} - z_1y_2\mathbf{i} - z_1z_2
\end{aligned}$$

...and regrouping into like complex terms:

$$\begin{aligned}
&= w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 \\
&+ w_1x_2\mathbf{i} + x_1w_2\mathbf{i} - z_1y_2\mathbf{i} + y_1z_2\mathbf{i} \\
&+ w_1y_2\mathbf{j} + y_1w_2\mathbf{j} + z_1x_2\mathbf{j} - x_1z_2\mathbf{j} \\
&+ w_1z_2\mathbf{k} + z_1w_2\mathbf{k} + x_1y_2\mathbf{k} - y_1x_2\mathbf{k}
\end{aligned}$$

$$\begin{aligned}
&= w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 \\
&+ (w_1x_2 + x_1w_2 - z_1y_2 + y_1z_2)\mathbf{i} \\
&+ (w_1y_2 + y_1w_2 + z_1x_2 - x_1z_2)\mathbf{j} \\
&+ (w_1z_2 + z_1w_2 + x_1y_2 - y_1x_2)\mathbf{k}
\end{aligned}$$

Note that in DirectX, D=BA. Thus, it expects a B, A order!

DirectX: *D3DXQuaternionMultiply*

Listing 13-13: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatMul(vmpQuat * const pqD,
           const vmpQuat * const pqA,
           const vmpQuat * const pqB)
{
  vmpQuat quat;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);
  ASSERT_PTR4(pqB);

  quat.x = pqA->x * pqB->w + pqA->y * pqB->z
         - pqA->z * pqB->y + pqA->w * pqB->x;
  quat.y = pqA->y * pqB->w - pqA->x * pqB->z
         + pqA->w * pqB->y + pqA->z * pqB->x;
  quat.z = pqA->z * pqB->w + pqA->w * pqB->z
         + pqA->x * pqB->y - pqA->y * pqB->x;
  pqD->w = pqA->w * pqB->w - pqA->z * pqB->z
         - pqA->y * pqB->y - pqA->x * pqB->x;
```

```
  pqD->x = quat.x;
  pqD->y = quat.y;
  pqD->z = quat.z;
}
```

Pseudo Vec (X86)

```
vQQZ dword 000000000h,080000000h,000000000h,080000000h
vQQY dword 080000000h,000000000h,000000000h,080000000h
vQQX dword 000000000h,000000000h,080000000h,080000000h

  mov   ecx,vB           ; Vector B
  mov   eax,vA           ; Vector A
  mov   edx,vD           ; Vector Destination
```

vmp_QuatMult (SSE) Aligned

The principles of vector-to-vector multiplication are similar to those of a quaternion-to-quaternion multiplication, except there is {XYZW} swizzling and some mixed addition and subtraction of vector elements. Using the exclusive OR to flip the appropriate signs followed by a vector summation allows the result to be calculated.

Listing 13-14: vmp_x86\chap13\quat3d\Quat3DX86M.asm

```
  movaps  xmm0,[eax]            ; {Aw Az Ay Ax}
  movaps  xmm4,[ecx]            ; {Bw Bz By Bx}
  movaps  xmm1,xmm0
  movaps  xmm2,xmm0
  movaps  xmm3,xmm0

  shufps  xmm1,xmm1,10110001b   ; {Az Aw Ax Ay}
  shufps  xmm2,xmm2,01001110b   ; {Ay Ax Aw Az}
  shufps  xmm3,xmm3,00011011b   ; {Ax Ay Az Aw}

  movaps  xmm5,xmm4             ; {Bw Bz By Bx}
  movaps  xmm6,xmm4             ; {Bw Bz By Bx}
  movaps  xmm7,xmm4             ; {Bw Bz By Bx}

  shufps  xmm4,xmm4,00000000b   ; {Bx Bx Bx Bx}
  shufps  xmm5,xmm5,11111111b   ; {Bw Bw Bw Bw}
  shufps  xmm6,xmm6,10101010b   ; {Bz Bz Bz Bz}
  shufps  xmm7,xmm7,01010101b   ; {By By By By}

  mulps   xmm0,xmm5         ; {AwBw AzBw AyBw AxBw}
  mulps   xmm1,xmm6         ; {AzBz AwBz AxBz AyBz} - + - +
  mulps   xmm2,xmm7         ; {AyBy AxBy AwBy AzBy} - + + -
  mulps   xmm3,xmm4         ; {AxBx AyBx AzBx AwBx} - - + +

  xorps   xmm1,oword ptr vQQZ ; {-AzBz +AwBz -AxBz +AyBz}
  xorps   xmm2,oword ptr vQQY ; {-AyBy +AxBy +AwBy -AzBy}
  xorps   xmm3,oword ptr vQQX ; {-AxBx -AyBx +AzBx +AwBx}

  addps   xmm0,xmm1
```

```
  addps   xmm2,xmm3
  addps   xmm0,xmm2

  movaps  [edx],xmm0       ; Save quat
```

Convert a Normalized Axis and Radian Angle to Quaternions

Note that the vector representing the axis must be normalized.

DirectX: *D3DXQuaternionRotationAxis*

Listing 13-15: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatRotAxis( vmpQuat * const pqD,
            const vmp3DVector * const pvA, float fAngle )
{
  float f;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pvA);

#ifndef DIRECTX_EQUIV      // If NOT DirectX
    // Cos is 180 {-PI...0...PI}. Divide by two.
  pqD->w = 0.5f * cosf(fAngle);

  // Quad normalization (sum of squares)
  fLen = vmp_QuatDotProduct( pqA, pqA );   // Sum of squares

  // Handle a normalized vector with a magnitude of zero.
// Too close to zero, thus infinity! The effect is negligible upon the vector!
  f = (SINGLE_PRECISION <= fLen) ? 1.0f/sqrtf(fLen) : 1.0f;
#else                      // DIRECTX equivalent
   // Cos is 180 degrees {-PI...0...PI}. Divide by 2.
  vmp_FSinCos(&f, &(pvD->w), fAngle * 0.5f);
#endif

       // pvA is normalized
  pqD->x = f * pvA->x;
  pqD->y = f * pvA->y;
  pqD->z = f * pvA->z;
}
```

Convert a (Unit) Quaternion to a Normalized Axis

DirectX: *D3DXQuaternionToAxisAngle*

Listing 13-16: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatToRotAxis( vmp3DVector * const pvD,
        float * const pfAngle, const vmpQuat * const pqA )
{
  float fLen, f;

  ASSERT_PTR4(pvD);
  ASSERT_PTR4(pqA);
  ASSERT_PTR4(pfAngle);

  // Correct back to true 360 degree angle in radians
  *pfAngle = 2.0f * acosf(pqA->w);         // radian/2

  // Quad normalization (sum of squares) (3 or 4) element!
#ifndef DIRECTX_EQUIV      // If NOT DirectX
  fLen = (pqA->x*pqA->x + pqA->y*pqA->y + pqA->z*pqA->z);
#else                      // DIRECTX equivalent
  fLen = vmp_QuatDotProduct( pqA, pqA );   // Sum of squares
#endif
                                           // Not too close to zero?
  f = (SINGLE_PRECISION <= fLen) ? 1.0f/sqrtf(fLen) : 1.0f;

  pvD->x = pqA->x * f;     // Quaternions back to radians
  pvD->y = pqA->y * f;
  pvD->z = pqA->z * f;
}
```

Quaternion Rotation from Euler (Yaw Pitch Roll) Angles

Yaw: y-axis, Pitch: x-axis, Roll: z-axis

DirectX: *D3DXQuaternionRotationYawPitchRoll*

Listing 13-17: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatRotYawPitchRoll( vmpQuat * const pqD,
                    float fYaw, float fPitch, float fRoll )
{
  float fYawSin, fPitchSin, fRollSin;
  float fYawCos, fPitchCos, fRollCos;
  float fPYCos, fPYSin, fPCYS, fPSYC;

  vmp_FSinCos( &fYawSin, &fYawCos, fYaw*0.5f );
  vmp_FSinCos( &fPitchSin, &fPitchCos, fPitch*0.5f );
  vmp_FSinCos( &fRollSin,  &fRollCos,  fRoll*0.5f );
```

```
  fPYCos = fPitchCos * fYawCos;
  fPYSin = fPitchSin * fYawSin;
  fPCYS =  fPitchCos * fYawSin;
  fPSYC =  fPitchSin * fYawCos;

  pqD->x = (fRollCos * fPSYC)  + (fRollSin * fPCYS);
  pqD->w = (fRollCos * fPYCos) + (fRollSin * fPYSin);

#ifndef DIRECTX_EQUIV      // If NOT DirectX
  pqD->y = (fRollSin * fPYCos) - (fRollCos * fPYSin);
  pqD->z = (fRollCos * fPCYS)  - (fRollSin * fPSYC);
#else                      // DIRECTX equivalent
  pqD->y = (fRollCos * fPCYS)  - (fRollSin * fPSYC);
  pqD->z = (fRollSin * fPYCos) - (fRollCos * fPYSin);
#endif
}
```

Quaternion Square

Listing 13-18: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatSquare(vmpQuat * const pqD,
              const vmpQuat * const pqA)
{
  float s;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);

  s = 2.0f * pqA->w;
  pqD->w = vmp_QuatDotProduct( pqA, pqA );
  pqD->x = s * pqA->x;
  pqD->y = s * pqA->y;
  pqD->z = s * pqA->z;
}
```

Quaternion Division

A quaternion division is the result of the product of a quaternion with the inverse of a quaternion.

$$q_1 = (w_1 \quad x_1\mathbf{i} \quad y_1\mathbf{j} \quad z_1\mathbf{k}) \qquad q_2 = (w_2 \quad x_2\mathbf{i} \quad y_2\mathbf{j} \quad z_2\mathbf{k})$$

$$\frac{q_1}{q_2} = q_2^{-1}\, q_1 = \frac{w_2 - x_2\mathbf{i} - y_2\mathbf{j} - z_2\mathbf{k}}{w_2^2 + x_2^2 + y_2^2 + z_2^2}\,(w_1 + x_1\mathbf{i} + y_1\mathbf{j} + z_1\mathbf{k})$$

Please note a few steps were skipped here to arrive at the following solution!

$$= \frac{w_2w_1 + x_2w_1 + y_2y_1 + z_2z_1}{w_2^2 + x_2^2 + y_2^2 + z_2^2} + \frac{(w_2x_1 - x_2w_1 - y_2z_1 + z_2y_1)\boldsymbol{i}}{w_2^2 + x_2^2 + y_2^2 + z_2^2}$$

$$+ \frac{w_2y_1 + x_2z_1 - y_2w_1 - z_2x_1)\boldsymbol{j}}{w_2^2 + x_2^2 + y_2^2 + z_2^2} + \frac{(w_2z_1 - x_2y_1 + y_2x_1 - z_2w_1)\boldsymbol{k}}{w_2^2 + x_2^2 + y_2^2 + z_2^2}$$

Listing 13-19: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatDiv(vmpQuat * const pqD,
           const vmpQuat * const pqA,
           const vmpQuat * const pqB)
{
  vmpQuat qTmp, qB;
  float s;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);
  ASSERT_PTR4(pqB);

  qB.x = -pqB->x;                        // conjugate
  qB.y = -pqB->y;
  qB.z = -pqB->z;
  qB.w =  pqB->w;

  vmp_QuatMul(pqD, pqA, &qTmp);
  vmp_QuatMul(&qB, &qB, &qB);          // square elements
                                       // Not too close to zero?
  s = (0.0f != qB.w) ? 1.0f / qB.w : 1.0f;

  pqD->x = qTmp.x * s;
  pqD->y = qTmp.y * s;
  pqD->z = qTmp.z * s;
  pqD->w = qTmp.w * s;
}
```

Quaternion Square Root

Listing 13-20: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatSqrt(vmpQuat * const pqD,
            const vmpQuat * const pqA)
{
  vmpQuat qT;
  float fx, fy, fMag, fLen;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);
```

```
  fLen = sqrtf(pqA->x * pqA->x + pqA->y * pqA->y
            + pqA->w * pqA->w);

  fLen = (0.0f != fLen) ? 1.0f / fLen : 1.0f;
  qT.x = pqA->x * fLen;
  qT.y = pqA->z * fLen;
  qT.z = pqA->z;
  qT.w = pqA->w * fLen;

  fMag = 1.0f / sqrtf( qT.w * qT.w + qT.x * qT.x );
  fx   =        sqrtf( (1.0f – qT.y) * 0.5f );
  fy   =        sqrtf( (1.0f + qT.y) * 0.5f );
  fLen =        sqrtf( fLen );

  pqD->x = qT.x * fLen * fx * fMag;
  pqD->y =        fLen * fy;
  pqD->z = qT.z;
  pqD->w = qT.w * fLen * fy * fMag;
}
```

(Pure) Quaternion Exponent

DirectX: *D3DXQuaternionExp* (requires normalized quaternion)

The following function is a pure quaternion to unit quaternion conversion. Pure: *w* is ignored in calculations.

Listing 13-21: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatExp(vmpQuat * const pqD,
           const vmpQuat * const pqA)
{
  float fMag, fSin;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);

  fMag = pqA->x*pqA->x + pqA->y*pqA->y + pqA->z*pqA->z;

  if (SINGLE_PRECISION <= fMag)       // Not too close to zero?
  {
    fMag = sqrt(fMag);
    vmp_FSinCos(&fSin, &(pqD->w), fMag);

    fMag = fSin / fMag;
    pqD->x = pqA->x * fMag;
    pqD->y = pqA->y * fMag;
    pqD->z = pqA->z * fMag;
  }
  else
  {
    pqD->w = 1.0f;                        // 1.0 = cos(0)
    pqD->x = pqD->y = pqD->z = 0.0f;
  }
}
```

(Unit) Quaternion Natural Log

This function calculates the natural log of a unit quaternion.

DirectX: *D3DXQuaternionLn*

Listing 13-22: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatLn(vmpQuat * const pqD,
          const vmpQuat * const pqA)
{
  float fMag, fSin;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);

  fMag = acosf( pqA->w );
  fSin = sinf( fMag );

  pqD->w = 0.0f;

  if (0.0f != fSin)
  {
    fSin = fMag / fSin;

    pqD->x = pqA->x * fSin;
    pqD->y = pqA->y * fSin;
    pqD->z = pqA->z * fSin;
  }
  else
  {
    pqD->x = pqD->y = pqD->z = 0.0f;
  }
}
```

Normalized Quaternion to Rotation Matrix

This function converts a normalized quaternion into a rotation matrix.

DirectX: *D3DXMatrixRotationQuaternion*

Listing 13-23: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatToMatrix(vmp3DMatrix dMx,
                const vmpQuat * const pqA)
{
  float fMag;
  float fxx, fyy, fzz, fwx, fwy, fwz, fxy, fxz, fyz;

  ASSERT_PTR4(dMx);
  ASSERT_PTR4(pqA);
```

```
  fxx = pqA->x * pqA->x;   // xx
  fyy = pqA->y * pqA->y;   // yy
  fzz = pqA->z * pqA->z;   // zz

  fMag = fxx + fyy + fzz + (pqA->w * pqA->w);
  if (0.0f < fMag)
  {
    fMag = 2.0f / fMag;

    fxx *= fMag;
    fyy *= fMag;
    fzz *= fMag;
  }

  fwx = pqA->w * pqA->x * fMag;   // wx
  fwy = pqA->w * pqA->y * fMag;   // wy
  fwz = pqA->w * pqA->z * fMag;   // wz

  fxy = pqA->x * pqA->y * fMag;   // xy
  fxz = pqA->x * pqA->z * fMag;   // xz
  fyz = pqA->y * pqA->z * fMag;   // yz

  dMx[0][0] = 1.0f - (fyy + fzz);
  dMx[0][1] =        (fxy + fwz);
  dMx[0][2] =        (fxz - fwy);
  dMx[0][3] = 0.0f;

  dMx[1][0] =        (fxy - fwz);
  dMx[1][1] = 1.0f - (fxx + fzz);
  dMx[1][2] =        (fyz + fwx);
  dMx[1][3] = 0.0f;

  dMx[2][0] =        (fxz + fwy);
  dMx[2][1] =        (fyz - fwx);
  dMx[2][2] = 1.0f - (fxx + fyy);
  dMx[2][3] = 0.0f;

  dMx[3][0] = 0.0f;
  dMx[3][1] = 0.0f;
  dMx[3][2] = 0.0f;
  dMx[3][3] = 1.0f;
}
```

Rotation Matrix to Quaternion

This function is used to convert a quaternion matrix back into a quaternion.

DirectX: *D3DXQuaternionRotationMatrix*

Listing 13-24: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatMatrix(vmpQuat * const pqD,
              const vmp3DMatrix aMx )
{
  float fSum, *pf, s;
```

```
int  i,j,k;

ASSERT_PTR4(aMx);
ASSERT_PTR4(pqD);

fSum = aMx[0][0] + aMx[1][1] + aMx[2][2] + aMx[3][3];

// check the diagonal

if (fSum >= 1.0f)        // Positive diagonal
{
  s = sqrtf(fSum) * 0.5f;
  pqD->w = s;
  pqD->x = (aMx[1][2] - aMx[2][1]) * s;
  pqD->y = (aMx[2][0] - aMx[0][2]) * s;
  pqD->z = (aMx[0][1] - aMx[1][0]) * s;
}
else                     // Negative diagonal
{                        // Order the diagonal elements
  pf = (float *)pqD;     // Float elements base {XYZ}W

  if (aMx[0][0] > aMx[1][1])
  {
    k = 0x9;     // 1001 00b  2 1 0   k j i
    i = 0;
  }
  else
  {
    k = 0x2;     // 0010 01b  0 2 1   k j i
    i = 1;
  }

  if (aMx[2][2] > aMx[i][i])
  {
    k = 0x4;     // 0100 10b  1 0 2   k j i
    i = 2;
  }

// {i,j,k} (n % 3) thus ; {0,1,2} {1,2,0} {2,0,1}
  j = k & 3;
  k = (k >> 2);

  // 'i' is index to biggest element

  s = 2.0f * sqrtf(aMx[i][i]
                - aMx[j][j] - aMx[k][k] + 1.0f);
  *(pf+i) = s * 0.5f;

  if (0.0f != s)
  {
    s = 1.0f / (2.0f * s);
  }
  else
  {
    s = 1.0f;
  }

  *(pf+j) = (aMx[i][j] + aMx[j][i]) * s;
```

```
    *(pf+k) = (aMx[i][k] + aMx[k][i]) * s;
    pqD->w =  (aMx[j][k] – aMx[k][j]) * s;
  }
}
```

Slerp (Spherical Linear Interpolation)

Interpolate between two quaternions using spherical linear interpolation from *pqA* to *pqB* distance *fDist*. When working with movement between two rotation angles, such as connecting bones in animations, spline type movement, etc., a method is needed to minutely change from one angle to another with a stepping distance, and this function is it!

DirectX: *D3DXQuaternionSlerp*

Listing 13-25: \chap13\quat3d\Quat3D.cpp

```
void vmp_QuatSlerp(vmpQuat * const pqD,
                   const vmpQuat * const pqA,
                   const vmpQuat * const pqB, float fDist)
{
  float fCos, fSin, fAngle, fSO, fS1;
  vmpQuat vT;

  ASSERT_PTR4(pqD);
  ASSERT_PTR4(pqA);
  ASSERT_PTR4(pqB);

  // Calculate dot product (cosine of angle between quaternions)!

  fCos = vmp_QuatDotProduct( pqA, pqB );
           // Replicated inversion
  if (0.0f > fCos)         // Quaternions angle Obtuse? (–)?
  {                        // Negative and zero?
    fCos = –fCos;          // Yes, then flip B! (force acute) (+)
    vT.x = –pqB->x;
    vT.y = –pqB->y;
    vT.z = –pqB->z;
    vT.w = –pqB->w;
  }
  else                     // acute angle (<=90 degrees)
  {
    vT.x =  pqB->x;
    vT.y =  pqB->y;
    vT.z =  pqB->z;
    vT.w =  pqB->w;
  }

    // Calculate coefficients
```

```
  fS0 = 1.0f - fDist;

  if (SINGLE_PRECISION < (1.0f - fCos)) // Far enough apart
  {
    fAngle = acosf(fCos);
    fSin = 1.0f / sinf(fAngle);

    fS0 = sinf(fS0   * fAngle) * fSin;
    fS1 = sinf(fDist * fAngle) * fSin;
  }
  else                                  // Too close together
  {
    fS1 = fDist;
  }

    // Final vector scalar multiplication(s) and summation quaternion

  pqD->x = fS0 * pqA->x + fS1 * vT.x;
  pqD->y = fS0 * pqA->y + fS1 * vT.y;
  pqD->z = fS0 * pqA->z + fS1 * vT.z;
  pqD->w = fS0 * pqA->w + fS1 * vT.w;
}
```

Only single-precision floating-point was discussed here, but the same methods can easily be used to handle double-precision.

Exercises

1. Write a set identity instruction for your favorite processor.
2. What is the difference between a unit quaternion and a pure quaternion?
3. In the "Quaternion Division" section of this chapter, the starting and ending identity equations are missing the middle steps. Write those missing equations.
4. Explain a gimbal lock. What happens when two rings are in the same position? Three rings?
5. Without using a quaternion, what would be some alternative methods to prevent gimbal lock?
6. What do you think the considerations were for quaternion math being handled differently, such as in the case of DirectX and this as well as other books' implementations?

Chapter 14
Geometry Engine Tools

The title of this chapter is slightly misleading, as this is not really about geometry engines, or 3D rendering, or vertex and pixel shading, etc. It's not even really about geometry tools, as there are other books that cover those topics. It is instead a sprinkling of issues in regard to the use of vector algorithms within those tools and some algorithms within the game outside the rendering code.

CD Workbench Files: /Bench/*architecture*/chap14/*project*/*platform*

	architecture		*project*	*platform*
PowerPC	/vmp_ppc/	**ASCIItoFloat**	**/AtoFlt/**	/mac9cw
X86	/vmp_x86/	**10pt. House**	**/House**	/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

We have come a long way and covered a lot of material learning foundations and tricks on how to use vector math instructions with the various platforms. Up until now, the focus has been on packed integer as well as packed single-precision floating-point, and not much has been said about double-precision floating-point, but that time has now come. The reason the previous chapters of this book have focused on single-precision floating-point is for purposes of speed and availability; that is the common denominator between the various processors. In the following table, the CPU column is meant to be inclusive of all instructions handled by processors related to the X86-based PC, PowerPC-based Macintosh, or MIPS-based game consoles.

Table 14-1: CPU instruction set with supported packed integer, single-precision, and double-precision, and scalar and packed data

CPU	Packed Integer	Scalar SPFP	Packed SPFP	Scalar DPFP	Packed DPFP
Pre-MMX	No	Yes	No	Yes	**No**
MMX	Yes	Yes	No	Yes	**No**
3DNow!	Yes	Yes	Yes	Yes	**No**
MMX Ext.	Yes	Yes	Yes	Yes	**No**
3DNow! Ext.	Yes	Yes	Yes	Yes	**No**
SSE	Yes	Yes	Yes	Yes	**No**
SSE2	Yes	Yes	Yes	Yes	**Yes**
PowerPC	No	Yes	No	Yes	**No**
AltiVec	Yes	Yes	Yes	Yes	**No**
Gekko	No	Yes	Yes	Yes	**No**
MIPS	No	Yes	No	Yes	**No**
MIPS-MMI	Yes	Yes	Yes	Yes	**No**
MIPS-3D	Yes	Yes	Yes	Yes	**No**

Notice that the column on the far right indicates in bold packed double-precision floating-point. Single-precision floating-point is fine and dandy, but the precision is very low and so accuracy can be lost during transformations. Since most processors used in games do not support packed double-precision with the exception of the SSE2, then single-precision is typically used.

Hint: Use packed double-precision floating-point math in tools. This is where speed is sometimes more important in a tool then in the game!

But when used in tools, why use a scalar when one can use packed floating-point, and why use single-precision when one can use double-precision to obtain the most accurate results? They can be saved to the Game Relational Database (GRDB) as single-precision floating-point values.

This brings up a question that I have for my readers. If you start a processing tool to convert art resources or game resources into a game database and then leave to have lunch, get a soda, have a snack, go to the bathroom, pick up your kids from school, or go home, then yell, "Me!"

Wow! That was loud! I could hear it reverberating across the country.

Those of you who have worked on games in the past, did you meet your timelines? Did you find yourself working lots of extra crunch time to meet a milestone? (We will ignore E^3 and the final milestones!) How often do you have to wait for a tool to complete a data conversion? Add up all that "waiting" time. What did your tally come to?

Here is a thought. Add a wee bit of tiny code to your program, and write the results to an accumulative log file. Then check it from time to time. See where some of that time is going.

Some people only believe in optimizing the game if there is time somewhere in the schedule. Management quite often counts the time beans and decides that getting the milestone met is much more important than early ongoing debugging or optimization. But think of the time savings if your tools were written with optimization. Just do not tell management about it, or they will think they can ship the product early.

3D rendering tools are expensive, and so programmers typically do not readily have access to a live tool. They sometimes write plug-ins, but quite often they will merely write an ASCII Scene Exporter (ASE) file parser to import the 3D data into their tools that generate the game databases. With this method, the programmer does not have to have a licensed copy of a very expensive tool sitting on his or her desk.

Hint: Prevent artist versus programmer wars by working out who gets the burden of the day-to-day conversion of art resources for the game application before the project coding begins. Use and document it in the Game Design, Technical Design, and especially the Art Bible documents, and treat these as living entities and not written in stone!

This little item brings up a trivial item of artist versus programmer wars. It all comes down to who will have the task of running the tools to export and convert data into a form loaded and used by a game application. Neither typically wants the task and consider it mundane, but it is nevertheless required. Artists need to run the tools occasionally to check results of their changes to art resources. Programmers occasionally need to run the tools to test changes to database designs, etc. But nobody wants to do it all the time. So my suggestion is to automate the tools and incorporate the who and what into the Game Design, Technical Design, and Art Bibles for the project. That way, there will be no misperception.

Okay, let's talk about something else related to vectors.

In this particular case, an ASE file is an ASCII export from 3D Studio Max. How many of you have actually written a parser and wondered where all your processing time had gone? Did you use file streaming reads to load a line at a time or a block read to read the entire file into memory?

I personally write ASE parsers by loading the entire file into memory, even when they are 20 MB or larger in size. The core ASE parser code included with this book can actually parse an entire 20 MB file and

convert about 1.15 million floating-point values from ASCII to doubles in a few seconds. But here is where it really gets interesting!

ASCII String to Double-Precision Float

Calling the standard C language function atof() to convert an ASCII floating-point value to single- or double-precision will add significant time to your processing time for those large ASE files!

Hint: Do not use the run-time C library atof()! Use the following functionality instead! By using this single function, this book will pay for itself quickly in time savings! And that is what it is all about!

I have good news for you. The following function works only on an X86 processor and will carve those hours back to something a lot more reasonable. It takes advantage of a little-known functionality within the floating-point unit (FPU) of the X86 processor.

The FPU loads and handles the following data types:

- (Four-byte) single-precision floating-point
- (Eight-byte) double-precision floating-point
- (Ten-byte) double-extended precision floating-point
- (Ten-byte) binary-coded decimal (BCD)

Did you know about all four of these?

This latter type is what is of interest to us. Converting an ASCII string to binary-coded decimal is easy as pie (or is it cake?). In BCD, for every byte, the lower 4-bit nibble and upper 4-bit nibble each store a value between 0 and 9 (think double-digit hex, only the upper six values A through F are ignored).

Table 14-2: ASCII numerical digit to hex and decimal values

ASCII	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
Hex	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39
Decimal	48	49	50	51	52	53	54	55	56	57
BCD	0	1	2	3	4	5	6	7	8	9
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001

Converting a BCD value from ASCII to a nibble is as easy as subtracting the hex value of 0x30, 0, or 48 decimal from the ASCII numerical value and getting the resulting value with a range of {0...9}.

```
byte ASCIItoBCD( char c )
{
  ASSERT(('0' <= c) && (c <= '9'));

  return (byte)(c - '0');
}
```

The FPU uses the first nine bytes to support 18 BCD digits. If the uppermost bit of the 10th byte is set, the value is negative. The value is positive if the bit is clear.

Table 14-3: Ten-byte BCD data storage. MSB in far-left byte (byte#9) is the sign bit and the rightmost eight bytes (#8...0) contain the BCD value pairs. The 18th BCD digit resides in the upper nibble of byte #8, and the first BCD digit resides in the lower nibble of byte #0.

Byte 9	8	7	6	5	4	3	2	1	0
S_	17 16	15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0

Setting the upper nibble of a byte is merely the shifting left of a BCD digit by four bits then logical OR'ing (or summing) the lower nibble.

```
byte BCDtoByte( byte lo, byte hi )
{
  return  (hi << 4) | lo;
}
```

How does this all work? Well, the FPU has a single instruction that loads a BCD value and converts it to an 80-bit (10-byte) double extended precision floating-point value that it stores on the FPU stack. This can then be written back to computer memory as double-precision floating-point — simple, fast, and minimal excess code and nothing time intensive.

Listing 14-1: vmp_x86\chap14\ase2vmp\util.cpp

```
unsigned char bcd[10];
double f;

__asm {
  fbld  tbyte ptr bcd    ; Load (80-bit) BCD
  fstp  f                ; Write 64-bit Double-Precision
}
```

The returned floating-point value contains the BCD number as an integer with no fractional component. For example:

```
byte bcd[10] = {0x68, 0x23, 0x45, 0x67, 0x89, 0x98, 0x87, 0x76, 0x65, 0x80};
```

The float returned is –657,687,988,967,452,368.0

At this point, the decimal place needs to be adjusted to its correct position using the product of an exponential 10^{-n}. This can be done

either with a simple table lookup or a call to the function pow (10,–e), but the table lookup is faster! And speed is what it is all about.

ASCII to Double

Note that the following code sample expects a normal floating-point number and no exponential. The ASE files do not contain exponential, just really long ASCII floating-point numbers, which is the reason this code traps for more than 18 digits.

Listing 14-2: vmp_x86\chap14\ase2vmp\util.cpp

```
double exptbl[] =                  // –e
{
  1.0,                   0.1,
  0.01,                  0.001,
  0.0001,                0.00001,
  0.000001,              0.0000001,
  0.00000001,            0.000000001,
  0.0000000001,          0.00000000001,
  0.000000000001,        0.0000000000001,
  0.00000000000001,      0.000000000000001,
  0.0000000000000001,    0.00000000000000001,
  0.000000000000000001
};                                 // Limit 18 places

double ASCIItoDouble( const char *pStr )
{
#ifdef CC_VMP_WIN32
  unsigned int dig[80], *pd;
  unsigned char bcd[10+2], *pb;
  double f;
  int       n, e;
  const char *p;

  ASSERT_PTR(pStr);

  *(((uint32*)bcd)+0) = 0;         // Clear (12 bytes)
  *(((uint32*)bcd)+1) = 0;
  *(((uint32*)bcd)+2) = 0;         // 2 + 2 spare bytes

  // Collect negative/positive – and delimiters are pre-stripped.

  p = pStr;
  if ('–' == *p)
  {
    *(bcd+9) = 0x80;               // Set the negative bit into the BCD
    p++;
  }

  // Collect digits and remember position of decimal point

  *dig = 0;                        // Prepend a leading zero
```

```
  e = n = 0;
  pd = dig+1;

  while (('0' <= *p) && (*p <= '9'))
  {
    *pd++ = (*p++ - '0');        // Collect a digit
    n++;

    // The decimal place is checked after the first digit as no
    // floating-point value should start with a decimal point!
    // even values between 0 and 1 should have a leading zero!  0.1
    if ('.' == *p)               // Decimal place?
    {                            // Remember its position!
      e = n;
      p++;
    }
  }

 // Check for a really BIG (and thus ridiculous) number

  if (n > 18)                    // More than 18 digits?
  {
    return atof(pStr);
  }

  if (e)                         // 0=1.0  1=0.1  2=0.01  3=0.001, etc.
  {
    e = n - e;                   // Get correct exponent!
  }

    // repack into BCD (preset lead zeros)
    // last to first digit

  n = (n+1)>>1;                  // Start in middle of BCD buffer
  pb = bcd;                      // Calc. 1st BCD character position

  while(n--)                     // loop for digit pairs
  {
    pd-=2;                       // Roll back to last 2 digits
    *pb++ = ((*(pd+0)<<4) | *(pd+1));          // blend two digits
  }

  __asm {
    fbld  tbyte ptr bcd   ; Load (10 byte) BCD
    fstp  f               ; Write 64-bit double-precision
  }

    return f * exptbl[e];                      // FASTER
//  return f * pow( 10.0, (double) -e );       // FAST
#else
  return atof(p);                              // Really SLOW
#endif
}
```

Okay, admittedly, this has little to do with vector processing, but it is definitely worth the write-up here due to the high optimization factor

that it delivers and the fact that four of those floating-point scalar values makes up a vector!

If you do not believe me about the speed, then replace all the atof() functions in your current tool with a macro to assign 0.0 and measure the difference in speed. Or better yet, embed the function atof() within this function and do a float comparison with the precision slop factor, since by now you should be very aware that you never, ever compare two floating points to each other to test for equivalence unless a precision slop factor (accuracy) is utilized.

Review:

```
bool vmp_IsDEqual(double fA, double fB, DOUBLE_PRECISION);
```

Besides, you should always test optimized code (vector based or not) in conjunction with slow scalar code written in C to ensure that the code is functioning as required.

One more thing: If you insist on using atof() or sscanf(), copy the ASCII number to a scratch buffer before processing it with either of these two functions because processing them within a 20 MB file dramatically increases the processing time by hours.

Okay, back on task!

ASE File Import — XZY to XYZ

Sorry for the back-to-basics tutorial, but for those of you not familiar with these little details, you will need to know them to have a better insight into vectorizing your code. For purposes of assisting that understanding, this book uses an ASE file for import of vertex data, as the file is in an ASCII format, which makes it an easier tutorial path than trying to parse a binary file.

Back in Chapter 9, "Vector Multiplication and Division," culling was briefly discussed in regards to the dot product, and it was noted that for game applications, the left-handed rule is generally followed and the vertices are kept in a clockwise direction. The data export from the ASE file is actually in an {XZY} order instead of the expected {XYZ}, and the vertices are actually in a right-handed counterclockwise order.

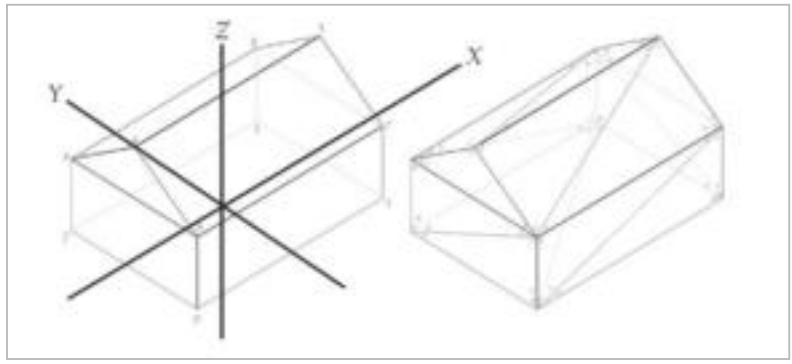


Figure 14-1: Ten-point 3D house object in XZY right-handed coordinate system. Z is up! Art compliments of Key Mayfield.

Now I should point out that some graphic developers keep their environment in a right-handed order. That is, their triangles are in a counterclockwise direction. Other developers set their triangles in left-handed order, a clockwise direction. Some of those, however, actually use the z-axis for their elevation information and y-axis for their horizon. Those developers are in luck because that is exactly the format that the data from an ASE is in. For the rest of you, it will require more work.

To make it all apparent, let's first examine the list of ten vertices. It has been extracted from an ASE file:

```
// *MESH_VERTEX_LIST {
// *MESH_VERTEX        0          -3.5430   -2.4530   -0.0000
     Data Type      Vertex #            XZY Coordinate
```

Set that list as a simple vector array to keep it similar.

```
vmp3DVector HouseVertexList[] =
{ //         X                 Z                 Y
  { -3.5430f, -2.4530f, -0.0000f },   // Mesh Vertex #0
  {  3.5382f, -2.4530f, -0.0000f },   // Mesh Vertex #1
  { -3.5430f,  2.4445f,  0.0000f },
  {  3.5382f,  2.4445f,  0.0000f },
  { -3.5430f, -0.0051f,  4.2178f },
  {  3.5382f, -0.0051f,  4.2178f },
  { -3.5430f, -2.4530f,  2.4776f },
  {  3.5382f, -2.4530f,  2.4776f },   // Mesh Vertex #7
  {  3.5382f,  2.4445f,  2.4776f },
  { -3.5430f,  2.4445f,  2.4776f }    // Mesh Vertex #9
};
```

Did you noticed the XZY ordering of the column labels where Z and Y are swapped?

The object requires 16 three-point faces (triangles) to form the shape. The ASE file uses a reference such as the following:

```
// *MESH_FACE_LIST {
// *MESH_FACE    2:            A: 0  B: 1  C: 7
            Mesh Face #         Mesh Vertices
```

Each corner of the triangle represented by an A, B, or C is a zero-based index into the vertex list.

```
uint HouseFaceList[] =
{ // A   B   C
    0, 2, 3,     // Mesh Face #0
    3, 1, 0,
    0, 1, 7,     // Mesh Face #2
    7, 6, 0,
    6, 7, 5,
    5, 4, 6,
    1, 3, 8,
    8, 7, 1,
    7, 8, 5,
    3, 2, 9,
    9, 8, 3,
    8, 9, 4,
    4, 5, 8,
    2, 0, 6,
    6, 9, 2,
    9, 6, 4      // Mesh Face #15
};
```

So for example, the third face (#2), using vertex indices 0, 1, 7, make up the triangular lower front of the house. Notice that the vectors bordering the edges of the face are in a counterclockwise pattern.

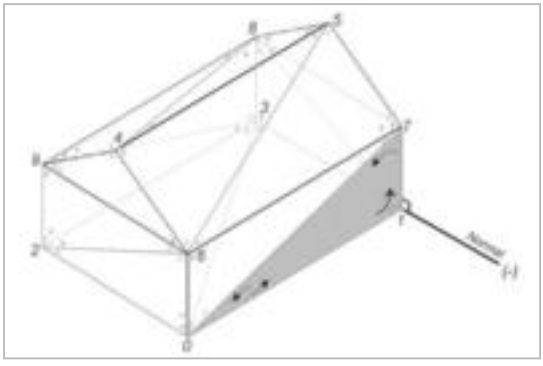


Figure 14-2: The same ten-point 3D house, but with face#2 highlighted, the counterclockwise orientation indicated by arrows, and the negative normal indicating the texture is properly oriented. Art compliments of Key Mayfield.

If the crossed elements {Z} and {Y} are left crossed, then if rendered on a left-handed system it will be rendered on its edge and inside out.

Within your tool, the vector is copied to a quad vector for purposes of conversion of AoS to SoA efficiency, but that will be discussed later. Regardless of the reasons, the {Z} and {Y} elements need to be uncrossed, and that is easily done with the following algorithm:

```
pQVec->z = pVec->y;
pQVec->y = pVec->z;
pQVec->x = pVec->x;
pQVec->w = 0.0f;
```

The important items to note are the ones indicated in bold. The algorithm copies a vector to a quad vector while correcting for the axis reversal. That is only part of the solution, as the vertices may now be in XYZ order, but they are still in a clockwise orientation.

The second part of the solution is a reverse of vector directions from right- to left-handed, and that is simply a matter of reversing the direction of all three vertices. So if vAB is vector AB, and vAC is vector AC, then vD would be the outer product (the perpendicular) to the plane that the two vectors intersect in the right-handed orientation.

```
vmp_QCrossProduct(&vD, &vAB, &vAC );
```

So to reverse for the left-handed operation, either negate the resulting vector:

```
vD.z = –vD.z          vD.y = –vD.y          vD.x = –vD.x
```

...or better yet, merely swap the source vectors to the cross-product! This effectively inverts the direction of each axis that makes up the new vector.

```
vmp_QCrossProduct(&vD, &vAC, &vAB );
```

Check out the listing vmp_x86\chap14\House\Bench.cpp on the companion CD. It is a demonstration of vertex manipulation from ASE XZY to left-handed XYZ form. So now that you know how to massage the data into a usable form, let's continue.

3D Render Tool to Game Relational Database

We touched on the importance of the optimization of tools and have also discussed the use of a string to BCD to double-precision floating-point to achieve a higher throughput. Now let's look into how to handle our floating-point data. Normally, a programmer would (not to oversimplify it) merely collect the polygon and face data and their associated properties a float at a time, store it in a three-float set of single-precision to represent the {XYZ} coordinates, transform the data such as with scaling and any possible needed rotations, and then write it out in a dedicated relational database to be loaded and used by the game. At this point, you are probably thinking, if we saved a lot of time just using the improved ASCII to float function, why would one need to implement additional vector methods?

Well, every little bit helps. The faster the tool, the shorter the wait time. The quicker the turnaround, the sooner you will be back onto a normal 40- to 50-hour work week. But then again, maybe you have no social life, or you do not want to go home because you would have to talk to your spouse or hear the screaming kids, or you like the company to buy you dinner every night. (They do buy you dinner on all of those late work nights, do they not? Oh, sorry to hear that! Time to work on that resume!)

Since we as developers typically have the computing power for the tools, we should use the double-precision math for all the pre-database calculations. In scalar form on an X86 with the coprocessor FPU, there is no time penalty, as all calculations are done in a 10-byte (double-extended precision floating-point) form. As you have hopefully been learning throughout this book, in a parallel form a much higher throughput of calculations is achieved. You may recall from Chapter 4, "Vector Methodologies," that we discussed the need to process data in an SoA (Structure of Arrays) format instead of the AoS (Array of Structures.) This latter mechanism is what most programmers typically use in their scalar implementations.

```
#define SOAD_ARY_MAX 8   // SoA double-precision version
```

SoA (Structure of Arrays)

```
typedef struct
{
  double x[SOAD_ARY_MAX];
  double y[SOAD_ARY_MAX];
  double z[SOAD_ARY_MAX];
  double w[SOAD_ARY_MAX];
} SoAD;    // 256-byte block
```

AoS (Array of Structures)

```
typedef struct AoSD_Type
{
  double x;
  double y;
  double z;
  double w;
} AoSD;

AoSD   aosTbl[SOAD_ARY_MAX];  // 256-byte block
```

This AoS looks very similar to a typical quad double-precision floating-point vector structure, because it is!

Listing 14-3: \inc???\vmp3D.h

```
typedef struct vmp3DQDVectorType
{
  double x;
  double y;
  double z;
  double w;
} vmp3DQDVector;
```

It only really works out for scalar operations because each component is typically dealt with differently, especially in the case of the ASE file as it only uses the three components {X,Y,Z}. Or I should say {X,Z,Y}, but that was just discussed. This implementation of an ASE to Game Relational Database (GRDB) is first loaded into a tree architecture of vectors, faces, and properties, and the ASCII floating-point data is loaded in an AoS format but in double-precision with the fourth component {W} set to zero.

In a second pass, each AoS containing the vertex data is converted into aligned blocks of SoA while still being kept in double-precision floating-point. Any unused fractional portion of the last block is padded to fill the block. This makes it easier for the next step, which is the manipulation of the data, scaling, translation, component swap, etc.

Normally this is where I use automation tools designed for the import of the entire game — all scenery, all characters, etc. — but to keep things simple for this book, I opted to use simple output. For example, in the case of textures, the file path to the texture is extracted from the ASE and exported in an enumeration order in the form of an automatically generated C file to embed within the body of a lookup table. I know it is not good C programming style, but I wanted something that you could easily adapt to whatever mechanism you wanted to put into place.

For example, you would generate a file such as the following:

Sample.c

```
Char *myTexturePaths[] = {

#include "SampleA_Tex.c"
#include "SampleB_Tex.c"
#include "SampleC_Tex.c"

};
```

And the automatically generated file would be as follows:

SampleA_Tex.c

```
    "C:\Game\N_Wall01.bmp",   // +0 Map #2
    "C:\Game\N_Wall02.bmp",   // +1 Map #4
    "C:\Game\N_Wall03.bmp",   // +2 Map #6
```

This would be used within a tool to convert art files to something more appropriate for the platform, as well as repackage them on 2048-byte alignment for CD sector block loading and indexed handling.

But before we begin to deal with all of that, there is a simple question you need to answer: How is the vertex and pixel data going to be rendered?

It is a simple question, but its answer dictates how the data is going to be exported. Is the rendering device an old PC that will require a software render or does it have hardware render capability? Is it a recent video card with programmable pixel and vertex shaders? Is it a PS2 console where you will be using VU1 code to handle the render processing? Or is it something else? All of the above? If it is DirectX related, then your vertex data is expected to be arranged in a certain way.

Ignoring for now the issues of vertices used for rendering, other vertices are used for terrain information for game physics, artificial intelligence, and rail systems. These rails can be very short (two points) and used for setting an object and a direction in which it is facing (a

simple vector!). Other rails are longer and used as paths for characters whether good guys, bad guys, simple scenery props, or the player character that the user controls, as well as camera glide paths.

Each vertex along the line path would be considered a waypoint and spline, or another line-following algorithm would be used to smooth out the bumps in the corners of each line. Regardless of the use, in the case of a landscape, as it is being rotated, scaled, translated, etc., each line path within the landscape would need to be manipulated as well. Since these supporting vertices have little to do with rendering of the scene (except for developing the game when the rails are made visible for debugging purposes), they have to be manipulated in parallel to the rendering operations. Since the rendering mechanism is isolated to handle only rendering, you have full control of how you want those vertices packaged. As such, you would want them in the best form for highest speed calculations, thus SoA!

Different paint programs have different methods of implementing this, and some game developers use different methods, such as laying a track down with a middleware tool during the conversion process, but my favorite is to allow the artist to lay the tracks down using the line tool. With this they can place these lines into their terrain. From a database point of view, these lines would be treated as a long list of vertices with all vertices within a line being sequentially ordered. A separate table would contain the list of indices to the first entry of their line within the list of vertices. By using enumerated lines, the next entry in the index table would be the beginning of the next line. The difference between indices would be the number of waypoints (vertices) within a line (rail).

The following table is an example of five rail (track/glide path) enumerations and their lengths. So the equivalent offset/size lookup table would be similar to what is shown in bold on the right. Note the addition of an extra index (#5) to indicate the end of the table, in essence the number of vertices total and the total of all the lengths.

Rail (Track) Table			
Index	**Name**	**Length of Rail**	**Lookup Offset**
0	Rail00	32	**0**
1	Rail01	2	**32**
2	Rail02	2	**34**
3	Rail03	2	**36**
4	Rail04	2	**38**
5			**40**

No matter how the game data is arranged, the table would effectively be used to index into the list of vertices. Again, let's examine a small portion of an ASE file used for a linear landscape. (A linear landscape is a landscape designed as a complete elevation terrain model and not as a cookie cutter-type landscape where the terrain is constructed on the fly by assembling scene props!) Some of the details have been extracted for this one shape object. Keep in mind that the SHAPE_VERTEX_KNOT is in an {XZY} order! So at this point, it would be up to you as to how you would manipulate the vertices — either as an entire set, a fixed block, or a few vertices upon demand.

```
*SHAPEOBJECT {
  *NODE_NAME "rail00"
  *NODE_TM {
    *NODE_NAME "rail00"
    *TM_ROTAXIS   0.00000   0.00000   0.00000

    *TM_ROTANGLE 0.00000
  }
  *SHAPE_LINECOUNT 1
  *SHAPE_LINE 0 {
    *SHAPE_VERTEXCOUNT 32
    *SHAPE_VERTEX_KNOT  0   23887.78906   -63951.60547   10921.57520
    *SHAPE_VERTEX_KNOT  1   24473.74023   -63756.69922   10921.57520
    *SHAPE_VERTEX_KNOT  2   24899.65820   -63405.74219   10921.57520
    *SHAPE_VERTEX_KNOT  3   25190.11719   -62846.81641   10921.57520
    *SHAPE_VERTEX_KNOT  4   25295.52734   -62108.67188   10921.57520
    *SHAPE_VERTEX_KNOT  5   25116.21484   -61312.30078   10921.57520
    *SHAPE_VERTEX_KNOT  6   24514.09961   -60748.00000   10921.57520
    *SHAPE_VERTEX_KNOT  7   23642.72656   -60463.22656   10999.52051
    *SHAPE_VERTEX_KNOT  8   22749.57031   -60331.46484   11183.67676
    *SHAPE_VERTEX_KNOT  9   21883.35156   -60357.81641   11358.09277
    *SHAPE_VERTEX_KNOT  10  20873.89844   -60357.81641   11468.44629
    *SHAPE_VERTEX_KNOT  11  19558.52344   -60357.81641   11516.72070
    *SHAPE_VERTEX_KNOT  12  18355.98438   -60176.35156   11516.72070
    *SHAPE_VERTEX_KNOT  13  17737.18359   -59843.15234   11516.72070
    *SHAPE_VERTEX_KNOT  14  17403.98242   -59414.75000   11516.72070
    *SHAPE_VERTEX_KNOT  15  17356.38281   -58653.14844   11516.72070
    *SHAPE_VERTEX_KNOT  16  17629.56641   -58132.47656   11516.72070
    *SHAPE_VERTEX_KNOT  17  18213.18359   -57939.14844   11516.72070
    *SHAPE_VERTEX_KNOT  18  18831.98633   -57939.14844   11516.72070
    *SHAPE_VERTEX_KNOT  19  19450.78711   -58034.34766   11516.72070
    *SHAPE_VERTEX_KNOT  20  19736.38867   -58319.94922   11516.72070
    *SHAPE_VERTEX_KNOT  21  20069.58984   -58795.94922   11516.72070
    *SHAPE_VERTEX_KNOT  22  20402.78906   -59414.75000   11516.72070
    *SHAPE_VERTEX_KNOT  23  20688.39063   -60033.55469   11516.72070
    *SHAPE_VERTEX_KNOT  24  20942.64844   -60779.73047   11516.72070
    *SHAPE_VERTEX_KNOT  25  21233.68555   -61417.84766   11468.44629
    *SHAPE_VERTEX_KNOT  26  21482.32813   -62061.20703   11389.28711
    *SHAPE_VERTEX_KNOT  27  21762.47852   -62678.21875   11250.94727
    *SHAPE_VERTEX_KNOT  28  21947.06836   -63150.90625   11139.12012
    *SHAPE_VERTEX_KNOT  29  22470.66797   -63531.70703   10984.61426
    *SHAPE_VERTEX_KNOT  30  22946.66992   -63817.30859   10921.57520
    *SHAPE_VERTEX_KNOT  31  23668.59375   -63932.20313   10921.57520
```

```
  }
}
```

A typical XZY Array of Structures:

```
vmp3DDVector Tbl[] =
{   //   X              Z              Y
  { 23887.78906,   –63951.60547,   10921.57520 },  // Knot #0
  { 24473.74023,   –63756.69922,   10921.57520 },  // Knot #1
   ...
};
```

A corrected quad Array of Structures:

```
vmp3DQDVector Tbl[] =      // Same as AoSD
{   //   X              Y               Z        W
  { 23887.78906,   10921.57520,   –63951.60547, 0.0 },  // Knot #0
  { 24473.74023,   10921.57520,   –63756.69922, 0.0 },  // Knot #1
   ...
};
```

When reorganized into the Structure of Arrays:

```
SoAD Tbl[] = {
{ {  23887.78906,  24473.74023,  24899.65820,  25190.11719
     25295.52734,  25116.21484,  24514.09961,  23642.72656 },   // x0...7
  {  10921.57520,  10921.57520,  10921.57520,  10921.57520
     10921.57520,  10921.57520,  10921.57520,  10999.52051 },   // y0...7
  { –63951.60547, –63756.69922, –63405.74219, –62846.81641
    –62108.67188, –61312.30078, –60748.00000, –60463.22656 },   // z0...7
  { 0.0, 0.0, 0.0, 0.0,   0.0, 0.0, 0.0, 0.0 }
},
{ {  22749.57031,  21883.35156,  20873.89844,  19558.52344
     18355.98438,  17737.18359,  17403.98242,  17356.38281 },   // x8...15
  {  11183.67676,  11358.09277,  11468.44629,  11516.72070
     11516.72070,  11516.72070,  11516.72070,  11516.72070 },   // y8...15
  { –60331.46484, –60357.81641, –60357.81641, –60357.81641
    –60176.35156, –59843.15234, –59414.75000, –58653.14844 },   // z8...15
  { 0.0, 0.0, 0.0, 0.0,   0.0, 0.0, 0.0, 0.0 },
},
   ...
};
```

When processing the data, the fourth element is merely skipped over, as it is merely a placeholder. In a real application, it would contain up to 32 information bits related to the waypoint (vertex). The SoA would require extra space for the fourth element, but it would be optimally faster processing then just using the AoS. In that case, if the {W} field does not exist, the {XYZ} vector is processed with misaligned memory. If the {W} field does exist, it needs to be coded around. This is a lesson that was made very clear in earlier chapters of this book.

The next generation of vertex and pixel shaders will be discussed in the next chapter. If working with DirectX, your only concern with the data is to either import the X file into your application or massage the vertex and face data into a form that is expected.

Note that only the first vertex is aligned properly and the data is really only 32-bit aligned. It will be up to you to custom modify the data structure to compensate for this problem.

Those of you working with older operating systems will find that the burden of vertex mathematics and culling is on your shoulders, so hopefully you will have learned the lessons taught earlier in this book.

Collision Detection

This is a perfect platform for this book, as rendering hardware is not going to do collision detection for you! This topic is related to physics and AI and is typically one of the most time-consuming operations for the CPU. I must point out again that this book is not about how to do collision detection. It merely takes some well-known collision detection functions and shows you how to vectorize them, although by now you should have the foundations to do so on your own. The vector code shown in this section uses vector functions that have been discussed in this book. These will be significantly faster than using scalar-based code, but if even more speed is needed, you can always blend the functionality of all the vector functions together in assembly. This will make tighter, faster code where aspects of processor optimization, such as pipelining, register usage, etc., can be used to advantage. Obviously doing so will increase code size, but optimizing code is a balancing act.

Is Point on Face?

The first item of importance is the placement of the objects within a landscape. So, in essence, it is the point at which you collide with (touch) the ground. Each character has an anchor point typically at their center of gravity between the soles of feet, paws, tentacles, whatever. To place the character, typically the {XZ} position is known, but it has to be positioned where it makes contact with the ground.

The character may be walking around within a 2D polygon fenced area or locked on a rail. The artist does not want to spend time twiddling a rail to match the elevation of the landscape exactly on a face-by-face basis, as it would be considered a waste of one's time. So the {XZ} axis of the rail is used for horizontal positioning, and then the face information is used to calculate the Y elevation of the base position (the point where it makes contact with the ground).

In some applications, the 3D objects are sometimes forced to 2D, removing the elevation component in an attempt to simplify code and speed up the math, since rotating a 2D marker around the y-axis (yaw) is very simple. Often however, there are issues of the elevation not being quite right. One of the vertex corners is used as a quick y-axis base. Even using a midpoint on two corners is not really effective, but it is quick:

$D_z=(A_z+B_z)*0.5$ $D_y=(A_y+B_y)*0.5$ $D_x=(A_x+B_x)*0.5$

Notice the vector math?

```
vmp3DVector vHalf = {0.5f, 0.5f, 0.5f};

vmp_VecAdd( &vD, &vA, &vB );
vmp_VecMul( &vD, &vD, vHalf );
```

This leads to some interesting artifacts when the ground is large and/or approaching a 45º angle or more of slope, thus not quite level. Even storing or calculating a midpoint will not be very accurate. With vector processing, we can speedily calculate a vector plane intersect. Of course, we are resolving the point of intersection.

Cat Whiskers

Have you ever noticed that dogs always seem to be in the news because they become stuck in all sorts of weird locations? Did you ever notice that it is almost never a domestic kitty cat? It is not because dogs are dumber than cats (although cat owners may think so), but because a cat's whiskers extend past the width of its body. A cat knows that if it attempts to go through a small opening, if the whiskers do not have the clearance, its body will not either, so it typically does not try. Unless it is in hot pursuit of a mouse, but that is a fable for a children's book.

You can think of the surface of a sphere as the end of a cat's whisker. One of the simplest collision detections is a bounding sphere to bounding sphere collision. The surface of a sphere is always the distance of the length of the radius from the center of that sphere.

$r = \sqrt{((B_z-A_z)^2 + (B_y-A_y)^2 + (B_x-A_x)^2)}$
if $(r < A_{radius} + B_{radius})$ then collided!

So by calculating the distance between the two center points of the two spheres and subtracting the radius of each of the spheres, if the distance becomes negative, their volumes intersect and are collided. If the distance is zero, their surfaces are touching. If positive, they do not intersect.

An optimization step, especially if there are multiple objects to test (which is typically the case), would be to square the sum of A_{radius} and B_{radius} so there would be no need for a square root until the actual distance is needed. A multiplication is much faster than a square root calculation.

$$r_{sum} = ((B_z - A_z)^2 + (B_y - A_y)^2 + (B_x - A_x)^2) \qquad \text{Sum of squares!}$$

We learned this trick back in Chapter 9, "Vector Multiplication and Division."

```
vmp_VecSub( &vD, &vB, &vA );          // Calculate a vector
vmp_DotProduct( &r, &vD, &vD );       // r=x²+y²+z²
```

You may recall that the sum of squares is calculated using the dot product but with the same vector used for both source inputs.

$$i = r_{sum} - (A_{radius} + B_{radius})^2$$

```
if (0 > i)          then collided!
else if (0 == i)    then touching.
else                no collision.
```

No matter what collision mechanism is used in your application, by using a sphere-to-sphere collision test as the preliminary collision check, some processing speed can be retrieved. Just set each sphere to the maximum limit of the collision detection shapes. So in essence, the sphere becomes the cat whisker!

But the game being developed is not about sports, where balls (spheres) bounce around and off each other, so how do we calculate it?

The solution to this problem is actually easy! It is a step beyond calculating a bounding box for a list of vertices. For that, it is merely the list of vertices for an object being scanned and the minimum and maximum {XYZ} limit values being collected.

Calculate a Bounding Box from Vertex List

Listing 14-4: \chap14\clash\Clash.cpp

```
void vmp_ClashBoundBox( vmp3DVector * const pvMin, // Lower
                        vmp3DVector * const pvMax, // Upper
                  const vmp3DVector * const pvList,
                  uint nCnt )
{
  ASSERT_PTR16(pvMin);
  ASSERT_PTR16(pvMax);
  ASSERT_PTR16(pvList);
```

```
  ASSERT(2<=nCnt);    // Need at least two points for limits

  nCnt--;
  vmp_VecCopy( pvMin, pvList );
  vmp_VecCopy( pvMax, pvList++ );

  // Loop for each vertex in list and if an individual
  // {XYZ} element exceeds the current minimum and maximum
  // limit, then set that element as the new limit!

  do {
    nCnt--;
    vmp_VecMin( pvMin, pvMin, pvList );   // D = Min(A,B)
    vmp_VecMax( pvMax, pvMax, pvList++ ); // D = Max(A,B)
  } while (nCnt);
}
```

Figure 14-3: The midpoints of the {XYZ} axis used as the center of the box and the center of the sphere. Art compliments of Ken Mayfield.

For the sphere, the center point is merely the midpoint of these limits of the bounding box. The radius is half of the distance between the minimum and maximum set of {XYZ} values (the center of the box) and a corner of the box. So in essence, all eight corners of the box touch the surface of the sphere so that the box is contained within the sphere. An alternative would be that the sphere is contained within the box so that the sphere touches the middle of each of the six sides of the box, but that is not what is being used here.

```
vmp3DVector vBoxL, vBoxU, vCenter;
float r, rsq;

vmp_ClashBoundBox( &vBoxL, &vBoxU, vList, nListCnt );
vmp_ClashBoundSphere( &vCenter, &r, &rsq, &vBoxL, &vBoxU );
```

Calculate a Bounding Sphere for a Box

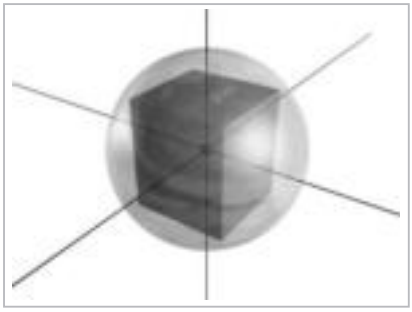


Figure 14-4: Bounding sphere encapsulating a box. Art compliments of Ken Mayfield.

Listing 14-5: \chap14\clash\Clash.cpp

```
void vmp_ClashBoundSphere( vmp3DVector * const pvCenter,
                    float * const pRadius,
                    float * const pRadiusSq,
                    const vmp3DVector * const pvMin,
                    const vmp3DVector * const pvMax )
{
  float rr;
  vmp3DVector v;

  ASSERT_PTR16(pventer);
  ASSERT_PTR4(pRadius);
  ASSERT_PTR4(pRadiusSq);
  ASSERT_PTR16(pvMin);
  ASSERT_PTR16(pvMax);

    // Bounding Box    C=(A+B)/2
  vmp_VecAdd( pvCenter, pvMin, pvMax );
  vmp_VecMul( pvCenter, pvCenter, &vHalf );

    // Internal Bounding Sphere   R=B–C
  vmp_VecSub( &vR, pvMax, pvCenter ); // center to box corner
 //             rr = r2 =x2+y2+z2    Note that the radius is still squared!
  vmp_DotProduct( &rr, &vR, &vR );    // sum of squares
  *pRadiusSq = rr;                    // Radius squared = rr = r2
  vmp_FSqrt( pRadius, r4 );           // radius of bounding sphere = √rr
}
```

Once the bounding box and sphere information is known, the parameters for other collision shape objects, such as cones and cylinders, can easily be calculated as well using Pythagorean theorems and scalar trigonometry.

vMin	Lower box coordinate
vMax	Upper box coordinate
pvCenter	Center of the sphere, box, cylinder, etc.
*pRadius	Radius of the sphere
*pRadiusSq	Radius squared

The only catch is that the orientation of other collision types, such as cylinders, needs to be known as they have three possible orientations, and so the axis {XYZ} that the lateral axis of the cylinder is parallel to needs to be known. Cones have six possible orientations along vector lines in positive and negative directions with the point at pvCenter.

I would recommend storing collision information, such as the center point, radius, and radius^2 values at the very least, in the data file with an object's mesh information. This is so you would not have to calculate them during game play, and this little amount of information can save a lot of time overall.

There are a multitude of other collision types, but these are the basic shapes used by most. By replacing your scalar-based code libraries with vector-based ones, your collision detection as well as your AI code speed should increase. With the advent of programmable vertex and pixel shaders, a large processing-intensive burden has been lifted from the CPU, freeing it for more physics, AI, multimedia, network programming, etc. But why stop there? Why just use slower scalar code with the freed-up processing time when this can actually be taken advantage of? Fantastic new functionality can be easily added to your application by using vector-based code. Keep the algorithms fast, and keep more CPU power in reserve for more added feature complexity. Have feature creep become a good thing!

Exercises

1. If in a left-handed coordinate system, {Z} is elevation and {Y} is the horizon, how do we correct for a left-handed coordinate system with {Y} as elevation and {Z} as the horizon?
2. Same as exercise 2, but if the normals were the same, how do we correct those?

Chapter 15
Vertex and Pixel Shaders

The next logical step in making 3D animation go faster is to allow a programmer to add his own code into the rendering pipeline. A simple scripting language is now implemented using an assembly style programming syntax and registers in a method of assembly language programming. In the previous chapters of this book, we discussed the internals of mathematical functionality. This laid a good foundation for those of you with access to one of the new video cards, such as nVIDIA's GeForce series 3 or higher, ATi's 8500 series, and Matrox's Parhelia-512, that supports programmable vertex and pixel shaders. It should be pointed out that if this chapter was not here, it would be noticeably missing, as it is very vector related! This chapter is going to give you a brief overview of the programmable vertex shaders and just a touch of the pixel shaders. It would actually take an entire book to go into detail about its full flexibility and functionality!

If you really want to jump in feet first, check out Wolfgang F. Engel's *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks,* from Wordware Publishing. See the references section for additional information, but refer back to this book for vector functionality.

Do not get bogged down by the Direct3D aspects, as OpenGL uses this technology and Macintosh has these graphics chips available to them as well.

With this new technology, you now have a choice. One choice is to visit one of the following web sites with your little shopping cart and scroll through their demo aisles until you come to a product functionality that is similar to what you are interested in. Drop the demo into your download cart and check out!

- nVIDIA (http://www.nvidia.com)
- ATi (http://www.ati.com)

- Matrox (http://www.matrox.com)
- 3D Labs (http://www.3dlabs.com)
- SiS (http://www.xabre.com)
- DirectX SDK (http://www.microsoft.com)

But that would be cheating, and you would not learn anything. So instead, how about using what you have learned, picking that item apart and learning how it functions, and more! It is all vector processing — just from a higher level.

This is a grand new technology, as rendering calculations are moved from the processor CPU(s) to the video GPU (Graphics Processor Unit). This relieves the CPU of that time-consuming burden, thus freeing its time for more time-worthy game processing, such as AI, game physics, event triggering, terrain following, etc. At the time of this book's publication, there were six instruction set versions available, and those have been grouped in terms of function enhancements: {1.0, 1.1}, {1.2, 1.3}, {1.4}, and very soon {2.0}, as the state of the technology is always advancing.

The vertex shaders are handled early in the pipeline and used in the manipulation of vertices. The pixel shaders are utilized late in the pipeline and are only designed to process pixels. Together they can be used to create fabulous special effect displays within a scene. Before beginning, however, the rendering pipeline should be examined.

Note the two gray boxes in the path on the right side of Figure 15-1. They indicate the position within the flow of logic of the vertex shader module and the pixel shader module. This rendering pipeline is usually referred to as a pipeline, but in reality the whole architecture is plumbing, which has some parallel rendering pipes. Whether the logic uses an older fixed semi-static flow, such as on the left, or the more robust dynamic flow on the right, it is all fluid!

Figure 15-1: Rendering pipeline (plumbing)

Video Cards

Before we get into it, we will need a video card that supports vertex shaders and pixel shaders. But which one do we get? If you have $300 to $400 laying around, then by all means get the top of the line! But if you are on a budget, you will need to decide what functionality you can do without. Get a bottom-end card and bump up after you have mastered the entry-level stuff or something mid-priced! But be careful; labels can sometimes be misleading. Examine Table 15-1 carefully.

Currently there are five manufacturers putting out chips: nVIDIA, ATi, Matrix, 3D Labs, and Xebeia. There are many flavors of video cards out there with GPUs supporting programmable vertex and pixel shaders, as well as the cards with all sorts of bells and whistles, but the primary task is finding out what instructions you will want to support, how fast you want it to be, and which chip the card needs to contain. The following table shows the latest model chips.

Table 15-1: Manufacturer with basic GPU and relative information

MFG	Chip	VidRam MB	Spec. Version	Vertex Shader	Render Pipes	Const
nVIDIA	GeForce2	32/64	1.0	0	2	0
	GeForce3 – TI	64/128	1.0 ... 1.1	1	4	96
	Xbox	----	1.0 ... 1.3	2	4	96
	GeForce4 – TI	64/128	1.0 ... 1.3	2	4	96
	GeForce4 – MX	64	---	0	2	0
ATi	Radeon™ 8500	64/128	1.0 ... 1.4	1	3	192
Matrox	Parhelia™–512	?	1.0 ... 2.0	4	16	256

I would recommend that you follow up on the companies' web sites and get additional statistical information, such as fill rates, operations per second, memory bandwidth, etc., to make an informed decision. The information in the above table is only enough to determine which instruction set your prospective card would be capable of supporting. When I personally buy new computer processors, I typically buy for the instruction set and not so much for speed, but to each their own! Also, do not make the mistake of buying the wrong GeForce4. Zero vertex shaders means it does not have any hardware support and is thus not applicable to what this chapter is about! However, it does have very fast software emulation. If you have little or no money, you can always use software emulation of the vertex shaders. It is much slower, but at least you can still test your vertex algorithms.

Vertex Shaders

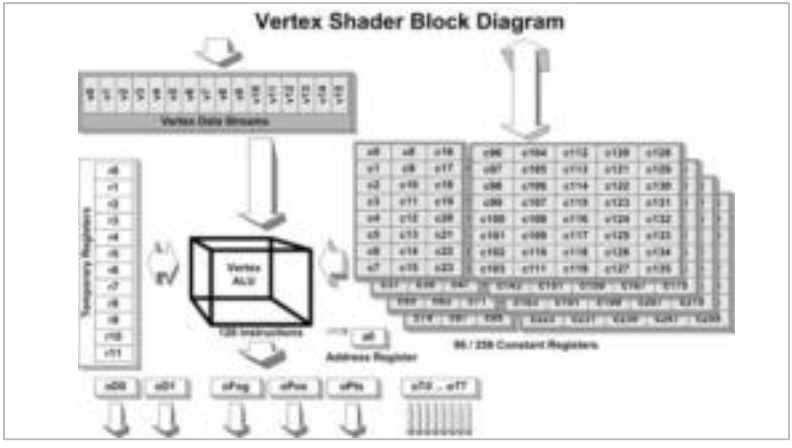


Figure 15-2: Vertex shader block diagram. Note that the grayed-out registers C_n and V_n are read only from the shader code.

Vertex shaders are used to manipulate vertices, a burden once handled by the CPU processor. Anything that requires vertex movement, i.e., flapping flags, flowing clothes, bouncing hair, particle fountains, water ripples, etc., can use this programming mechanism instead.

- *r0 ... r11* — The 12 single-precision floating-point temporary registers are used as scratch registers to temporarily save vertex data in various stages of processing.
- *c0 ... c95* and *c96 ... c255* — The standard 96 constant registers (and the additional 160 on the Parhelia), are each read-only quad single-precision floating-point vectors. They are set either from the use of the *def* instruction or by calling an external function from an application. Only one vector constant can be used per instruction, but the elements can be negated and/or swizzled. These can only be read by the vertex shader code or from the game application through an API interface. Access is through c[#] or *c*[*a0.x* + #]. For Direct3D, see *SetVertexShaderConstant*(), which can be found in the DirectX SDK.
- *v0 ... v15* — The 16 read-only vertex data registers each represent a stream of single-precision floating-point and is used as the mechanism to route the data into the Vertex ALU. Only one vertex can be used per instruction, but the elements can be negated and/or swizzled.

The output registers are primarily write-only vectors and scalars and are used to route the processed data to the graphics pipeline for the next pipeline processing stage.

- *a0* — This is a scalar write-only address register, which is used as an index offset into the table of registers. It was introduced with version 1.1. Only one use of a0 as a variable index is allowed per instruction: *c*[*a0.x* + #]. It can be thought of as a base address plus offset.
- *oD0* — The Vertex Diffuse Color Register is a write-only vector that is interpolated and written to the pixel shader color input register *v0*.
- *oD1* — The Vertex Specular Color Register is a write-only vector that is interpolated and written to the pixel shader color input register *v1*.
- *oFog* — The Vertex Fog Factor is a write-only vector of which only the scalar {X} element is interpolated and routed to the fog table.

- *oPos* — The Vertex Position Register is a write-only vector that contains the position within homogeneous clipping space.
- *oPts* — The Vertex Size Register is a write-only vector of which only the scalar {X} element containing the point size is used.
- *oT0 ... oT3* and *oT4 ... oT7* — Texture Coordinates {0...7}. These write-only vectors are used as the texture coordinates and routed to the pixel shader. Use {XY} for a 2D texture map. GeForce3 uses oT0 ... oT3 and Radeon uses oT0 ... oT7.

For access of registers, each instruction can negate or swizzle the elements.

r0.xzwy, r1.xyzw	**r0.zw, r1.xy**	**r0.xyzw, –r1.xywz**
r0.x r1.x	r0.z r1.x	r0.x –r1.x
r0.z r1.y	r0.w r1.y	r0.y –r1.y
r0.w r1.z		r0.z –r1.w
r0.y r1.w		r0.w –r1.z

For additional information, review the "Swizzle, Shuffle, and Splat" section in Chapter 5.

Before beginning, note that if you are working with Direct3D version 8.0 or 8.1, only vertex shader instructions for 1.0 through 1.1 are supported. In Direct3D version 9.0 or beyond, instructions up to 2.0 are supported. Unfortunately, at the time of publication, version 2.0 was not yet available to the public, and so this chapter is an overview of vertex shader versions from 1.0 to 1.1 and DirectX3D version 8.1. As version 8.1 is readily available, do not use 8.0, as it would have restrictions to what it can do due to being an older version. In keeping with the layout organization of this book, the instructions will be given in a similar fashion. Note that there are also macros, and they will be indicated in italics.

Table 15-2: Programmable vertex instructions and their relationship with the version of Direct3D and vertex shader versions. The ☺ indicates the instruction is supported for that version. Note the wire frame smiley face represents vertices!

		Direct3D 8.0 ... 8.1	
Instruction		**Version 1.0**	**1.1**
Assembly (scripting) Command			
vs	Version (Vertex Shader)	☺	☺
def	Definition of a constant	☺	☺
Data Conversions			
mov	Copy	☺	☺

Instruction		Direct3D Version	8.0 1.0	... 8.1 1.1
arl	Address Register Load			☺
frc	*Fractional portion of float*		☺	☺
Add/Sub/Mul/Div				
add	Addition		☺	☺
sub	Subtraction		☺	☺
mul	Multiply		☺	☺
mad	Multiply–Add		☺	☺
dp3	Dot Product (Vec)		☺	☺
dp4	Dot Product (QVec)		☺	☺
rcp	Reciprocal		☺	☺
Special Functions				
min	Minimum		☺	☺
max	Maximum		☺	☺
slt	Set if (<)		☺	☺
sge	Set if (>=)		☺	☺
rsq	Reciprocal Square Root		☺	☺
dst	Distance Vector		☺	☺
expp	Exponential 2^x		☺	☺
exp	*Exponential* 2^x full precision		☺	☺
lit	Lighting		☺	☺
logp	$Log_2(x)$ partial		☺	☺
log	*$Log_2(x)$ full precision*		☺	☺
Matrices				
m3x2	*Apply 3x2 matrix to vector*		☺	☺
m3x3	*Apply 3x3 matrix to vector*		☺	☺
m3x4	*Apply 3x4 matrix to vector*		☺	☺
m4x3	*Apply 4x3 matrix to vector*		☺	☺
m4x4	*Apply 4x4 matrix to vector*		☺	☺

Before we dive into the deep end of the pool, I should reiterate that this chapter is not about special effects or any other vertex manipulations, but it shows the processing functionality of vectors within the programmable Graphics Processor Unit (GPU).

Vertex Shader Definitions

vs — Definition for the version of the code written for the vertex shader

vs.*MajVer.MinVer*

This is an assembly language definition and not an instruction. It is for setting the version that the code was written for and must be the first declaration in a code fragment. *MajVer* is the major version number, and *MinVer* is the minor version number of the vertex shader for which

the code is targeted — current range {1.0, ..., 1.1}. There can only be one version definition per code block.

Example code:

```
vs.1.1                          // Uses 1.1 Vertex Shader Code
```

def — Definition of a constant

def *Dst, aSrc, bSrc, cSrc, dSrc*

This declaration is used to define values within the constant registers by the code of the vertex shader code before it is executed. This instruction must occur after the version instruction but before any arithmetic instruction. This is not a programming instruction but a definition, and so it does not use up any of the 128 instruction code space. The constant value can only be read by the shader code and not written.

Example code:

```
vs   1.1                          // Version 1.1
def  c0, 1.0f, 0.0f, 1.0f, 0.0f   // Set c0 register {1,0,1,0}
```

An alternative to this is writing or reading the value directly by using the provided API for access from the application by the CPU.

Vertex Shader Assembly

So let's peek at the file architecture for this graphics processor assembly language. A vertex shader script can exist in a *.vsh file. As such, it would be ordered similar to the following:

Listing 15-1: DXSDK\samples\Multimedia\Media\fogshader.vsh

```
   // Fog Shader code sample from DirectX SDK 8.1
   // v0=vector, c8,c9,c10,c11=matrix, c12=limit

vs.1.0                          // Version 1.0
def c40, 0.0f,0.0f,0.0f,0.0f    // c40={0,0,0,0}

m4x4 r0,v0,c8          // r = v0 × [c8,c9,c10,c11]        D=AB
  // r0x = (v0x * c8x)  + (v0y * c8y)  + (v0z * c8z)  + (v0w * c8w)
  // r0y = (v0x * c9x)  + (v0y * c9y)  + (v0z * c9z)  + (v0w * c9w)
  // r0z = (v0x * c10x) + (v0y * c10y) + (v0z * c10z) + (v0w * c10w)
  // r0w = (v0x * c11x) + (v0y * c11y) + (v0z * c11z) + (v0w * c11w)
                       // saturate low of 0.0
                       // r0z = (c40z > r0z) ? c40z : r0z
max r0.z,c40.z,r0.z    // r0z = (0      > r0z) ?   0    : r0z
  // clamp {w} to near clip plane
max r0.w,c12.x,r0.w    // r0w = (c12x > r0w) ? c12x : r0w
mov oPos,r0            // oPosxyzw = r0xyzw
```

```
add r0.w,r0.w,–c12.x  // r0_w = r0_w – c12_x
  // Load into diffuse
mul r0.w,r0.w,c12.y   // r0_w = r0_w × c12_y

  // Set diffuse color register
mov oD0.xyzw,r0.w     // oD0_x = oD0_y = oD0_z = oD0_w =    r0_w
mov oT0.x,r0.w        ;  oT0_x = r0_w      Set 2D texture X
mov oT0.y,c12.x       // oT0_y = c12_x     Set 2D texture Y
```

Note that either the assembly language comment specifier of ";" or the C++ style of comment "//" can be used for remarks. But for reasons of cosmetics, you should really not mix and match. Use one or the other but not both!

Vertex Shader Instructions (Data Conversions)

mov — Copy register data to register (d = a)

mov *Dst*, *aSrc*	1.0	1.1
	☺	☺

This instruction moves the referenced source register *aSrc* to the destination register *Dst*.

Pseudo code:

```
#ifdef (1.0 == Version)
        d_w=a_w     d_z=a_z     d_y=a_y     d_x=a_x
#else
    if (a0  Src)     // If not the register index offset.
        d_w=a_w     d_z=a_z     d_y=a_y     d_x=a_x

    else
        *(int*)&d_x = int( floor( a0_x ))

#endif
```

Example code:

```
mov oD0, c0
mov oD0, r4
mov r1,  c9
mov r0.xz, v0.xxyy        ; r0_x = v0_x,   r0_z = v0_y
mov r0.yw, c4.yyww        ; r0_y =  0.0,   r0_w =  1.0
```

arl — Address register load a0

arl *Dst*, *aSrc*	1.0	1.1
		☺

Write register base offset for use in referencing constant registers.
This is not available in 1.0 .. 1.3

frc — Return fractional component of each input of {XY}.

frc *Dst*, *aSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro removes the integer component from the source *aSrc*, leaving the fractional component of the {XY} elements, which is stored in the destination *Dst*.

```
  123.456
– 123.0
---------
    0.456
```

Pseudo code:

```
dx = ax – floor(ax)
dy = ay – floor(ay)
 // The {ZW} elements are ignored.
```

Example code:

```
frc r0.xy, r0.x    ; Fraction of {X} is stored in {XY}
```

Vertex Shader Instructions (Mathematics)

add — Addition (d = a + b)

add *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction sums each of the specified elements of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

Pseudo code:

```
dw=aw+bw   dz=az+bz   dy=ay+by   dx=ax+bx
```

Example code:

```
add r0, r0, –c24
add r0, c23.x, r3
add oD0,r0,  r1
add r4.x, r4.x, c7.x
```

sub — Subtraction (d = a – b)

sub *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction subtracts each of the specified elements of the source *bSrc* from the source *aSrc* and stores the result in the destination *Dst*.

Pseudo code:

$d_w=a_w-b_w$ $d_z=a_z-b_z$ $d_y=a_y-b_y$ $d_x=a_x-b_x$

Example code:

```
sub r2, r1, r0
sub r0, c23.y,r3
sub oD0,r0, r1
```

mul — Multiply (d = ab)

mul *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in the product of each of the specified elements of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

Pseudo code:

$d_w=a_wb_w$ $d_z=a_zb_z$ $d_y=a_yb_y$ $d_x=a_xb_x$; mul r2,r0,r1
$d_z=a_zb_z$ $d_y=a_yb_y$ $d_x=a_xb_x$; mul r0.xyz, r0.xyz, r11.xyz

Example code:

```
mul r2, r0, r1
mul r2, r2, r2
mul r5, r5, c15
mul r0.xyz, r0.xyz, r11.xyz
```

The equation of squares is extremely simple here, as it is the product of itself!

```
mul r0,r0,r0            ; {r0w^2 r0z^2 r0y^2 r0x^2}
```

mad — Multiply/Add (d = ab + c)

mad *Dst*, *aSrc*, *bSrc*, *cSrc*	1.0	1.1
	☺	☺

This instruction results in the product of each of the specified elements of the source *aSrc* and the source *bSrc*, then sums the elements of the source *cSrc* and stores the result in the destination *Dst*.

Pseudo code:

$d_w=a_wb_w+c_w$ $d_z=a_zb_z+c_z$ $d_y=a_yb_y+c_y$ $d_x=a_xb_x+c_x$

Example code:

```
mad oT0.xyz, r2, r0, r1
mad oT0,     r1, c8, c3
```

Note that the elements can be negated as a whole and/or crossed:

```
mul r0, r7.zxyw,  r8.yzxw
mad r5, r7.yzxw, –r8.zxyw, r0
```

...so the quad vector multiply: mul r0, r7.zxyw, r8.yzxw

$r0_x = r7_z r8_y$
$r0_y = r7_x r8_z$
$r0_z = r7_y r8_x$
$r0_w = r7_w r8_w$

...followed by the multiply-add: mad r5, r7.yzxw, –r8.zxyw, r0

$r5_x = -r7_y r8_z + r0_x$
$r5_y = -r7_z r8_x + r0_y$
$r5_z = -r7_x r8_y + r0_z$
$r5_w = -r7_w r8_w + r0_w$

...should look very familiar to you! I hope it does, as it is the cross product!

$D_x = A_y B_z – A_z B_y$	Dx = Ay*Bz – Az*By;
$D_y = A_z B_x – A_x B_z$	Dy = Az*Bx – Ax*Bz;
$D_z = A_x B_y – A_y B_x$	Dz = Ax*By – Ay*Bx;
$D_w = 0.0$	Dw = 0.0;

dp3 — Three-element {XYZ} dot product (d = a • b)

dp3 *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in the dot product of the source *aSrc.xyz* and the source *bSrc.xyz* and stores the replicated scalar result in each element of the destination. The default is *Dst.xyzw*. See m3x3, m3x3, and m3x4 for use of this instruction in 3xn matrix operations.

Pseudo code:

```
dw=dz=dy=dx=   ax*bx + ay*by + az*bz          ; dp3   d,    a, b
dx=            ax*bx + ay*by + az*bz          ; dp3   d.x, a, b
```

Example code:

```
dp3 r2,r0,r1
dp3 r11.x,r0,r0          ; r11x = r0xr0x + r0yr0y + r0zr0z
```

dp4 — Four-element {XYZW} dot product (d = a • b)

dp4 *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in the dot product of the source *aSrc.xyzw* and the source *bSrc.xyzw* and stores the replicated scalar result in each element of the destination. The default is *Dst.xyzw*. See m4x3 and m4x4 for use of this instruction in 4xn matrix operations.

Pseudo code:

```
dw=dz=dy=dx=   ax*bx + ay*by + az*bz + aw*bw      ; dp4   d,   a, b
dy=            ax*bx + ay*by + az*bz + aw*bw      ; dp4   d.y, a, b
```

Example code:

```
dp4 r2,   r0,r1
dp4 r5.y, v0, c3      ; r5y=   v0x*c3x + v0y*c3y + v0z*c3z + v0w*c3w
```

rcp — Reciprocal of the source scalar (d = 1/a)

rcp *Dst*, *aSrc*	1.0	1.1
	☺	☺

This instruction results in the reciprocal of the source *aSrc* and stores the replicated scalar result in each specified element of the destination. Special case handling is utilized if a source is equal to 1.0 or 0.0. The default is *Dst.xyzw*, Src.x

Pseudo code:

```
if (0.0 == ax)                          // 1/0 Divide by zero
    r =   +∞                            // Positive infinity

else if (1.0 == ax)                     // 1/1 = 1
    r =   1.0

else                                    // 1/x
    r =   1.0/ax

    dw=dz=dy=dx=   r
```

Note that unlike most reciprocal instructions, if the denominator is zero, there is no exception (SNaN or QNaN) and/or the data is not marked as invalid. Instead, it is set as a positive infinity, thus keeping the data valid for additional operations.

Example code:

```
rcp r2,   r0          ; r2w=r2z=r2y=r2x=   1/r0x
rcp r0.z, r0.z
```

```
rcp r1.y,  r1.x
rcp r2.yw, r2.y
rcp r7.w,  r7.w
```

Division — If you recall from Chapter 9, "Vector Multiplication and Division," a division is as simple as multiplying the source value numerator by the reciprocal, which is the denominator:

```
    d = a/b = a * 1/b

rcp r0.x, r2.x          // d_x = 1/b_x
mul r0.x, r1.x, r0.x    // d_x = a_x/b_x = a_x * 1/b_x
```

Vertex Shader Instructions (Special Functions)

min — Minimum (d = (a < b) ? a : b)

min *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in the selection of the lower value from each element of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

Pseudo code:

```
d_x= (a_x < b_x)  ? a_x : b_x
d_y= (a_y < b_y)  ? a_y : b_y
d_z= (a_z < b_z)  ? a_z : b_z
d_w= (a_w < b_w)  ? a_w : b_w
```

Example code:

```
min r2, r0, r1
min r0, r0, c4.y
```

max — Maximum (d = (a > b) ? a : b)

max *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in the selection of the higher value from each element of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

Pseudo code:

```
d_x= (a_x > b_x)  ? a_x : b_x
d_y= (a_y > b_y)  ? a_y : b_y
d_z= (a_z > b_z)  ? a_z : b_z
d_w= (a_w > b_w)  ? a_w : b_w
```

Example code:

```
max r2, r0, r1
max r0, r0, c4.y
```

slt — Set if less than (d = (a < b) ? 1.0 : 0.0)

slt *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in a comparison of each selected element of the source *aSrc* and the source *bSrc* and stores 1.0 if less than and 0.0 if not as the result in the destination *Dst*.

Pseudo code:

```
dx= (ax < bx)  ? 1.0 : 0.0
dy= (ay < by)  ? 1.0 : 0.0
dz= (az < bz)  ? 1.0 : 0.0
dw= (aw < bw)  ? 1.0 : 0.0
```

Example code:

```
slt r2,c4,r0
```

sge — Set if greater or equal than (d = (a >= b) ? 1.0 : 0.0)

sge *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction results in a comparison of each element of the source *aSrc* and the source *bSrc* and stores 1.0 if greater than or equal to and 0.0 if not as the result in the destination *Dst*.

Pseudo code:

```
dx= (ax >= bx)  ? 1.0 : 0.0
dy= (ay >= by)  ? 1.0 : 0.0
dz= (az >= bz)  ? 1.0 : 0.0
dw= (aw >= bw)  ? 1.0 : 0.0
```

Example code:

```
sge r2,c2,r3
```

rsq — Reciprocal square root of the source scalar (d = $1/\sqrt{a}$)

rsq *Dst*, *aSrc*	1.0	1.1
	☺	☺

This instruction results in the reciprocal square root of the source *aSrc* and stores the replicated scalar result in each element of the destination.

Special case handling is utilized if the source is equal to 1.0 or 0.0. The default is *Dst.xyzw*, *aSrc.x*.

Pseudo code:

```
if (0.0 == a_x)                          // 1/0   Divide by zero
    r =   +∞                             // Positive infinity

else if (1.0 == a_x)                     // 1 = 1/1 = 1/√1
    r =   1.0

else                                     // 1/√1
    r =   1.0/√a_x

    d_w=d_z=d_y=d_x =   r
```

Example code:

```
rsq r1,    c3          //  r1_w = r1_z = r1_y = r1_x =    1/√c3_x
rsq r1.y, c3.y         //           r1y =                 1/√c3y
```

Square Root — If you remember your square root formulas from Chapter 10, "Special Functions," you should remember that multiplying a reciprocal square root by the original number returns a square root! $\sqrt{x} = x * 1/\sqrt{x}$

```
rsq r1.x, c3.x         //  1/√b_x
mul r1.x, r1.x, c3.x   //  a_x/√b_x
```

dst — Distance vector

dst *Dst*, *aSrc*, *bSrc*	1.0	1.1
	☺	☺

This instruction calculates the distance between the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

The *aSrc* is assumed to be the source vector $\{\#,d^2,d^2,\#\}$ and *bSrc* is the vector $\{\#,1/d,\#,1/d\}$, and the result *Dst* is $\{1,d,d^2,1/d\}$. The # symbol is a "I do not care!"

Pseudo code:

```
d_x=1.0     d_y=a_y b_y     d_z=a_z     d_w=b_w
```

Example code:

```
   // Find the distance from v1 to the origin {0,0,0}.
mov r1.xyz, v1.xyz          // vec = {#ZYX} Position
dp3 r1.yz, r1, r1           // d={ZY##} = sum of squares
rsq r2.y, r1.y              // {# # 1/√d #}
rcp r2.yw, r2.y             // {√d √d # #} = 1/(1/√d)
dst r0,r1,r2                //  = r1_#yz#  r2_#yw#
```

expp — Exponential 2^x — precision 10 bit

expp *Dst*, *aSrc*	1.0	1.1
	☺	☺

This instruction calculates the exponential number using the source *aSrc* and stores the result in the destination *Dst*.

Pseudo code:

```
uint32 m

w = floor(a_w)
t   = pow(2, a_w)
d_x = pow(2, w)
d_y = a_w – w

    // Reduced precision exponent
m = *((uint32*)&t) & 0xffffff00
d_z = *(float*)&m
d_w = 1.0
```

Example code:

```
expp r1.x, c6.y
expp r5.yw, r5.xxxx
```

exp — Exponential 2^x — precision 19 bit

exp *Dst*, *aSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro calculates the exponential number using the source *aSrc* and stores the result in the destination *Dst*. See *expp*.

Pseudo code:

```
d_x = d_y = d_z = d_w = pow(2, a_w)
```

Example code:

```
exp r1.x,   c6.y
```

lit — Lighting coefficients — reduced precision

lit *Dst*, *aSrc*	1.0	1.1
	☺	☺

This instruction calculates the lighting coefficient for the source *aSrc* using two dot products and an exponent and stores the result in the destination *Dst*.

Pseudo code:

```
const float MAXPOWER = 127.9961

a_x = Normal•LightVector
a_y = Normal•HalfVector
a_z = 0.0
a_w = exponent      ; Exponent of  –128.0 ... 128.0

  // The following code fragment shows the operations performed.
d_x = d_w = 1.0
d_y = d_z = 0.0

power = a_w

if ((power < –MAXPOWER) || (MAXPOWER < power))
  power = –MAXPOWER                              // 8.8 fixed point format

if (0.0 < a_x)                                   // positive
{
  d_y = a_x
                                  // Allowed approx. is EXP(power * LOG(a_y))
  if (0.0 < a_y)  d_z = pow(a_y, power)           // positive
}
```

logp — $\log_2(x)$ — precision 10 bit

logp *Dst*, *aSrc*	1.0	1.1
	☺	☺

This instruction calculates a partial log using the source *aSrc* and stores the result in the destination *Dst*.

Pseudo code:

```
v = |a_w|

if (0.0  v)
{
    int i = (int)(*(DWORD*)&v >> 23) – 127

    d_x = (float)I                                // exponent
    i = (*(uint*)&v & 0x7FFFFF) | 0x3f800000
    d_y =  *(float*)&i                            // mantissa

    v = log (v) / log (2.0)
    i = *(uint*)&v & 0xffffff00

    d_z = *(float*)&i;
    d_w = 1.0
}
else                                              // a_w is zero!
{
    d_x = d_z = MINUS_MAX()
    d_y = d_w = 1.0
}
```

Example code:

```
logp r0, r0.w
```

log — $\log_2(x)$ full precision

log *Dst*, *aSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro calculates a full precision log using the source *aSrc* and stores the result in the destination *Dst*.

Pseudo code:

```
v = |aSrc.w|

if (0.0 != v)
  v = (log (v)/log (2))

else
  v = MINUS_MAX()

Dst.x = Dst.y = Dst.z = Dst.w = v
```

Example code:

```
log r0, r0.w
```

Vertex Shader Instructions (Matrices)

m3x2 — Apply 3x2 matrix to vector (d = aB)

m3x2 *Dst*, *aSrc*, *bSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro applies a 3x2 matrix of the two sequential registers, beginning with the source *bSrc* {+0, +1} to the {XYZ} vector referenced by the source *aSrc*, and stores the result in the destination vector *Dst*.

Pseudo code:

In the following, *a* is the vector, *b*[0] is the first row, and *b*[1] is the second row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[0]_y) + (a_z * b[0]_z)$$
$$d_y = (a_x * b[1]_x) + (a_y * b[1]_y) + (a_z * b[1]_z)$$

Macro equivalent:

```
m3x2  r5,v0,c3

dp3 r5.x, v0, c3   ; 1st row
dp3 r5.y, v0, c4   ; 2nd row
```

Example code:

```
m3x2 r5,v0,c3          ;c3 1st row, c4 2nd row
```

m3x3 — Apply 3x3 matrix to vector (d = aB)

m3x3 *Dst*, *aSrc*, *bSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro applies a 3x3 matrix referenced by the three sequential registers, beginning with the source *bSrc* {+0, +1, +2} to the {XYZ} vector referenced by the source *aSrc*, and stores the result in the destination vector *Dst*.

Pseudo code:

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, and *b*[2] is the third row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[0]_y) + (a_z * b[0]_z)$$
$$d_y = (a_x * b[1]_x) + (a_y * b[1]_y) + (a_z * b[1]_z)$$
$$d_z = (a_x * b[2]_x) + (a_y * b[2]_y) + (a_z * b[2]_z)$$

Macro equivalent:

```
m3x3  r5,v0,c3

dp3 r5.x, v0, c3   ; 1st row
dp3 r5.y, v0, c4   ; 2nd row
dp3 r5.z, v0, c5   ; 3rd row
```

Example code:

```
m3x3 r5,v0,c3          ;c3 1st row, c4 2nd row, c5 3rd row
m3x3 r8,v3,c0
```

m3x4 — Apply 3x4 matrix to vector (d = aB)

m3x4 *Dst*, *aSrc*, *bSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro applies a 3x4 matrix referenced by the four sequential registers, beginning with the source *bSrc* {+0, ..., +3} to the {XYZ} vector referenced by the source *aSrc*, and stores the result in the destination vector *Dst*.

Pseudo code:

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, *b*[2] is the third row, and *b*[3] is the fourth row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[0]_y) + (a_z * b[0]_z)$$
$$d_y = (a_x * b[1]_x) + (a_y * b[1]_y) + (a_z * b[1]_z)$$

$d_z = (a_x * b[2]_x) + (a_y * b[2]_y) + (a_z * b[2]_z)$
$d_w = (a_x * b[3]_x) + (a_y * b[3]_y) + (a_z * b[3]_z)$

Macro equivalent:

```
m3x4  r5,v0,c3

dp3 r5.x, v0, c3   ; 1st row
dp3 r5.y, v0, c4   ; 2nd row
dp3 r5.z, v0, c5   ; 3rd row
dp3 r5.w, v0, c6   ; 4th row
```

Example code:

```
m3x4 r5,v0,c3       ;c3 1st row, c4 2nd row, c5 3rd row, c6 4th row
```

m4x3 — Apply 4x3 matrix to vector (d = aB)

m4x3 *Dst*, *aSrc*, *bSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro applies a 4x3 matrix referenced by the three sequential registers, beginning with the source *bSrc* {+0, +1, +2} to the {XYZW} vector referenced by the source *aSrc*, and stores the result in the destination vector *Dst*.

Pseudo code:

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, and *b*[2] is the third row of the matrix.

$d_x = (a_x * b[0]_x) + (a_y * b[0]_y) + (a_z * b[0]_z) + (a_w * b[0]_w)$
$d_y = (a_x * b[1]_x) + (a_y * b[1]_y) + (a_z * b[1]_z) + (a_w * b[1]_w)$
$d_z = (a_x * b[2]_x) + (a_y * b[2]_y) + (a_z * b[2]_z) + (a_w * b[2]_w)$

Macro equivalent:

```
m4x3  r5,v0,c3

dp4 r5.x, v0, c3   ; 1st row
dp4 r5.y, v0, c4   ; 2nd row
dp4 r5.z, v0, c5   ; 3rd row
```

Example code:

```
m4x3 r5,v0,c3          ;c3 1st row, c4 2nd row, c5 3rd row
```

m4x4 — Apply 4x4 matrix to vector (d = aB)

m4x4 *Dst*, *aSrc*, *bSrc*	1.0	1.1
(**MACRO**)	☺	☺

This macro applies a 4x4 matrix referenced by the four sequential registers, beginning with the source *bSrc* {+0, ..., +3} to the {XYZW} vector

referenced by the source *aSrc*, and stores the result in the destination vector *Dst*.

Pseudo code:

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, *b*[2] is the third row, and *b*[3] is the fourth row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[0]_y) + (a_z * b[0]_z) + (a_w * b[0]_w)$$
$$d_y = (a_x * b[1]_x) + (a_y * b[1]_y) + (a_z * b[1]_z) + (a_w * b[1]_w)$$
$$d_z = (a_x * b[2]_x) + (a_y * b[2]_y) + (a_z * b[2]_z) + (a_w * b[2]_w)$$
$$d_w = (a_x * b[3]_x) + (a_y * b[3]_y) + (a_z * b[3]_z) + (a_w * b[3]_w)$$

Macro equivalent:

```
m4x4  r5,v0,c3

dp4 r5.x, v0, c3   ; 1st row
dp4 r5.y, v0, c4   ; 2nd row
dp4 r5.z, v0, c5   ; 3rd row
dp4 r5.w, v0, c6   ; 4th row
```

Example code:

```
m4x4 r5,v0,c3       ;c3 1st row, c4 2nd row, c5 3rd row, c6 4th row
```

Normalization

Let's carry this another step forward to the normalization of numbers. Earlier in this chapter a reciprocal was shown as a division:

```
rcp r0.x, r2.x          // d_x = 1/b_x
mul r0.x, r1.x, r0.x    // d_x = a_x/b_x = a_x * 1/b_x
```

And a reciprocal square root as a square root:

```
rsq r1.x, c3.x          //  1/√b_x
mul r1.x, r1.x, c3.x    //   ax/√b_x
```

It was not discussed or shown how to handle the divide by zero in both cases. This would normally present a problem because a divide by zero has an invalid solution, and quite often it must be trapped and converted to a value of one, as divide by zero is in essence infinity, and a denominator of zero has an infinitesimal affect on a value. So in essence, the value remains the same.

But for programmable vertex and pixel shaders, they are trapped, and a positive infinity is returned. The interesting thing here is that the product of zero and infinity is zero!

Hint: The product of zero and any value (including infinity) is zero!

```
if (0.0 == a_w)                          // 1/0   Divide by zero
    r =   +∞                             // Positive infinity

else if (1.0 == a_w)                     // 1/1 = 1
    r =   1.0

else                                     // 1/x or 1/√x
    r =   1.0/a_w          or            r =   1.0/sqrt(a_w)
```

They both require a reciprocal — in essence, a division. But that presents the old problem of a divide by zero. The coding of quaternions, such as in the case of an element being too close to zero, was discussed in Chapter 12, "Matrix Math." When dealing with a normalization, losing precision in the process of manipulating pixels is not a problem.

```
dp4  r0.w, r1.xyz, r1.xyz      // d_w=a_x²+a_y²+a_z²
rsq  r0.w, r0.w                // d_w = 1/√d_w
mul  r0.xyz, r1.xyz, r0.w      // {a_z(1/d_w) a_y(1/d_w) a_x(1/d_w)}
```

If precision is required because the output of the normalization is being used for additional calculations, then use limits to trap for when the denominator approaches zero. To resolve this problem, there is no branching or Boolean masking, to the detriment of this assembly code. Hopefully, those instructions will be added to a new version soon. And yet, a normalization is needed, so what can be done?

A comparison in conjunction with a constant can be utilized! Remember that the sum of squares is never a negative number, so the value is {0.0, ..., r}!

slt — Compare less than (d = (a < b) ? 1.0 : 0.0)

```
def c0, 0.0000001, 0.0000001, 0.0000001, 0.0000001

  // "r1" too close to zero?
slt r3, r1, c0           // s = (a < 0.0000001) ? 1.0 : 0.0
sge r4, r1, c0           // t = (a >= 0.0000001) ? 1.0 : 0.0

  // Complement masks, d[]=1 if too close to zero, t[]=1 if not!
  // If too small, 1.0 = (0.00000001*0)+1.0
  // If okay a = (a * 1) + 0
mad r0, r1, r4, r3       // d[] = (a[] * t[]) + s[]
```

Quaternions

As the children's story, *The Monster Bed*, goes: "'I'm frightened! I'm frightened!' the wee monster said." Quaternions are no monster. In fact, they ought to be fun and simple! This should be a rehash of the quaternions explained in Chapter 13, "Quaternion Math."

$$q = w + xi + yj + zk$$

Some quaternions are the same as quad vectors, such as in the following table:

Table 15-3: Quaternion to vertex shader assembly instruction similarities

Qmov	mov	Move	d=a
		$= a_w + a_x\mathbf{i} + a_y\mathbf{j} + a_z\mathbf{k}$	
Qadd	add	Addition	d=a+b
		$= a_w+b_w + (a_x+b_x)\mathbf{i} + (a_y+b_y)\mathbf{j} + (a_z+b_z)\mathbf{k}$	
Qsub	sub	Subtraction	d=a–b
		$= a_w-b_w + (a_x-b_x)\mathbf{i} + (a_y-b_y)\mathbf{j} + (a_z-b_z)\mathbf{k}$	
Qdp	dp4	Dot Product	d=a•b
		$= a_wb_w + (a_xb_x)\mathbf{i} + (a_yb_y)\mathbf{j} + (a_zb_z)\mathbf{k}$	

With some quaternions, it is merely a matter of swizzling elements with standard assembly: r0=*Dst*, r1=*aSrc*.

Quaternion Conjugate

$$\bar{q} = w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$$

Listing 15-2

```
mov  r0,  –r1            // {–a_w  –a_z  –a_y  –a_x}
mov  r0.w, r1.w          // { a_w  –a_z  –a_y  –a_x}
```

Quaternion Multiplication

$$\begin{aligned} &= w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2 \\ &+ (w_1x_2 + x_1w_2 - z_1y_2 + y_1z_2)\mathbf{i} \\ &+ (w_1y_2 + y_1w_2 + z_1x_2 - x_1z_2)\mathbf{j} \\ &+ (w_1z_2 + z_1w_2 + x_1y_2 - y_1x_2)\mathbf{k} \end{aligned}$$

Listing 15-3

```
// d_x = (a_x * b_w) + (a_y * b_z) – (a_z * b_y) + (a_w * b_x)
// d_y = (a_y * b_w) – (a_x * b_z) + (a_w * b_y) + (a_z * b_x)
// d_z = (a_z * b_w) + (a_w * b_z) + (a_x * b_y) – (a_y * b_x)
// d_w = (a_w * b_w) – (a_z * b_z) – (a_y * b_y) – (a_x * b_x)

mul r3, r1.yxwz, r2.z
mul r4, r1.zyxy, r2.y
mul r5, r1.wzyx, r2.x

mov r3.yw, –r3.yw
mov r4.xw, –r4.xw
mov r5.zw, –r5.zw

mad r0, r1, r2.w, r3
add r0, r0, r4
add r0, r0, r5
```

The rest of these quaternions appear at first glance to have a wee bit of a problem. Or do they? They require a reciprocal — in essence, a division. But that presents the old problem of a divide by zero. For the coding of quaternions discussed in Chapter 13, "Quaternion Math," there is that case of an element being too close to zero, but we can use what we learned previously in the normalization section of this chapter.

Quaternion Normalization

Now notice that same logic used in the following code! When the denominator is zero, the positive infinity is used to its advantage.

$$norm(q) = w + xi + yj + zk$$

Listing 15-4

```
dp4  r0.w, r1, r1          // dw=ax²+ay²+az²+aw²
rsq  r0, r0.w              // dw = 1/√dw          or   1/+∞ = 1/√0
mul  r0, r1, r0            // {aw(1/dw)  az(1/dw)  ay(1/dw)  ax(1/dw)}
```

Quaternion Magnitude

$$q_1q_2 = \sqrt{(w^2 + x^2\mathbf{i} + y^2\mathbf{j} + z^2\mathbf{k})}$$

Listing 15-5

```
dp4  r0.w, r1, r1          // r=ax²+ay²+az²+aw²
rsq  r0.x, r0.w            // dx = 1/√r
mul  r0.x, r0.x, r0.w      // dx=r*dx
```

Quaternion Inverse

$$q^{-1} = \frac{\bar{q}}{norm(q)} = \frac{w - xi - yj - zk}{w^2 + x^2 + y^2 + z^2}$$

Listing 15-6

```
dp4  r0.w, r1, r1          // dw=ax²+ay²+az²+aw²
rcp  r0, r0.w              // dxyzw = 1/dw
mul  r0, r1, -r0.w         // {-awdw  -azdw  -aydw  -axdw}
mov  r0.w, -r0.w           // { awdw  -azdw  -aydw  -axdw}
```

Pixel Shaders

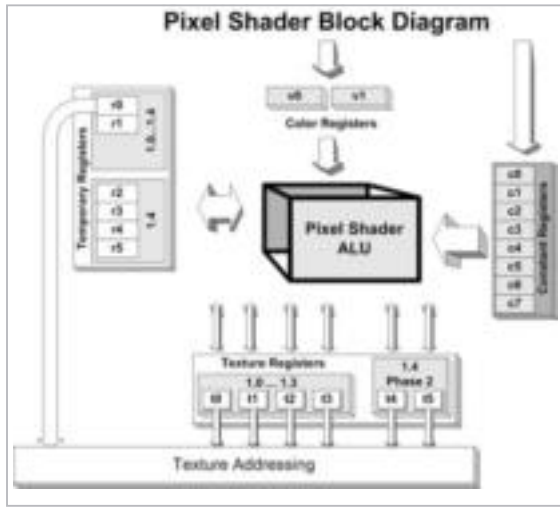


Figure 15-3: Pixel shader block diagram

- *r0 ... r1* and *r2 ... r5* — The first two temporary registers {r0, r1} are used as scratch registers to temporarily save vertex data in various stages of processing for version {1.0 ... 1.3}. The last four temporary registers {r2, ..., r5} are used for version 1.4.
- *c0 ... c7* — The eight constant read-only registers are each a quad single-precision floating-point vector. They are set either from the use of the *def* instruction or from calling an external function from an application. These can only be read by the shader code or from the game application through an API interface.
- *v0 ... v1* — The two read-only color registers, *v0* (the diffuse) and *v1* (the specular)
- *t0 ... t3* — For pixel shader version 1.1 to 1.3
- *t4 ... t5* — For pixel shader version 1.4

Note in the following table that there are differences between versions 1.0 through 1.3. This was supported by nVIDIA and the introduction of the ATi chipset that covers that same range of versions as well as the addition of version 1.4, which handles textures differently.

Table 15-4: Programmable vertex instructions and their relationship with the version of Direct3D and pixel shader versions. The) indicates the instruction is supported for that version. The solid smiley face represents texturing.

		Direct3D	8.0		8.1		
Instruction		**Version**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**
Assembly (Scripting) Command							
ps	Version (Pixel Shader)		)	)	)	)	)
def	Definition of a Constant		)	)	)	)	)
Data Conversions							
mov	Copy		)	)	)	)	)
nop	No operation		)	)	)	)	)
texld	Loads RGBA using Tex. Coords.						)
Add/Sub/Mul/Div							
add	Addition		)	)	)	)	)
sub	Subtraction		)	)	)	)	)
mul	Multiply		)	)	)	)	)
mad	Multiply–Add		)	)	)	)	)
lrp	Linearly Interpolate by Proportion		)	)	)	)	)
dp3	Dot Product (Vec)		)	)	)	)	)
dp4	Dot Product (QVec)				)	)	)
texdp3	Texture dot product				)	)	
texdp3tex	Texture dot product for 1D tex tbl.				)	)	
Special Functions							
phase	Transition between phase 1 and 2.						)
bem	Bump Environment Map xform						)
cmp	Comparison (#>=0)				)	)	)
cnd	Comparison (r0.a > 0.5) ? b : c		)	)	)	)	)
	Comparison (a > 0.5) ? b : c						)
Texture							
tex	Load Color RGBA from texture.		)	)	)	)	
texcoord	Interp. TexCoord. UVW1.		)	)	)	)	
texbem	Apply fake bump map xform.		)	)	)	)	
texbeml	Apply fake bump map xform w/lum.		)	)	)	)	
texcrd	Copy TexCoord. UVW1.						)
texdepth	Calc. Depth values.						)
texkill	Cull pixel if any of (UVW) is zero.		)	)	)	)	)
texreg2ar	Intrp. Alpha & Red.		)	)	)	)	
texreg2gb	Intrp. Green & Blue.		)	)	)	)	
texreg2rgb	Intrp. Red, Green & Blue.				)	)	
Texture Matrices							
texm3x2depth	Calc. Depth to test pixel.					)	
texm3x2pad	1st row mul. of 2 row mtx mul.		)	)	)	)	
texm3x2tex	Last row mul. of 3x2 mtx mul.		)	)	)	)	
texm3x3pad	1st or 2nd row of 3x3 mul.		)	)	)	)	
texm3x3	3x3 matrix mul. Use w/'3x3pad'.				)	)	
texm3x3tex	3x3 matrix mul. Use for tex tbl.		)	)	)	)	

		Direct3D	8.0		8.1		
Instruction		**Version**	**1.0**	**1.1**	**1.2**	**1.3**	**1.4**
texm3x3spec	3x3 matrix mul. Specular reflect		)	)	)	)	
texm3x3vspec	3x3 matrix mul. Var. Spec. reflect		)	)	)	)	

There are similarities as well as multiple differences between the vertex shader and pixel shader instruction sets. As this book is about vector processing and not about vertex and/or pixel shaders, it will not delve any deeper into the realm of either. The purpose for its inclusion here is to utilize vector processing as part of its instruction sets and also to whet your appetite. Also, the same techniques shown in the vertex shader section of this chapter apply directly to this section on pixel shaders.

If you wish for more information, either watch for and purchase a copy of my next book related to this subject matter from an introductory point of view or purchase a copy of Wolfgang F. Engel's *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks* from Wordware Publishing.

There are a multitude of other equations that you can write since this chapter was just a sampling. So take some time and experiment! Check out the functionality of the pixel shader functions and practice writing algorithms. Just remember that each vertex and pixel will be calling the function, so you need to keep it as small and lightweight as possible. Some chips may have parallel pipelines, but be efficient anyway!

So where do we go from here? Is there more that can be done with these instructions? Check out the document, "Where Is that Instruction? How to Implement 'Missing' Vertex Shader Instructions!," which is available for download from the nVIDIA web site.

Download all the SDKs and tools and experiment. Above all, practice, practice, practice! Make this one of your specialties for when you go on job interviews. Or forget everything you have read so far in this book. Being a 3D programmer is nothing more then calling APIs that someone else wrote anyway, right? You do not need to know vectors because someone else can always construct the mathematic sections of the game code.

Exercises

1. Which registers can only be read by the vertex shader?
2. Which registers can be written to by the vertex shader?
3. What is the advantage of having 256 constant registers versus 96?
4. Write a VSH file definition to indicate the code is written for version 2.0.
5. Write a function to copy vector a_{xyzw} to b_{wyzx}.
6. Write a function to calculate the determinate of a 2x2 XY matrix and a 3x3 XYZ matrix.
7. Write a CosAng (Cosine Angle) function that uses two vectors similar to that in Chapter 11 but using vertex shader assembly code.
8. Write a three-element dot product without using the *dp3* instruction.
9. Write a four-element dot product without using the *dp4* instruction.
10. What is the result for all input values of r0?

```
def c0, 0.0, 0.0, 0.0, 0.0
slt r0, r0, c0
slt r0, –r0, c0
add r0, r0, –r1
```

Chapter 16
Video Codec

CD Workbench Files: /Bench/*architecture*/chap16/*project*/*platform*

architecture		*project*		*platform*
PowerPC	/vmp_ppc/	**MotionComp.**	/MComp/	/mac9cw
X86	/vmp_x86/	**YUV12 to RGB**	/yuv12/	/vc6
MIPS	/vmp_mips/			/vc.net
				/devTool

Computer animation technology has been around for some time now. Not to over simplify it, but multiple companies have come up with their own CODEC (compression and decompression) schemes to save image information in a non-lossy file format, but in the end the file becomes too large. Photographic images require too much storage space when digitized and stored onto a computer in digital form. To help resolve this, the JPEG (Joint Photographic Experts Group) created a still image compression method in 1991 with the same name.

This uses a lossy method to allow a loss of exactness of picture in an attempt to reduce the amount of storage space required but still maintain a degree of picture quality by comparing adjacent pixels and culling those that are similar and close to each other based upon the percentage factor of lossy. If set too low, the image will deteriorate beyond recognition, but this was only designed for single images.

A method was needed to store animations, so a variation of this scheme, MJPEG (Motion-JPEG), was adopted, but this was short lived. Just kidding! Now that I have startled a few of you, the MJPEG is still a viable CODEC; it's just not spoken of as much as when it first came out.

MPEG (Moving Picture Experts Group) is another CODEC that was put out in 1988 and in recent years has been coming on strong and used in increasing numbers of consumer and industrial device applications.

In the arena of listening pleasure, MP3 (MPEG-3) is used for digital playback of audio. There has been a rush to market MP3 playback

devices where hours of favorite audio files can be downloaded to a portable playback unit for one's listening pleasure and convenience.

Even DVD (Digital Versatile Disk) technology uses MPEG video and audio compression. Its large storage capacity of 17 GB is higher than the 540 to 720 MB of a CD-ROM.

With the current release of MPEG-4, there has been a gold rush of better, smaller, and faster compression methods.

Right now, MPEG-4, AVI, and Quick Time MOV files are at the forefront of the video industry as the storage of choice for digital video. Digital video is being used in Internet video, cable television, direct satellite broadcast, and computer games, as well as other avenues of video delivery mechanisms. In some cases, hardwired-based encoders and decoders are utilized and in some cases not. Since this book is oriented toward vector mathematics in video games, that is where our focus will reside.

These days, a game is not a game unless it has movies or FMVs (Full Motion Video), whether they be for an introduction to open the game, to denote the beginning of a level, cut-scenes within a level, a reward for a successful completion of a level, or wallpaper for the game credits. Games that are limited on media space will tend to use choreographed sprites to simulate a movie. If this is the case for your game, just skip this chapter. But if not or if you are genuinely interested in movies or video streams and their CODEC, such as Digital Video (DV), DivX;–), or MPEG, then by all means read on.

Part of my job over the years has been reverse engineering video CODECs and designing and developing new proprietary ones (although the Millennium Copyright Act of 1998 has put a legal kink in that skill set of mine and transferred that sort of engineering work to other countries that do not acknowledge United States laws since they do not apply to them). When I was an employee of a casino gaming company, I developed a graphics engine that had the option of playing a movie in the background like animated wallpaper and would then paint animated graphic text and sprites on the foreground using either a standard off-the-shelf CODEC or a custom proprietary one that I developed. It was a pretty good sight to see. In fact, in one particular case, if you walk into some of the major casinos, you might just catch sight of a 42-inch plasma display as part of a Slot Machine Bonusing System doing just that — running a soccer film clip, a "Three Stooges" film short, a commercial, or a movie clip.

Movies are, in reality, animations that are made up of a series of frames that play back at a particular frame rate. Each frame is

considered to be a key frame or delta frame. A *key frame* contains all the information necessary to generate an image. A *delta frame* merely contains the drawing changes necessary to modify the last frame to appear as the current frame. A movie is considered to be lossy or lossless. A lossy animation is where the frames do not appear exactly as the original, but depending on the loss factor, they might appear similar to the original. The trick is to lose as much data as possible and still look like the original image but be as small as possible. The frames are typically delta-based with periodic key frames to keep the animation within expected parameters. If there are no key frames or lossy frames in use, then by the end of a few seconds, the frames may appear nothing like the original animation. They therefore need to be used at appropriate intervals, keeping in mind that key frames typically require more storage space than delta frames.

This is akin to taking a single compass direction (angular vector) reading in the morning at the beginning of your trek across the southern icepack and then being discovered the next season frozen in the ice due to having missed your target.

On the contrary, if you use waypoints and realign your course periodically, you become very likely to arrive at the destination on target without being lost or bewildered. Each of those waypoints would be considered a key frame.

Recently, I was the lead in the skunk works division of my employer. A co-worker and I were loaned to the Project Mayo group in which we were tasked with assisting in the optimization of their open source DivX;–) compression technology for the Macintosh using PowerPC and AltiVec assembly language. Since it was open source and a vectorizable product, this seemed to be an excellent forum to discuss the optimization of some of its components, and so some MPEG methods will be discussed as vector-based algorithms, drastically increasing their processing speed.

DivX;–) is a legal derivative of the MPEG-4 video compression that uses an MPEG-2 video stream in conjunction with an MPEG 1 Layer 3 (MP3) audio stream without the overhead of its object dynamics. This is a digital alternative for private citizens (or pirates to do their thing) to be able to inexpensively archive their movies.

There are three components that are of interest to us in this book:

- Motion compensation
- Inverse Discrete Cosine Transform
- YUV color conversion

These will be discussed individually in terms of optimization issues and their solutions.

Motion Compensation

Motion compensation is used in both the encode and decode process. In the decode process, which is of interest to us here, it is used for the averaging of pixel macro blocks. Different equations are utilized depending on the desired result. A motion macro block comes in two sizes, an 8x8 byte and a 16x16 byte, and is used for measuring the amount of movement of a related square pixel-sized image.

Okay, okay, if you really want detailed information, you will find additional information in the references section of this book. We are not much interested in the functionality here, only the vectorization of that functionality, although knowing one definitely is an advantage to the implementation of the other.

Horizontal and/or Vertical Averaging with Rounding or Truncation

Note that the following image is a sample 16x16 data macro block from a much larger image. It is a single-byte array captured from an actual DivX;–) video motion compensation decode between two frames. I could have included a frame here, but then I would have had to get permission to print the anime frame, which was not worth the hassle and unimportant to what needs to be discussed. The raw data in text form looks uninteresting and can be found on the companion CD, but it is graphically much more interesting.

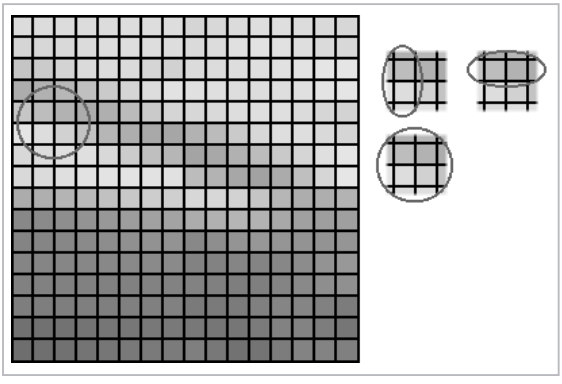


Figure 16-1: A 16x16 array of 8-bit macro blocks

Table 16-1: The value of the encircled 2x2 macro block from Figure 16-1

	1	2
4	0xBD	0xB2
5	0xDD	0xD1

The following are formulas needed for individual processing of those macro blocks:

Horizontal {8x8}, {16x16}

$$\frac{(col[0...n-1]+col[1...n])}{2}$$

$$\frac{(BD+B2)}{2} = B7$$

Horizontal rounding {8x8}, {16x16}

$$\frac{(col[0...n-1]+col[1...n]+1)}{2}$$

$$\frac{(BD+B2+1)}{2} = B8$$

Vertical {8x8}, {16x16}

$$\frac{(row[0...n-1]+row[1...n])}{2}$$

$$\frac{(BD+DD)}{2} = CD$$

Vertical rounding {8x8}, {16x16}

$$\frac{(row[0...n-1]+row[1...n]+1)}{2}$$

$$\frac{(BD+DD+1)}{2} = CD$$

Horizontal and vertical {8x8}, {16x16}

$$\frac{(col[0...n-1]+col[1...n]+row[0...n-1]+row[1...n]+1)}{4}$$

$$\frac{(BD+DD+B2+D1)}{4} = C7$$

Horizontal and vertical rounding {8x8}, {16x16}

$$\frac{(col[0...n-1]+col[1...n]+row[0...n-1]+row[1...n]+2)}{4}$$

$$\frac{(BD+DD+B2+D1+1)}{4} = C7$$

One additional item that should be noted is that neither the 8-byte nor 16-byte rows are guaranteed to be memory aligned. There is nothing that can be done about this, as it is the nature of this use of the algorithm that when a macro block is encoded as it is moving, it is grouped into a fixed 8x8 or 16x16 block form. But the difference of movement between two frames will not be a jump of eight pixels; it will be more along the lines of one or two pixels. As it takes up too much time to align memory before processing it, in this particular case, out-of-alignment logic should be used, as there is little to no control over it. You can, however, predetermine if the first upper-left macro pixel is aligned and, if so, jump to aligned code for the entire block. This will enhance your throughput and gain higher frame rates. However, the cost is an increase in code size to support both methods.

An item to note is that only the "horizontal rounded motion compensation" will be detailed here to help minimize the printed code in this book. Besides, they are all variations of each other, and by now you should have an idea of how to implement them using this one function as a foundation.

Horizontal 8x8 Rounded Motion Compensation

As an example, this will expand upon 8x8 horizontal rounding. Adjacent bytes are averaged for 8+1 columns wide and 8+1 rows high. Please note that an extra column (ninth) and an extra row (ninth) are calculated and thus need to exist. The stack argument *iStride* is (±) signed, so it will advance to the next scan line in a positive or negative direction. For a top down, the *Src* is passed with the beginning of the top scan line. For a bottom up, the *Src* is passed with the beginning of the bottom-most scan line.

Pseudo Vec

The following is the 8x8 horizontal rounding in its simplistic form so it is easily understood. Keep in mind that this implementation is slow!

Listing 16-1: \chap16\MComp\MComp.cpp

```
void vmp_MCompHz8R(uint8 *Dst, uint8 *Src, int iStride)
{
  uint dy, dx;

  ASSERT_PTR(Dst);
  ASSERT_PTR(Src);
  ASSERT_ZERO(iStride);

  for (dy = 0; dy < 8; dy++)
  {
    for (dx = 0; dx < 8; dx++)
    {
      Dst[dx] = (uint8)((Src[dx+0] + Src[dx+1] + 1) >> 1);
    }
     // iStride (+) is top down and (–) is bottom up orientation
    Src += iStride;
    Dst += iStride;
  }
}
```

The following code is a simulated version as well but utilizes Boolean masking and pointer logic to process four array elements simultaneously instead of individually. This code is much faster and is portable to 32-bit processors. The following averaging method was lightly touched on in the section "Vector Integer Addition" in Chapter 8. The idea is that a summation of two 8-bit values actually has a carry to a ninth bit, but if simulating a parallel operation with a packed data width of 16, 32, 64, or 128 bits, the ninth bit actually contaminates the LSB of the adjacent 8-bit value. By using a process of a pre-right shift for each 8-bit value, adding the result, and then adding the logical OR value of the lowest bit of each of the original 8-bit values, the same result can be achieved without affecting the ninth bit.

You may recall the section "Vertical Interpolation with Rounding" in Chapter 4, "Vector Methodologies." The principals are essentially the same.

$$\frac{(col[0\ldots n-1]+col[1\ldots n]+1)}{2} \quad \text{thus} \quad \frac{(A+B+1)}{2}$$

Recall the algebraic law of distribution: (b+c)/a = b/a + c/a

So:

$$\frac{col[0\ldots n-1]}{2} + \frac{col[1\ldots n]}{2} + \frac{1}{2}$$

Each original value can be pre-shifted right by one bit, and the effect of adding one before the division by two is the same as a logical OR of the result of the least significant bit of both values A and B.

(A+B+1)/2	0	1
0	00	01
1	01	01

Thus:

$$\frac{(A+B+1)}{2} = (A+B+1)>>1 = (A>>1) + (B>>1) + ((A\&1) \mid (B\&1))$$

Is it beginning to come back to you? Okay, now the optimized C code!

```
uint eMASK01 = 0x01010101;
uint eMASK7F = 0x7F7F7F7F;
```

In the following listing, note that the eighth column is rounded with the ninth column, and the eighth row is rounded with the ninth row. So when implementing your own algorithm, make sure there is a ninth row and column! Typically, the actual image processing buffer has an extra 16 columns and one extra row to prevent any possible out-of-bounds error.

Listing 16-2: \chap16\MComp\MComp.cpp

```
void vmp_MCompHz8R( uint8 *pbDst, uint8 *pbSrc, int iStride)
{
  register uint32 c0, c1, d0, d1;
  const uint32 *pSrc;
  uint32 *pDst;
  int dy;

  ASSERT_PTR(pbDst);
  ASSERT_PTR(pbSrc);
  ASSERT_ZERO(iStride);

  iStride >>= 2;        // # of 32-bit words

  pSrc = (uint32*) pbSrc;
  pDst = (uint32*) pbDst;
```

Endian-less quad 4x8-bit average (a+b+1)/2 has no carry between 8-bit sections but handles two 32-bit sections and uses the law of distribution:

```
a(b+c)=ab+ac
```

```
  dy = 8;
  do {                  // Horizontal interpolation with rounding
    d0 = *(pSrc+0);
    d1 = *(pSrc+1);

#ifdef VMP_BIG_ENDIAN   // Big endian
    c0 = (d0 << 8) | (d1>>24);
    c1 = (d1 << 8) | (*(pSrc+2)>>24);
#else                   // Little endian
    c0 = (d0 >> 8) | (d1<<24);
    c1 = (d1 >> 8) | (*(pSrc+2)<<24);
#endif
```

Note the masking after the shift to strip the bits! If before the shift, then a value of FE (11111110b) would have been used instead of 7F (01111111b). The point is to get the effect of that LSB cleared!

```
    *(pDst+0) = ((c0>>1) & eMASK7F) + ((d0>>1) & eMASK7F)
              + ((d0 | c0) & eMASK01);
    *(pDst+1) = ((c1>>1) & eMASK7F) + ((d1>>1) & eMASK7F)
              + ((d1 | c1) & eMASK01);
    pSrc += iStride;       // Advance to next scan line
    pDst += iStride;
  } while (--dy);          // Loop for 8 scan lines
}
```

Now note that for the 8x8 arrays, one row is in reality 64 bits wide and not 128 bits, and so some special handling would need to be done for the reading and writing of the values by the processors with 128-bit data registers!

Pseudo Vec (X86)

```
mov   eax,dword ptr [pbSrc]
mov   edx,dword ptr [pbDst]
mov   ecx,8
```

Horizontal 8x8 Rounded Motion Compensation (MMX)

Well at the very least, the target computer will have MMX capability, but unfortunately the basic MMX does not support a *pavgusb* instruction. This was not introduced until the next generation of processors that supported the SSE and 3DNow! instruction set. So in this particular case, the parallel averaging algorithm of (A+B+1)>>1 has to be simulated. The good news though is that the 8x8 macro block size is supported nicely by a 64-bit register size. The data width of this instruction is only a 64-bit half-vector but works out well (except for the *pavgusb* issue!).

$$D_n = \frac{(A_n + A_{n+1} + 1)}{2}$$

Please review Chapter 10, "Special Functions" for more information.

Listing 16-3: vmp_x86\chap16\MComp\MCompX86M.asm

```
  movq  mm7,ZeroD
  movq  mm6,OneHD

next:
  movq  mm0,[eax+0]      ; A= [Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0]
  movq  mm1,[eax+1]      ; B= [Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1]

    ; (An+Bn+1)÷2       (simulate pavgusb instruction)
  movq  mm2,mm0
  movq  mm3,mm1
  punpcklbw mm0,mm7      ; 0:A  Low  [_A3 _A2 _A1 _A0]
  punpcklbw mm1,mm7      ; 0:B  Low  [_B3 _B2 _B1 _B0]
  punpckhbw mm2,mm7      ; 0:A  High [_A7 _A6 _A5 _A4]
  punpckhbw mm3,mm7      ; 0:B  High [_A7 _A6 _A5 _A4]

  paddusw   mm0,mm1      ; A+B Low  [A3+B3 A2+B2 A1+B1 A0+B0]
  paddusw   mm2,mm3      ; A+B High [A7+B7 A6+B6 A5+B5 A4+B4]
  paddusw   mm0,mm6      ; +1 Low   [A3+B3+1  ...  A0+B0+1]
  paddusw   mm2,mm6      ; +1 High  [A7+B7+1  ...  A4+B4+1]
  psrlw mm0,1            ; >>1          (A3...0+B3...0+1)÷2
  psrlw mm2,1            ; >>1          (A7...4+B7...4+1)÷2
  packuswb mm0,mm2       ; H...L    [D7 D6 D5 D4 D3 D2 D1 D0]

  movq  [edx],mm0        ; Save 64-bit (8-byte) result

  add   eax,iStride
  add   edx,iStride
  dec   ecx
  jne   next             ; Loop for next scan line
```

This was done by unpacking 8-bit bytes into 16-bit half-words, performing the calculations as half-words, and repacking back to 8 bit for the final solution.

Horizontal 8x8 Rounded Motion Compensation (3DNow!)

The AMD 3DNow! instruction set contains a *pavgusb* instruction that individually averages eight unsigned 8-bit values: (A+B+1)>>1

Listing 16-4: vmp_x86\chap16\MComp\MCompX86M.asm

```
next:
  movq  mm0,[eax+0]     ; A= [Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0]
  movq  mm1,[eax+1]     ; B= [Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1]
  add   eax,iStride

  pavgusb mm0,mm1       ;       (Y7...0+Y8...1+1)÷2

  movq  [edx],mm0   ; Save 64-bit (8-byte) result
  add   edx,iStride
  dec   ecx
  jne   next
```

Of course, if memory for code space is abundant and speed is more important than compactness, merely using a REPEAT 8 expansion macro instead of the loop will speed up the code, but the code size representing the function loop would increase from 37 to 266 bytes!

Horizontal 8x8 Rounded Motion Compensation (MMX+)

The AMD 3DNow! MMX+ Extensions include a *pavgb* instruction in addition to their earlier *pavgusb* instruction.

Horizontal 8x8 Rounded Motion Compensation (SSE)

Replace the *pavgusb* in the previous 3DNow! code sample with a 64-bit MMX *pavgb* instruction, which was introduced for SSE. For SSE2, there is no payoff due to the 128-bit data width of the XMM register.

Horizontal 16x16 Rounded Motion Compensation

This chapter can really get verbose with code snippets due to all the flavors and processor types, but just for a little clarity, the 3DNow! and SSE instruction sets are similar again, replacing the instruction *pavgb* with *pavgusb*.

Horizontal 16x16 Rounded Motion Compensation (3DNow!)

Listing 16-5: vmp_x86\chap16\MComp\MCompX86M.asm

```
next:
  movq  mm0,[eax+0]     ; A= [Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0]
  movq  mm2,[eax+1]     ; B= [Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1]
  movq  mm1,[eax+8]     ;    [YF YE YD YC YB YA Y9 Y8]
  movq  mm3,[eax+8+1]   ;    [Y10 YF YE YD YC YB YA Y9]

  pavgusb mm0,mm2       ; (Y7...0+Y8...1+1)÷2
  pavgusb mm1,mm3       ; (Y10...9+YF...8+1)÷2

  movq  [edx+0],mm0     ; Save lower 64-bits (8-byte) result
  movq  [edx+8],mm1     ; Save upper 64-bits (8-byte) result

  add   eax,iStride
  add   edx,iStride
  dec   ecx
  jne   next
```

Remember that the SSE is pretty much a parallel single-precision floating-point and not much in dealing with packed integers. The SSE2, on the other hand, expanded all the packed integer functions supported by MMX into a 128-bit form.

Horizontal 16x16 Rounded Motion Compensation (SSE2)

Even simpler, note the use of the unaligned memory access. Due to the nature of the use of this function in this particular application, as explained earlier, it is almost always unaligned.

Listing 16-6: vmp_x86\chap16\MComp\MCompX86M.cpp

```
next:
  movdqu xmm0,[eax+0]   ; [YF  YE YD YC  ...  Y3 Y2 Y1 Y0]
  movdqu xmm1,[eax+1]   ; [Y10 YF YE YD  ...  Y4 Y3 Y2 Y1]
  add    eax,iStride

  pavgb  xmm0,xmm1      ; (YF...0+Y10...1+1)÷2

  movdqu [edx],xmm0     ; Save 128-bit (16-byte) result
  add    edx,iStride
  dec    ecx
  jne    next
```

Pseudo Vec (PowerPC)

The PowerPC is not as forgiving as the X86 about misaligned memory, and so two cases need to be dealt with — one if the memory is aligned and the other if not, but the code is not included here. I did not think you wanted to look at four pages of code for just one function. So instead, only the highlights are shown.

Horizontal 8x8 Rounded Motion Compensation (PowerPC)

Effectively the following equation is at the core of this function using 32-bit data access, and four bytes are handled in parallel using simulated functionality. By now you should recognize the basic pattern:

```
*(pDst+0) = ((c0 >> 1) & eMASK7F)
          + ((d0 >> 1) & eMASK7F) + ((d0 | c0) & eMASK01);
```

There is, however, a catch. The PowerPC coprocessor has an exception fault on any misaligned memory access (the AltiVec coprocessor is the one that ignored the lower four address bits) even if the data is only 32 bits in size. Thus, the data address has to be corrected. Also, when following the comments within braces [], remember that this processor is in big endian, so more significant bytes will be on the right; thus the byte numbering is reversed. The Y, representing Y0, Y1, etc., is dropped for sake of simplicity.

First check for alignment:

Listing 16-7: vmp_ppc\chap16\MComp\MCompPPC.cpp

```
andi  r0,r4,3
cmpwi r0,0
beq   alignok         // Branch if aligned already
                      // r0={1...3} offset
li    dy,-4           // =FFFFFFFC
slwi  lshift,r0,3     // x8  1=8, 2=16, 3=24 bits to shift
and   r4,r4,dy        // Set pbSrc to mod 4 = 0

    // Note: r4 is either pbSrc -1,-2,-3 actual data!
    // Generate shifting masks

li    rshift,32
li    rmask,-1              //  =FFFFFFFF
sub   rshift,rshift,lshift  // (n=32-n)
srw   lmask,rmask,rshift    //  [___X] [__XX] [_XXX]
xor   rmask,rmask,lmask     //  [X___] [XX__] [X___]
```

```
      // Top of scan line loop
remX:                       //   1=8   2=16   3=24 (lshift)
  lwz   c0,0(r4)            // [X012] [XX01] [XXX0]
  lwz   c1,4(r4)            // [3456] [2345] [1234]
  lwz   d1,8(r4)            // [789A] [6789] [5678]

  slw   c0,c0,lshift        // [012_] [01__] [0___]
  rlwnm c1,c1,lshift,0,31 // [4563] [4523] [4123]
  rlwnm d1,d1,lshift,0,31 // [89A7] [8967] [8567]

  and   r0,c1,lmask         // [___3] [__23] [_123]
  and   c1,c1,rmask         // [456_] [45__] [4___]
  and   d0,d1,lmask         // [___7] [__67] [_567]
  or    c0,c0,r0            // [0123] [0123] [0123]
  or    c1,c1,d0            // [4567] [4567] [4567]

      // c0=[0123]   c1=[4567]   d1=[8XXX]
      //  d1:c1:c0 Source data now properly aligned!
```

Note: The rest of the code for this function is available on the companion CD.

Non-Rounded Motion Compensation

You might remember this from Chapter 4, "Vector Methodologies."

Table 16-2: On the right is the averaging (with rounding) result of the least significant bits of two numbers, which we worked in detail in the previous code samples. The left shows all the combinations of the result of the summation of the least significant bits and the resulting carry without rounding [Carry | Bit#0].

(A+B)/2	0	1
0	00	00
1	00	01

(A+B+1)/2	0	1
0	00	01
1	01	01

So while the table on the right is effectively a logical OR operation, the table on the left is a logical AND. With a slight modification to the equation, a similar simulated effect is delivered using the same algebraic law of distribution:

```
(b+c)/a = b/a + c/a
```

Use the results of the logical AND instead. Thus:

$$\frac{(A+B)}{2} = (A+B)>>1 = ((A>>1)+(B>>1)) + (A\&B\&1)$$

Motion Compensation (AltiVec)

The alignment issues of this function are a serious problem, as the AltiVec requires all data to be properly aligned to be accessed properly. So there are three choices:

1. Do not implement this function on AltiVec.
2. If unaligned, then call the PowerPC version of the code.
3. More work! Align the data!

The first two are doable, but the third looks a little tough. Why not try option number three first before writing it off?

If data was aligned, it would be easy! But where is the challenge in that?

Pseudo Vec (MIPS)

The MMI instructions for MIPS do not support a byte averaging instruction, so it must be handled manually with individual instructions. Also, since this function can be on any alignment, only variations of the first slow C function and the following assembly can be used. The optimized C cannot, as it expects the data to be at least on a 4-byte boundary, of which it typically is not due to the functionality of this implementation.

```
OneH0: .dword 0x0001000100010001,0x0001000100010001
```

Horizontal 8x8 Rounded Motion Compensation (MMI)

Listing 16-8: vmp_mips\chap16\MComp\MCompMMI.s

```
  la    t4,OneH0
  lq    t4,0(t4) // 0001000100010001 0001000100010001
  li    t5,8     // # of scan lines

// Load 1st and 2nd 64 bits S and R from memory. Using the 64-bit unaligned
// memory to access the data, it can be loaded on a byte-aligned basis.

$H0:
  ldl   t0, 7(a1)       // S {63...0} [Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0]
  ldl   t1, 8(a1)       // R {63...0} [Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1]
  ldr   t0, 0(a1)       // S {63...0} [Y7 Y6 Y5 Y4 Y3 Y2 Y1 Y0]
  ldr   t1, 1(a1)       // R {63...0} [Y8 Y7 Y6 Y5 Y4 Y3 Y2 Y1]
  add   a1,a1,a2        //  pbSrc += iStride

// S and R (t0 and t1) now each contain 128-bit aligned data expanded
// from 8 bit to 16 bit.

  pextlb t0,zero,t0     //  [_Y7 _Y6 _Y5 _Y4 _Y3 _Y2 _Y1 _Y0]
  pextlb t1,zero,t1     //  [_Y8 _Y7 _Y6 _Y5 _Y4 _Y3 _Y2 _Y1]

// Now average t1:t0  (A + B + 1)>>1

  paddh t0,t0,t1        // S+R [Y7+Y8 Y6+Y7 ... Y1+Y2 Y0+Y1]
  paddh t0,t0,t4        //  [Y7+Y8+1 Y6+Y7+1 ... Y1+Y2+1 Y0+Y1+1]
  psrlh t0,t0,1         //  (Yn+Yn+1+1)÷2
```

```
// Repack the data from 16 bit back to 8 bit. We only care about the lower
// eight bytes.

  ppacb t0,t0,t0        // [D7 D6 ... D1 D0 D7 D6 ... D1 D0]

// Save result to (unaligned) memory

  sdl   t0, 7(a0)       // pwD {63...0}
  sdr   t0, 0(a0)
  add   a0,a0,a2        // pDst += iStride

  addi  t5,t5,-1
  bne   zero,t5,$H0     // Loop for 8 scan lines
  BDELAY                // nop = Branch Delay Slot
```

All the other horizontal and vertical averaging functions are variations of this same code. Due to the unaligned access instruction, it is a very easy algorithm to implement.

Inverse Discrete Cosine Transform (IDCT)

I had to debate this topic for a while as to its inclusion into the book. The specification was set by the IEEE and is for sale by them on their web site (www.ieee.org): 1180-1990 *IEEE Standard Specifications for the Implementations of 8x8 Inverse Discrete Cosine Transform* (ISBN 1-5593-7098-X). This is the specification used to ensure the compatibility of the multitude of IDCT algorithms. In addition, there is actual assembly code for MMX, SSE2, AltiVec, and other processors available for download from the individual processor manufacturer web sites. There is also currently a market out there for high-speed hardware/software solutions, and any reprinting of an algorithm for this "commercial" book would be in violation of someone's copyright.

So there is no IDCT code in this book or on the companion CD. For your private use, feel free to download the code from the Internet, and figure out how it works! See if you think they did a good job or if there is an alternative method that you would have implemented!

Basically, a DCT (Discrete Cosine Transform) is used to encode 8x8 macro blocks within a frame, and an IDCT (Inverse Discrete Cosine Transform) is used to decode those same macro blocks into an image. The basic algorithm used in this particular type of application is as follows:

Equation 16-1: IDCT (Inverse Discrete Cosine Transform)

$$S(y,x) = \sum_{u=0}^{7} \sum_{v=0}^{7} a(u)\, a(v) f(v,u) \cos \frac{(2x+1)\pi u}{16} \cos \frac{2(y+1)\pi v}{16}$$

Discretely coding the equation requires a grand total of 4096 multiplications and 4032 additions to support transforming a single 8x8 two-dimensional matrix. An image is made up of multiples of these blocks, and by now you should have an understanding of the need for speed and the necessity of minimizing zinger code as much as possible. This might also bring to focus the reason that there are so many companies attempting to invent the fastest algorithmic mechanism to feed their profit margins.

Something else to keep in mind is that vectors are 16 byte, not 8 byte, and so register sizes of 64 bit have an advantage in this particular case.

YUV Color Conversion

The decode process generates a YUV, which is typically converted to RGB (red, green, and blue). As 32-bit RGB is my personal favorite, that is what will be discussed. It is relatively easy to convert the YUV to RGB 24 bit, 16 bit, 5:5:5, or 5:6:5 and a little more difficult to 8 bit using an indexed palette lookup.

The YUV12 to RGB Color Conversion algorithm is typically used in MPEG-2 decoding in the conversion of a YUV12 encoded image into an RGB-based image. Since frame rates need to be the highest possible for the smoothest playback of a movie, the faster a frame can be decompressed, the better. Discussed here will be the basic functionality using the C programming language, followed by two different methods of vector implementation.

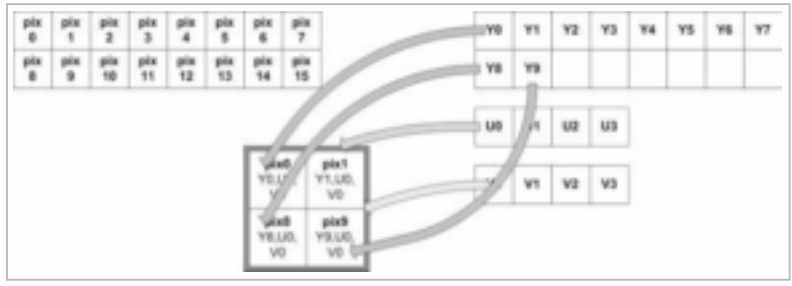


Figure 16-2: Relationship of each 2x2 block of pixels in conjunction with each pixel having a unique y but all sharing the same u and v

Each pixel consists of a Y, U, and V value. There are four times as may Y coordinates as UV coordinates that are shared for each quad 2x2 array of pixels.

```
pix0=Y0,U0,V0  pix1=Y1,U0,V0  pix2=Y2,U1,V1    pix3=Y3,U1,V1
pix8=Y8,U0,V0  pix9=Y9,U0,V0  pix10=Y10,U1,V1  pix11=Y11,U1,V1
```

YUV12 to RGB32

Pseudo Vec

The following macro is used to limit the unsigned byte value to the saturation ranges of a minimum of 0 and a maximum of 255:

```
#define SAT_RGB8(a)(a)>255 ? 255 : (a)<0 ? 0 : (a)
```

For more details, check out the Min() and Max() functionality in Chapter 10, "Special Functions."

The following function is one of the DivX;–) group's original open source C algorithm that I have cleaned up somewhat to demonstrate the use of integer multiplication in conjunction with data unpacking/packing logic to convert each of the 8-bit sets representing YUV into a 32-bit RGB frame buffer. Review Chapter 5, "Vector Data Conversion," for the optimal data bit reduction solution with saturation. By optimizing the original C code to that of the following on an X86 processor, an approximate 28% increase in speed is achieved.

Listing 16-9: \chap16\yuv12\yuv12.cpp

```
void vmp_yuv12toRGB32(
           uint8 *pBaseY, uint nStrideY,
           uint8 *pBaseU, uint8 *pBaseV, uint nStrideUV,
           void  *pImage, uint nWidth, uint nHeight,
                          int iStride)
{
  uint w, suv;
  byte *pImg;

  pImg = (uint8*)pImage;
  iStride -= (nWidth<<2);  // Remaining bytes to skip
  nStrideY -= nWidth;      // remaining Y table skip
  nWidth >>= 1;
  suv = 0;

  do {                     // Loop for each row
    uint8 const *pu, *pv;

    pu = pBaseU;
    pv = pBaseV;
    w = nWidth;
```

```
    do {                // Loop for each pixel pair
      signed int u,v, y0,y1, ur,ug,vg,vb, r,g,b;

      u = *pu++ – 128;
      v = *pv++ – 128;
      y0 =  0x2568 * (*pBaseY++ – 16);
      y1 =  0x2568 * (*pBaseY++ – 16);
      ur =  0x3343 * u;
      ug = –0x1a1e * u;    // 0xe5e2
      vg = –0x0c92 * v;    // 0xf36e
      vb =  0x40cf * v;

#ifdef VMP_BIG_ENDIAN
      r = (y0+   ur) >> 13; pImg[3] = SAT_RGB8(r);
      g = (y0+vg+ug) >> 13; pImg[2] = SAT_RGB8(g);
      b = (y0+vb   ) >> 13; pImg[1] = SAT_RGB8(b);
      pImg[0] = 0;
      r = (y1+   ur) >> 13; pImg[7] = SAT_RGB8(r);
      g = (y1+vg+ug) >> 13; pImg[6] = SAT_RGB8(g);
      b = (y1+vb   ) >> 13; pImg[5] = SAT_RGB8(b);
      pImg[4] = 0;
#else
      r = (y0+   ur) >> 13; pImg[0] = SAT_RGB8(r);
      g = (y0+vg+ug) >> 13; pImg[1] = SAT_RGB8(g);
      b = (y0+vb   ) >> 13; pImg[2] = SAT_RGB8(b);
      pImg[3] = 0;
      r = (y1+   ur) >> 13; pImg[4] = SAT_RGB8(r);
      g = (y1+vg+ug) >> 13; pImg[5] = SAT_RGB8(g);
      b = (y1+vb   ) >> 13; pImg[6] = SAT_RGB8(b);
      pImg[7] = 0;
#endif
      pImg += 8;
    } while( --w );

    pBaseY += nStrideY;    // Next Y scan line
    pBaseU += suv;         // Next or same U scan line
    pBaseV += suv;         // Next or same V scan line
    pImg   += iStride;     // Next image scan line
    suv ^= nStrideUV;      // every other scan line
  } while (--nHeight);     // Loop for # of scan lines
}
```

One other item to note is that the RGB orientation is different between big and little endian. In this particular case, the RGB order depends on the video hardware. PC is little endian and Macintosh is big endian; therefore, the byte ordering is reversed!

The arguments pBaseY, pBaseU, and pBaseV are byte arrays containing the list of YUV values. The stride(s) indicate the byte count per scan line of each list of Y/U/V elements. The YUV blocks actually consist of groupings of arrays of pixels (macro blocks), each containing a matrix block eight columns wide by eight rows high in size, which works out nicely for a processor with 128-bit data width. The image being rendered has a width and height also divisible evenly by 16, so

with 32 bits of RGB, eight pixels can be calculated simultaneously and written as two sets of four pixels. The floating-point version of the equations to convert the YUV to RGB for each pixel is as follows:

```
B = 1.164 (Y–16) +2.018(U–128);
G = 1.164 (Y–16) – 0.391(U–128) – 0.813(V–128);
R = 1.164 (Y–16) + 1.596(V–128);
```

By converting to a series of integer ratios, an increase in speed can be achieved by using an equation similar to the following:

```
B=Sat8(( 0x2568 (Y–16) +0x40CF(U–128) ) >> 13);
G=Sat8(( 0x2568 (Y–16) – 0x0c92(U–128) – 0x1A1E(V–128) ) >> 13);
R=Sat8(( 0x2568 (Y–16) + 0x3343(V–128) ) >> 13);
```

The 16x16 alignment and individual array orientation of the data makes this a good vectorization training model for this book. The data dynamics should be noted, as they will be an indicator of data flow. YUV begins as 8-bit data, followed by an expansion to 16 bit and a displacement, some multiplication, some addition, a right shift to reduce the value, and then a repacking of the data back to 8 bit using saturation, limiting the values n={0...255}.

```
saturation = Min( Max( n, 0 ), 255 );
```

The following code snippets are from an MMX sample, which uses an AoS-type implementation and achieves a 463% increase in speed over the original C code. By examining the formula, a horizontal (AoS) pattern emerges. If the expected end result of the data is arranged early in the process, the result progresses nicely. One item to note is that in a 16-bit calculation, –0x0C92 is equivalent to 0xF36E, and –0x1A1E is equivalent to 0xE5E2 when represented in 16 bits.

Examine the companion CD for the actual code sample. The code is fine and dandy but a little long for inclusion in the pages of this book. Another item to note is that after the 32-bit result of the 16x16-bit multiplication and summations, 13 bits of information is thrown away. This can be used to its advantage. As the 16-bit constants in the equation are {0x3343, 0x40cf, E5E2, F36E} and the resulting product between these and an 8-bit signed value is no more than 24 bits in size, 8 bits are left to play with. If each Y, U, or V is pre-shifted left by three bits, the result would need to be arithmetically shifted right by three bits for a total right shift of 16 bits to balance the equation. If the processor supports 16-bit multiplication of only the upper or lower 16-bit components, time can be saved. For this problem, only an upper 16-bit multiplication needs to be done, and the ending results would be in the desired 16-bit form.

Pseudo Vec (X86)

The following MMX sample code is a vectorization with a very untangled optimization to keep the code readable and still achieve a significant increase in speed from the original C code. For something a little more interwoven, check out the companion CD.

The following masks are used for color multiplication factors, displacement adjustments, and bit flipping:

Listing 16-10: vmp_x86\chap16\yuv12\yuv12X86M.asm

```
ZeroD    dq  0000000000000000h,0000000000000000h
ColorY   dq  0256825682568256h,0256825682568256h ; 2568h
ColorUR  dq  0334334334334334h,0334334334334334h ; 3343h
ColorUG  dq  0E5E2E5E2E2E2E2E2h,0E5E2E5E2E5E2E5E2h ;-1a1eh
ColorVG  dq  0F36EF36EF36EF36Eh,0F36EF36EF36EF36Eh ;-0c92h
ColorVB  dq  040CF40CF40CF40CFh,040CF40CF40CF40CFh ; 40cfh
DiffUV   dq  0FC00FC00FC00FC00h,0FC00FC00FC00FC00h ; -128*8
DiffY    dq  0FF80FF80FF80FF80h,0FF80FF80FF80FF80h ; -16*8
```

vmp_yuv12toRGB32 (MMX)

```
    mov   eax,nWidth
    shr   nWidth,3        ; nWidth >>= 3;   n/8

      ; Set up pointers and strides

    sub   nStrideY,eax    ; nStrideY -= nWidth
    mov   ebx,pImg
    mov   esi,pBaseY
    shl   eax,2           ; x4
    mov   edi,pBaseU
    mov   edx,pBaseV
    sub   iStride,eax     ; iStride -= (nWidth*4)
    mov   eax,0           ; (Flip Flip)

      ; Vertical loop

 Vrt: mov ecx,nWidth

      ; Horizontal loop
```

Before beginning to design your function, it is sometimes reasonable to work the algorithm from both ends in opposite directions. Figure out what is expected based upon how much data is loadable and the expected end result. As MMX uses 64 bits (8 bytes) and it is foreseeable that all eight registers are going to be very busy, then the function will probably have to only deal with half a vector (8 bytes at a time.) As Y, U, and V are 8 bits each, eight data elements are the natural data size to handle.

Input:

$[y_7\ y_6\ y_5\ y_4\ y_3\ y_2\ y_1\ y_0]$ $[y_F\ y_E\ y_D\ y_C\ y_B\ y_A\ y_9\ y_8]$
$[u_7\ u_6\ u_5\ u_4\ u_3\ u_2\ u_1\ u_0]$ $[u_7\ u_6\ u_5\ u_4\ u_3\ u_2\ u_1\ u_0]$
$[v_7\ v_6\ v_5\ v_4\ v_3\ v_2\ v_1\ v_0]$ $[v_7\ v_6\ v_5\ v_4\ v_3\ v_2\ v_1\ v_0]$

Output:

$0\ b_7\ g_7\ r_7 \quad 0\ b_6\ g_6\ r_6 \quad \ldots \quad 0\ b_1\ g_1\ r_1 \quad 0\ b_0\ g_0\ r_0$

Keep in mind that, based upon Figure 16-2, which displays a 2x2 Y table sharing the same U and V data elements, the upper four data elements of $U_{7...4}$ and $V_{7...4}$ can be afforded to be temporarily lost, as they are not needed.

```
Hrz:
  movq  mm2,[esi]       ; [y7 y6 y5 y4 y3 y2 y1 y0]
  movq  mm0,[edi]       ; [u7 u6 u5 u4 u3 u2 u1 u0]
  movq  mm1,[edx]       ; [v7 v6 v5 v4 v3 v2 v1 v0]
  add   esi,8
  add   edi,4
  add   edx,4
```

Since the smallest packed multiplication is a 16-bit half-word and these are 8-bit values, the data must be expanded before the displacement of –128 or –16 to maintain the data sign bit. As unpacking does not extend the sign bit, it is much easier to unpack the 8-bit unsigned values from one register into 16 bits stored in two separate registers, and then treat it as a signed number when calculating the negative displacement!

```
  movq  mm3,mm2         ; [y7 y6 y5 y4 y3 y2 y1 y0]
  punpcklbw mm2,ZeroD   ; [ 0 y3   0 y2   0 y1   0 y0]
  punpckhbw mm3,ZeroD   ; [ 0 y7   0 y6   0 y5   0 y4]
  punpcklbw mm0,ZeroD   ; [ 0 u3   0 u2   0 u1   0 u0]
  punpcklbw mm1,ZeroD   ; [ 0 v3   0 v2   0 v1   0 v0]
```

You may recall part of an equation in the original C code that shifted the data right by 13 bits after all the calculations, thus throwing away those lower bits. Since the integer-based packed multiplication tends to return the upper or lower 16 bits, this feature can be taken advantage of. If by left shifting by 3 bits and performing the calculations and then using the multiplication that returns the upper 16 bits, there's no need for the post down shift, as the multiplication would have handled it automatically.

```
(n >> 13) is equivalent to (n << 3)>>16.
```

Note that they are equivalent and not necessarily the same, as there will be a slight difference in solutions since results will vary slightly due to the difference in the number of down shifted bits, but this function is lossy based so it does not matter too much. As mentioned, the shift left

by three is in effect the result of a product of eight. Remember the original equation was:

```
(u–128)
```

...so:

```
8(y–16)     =     (8*y)–(8*16)     =     8y–128     =     8y+ –128
8(u–128)    =     (8*u)–(8*128)    =     8u–1024    =     8u+ –1024
8(v–128)    =     (8*v)–(8*128)    =     8v–1024    =     8v+ –1024
```

```
  psllw  mm0,3        ; [U3 U2 U1 U0]*8
  psllw  mm1,3        ; [V3 V2 V1 V0]*8
  psllw  mm2,3        ; [Y3 Y2 Y1 Y0]*8
  psllw  mm3,3        ; [Y7 Y6 Y5 Y4]*8
```

Now perform the calculations:

```
  paddsb mm0,DiffUV   ; [U3 U2 U1 U0]  (8*u)–(8*128)
  paddsb mm1,DiffUV   ; [V3 V2 V1 V0]
  paddsw mm2,DiffY    ; [Y3 Y2 Y1 Y0]  (8*y)–(8*16)
  paddsw mm3,DiffY    ; [Y7 Y6 Y5 Y4]

  movq   mm6,mm0
  pmulhw mm0,ColorUR  ; [UR3 UR2 UR1 UR0]  ur = 0x3343*u
  pmulhw mm6,ColorUG  ; [UG3 UG2 UG1 UG0]  ug =–0x1a1e*u

  movq   mm4,mm0      ; [UR3 UR2 UR1 UR0]
  punpcklwd mm0,mm0   ; [UR1 UR1 UR0 UR0]
  punpckhwd mm4,mm4   ; [UR3 UR3 UR2 UR2]

  movq   mm7,mm1      ; [ V3  V2  V1  V0]
  pmulhw mm1,ColorVB  ; [VB3 VB2 VB1 VB0]  vb = 0x40cf*v
  pmulhw mm7,ColorVG  ; [VG3 VG2 VG1 VG0]  vg =–0x0c92*v

  movq   mm5,mm1      ; [VB3 VB2 VB1 VB0]
  punpcklwd mm1,mm1   ; [VB1 VB1 VB0 VB0]
  punpckhwd mm5,mm5   ; [VB3 VB3 VB2 VB2]

  pmulhw mm2,ColorY   ; [ Y3  Y2  Y1  Y0]  y = 0x2568*y
  pmulhw mm3,ColorY   ; [ Y7  Y6  Y5  Y4]  y = 0x2568*y

  paddsw mm7,mm6      ; [UG3+VG3 UG2+VG2 UG1+VG1 UG0+VG0]
  movq   mm6,mm7      ; [UG3+VG3 UG2+VG2 UG1+VG1 UG0+VG0]
  punpcklwd mm7,mm7   ; [UG1+VG1 UG1+VG1 UG0+VG0 UG0+VG0]
  punpckhwd mm6,mm6   ; [UG3+VG3 UG3+VG3 UG2+VG2 UG2+VG2]

  paddsw mm0,mm2      ; r=[#r3 #r2 #r1 #r0]
  paddsw mm4,mm3      ; r=[#r7 #r6 #r5 #r4]
  paddsw mm1,mm2      ; b=[#b3 #b2 #b1 #b0]
  paddsw mm5,mm3      ; b=[#b7 #b6 #b5 #b4]
  paddsw mm2,mm7      ; g=[#g3 #g2 #g1 #g0]
  paddsw mm3,mm6      ; g=[#g7 #g6 #g5 #g4]

  packuswb mm0,mm4    ;r=[r7 r6 r5 r4 r3 r2 r1 r0]
  packuswb mm1,mm5    ;b=[b7 b6 b5 b4 b3 b2 b1 b0]
  packuswb mm2,mm3    ;g=[g7 g6 g5 g4 g3 g2 g1 g0]
```

At this point, there are eight red, green, and blue 8-bit data elements that now need to be interleaved into a 32-bit RGB sequence with a zero in the upper bits and red in the lower bits:

```
[3]=0   [2]=b   [1]=g   [0]=r
```

Since the unpack operation can be used to interleave data elements from two registers, it can be used in two passes. The first pass is to interleave the odd (green and zero) elements and even (red and blue) elements, then a second pass is to interleave the odd and even elements together into the {0, Blue, Green, Red} sequence. Note that each time two registers are interleaved, the equivalent of half of each register is ignored (wasted), thus it needs to be performed twice to handle the lower and upper halves separately.

```
movq  mm4,mm0       ;r
movq  mm3,mm2       ;g

punpcklbw mm4,mm1   ;br   [b3 r3 b2 r2 b1 r1 b0 r0]
punpckhbw mm0,mm1   ;br   [b7 r7 b6 r6 b5 r5 b4 r4]
punpcklbw mm3,ZeroD ;0g   [ 0 g3  0 g2  0 g1  0 g0]
punpckhbw mm2,ZeroD ;0g   [ 0 g7  0 g6  0 g5  0 g4]

movq  mm5,mm4       ;br   [b3 r3 b2 r2 b1 r1 b0 r0]
movq  mm1,mm0

punpcklbw mm4,mm3   ;0bgr [ 0 b1 g1 r1  0 b0 g0 r0]
punpckhbw mm5,mm3   ;0bgr [ 0 b3 g3 r3  0 b2 g2 r2]
punpcklbw mm0,mm2   ;0bgr [ 0 b5 g5 r5  0 b4 g4 r4]
punpckhbw mm1,mm2   ;0bgr [ 0 b7 g7 r7  0 b6 g6 r6]

movq  [ebx+0],mm4   ;bgr1 bgr0
movq  [ebx+8],mm5   ;bgr3 bgr2
movq  [ebx+16],mm0  ;bgr5 bgr4
movq  [ebx+24],mm1  ;bgr7 bgr6
add   ebx,32

dec   ecx
jnz   Hrz           ; Loop for width of image

    ; Odd/Even – Calculate starting positions to process next scan line

mov   ecx,nStrideUV  ; UV stride
mov   edi,pBaseU     ; pU (old)
mov   edx,pBaseV     ; pV (old)
and   ecx,eax        ; ? 0 : stride
xor   eax,0FFFFFFFFh ; Flip mask

add   esi,nStrideY   ; pY   += stride adj.
add   ebx,iStride    ; pImg += stride adj.

add   edi,ecx        ; pU += (odd) ? 0 : next
add   edx,ecx        ; pV +=

mov   pBaseU,edi     ; pU (Save even=old or odd=new pointer)
```

```
mov   pBaseV,edx  ; pV
dec   nHeight
jne   Vrt         ; Loop for height of image
```

vmp_yuv12toRGB32 (SSE2)

The really nice thing about the SSE2 instruction set is that the integer instructions are actually extensions of the same MMX instructions. Therefore, by using *xmm#* instead of *mm#*, and of course the appropriate memory movement instruction such as *movdqu* and *movdqa* instead of *movq*, the code looks virtually the same. Of course, the number of loops per scan line will be reduced by half. Check out the companion CD and compare them side by side!

Pseudo Vec (PowerPC)

As you are hopefully well aware, the basic PowerPC instruction set is only 32 bit and not 128 bit, so all those single board and G3 Macintosh computers have to simulate the vector algorithms. This is normally not a problem because since there are 32 of the registers (of which quite a few are available for application use) processor pipelining can be taken advantage of. The tricks that were discussed in earlier chapters allow pseudo 2x16-bit and 4x8-bit operations within each register. The problem is that the integer multiplications have to be handled individually as a scalar. The equations for the separate RGB components are expanded, such as the following:

```
Y -= 16;          U -= 128;          V -= 128;

R = (0x2568*Y + 0x0000*V + 0x3343*U) / 0x2000;
G = (0x2568*Y - 0x0c92*V - 0x1a1e*U) / 0x2000;
B = (0x2568*Y + 0x40cf*V + 0x0000*U) / 0x2000;

R = R>255 ? 255 : R;          R = R<0   ?   0 : R;
G = G>255 ? 255 : G;          G = G<0   ?   0 : G;
B = B>255 ? 255 : B;          B = B<0   ?   0 : B;

#define VAL_Ru    0x3343    // Ru   =13123v
#define VAL_Gv    0x0c92    // Gv   =3218u
#define VAL_RGBy  0x2568    // RGBy =9576y
#define VAL_Gu    0x1a1e    // Gu   =6686u
#define VAL_Bv    0x40cf    // Bv   =16591v
```

So there are three methods that can be done here:

- Integer multiplication
- Integer guesstimation
- Table lookup

Integer Multiplication

This is the simplest but not the fastest solution, as it just uses the regular expression.

Integer Guesstimation

It is not exactly a guess but more of taking advantage of the case of this implementation's lossy. The actual Gv, Ru, and Y multiplications still occur, but there is an interesting item to notice:

```
Ru = 13123u          Gv = 3218v          Y = 9576y
```

The remaining two products:

```
Gu = 6686u          Bv = 16591v
```

...are close to the results of Ru and Gv with some additional shifting logic.

```
Gu = (Ru ÷ 2)        6686u   ~    6561u = ((13123u) ÷ 2)     98.13%
Bv = (Gv × 5)       16591v   ~   16090v = ((3218v) × 5)      96.98%
```

They are not exact, so they are only used on really slow computers to save the two multiplication calculations.

Table Lookup

With table lookups, there is the little problem of the data cache to contend with, but having each value precomputed and merely retrieving the value from a table is definitely faster than actually doing the multiplication. It does require, however, 3 K of memory to contain the tables. It would be up to your implementation whether to have static tables or dynamically allocated ones, as it would depend upon your memory footprint requirements.

```
#define TBLMAX

static int UTblMult[ TBLMAX ];
static int VTblMult[ TBLMAX ];
static int YTblMult[ TBLMAX ];

  for ( int i = 0; i < TBLMAX; i++ )
  {
    int uv, y;

    uv = i – 128;
    y  = i – 16;

    VTblMult[i] = VAL_Gv   * uv;
    UTblMult[i] = VAL_Ru   * uv;
    YTblMult[i] = VAL_RGBy * y;
  }
```

The results are the same from the original equation; only the negative displacement and product are precomputed.

vmp_yuv12toRGB32 (PowerPC)

There is no need to cut and paste the entire body of the code here, as it is pretty much a scalar operation, although two pixels are handled simultaneously to take advantage of the UV multiplications. From single-byte loads, shifting, multiplication, and branchless min/max operation to handling the saturation is all 32-bit processing. It is not until the final stage when instead of having eight individual bytes written, the RGB components that make up the pixels are blended before being written out as two 32-bit values.

Listing 16-11: vmp_ppc\chap16\yuv12\yuv12PPC.cpp

```
rlwimi rd0,gn0,8,16,23  // e = [__gr]
rlwimi rd1,gn1,8,16,23  // o = [__gr]
rlwimi rd0,bl0,16,8,15  // e = [_bgr]
rlwimi rd1,bl1,16,8,15  // o = [_bgr]
stw    rd0,0(r3)        // Write even ORGB pixel
stw    rd1,4(r3)        // Write odd  ORGB pixel
```

There is one interesting thing about the function, however, and that is the butterfly switch to allow for branchless coding. As scan line pairs use the same U and V tables, a logic gate determines whether to rewind the U and V pointers to the beginning of their scan line or advance to the beginning of the next one. Without having to branch and conserve registers so that our UV base pointers become our actual pointers, this becomes a necessity.

```
suv = -(nWidth >> 1);            // # of UV's needed
iDir = suv ^ (nStrideUV + suv);  // Butterfly Key
```

The register containing the suv value starts by adding a negative displacement so that at completion of a scan line, the addition will put the pointer back at the beginning of the same scan line. The iDir value contains a mask of the difference between advancing to the beginning of the next scan line or rewinding to the beginning of the current. The butterfly component is that each time the iDir is logical XOR'd with the current suv value, the other value becomes the new value. Then it is merely a matter of adding the updated suv at the appropriate time.

```
add   r9,r9,suv       // U Table same/next scan pair
add   r10,r10,suv     // V Table same/next scan pair
xor   suv,suv,iDir    // Forward/Reverse
```

Chapter 17
Vector Compilers

Vector compilers are designed and built to help you do more within your game or embedded program, not to help reduce costs.

Note: A vector compiler will not help reduce costs!

If your company wants to make more money, it needs to design and develop better games, not try to cut corners. A vector compiler is a mechanism to get your C code application up and running with faster throughput in a minimal amount of time. It will not be optimal, but it is a solution. Think of it as a tool to help build a better product and possibly assist in proving new ideas and technology.

It is not meant to replace the need for writing any assembly code, only to supplement it. Management can go manage other things to keep themselves busy (I am being nice here but do not tell them this if you wish to keep your job), as vector compilers are not meant to replace assembly language programmers. Therefore, do not feel threatened (nor should you feel that this gives you job security. Trust me!).

A few narrow-minded university professors are spreading their unintelligible, unfounded, uninformed wisdom about abolishing assembly language programming, as it is not needed anymore. Little do they know!

A compiler cannot beat a good assembly language programmer in terms of writing fast code. Plus, what do you think is inside that compiler? Inside those optimized processor targeted libraries that you are linking to your code? Inside that Flash ROM booting up the computer? Inside that low-level graphics algorithm managing those sprites and 3D objects? C code? Ha! (At least for your sake, I hope not!) In a worst case, the programmer can always take the intermediate output files of the compiler and hand optimize them with or without tools and possibly make the code even faster. Note that the compiler may not always be beaten, as sometimes it is a tie, but the compilers are not perfect yet! There is not much of any artificial intelligence in them as of yet, only a list of patterns and techniques for them to test for and respond to.

If the truth be told, those of us who are very good at assembly language programming are members of a secret society. We have a secret handshake, chant, robes, and everything. Those other guys are just jealous and have assembly code envy.

Besides, you have seen the movie — the computers build other computers that write computer code that humans cannot read and then take over the world.

Okay, I am off my soapbox and fantastical vision now, so let's continue. I am only going to paint some broad strokes here, as these compilers come with online manuals and have trial versions that are downloadable. So it will be up to you to further investigate them on your own.

Codeplay's Vector C

A trial version is available for download from http://www.codeplay.com.

The Vector C compiler by Codeplay, which supports C and C++, is available on multiple platforms: X86 for Win32 and Linux as well as the Emotion Engine and VU coprocessors of the Sony PlayStation 2.

In order to use a vector compiler, it needs to be micromanaged — in a sense, told which blocks of code you want vectorized and how to go about doing it. This allows you to control different code segments, such as in the case of 3DNow! versus SSE code, etc. The following information is related to the public accessible version of the compiler for the X86 processor line as the MIPS EE and VU versions are under a non-disclosure agreement between manufacturers.

Source and Destination Dependencies

There is a problem when the destination also happens to be a source argument to a function. It limits the optimization ability of the compiler. You have seen something similar in pseudocode samples for matrix multiplication. In that particular case, data values were still being used as source input after some output data was written, contaminating the source input. Protection against this is needed just in case the destination also happened to be one of the source arguments. A temporary destination was used and eventually the final solution was copied to the true destination.

To help resolve this in the case of Codeplay's compiler, a restriction needs to be placed, such as:

```
void vmp_fn( int * restrict d, int * const restrict a, int * const restrict b)
```

Without the *restrict* keyword, the compiler cannot safely shuffle the memory accesses for the best throughput, since a destination may be set before it is used as a source! The keyword *restrict* essentially indicates to the compiler to assume that it is safe, since there should be no relationship between the source and destination memory being referenced.

To reinvest in our sample programs, portions of the AoS sample from Chapter 4, "Vector Methodologies," are being reused here!

Local Stack Memory Alignment

Stack memory sometimes needs to be aligned and so using a declaration such as:

```
__declspec(align(16))
```

...in the following:

```
int main( int argc, char *argv[] )
{
  __declspec(align(16)) SoABlk soaTmp;
  __declspec(align(16)) AoS    aosSrc[ SOA_MAX ],
                               aosDst[ SOA_MAX ];
```

...instead of:

```
  SoABlk soaTmp;
  AoS    aosSrc[ SOA_MAX ], aosDst[ SOA_MAX ];
```

...guarantees the alignment to the specified amount such as 16 bytes (128 bits) as in this particular case. And since the memory alignment is guaranteed, you can alter your assertions within your called functions:

```
static void vmp_SoA8_AoS( SoABlk *pDAry,
                          const AoS *pSAry, uint nCnt )
{
  ASSERT_PTR4(pSAry);
  ASSERT_PTR4(pDAry);

  ASSERT_PTR16(pSAry);
  ASSERT_PTR16(pDAry);
```

Other declarations, such as:

```
__declspec(align(8))
```

...would align to 8 bytes (64 bits) as well.

Structures Pushed on Stack (Aligned)

In all the examples in this book, vectors were never pushed onto the stack. Instead, a base pointer was passed and the structure was accessed from within the called function. If the data needs to be manipulated without affecting the original vector, that data has to be copied and manipulated. An alternative to this method would be to push the vector onto the stack, but there is the risk of the data not being aligned properly. By coding a function's passed arguments in its declaration, the vector compiler knows which arguments need to have stack pointer adjustment code in the calling function and in the called function as well.

Floating-Point Precision

The compiler needs to be told whether the single-precision floating-point vector block you wish to vectorize is to use the instructions that support standard precision or if it can use the faster instructions that use estimated precision. The following declaration does just that:

```
__hint__((precision(12)))
```

As was discussed earlier in this book in regards to reciprocals and square roots, there are estimated versions of their instructions for speed and methods to increase the precision to normal if necessary. The differences between processors should be kept in mind when attempting to use this functionality.

Intel's C++ Compiler

Intel's C++ Compiler for Windows has a trial demo version for download at http://developer.intel.com/software/products/eval/.

It takes full advantage of the instructions supported by the Intel X86 processor family, but support of the AMD family is another story. This is a great compiler if you are targeting only Intel. As Mel Brooks would say, "It's good to be the king!" As the Intel line of processors is not the only one, the best solution would be to compile most of your code with this compiler, but when specific AMD functionality is needed, such as 3DNow!, then use the Vector C++ compiler by Codeplay. The same principles of instruction types and memory as applied to the vector C compiler applies here and any other vector compiler.

Other Compilers

For your other platforms, you will have to settle for just using the scalar tools that are available. There are vector libraries available from the manufacturers and/or middleware providers, but you will have to wait and see when vector compilers will be made available to you. Software tools are being developed all the time, and the bigger the market for a processor, the bigger the profits, thus the more likely that advanced tools will be written and become available. For example, the PS2 has been out since around November 2000 in the United States, and embedded processors utilizing subsets of its capability were announced as being available in 2003. Sure there is a lag, but with the increase in the market, there will be a boon of new tools. In May 2002, home consumers were able to purchase the Linux Kit for the PS2, which contained programming manuals, hard disk, TCP/IP connection, and a Linux interface to start them on their road to game programming. That alone will help to open publicly available development tools, as well as warm up a new flock of fledgling game console programmer wannabes.

Wrap-up

As you can see, there is a little bit of preparation that you need to do to prepare your code for proper compilation, but this is an insignificant amount of time as opposed to the amount of time necessary to code those same code blocks for the individual processors by hand. This should be thought of as a time savings to get your demonstration versions up and running quicker so that you can see whether or not your concepts and implementations are working as you planned. Why expend your time optimizing code that you may have to change or throw away due to part of the game not working and you having to take a different path? Once you prove those concepts and start to lock things down, investigate the implementation of that code into an assembly form and whether or not it needs to be optimized as such. The code generated by the vector compiler may be fast enough for your purposes and may not need to be optimized any further.

There are other compiler options that we do not need to get into because if you have a copy of the compiler, the manuals come with it. If you do not have a copy, do not fret because time-limited demos are available for download from the Codeplay and Intel web sites.

Chapter 18
Debugging Vector Functions

Debugging code written for vector processors can either be complicated or very simple, depending on your toolset. The first thing to remember is to have a good IDE (Integrated Development Environment). With this and the proper processor package, the packed data vector registers can be immediately dumped and examined to verify that the data is as expected.

Visual C++

If using Visual C++ version 7 or higher, then your compiler has already been updated with the latest and greatest processor information, but if using Visual C++ version 6, then you need to download and install the service packs:

http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/default.asp.

...and the processor packs if you are running service pack 4:

http://msdn.microsoft.com/vstudio/downloads/ppack/Beta/download.asp.

If you are using a version older than Visual C++ version 6, you will need to rely heavily on functions that you can call to dump the contents of your registers.

Figure 18-1: About Visual C++ version 6.0

Figure 18-2: About Visual C++ .NET version 7.0

You can find out which compiler version you are using by choosing the Help | About menu option on your menu bar. In your particular case, your Product ID number will be displayed. In this sample case, I have erased my numbers from the figures for my own security protection.

Note in the following Visual C++ dump for an X86 type processor under VC6 that only MMX and not XMM registers are supported in this sample. Since this is from a run using a 3DNow! processor, there is a dump of the MMX registers in hex (MM#) as well as floating-point denoted by (MM##) in upper and lower 32-bit single-precision floating-point values. MM00 in the bottom portion of the following figure represents the lower single-precision value of the MM0 register.

Figure 18-3: Register dump from Visual C++ 6.0 (3DNow! Extensions)

In some particular cases, such as an older version of the compiler, the extended registers are not immediately available. In the newer version, such as the VC .NET toolset, that is no longer an issue! In the following figure, selective dumps of the other registers have been made available. Using at least version 7 of the compiler allows the active register sets to be enabled and viewed in the window.

Figure 18-4: Register dump from Visual C++ .NET version 7 (3DNow! Professional Extensions). Note the 3DNow! and SSE supported registers as well as the extra pop-up menu of register type selections that you can select for your window. The enabled selections are based upon the processor type running on the computer that is being debugged.

The one thing that you will really need while developing assembly code is a second monitor. This is because you will need lots of screen real estate for all the various windows that you will need for debugging your assembly code.

Since we are talking here about Win32 code, whether it be targeted for the Xbox or PC platforms, you are most likely dealing with Direct3D or the less likely OpenGL. If it is Direct3D, then it is in your best interest to install the debug versions of the DirectX dynamic-link libraries (DLLs). They will run slower, but if something weird happens they will give you more information about the type of failure. Remember that your application will run that much faster on a retail version of the DLLs!

Another handy debugging tool for vectors is the use of multi-colored pens. Using multiple colors on graph paper can produce layered results, giving depth and helping to make complex problems easier to understand.

The biggest help in developing vector-based applications above all is the symbolic debugger. The program counter can be followed throughout the code, and the general-purpose and SIMD registers can be monitored by observing the changes to memory as the results are written.

Using dual monitors allow more screen real estate for multiple diagnostic viewing windows.

Other Integrated Development Environments

For the AltiVec and other processors, the Code Warrior program is pretty handy in allowing the full debugging of vector code down to the assembly level: http://www.codewarrior.com.

I have found the Macintosh Programmer's Workshop (MPW) package with the SADE tool from Apple is oriented to higher language level debugging. There is the MacsBug program, but my favorite is Code Warrior: http://developer.apple.com/tools/mpw-tools/.

For MIPS, there is pretty much just the GNU C toolset, although CodeWarrior and SN Systems have development tools as well. The ProDG debugging tool is pretty cool, but they both come with a hefty price tag for console development: http://www.snsys.com.

Tuning and Optimization

There are multiple programs on the market for tuning and optimizing your application. Two popular performance monitoring applications are the VTune program by Intel for use with Intel's chips: http://developer.intel.com/software/products/eval

...and the AMD Code Analyst for use with AMD's chips: http://www.amd.com

These programs allow you to balance your pipelining, monitor the overall performance of your application, and make recommendations on how to balance your code to make it as tight as possible. By setting up monitoring zones indicated by start and stop markers, the application can be benchmarked between those zones.

Dang that 1.#QNAN

If you are trying to develop code that contains FPU versus MMX, especially if using a 3DNow! processor, do yourself a favor and first debug under SSE for floating-point and/or SSE2 for integer. SSE will not have to go through the dreaded EMMS switch, so there is no need for vmp_SIMDEntry() or vmp_SIMDExit(), although technically they are the same thing. Do yourself a favor and do your 3DNow! debugging last. If you encounter a floating-point value with a value of 1.#QNAN, this is typically the hidden reason a switch is between FPU and MMX.

Throughout this book, I have been oversimplifying, using merely vmp_SIMDEntry() and vmp_SIMDExit() to block off sections of code, but it is actually more complicated than that, as you truly need to differentiate between MMX and FPU. Remember that MMX can be floating-point on an AMD or integer or logical bit for all X86 processors. In this book's case, I did not use them on every function call like most books tell you to do. Instead, I used them between blocks of code, but for some processors, such as SSE, I overused them. Having multiple flavors of SIMD that can stub out helps to lighten your code even more so.

Print Output

To printf or not to printf, that is the question! Well, not to butcher Shakespeare, but sometimes the old ASCII text output is an excellent method of tracking down obscure bugs. This is not meant to displace the use of full-blown symbolic debuggers, but sometimes the old-fashioned printf is better. Note I said sometimes!

This is where things get interesting! (I keep saying that in this book, do I not?) With the C++ cout IOStream class, it would have to be overloaded and the output rerouted to your own device. The printf function is a little harder to capture, although the standard output could be reassigned. Some platforms, such as Win32, do not use printf. Instead, they use a string output function such as OutputDebugString(). What I personally like to do is create a thin abstraction layer (whether it be written in C++ or C) of log functions that route to one or more specified output devices. This is set up for two types of output, Debug, which is not shipped with the game, and Release, which is. For example:

```
LogWrite()
LogRDWrite()
```

They are both similar to the functionality of printf(), except they do not return a count of bytes written, and the function LogWrite() is a debug version that gets stubbed to nothing in a release build. LogRDWrite() is compiled for both a release and debug skew, which gets shipped with the code. This helps to simplify customer support's job when something really bad happens! There are other functions in the log module, but these two are at the core, and they would call printf() and/or OutputDebugString(), etc. The idea is that your cross-platform application would call the log functions and be abstracted for the individual platform.

When I develop Win32 applications, I frequently route this output to one or all of the following: the function OutputDebugString(), a Notepad window, a text file, a high-speed serial port with hardware handshake, or a TCP/IP communications setup for data collection on another computer. I do something similar for other embedded platforms. With a smart terminal there, I can back-scroll through live memory dumps to track behaviors.

One of the methods I frequently use with this type of diagnostics is to set threshold values to enable the trigger, which in effect allows the capture and printing of selected data. This helps to minimize being

inundated with massive amounts of data, which is too verbose and hinders rather than helps detecting problems.

As a superset of this functionality, the following functions are generated for even more detailed information on a generic basis!

For platforms that do not offer you the advantage of an IDE, or if you need to track problems and history dumps of memory or registers, some of the following functions should come in handy.

Float Array Print

Print dump a list of single-precision floats (four per line):

Listing 18-1: \chap18\vecBug\vecBug.cpp

```
void vmp_FloatPrint( const float * pf, uint nCnt )
{
  unsigned int n;
  int i;

  ASSERT_PTR4(pf);
  ASSERT_ZERO(nCnt);

  vmp_SIMDEntry();

  do {
    i = (--nCnt >= 4) ? 3 : (nCnt & 3);
    nCnt -= i;

    printf( "%8.8x ", pf );

    do {
      printf( "%lf ", (double)(*pf) );
      pf++;
    } while (i-- > 0);

    printf( "\n" );
  } while (nCnt);

  vmp_SIMDExit();
}
```

Once a float array function is in place, creating other dependent functions, such as the following, is a snap! Also by creating a debug versus release version, the debug would be the function type definitions, but for release, they would be empty macros, thus stubbing the code to nonexistence. Use something similar to the following:

```
#ifdef USE_LOG
void vmp_FloatPrint( const float * pf, uint nCnt );
#else
#define vmp_FloatPrint( pf, nCnt )   // stub
#endif
```

The same would need to be implemented in the logging header file for any other logging function that would need to be stubbed in a release build.

Vector Print

Listing 18-2: \chap18\vecBug\vecBug.cpp

```
void vmp_VecPrint( const vmp3DVector *pVec )
{
  vmp_FloatPrint( (float *)pVec, 3 );
}
```

Quad Vector Print

Listing 18-3: \chap18\vecBug\vecBug.cpp

```
void vmp_QVecPrint( const vmp3DQVector *pVec )
{
  vmp_FloatPrint( (float *)pVec, 4 );
}
```

Quaternion Print

Listing 18-4: \chap18\vecBug\vecBug.cpp

```
void vmp_QuatPrint( const vmpQuat *pVec )
{
  vmp_FloatPrint( (float *)pVec, 4 );
}
```

Matrix Print

This function dumps a single-precision floating-point matrix. It expands single-precision to double-precision to help minimize any problems of ASCII exponential representations.

Listing 18-5: \chap18\vecBug\vecBug.cpp

```
void vmp_MatrixPrint( const vmp3DMatrix Mx )
{
  ASSERT_PTR4(Mx);

  vmp_SIMDEntry();

  printf( "%8.8x ", Mx );          // Print Address
  printf( "%lf %lf %lf %lf\n",   (double)Mx[0][0],
      (double)Mx[0][1], (double)Mx[0][2], (double)Mx[0][3]);
  printf( "         %lf %lf %lf %lf\n",   (double)Mx[1][0],
```

```
      (double)Mx[1][1], (double)Mx[1][2], (double)Mx[1][3]);
  printf( "         %lf %lf %lf %lf\n",   (double)Mx[2][0],
      (double)Mx[2][1], (double)Mx[2][2], (double)Mx[2][3]);
  printf( "         %lf %lf %lf %lf\n\n", (double)Mx[3][0],
      (double)Mx[3][1], (double)Mx[3][2], (double)Mx[3][3]);

  vmp_SIMDExit();
}
```

Memory Dump

There is nothing sweeter than a memory dump similar to the old DOS (Disk Operating System) debug.exe program. A combination of address, hex data, and ASCII equivalents are all dumped on an up to 16-byte ASCII string. Even when you have a memory dump window within your IDE, it can typically only handle one dump at a time, and if you are trying to track a nasty little problem or verify proper functionality, sometimes a print trail is far superior. So with this in mind, the following should be a handy little tool for your development cycle:

Listing 18-6: \chap18\vecBug\vecBug.cpp

```
void vmp_LogMem( const void * const vp, uint size )
{
  uint  x, y, run, tail, col, n;
  char *p, buf[256];
  byte *mp;

  ASSERT_PTR(vp);
  ASSERT_ZERO(size);

  mp = (byte *) vp;

  run = 16;
  col = tail = 0;

    //  For all lines

  for ( y = 0; y < size;  )
  {
    if ( size-y < run )
    {
      run = size-y;          // Display only requested!
      n = 16 - run;          // trailing places
      tail = n * 3;          // trailing spaces
      tail += ((n+3) >> 2);  // column separators
    }
                             // Print address
    p = buf + sprintf(buf, "%8.8x ", mp );

      // Hex values

    for ( x = 0; x < run; x++ )
```

```
    {
      p += sprintf( p, " %2.2x", *(mp+x) );

      if (( ++col % 4 ) == 0 )
      {
        *p++ = ' ';
      }
    }

      //  Any trailing spaces ?

    while ( tail-- )
    {
      *p++ = ' ';
    }

      //  ASCII chars

    for ( x = 0; x < run; x++ )
    {
      if (( 0x20 <= *mp ) && ( *mp <= 0x7f ))
      {
        *p++ = *mp++;
      }
      else
      {
        *p++ = '.';
        mp++;
      }
    }

    *p++ = '\r';
    *p++ = '\n';
    *p = 0;
    printf( buf );

    y += run;
    run = 16;
    tail = col = 0;
    }

  printf( "\r\n" );
}
```

Test Jigs

One sometimes needs data to load vector algorithms during testing. These functions may seem downright silly, but they do come in handy from time to time.

Matrix Test Fill

The following function does just that. It fills a matrix with a sequential set of single-precision floating-point whole numbers. This helps to keep the floating-point stored in memory in a recognizable form when just doing a memory dump.

Listing 18-7: \chap18\vecBug\vecBug.cpp

```
void vmp_MatrixTestFill( vmp3DMatrix Mx )
{
  uint n;
  float *pF, f;

  f = 0.0f;              // Starting value of 0.0f
  pF = (float *)Mx;
  n = 16;

  do {
    *pF = f;
    f += 1.0f;           // Advance value by 1.0f
    pF++;
  } while (--n);
}
```

Matrix Splat

The function splats (replicates) a single-precision floating-point value into all 16 elements of a matrix.

Listing 18-8: \chap18\vecBug\vecBug.cpp

```
void vmp_MatrixSplat( vmp3DMatrix Mx, float f )
{
  uint n;
  float *pF;

  pF = (float *)Mx;
  n = 16;

  do {
    *pF = f;
    pF++;
  } while (--n);
}
```

There are other functions and test jigs that are available for use in the debugging of vector code, and they are available for purchase from middleware providers. They are also something that you can develop using your imagination.

Chapter 19
Epilogue

If you actually read this entire book and did not skip around here and there, you should have found a diverse spectrum of topics related to the computer processing of vectors using all the flavors of the X86 instruction set up through SSE2, PowerPC with AltiVec, and the new Multi-Media extensions to the MIPS processors, whether they contained three fields or four and were standard vectors or quaternions. You would have found how to use those in stock standard equations, such as distance, dot products, cross products, and matrices, just to name a few. You discovered the terrible penalty (and hidden problems) of not having your data properly aligned, how to detect problems early in your development cycle, the importance of using assertions and constant declarations within your code, and how vector processing pays off in some instances and not in others.

Also contained within these pages were answers to some of the most frequently asked questions that are typically encountered in a job interview and my insight into that and other topics of game or embedded development. A companion CD that contains practical problems and solutions despite the fact that they were not heavily optimized and algorithmically tuned but should have been understandable is included as well. There was even a peek into the new programmable vertex and pixel shaders that are out on the market and require an understanding of vector-based assembly code to program with, for which this book should have helped you prepare.

The principles found throughout this book can be applied to your project to make it that much better. There have been programming tricks, as well as other foundational elements for you to use and build upon to enhance your knowledge of vectors and game building from a vector point of view. That is what 3D games are all about! I hope you have found this book to be informative, learned at least one new item, and got your money's worth. Please feel free to contact me and let me know what you would like to see in any future revision of this book and your opinion on the growth of the industry.

New processors come out periodically with enhanced instruction sets, and so revisions of this book make perfect sense, especially with the Intel Itanium and AMD X86-64 chips on the horizon. However, you should also realize that because this book was designed and written with multiple platforms in mind, technological enhancements and characteristics of one processor easily carries over to another. The supporting registers and method of data access may change, but the implementation is generally along the same lines; without having had multiple processor families discussed in this book, that would not have been possible.

I am sorry the code included with this book is not heavily optimized, but as mentioned in the beginning of this book, "optimized code is difficult to understand." An unmentioned reason for not doing so is one of greed! As I mentioned in regards to the Inverse Discrete Cosine Transform (IDCT), there is gold in them thar' hills! Companies make money writing heavily optimized — thus very, very fast — algorithms. You too can possibly make lots of money from the comfort of your home writing optimized libraries targeted for a specific processor! In fact, consider any algorithms you find in this book that you need in your code as a homework assignment of optimization.

I hope this book and its related topics were as pleasant for you to read as it was fun for me to research and write about. In the process of writing this book, I too learned a few new things and that is what it is all about!

Please send an e-mail to books@leiterman.com to register your e-mail address as a purchaser of this book. Insert the word "Register" into the Subject field, and in the message section, list your processors of interest and any other comment. I can add you to an update notification list to let you know when additional code samples or chapter supplements are available for download. You can also just make a statement or remark about this book. A side benefit for me is that I can chart the sales curve of the book, which will give me an indicator as to the interest into this topic, as well as note likes and dislikes to keep them in mind. For additional information or related links, check out the associated web site: http://www.leiterman.com/books.html.

Oh, and one last item: Buy my brother's books as well. He has a large family too and can use some extra income!

Appendix A

Data Structure Definitions

When dealing with three-dimensional transformation, data structures are needed to contain the data type. The data structures listed here will be discussed in terms of this book.

Integer 2D Point

This is mostly used within a 2D game but has uses within a 3D game as well. It is used for specifying a starting point within a bitmap image with the origin {0,0} located in the upper-left corner. This is primarily used as the starting point of a texture within a bitmap image. It is also used in relationship to the anchor point of a font baseline as where to position it within an image or two-triangle quad polygon tile.

```
typedef struct iPosType
{
  int   x;
  int   y;
} iPos;
```

Integer 2D Size

Normally bitmaps range from 0 to w–1 for width and 0 to h–1 for height and are never negative. With that in mind, only unsigned arguments are used.

```
typedef struct iSizeType
{
  uint  w;      // Width
  uint  h;      // Height
} iSize;
```

Integer Rectangle

Some rectangle structures are designed with a field definition, such as {x1,y1,x2,y2}, but any movement has to be translated with the modification of four coordinates. With the following structure {x,y,w,h}, only the two fields {x,y} need to be modified.

```
typedef struct iRectType
{
  int   x;      // X coord.
  int   y;      // Y coord.
  uint  w;      // Width
  uint  h;      // Height
} iRect;
```

3D Vector (Integer)

```
typedef struct iVector3DType
{
  int   x;
  int   y;
  int   z;
} iVector3D;
```

3D Quad Vector (Integer)

```
typedef struct iQVector3DType
{
  int   x;
  int   y;
  int   z;
  int   w;
} iQVector3D;
```

3D Vector (Floating Point)

```
typedef struct vmp3DVector
{
  float x;
  float y;
  float z;
} vmp3DVector;


  vmp3DVector point =  {0.0, 0.0, 0.0};
  vmp3DVector vector = {1.0, 2.0, 3.0};
```

3D Quad Vector (Floating Point)

```
typedef struct vmp3DQVector
{
  float x;
  float y;
  float z;
  float w;
} vmp3DQVector;


vmp3DQVector point =  {0.0, 0.0, 0.0, 0.0};
vmp3DQVector vector = {1.0, 2.0, 3.0, 0.0};
```

Quaternion (Single-Precision Floating-Point)

```
typedef struct vmpQuat
{
  float x;
  float y;
  float z;
  float w;
} vmpQuat;
```

Appendix B
Glossary

— A number

AI — Artificial intelligence. Simulation by a computer to simulate human behavior and intelligence

alpha channel — A field within an RGBW (Red, Green, Blue, Alpha) color value representing the level of opacity and/or transparency

ALU — Algorithmic Logic Unit

AoS — Array of Structures

ASCII — American Standard Code for Information Interchange. A 7-bit numerical representation used to represent control characters and the standard English language character set

ASE — ASCII **S**cene **E**xporter (3D Studio MAX)
Application **S**pecific **E**xtension (MIPS-3D)

BCD — Binary Coded Decimal notation

bi-Endian — A byte ordering of either big endian or little endian supported by a processor

	(Byte)	0	1	2	3	
0x1A2B3C4D		1A	2B	3C	4D	Big
		4D	3C	2B	1A	Little

Note that the byte is endianless; that is, whether it is big endian or little endian, the MSB (most significant bit) is bit #7 and the LSB (least significant bit) is bit #0.

big endian — The byte ordering typically used by large mainframes. For purposes of this book, that would include the EE, VU, PowerPC-AltiVec, and PowerPC-Gekko processors.

	0	1	2	3
0x1A2B3C4D	1A	2B	3C	4D

BLIT — The process of transferring one or more blocks of data. The etymology of the word is Bacon, Lettuce, and Interactive Tomato.

CD — Compact Disc (540 MG to 700 MG)

CODEC — Compression/decompression

compiler — A software tool that converts symbolic source code into object code

coprocessor — A secondary processor that adds enhanced functionality to a primary processor

CPU — Central Processing Unit

culling — A process of reducing the number of polygons needed to be passed to the rendering engine

delta frame — The compression information to alter the current frame to be similar to the next frame in an animated sequence

diffuse reflection — A component of reflected light that is diffused in all directions

Digital Millennium Copyright Act — *See* Appendix C

DOS — Disk Operating System

double-precision — A long format (64-bit) storage of floating-point data

DSP — Digital Signal Processing

DV — Digital Video

DVD — Digital Versatile Disk (storage capacity 17 GB)

EE — Emotion Engine

extended double-precision — An extra long format (80-bit) storage of floating-point data

fixed-point — A number in which the decimal point is fixed to a number of places

floating-point — A number in which the decimal point is floating, thus it can be in any position. But it is typically stored in a sign, exponent, and mantissa component.

FMV — Full Motion Video

fogging — The blending of color with an image to increasingly obscure distant objects

FPU — Floating Point Unit

GCC — GNU C Compiler

Gekko — (1) The superset of the PowerPC processor manufactured by IBM for the Nintendo GameCube. (2) A lizard (gecko)

GFLOPS — **G**iga (Billion) **FLO**ating-point operations **P**er **S**econd

GNU — **G**nu is **n**ot **U**NIX

GPU — Graphics Processing Unit

GRDB — Game Relational Database

GSCube — A product based upon interconnecting 16 Emotion Engines and Graphic Synthesizers

IDE — Integrated Development Environment

IEEE — Institute of Electrical and Electronics Engineers

JPEG — Joint Photographic Experts Group

key Frame — An image containing a complete set of data to recreate itself

little endian — The byte ordering used by most modern computers. For purposes of this book, that would include the X86 and MIPS processor. Although a MIPS processor can be configured for big endian, for game consoles it is used in a little-endian configuration.

	0	1	2	3
0x1A2B3C4D	4D	3C	2B	1A

LSB — Least significant bit. The related bit depends upon the endian orientation.

MDMX — MIPS Digital Media Extension (MaDMaX)

MFLOPS — **M**illion **FLO**ating point operations **P**er **S**econd

MIMD — Multiple Instruction Multiple Data. A computer instruction that performs the multiple instructions in parallel for a series of isolated packed data blocks

MIPS — **M**illion **I**nstructions **P**er **S**econd. *See* Appendix C for MIPS organization.

MMX — Multimedia Extensions

MPEG — Moving Picture Experts Group

MSB — Most significant bit. The related bit depends upon the endian orientation.

open source — A standard of the Open Source Initiative (OSI) that makes available the source code of a computer program free of charge to the general public.

PETAFLOPS — **10^{15} FLO**ating-point operations **P**er **S**econd

polygon — In the context of 3D rendering, a graphical primitive within a closed plane consisting of a three-sided (triangle) or four-sided (quadrilateral) representing a face typically covered by a texture

PS2 — PlayStation 2

RGB — Red Green Blue

scalar processor — A processor that can perform only one instruction on one data element at a time. *See* superscalar processor *and* vector processor.

SIMD — Single Instruction Multiple Data. A computer instruction that performs the same instruction in parallel for a series of isolated packed data blocks

Single-precision floating-point — A standard format (32-bit) storage of floating-point data

SoA — Structure of Arrays

specular reflection — A component of reflected light at a point on a surface regulated by the direction of the incidental light source in conjunction with the viewing angle in relation to the normal of the surface

squirrelly — A term I use in response to an attempt to explain the behavior of a piece of code

superscalar processor — A processor that performs similar to a scalar processor but can handle multiple data operations simultaneously. *See* scalar processor *and* vector processor.

TERAFLOPS — **T**rillion **FLO**ating-point operations **P**er **S**econd

texture — A 2D image that is mapped upon a 3D wireframe polygon to represent its surface

"thinking out of the box" — An expression used to indicate creative thinking outside the limits of what is typically considered normal (what this book is trying to equip you to be able to do!)

vector — (1) A pointer to code or data typically used in a table (vector table). (2) A one-dimensional array. (3) A line defined by starting and ending points

vector processor — A processor that performs an instruction on an entire array of data in a single step. *See* scalar processor *and* superscalar processor.

vertex — The intersection of two vectors used to define a corner of a polygon, for example, three corners of a triangle, eight corners of a cube

vertex normal — A direction vector perpendicular to the plane intersecting the three vertices of a triangle

VU — Vector Units

w-buffer — A rectangular representation of the image buffer used to store the distance of each pixel of the image from the camera. The range of possible Z values are linearly distributed between the camera and a point in 3D space depicted as infinity. The distances from the camera are finer in resolution then those closer to infinity, allowing for a more refined depth of view.

z-buffer — A rectangular representation of the image buffer used to store the distance of each pixel of the image from the camera. The range of possible Z values are uniformly distributed between the camera and a point in 3D space depicted as infinity.

Alignment Macros

```
#define ALIGN2(len) ((len+1) & ~1)    // round up to 16bits
#define ALIGN4(len) ((len+3) & ~3)    // round up to 32bits
#define ALIGN8(len) ((len+7) & ~7)    // round up to 64bits
#define ALIGN16(len) ((len+15) & ~15) // round up to 128bits
#define ALIGN32(len) ((len+31) & ~31) // round up to 128bits

#define ALIGN2048(len) (( len + 2047 ) & ~2047 )
                                      // round up to 1 CD Sector
```

Algebraic Laws Used in This Book

Additive Identity	n + 0 = 0 + n = n
Multiplicative Identity	n1 = 1n = n

Additive Inverse	a – b = a + (–b)

Commutative Law of Addition	a + b = b + a
Commutative Law of Multiplication	ab = ba

Distributive	a(b+c) = ab+ac	(b+c)/a = b/a + c/a

Appendix C
References

AMD (Advanced Micro Devices): http://www.amd.com

Intel: http://www.intel.com

Millennium Copyright Act: http://www.loc.gov/copyright/legislation/dmca.pdf

ASE File Format

3DS Max ASCII Scene Exporter Description: http://www.solosnake.fsnet.co.uk/main/ase.htm

Game Development Links

Gamasutra — Game Developers: http://www.gamasutra.com

Game Developer magazine: http://www.gdmag.com

Game Development Magazine: http://www.gignews.com

Game Development Search Engine: http://www.game-developer.com

GDC — Game Developers Conference: http://www.gdconf.com

Online community for game developers of all levels: http://www.gamedev.net

MIPS

An Architecture Extension for Efficient Geometry Processing: http://www.hotchips.org/pubs/hc99/hc11_pdf/hc99.s8.1.Thekkath.pdf.

Farquhar & Bunce. *The MIPS Programmer's Handbook.* Morgan Kaufmann Publishers, Inc. 1994. ISBN 1-55860-297-6.

Green Hills Software, Inc.: http://www.ghs.com

MIPS-3D Graphics Extension. MIPS Technologies, Inc. MIPS-3D.pdf.

MIPS64™ 5K Processor Core Family Software User's Manual. MD00012 Revision 2.06, June 28, 2001. MIPS Technologies, Inc. MD00012-2B-5K-SUM-02.06.pdf.

MIPS64™ Architecture for Programmers — Volume I: Introduction to the MIPS64™ Architecture. MD00083 Revision 0.95, March 12, 2001. MIPS Technologies, Inc. MD00083-2B-MIPS64INT-AFP-00.95.pdf.

MIPS64™ Architecture for Programmers — Volume II: The MIPS64™ –Instruction Set. MD00087 Revision 0.95, March 12, 2001. MIPS Technologies, Inc. MD00087-2B-MIPS64BIS-AFP-00.95.pdf.

MIPS64™ Architecture for Programmers — Volume IV-c: The MIPS-3D™ Application-Specific Extension to the MIPS64™ Architecture. MD00099 Revision 1.11, March 12, 2001. MIPS Technologies, Inc. MD00099-2B-MIPS3D64-AFP-01.11.pdf.

MIPS Extension for Digital Media with 3D. MIPS Technologies, Inc. isa5_tech_brf.pdf.

MIPS Technologies, Inc.: www.mips.com.

Multimedia-Erweiterungen von MIPS by Steffen Sclotzer, September 26, 1999: http://www2.informatik.uni-jena.de/~mau/seminare/SS99/schloetze/Prozsem3.pdf

Sweetman, Dominic. *See MIPS Run*. Morgan Kaufmann Publishers, Inc. 1999. ISBN 1-55860-410-3.

JPEG

JPEG (Joint Photographic Experts Group): http://www.jpeg.org

MPEG

Mitchell, Joan L. (Editor), Didier J. Legall (Editor), William B. Pennebaker (Editor), Chad E. Fogg (Editor). *MPEG Video Compression Standard*. Chapman & Hall Publications. November 1996. ISBN: 0412087715.

MPEG (Moving Picture Experts Group): http://www.mpeg.org

Project Mayo — Open DivX;-): http://www.projectmayo.com

Watkinson, John. *MPEG-2*. Butterworth-Heinemann, 1998. ISBN 02405015-102.

SHA-1

Secure Hash Algorithm: http://www.itl.nist.gov/fipspubs/fip180-1.htm.

Vertex and Pixel Shaders

ATI: http://www.ati.com

Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks. Wolfgang F. Engel (Editor). Wordware Publishing: ISBN: 1556220413.

nVIDIA: http://www.nvidia.com

Dreamcast

Hitachi SH7750 (SH-4 Series) RISC Processor Manuals: http://www.hitachi-eu.com/hel/ecg/products/micro/32bit/sh_4.html

Macintosh

AltiVec Forum: http://forum.altivec.org

Apple Macintosh Developers program:
http://www.apple.com/developer
http://developer.apple.com/tools

Nintendo GameCube

"GameCube clears path for game developers." 17 May, 2001: http://www.eetimes.com/story/OEG20010516S0056

Interview: IBM Details Gekko (Part 1), IGN.COM. 12, Dec. 2001: http://cube.ign.com/articles/100/100445p1.html

Nintendo: http://www.nintendo.com

Personal Computer

Charlie Chaplin IBM Promotion: http://www.multimania.com/myibm/pub_chaplin.htm

IBM Personal Computer: http://www-1.ibm.com/ibm/history/catalog/itemdetail_57.html

Sony PlayStation2 Game Console

(*See also* MIPS)

"Game console 'could be used in missiles,'" by Coin Joyce. 17 April, 2000: http://www.telegraph.co.uk, Issue#1788.

Linux Kit manuals:

INST_0E.PDF	EE Core Instruction Set Manual
COREUM_E.PDF	EE Core User's Manual
EEOVER_E.PDF	EE Overview
EEUSER_E.PDF	EE User's Manual
GSUSER_E.PDF	GS User's Manual
VU0E.PDF	VU User's Manual

Playstation 2 Developer Network (Linux Community): http://playstation2-linux.com/

Sony PlayStation2: http://www.scea.com

"Toshiba and MIPS Technologies Agree to Joint Development of Next Generation RISC-based Microprocessors," WashingtonPost.com, 18 February, 2002: http://www.newsbytes.com/bizwire/02/356390.html

Toshiba / ArTile — TX-79:
http://www.semicon.toshiba.co.jp/prd/risc/ft_risc.html
http://www.semicon.toshiba.co.jp/prd/risc/pdf/TX79Architecture.pdf

"Toshiba, MIPS to develop fastest 64-bit embedded processor based on 'Amethyst' core." Semiconductor Business News, 18 February, 2002: http://www.siliconstrategies.com/story/OEG20020218S0033

TP 15.1 — *A Microprocessor with a 128b CPU, 10 Floating Point MACs, 4 Floating-Point Dividers, and an MPEG2 Decoder*, 1999 IEEE International Solid-State Circuits Conference:
http://www.ieee.org.conferences
0-7803-5129-0/99; 1999_15_1.pdf
1999 ISSCC Slide Supplement.
1999_s15_1.pdf

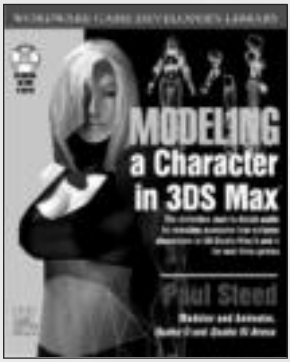
TP 15.2 — *A High Bandwith Superscalar Microprocessor for Multimedia Applications*, 1999 IEEE International Solid-State Circuits Conference:
http://www.ieee.org.conferences
0-7803-5129-0/99; 1999_15_2.pdf
1999 ISSCC Slide Supplement.
1999_s15_2.pdf

Why Iraq's buying up Sony Playstation 2s — "Intelligence experts fear games bundled for military applications," by Joseph Farah. 19 December, 2000: http://www.worldnetdaily.com

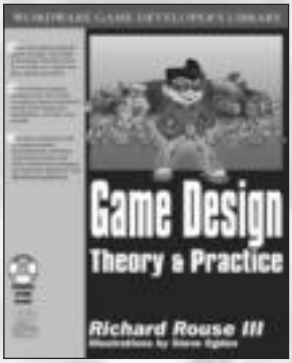
Index

Looking for more?

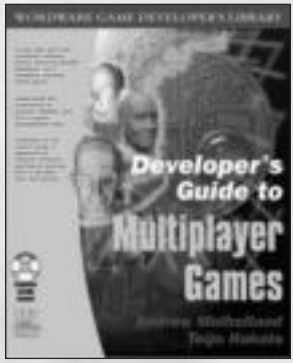
Check out Wordware's market-leading Game Developer's Library featuring the following new releases.



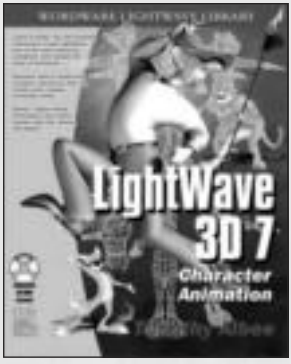
Modeling a Character in 3DS Max
1-55622-815-5
$44.95
7½ x 9¼ 544 pp.



Game Design: Theory & Practice
1-55622-735-3
$49.95
7½ x 9¼ 544 pp.

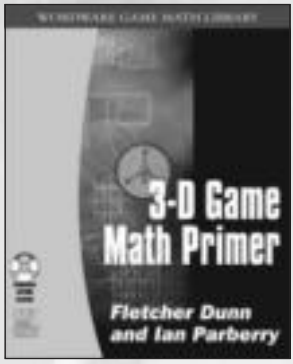


Developer's Guide to Multiplayer Games
1-55622-868-6
$59.95
7½ x 9¼ 608 pp.



LightWave 3D 7 Character Animation
1-55622-901-1
$49.95
7½ x 9¼ 360 pp.

Advanced 3-D Game Programming Using DirectX 8.0
1-55622-513-X
$59.95
7½ x 9¼ 592 pp.

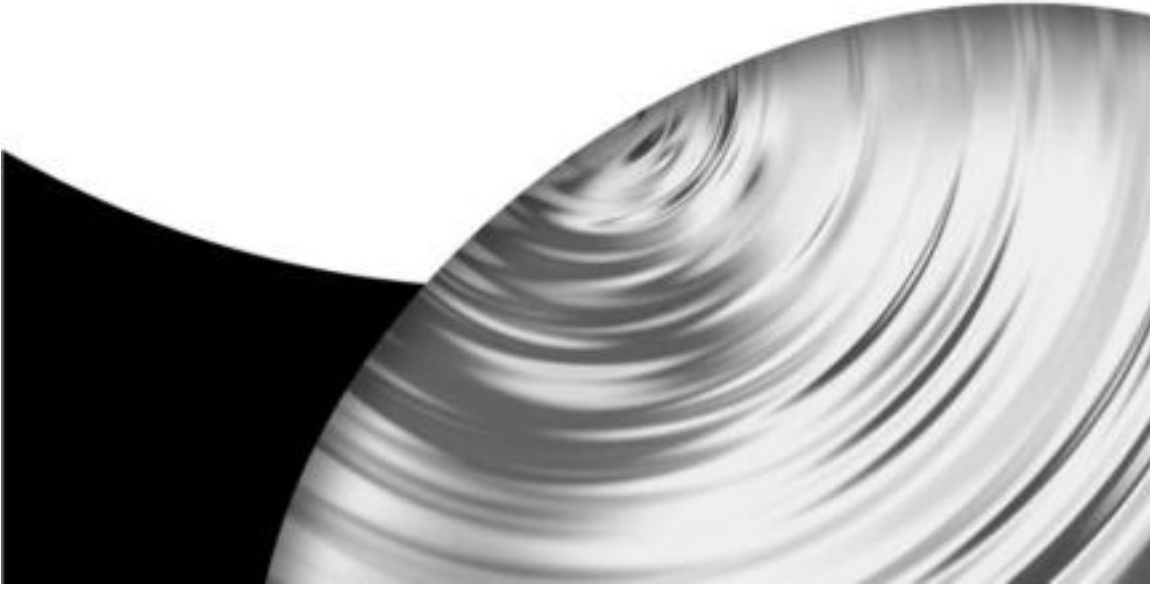


3D Math Primer for Graphics and Game Development
1-55622-911-9
$49.95
7½ x 9¼ 448 pp.

Visit us online at www.wordware.com for more information.
Use the following coupon code for online specials: ShaderX-0413

nVSDK

developer.nvidia.com



3D Graphics Programming With Direct3D

Examines the premier 3D graphics programming API on the Microsoft Windows platform. Create a complete 3D game engine with animated characters, light maps, special effects, and more.

3D Graphics Programming With OpenGL

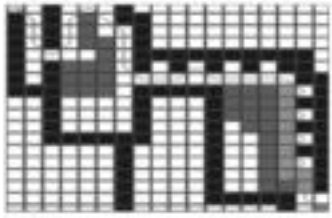
An excellent course for newcomers to 3D graphics programming. Also includes advanced topics like shadows, curved surfaces, environment mapping, particle systems, and more.

Advanced BSP/PVS/CSG Techniques

A strong understanding of spatial partitioning algorithms is important for 3D graphics programmers. Learn how to leverage the BSP tree data structure for fast visibility processing and collision detection as well as powerful CSG algorithms.

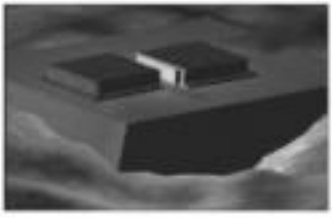
Real-Time 3D Terrain Rendering

Take your 3D engine into the great outdoors. This course takes a serious look at popular terrain generation and rendering algorithms including ROAM, Rottger, and Lindstrom.



Path Finding Algorithms

Study the fundamental art of maneuver in 2D and 3D environments. Course covers the most popular academic algorithms in use today. Also includes an in-depth look at the venerable A*.



Network Game Programming With DirectPlay

Microsoft DirectPlay takes your games online quickly. Course includes coverage of basic networking, lobbies, matchmaking and session management.

MORE COURSES AVAILABLE AT

www.GameInstitute.com


THIS
CHANGES
EVERYTHING
NOW WITH
128
MB
OF DDR
MEMORY!
RADEON 8500
Brand yourself a warrior with the groundbreaking, high-resolution 3D graphics of RADEON™ 8500 now with 128MB of memory for lightning fast 3D gaming. Get the most out of today's hottest 3D games and experience the most immersive 3D gaming imaginable. RADEON™ 8500 changes everything.
ATI.COM

About the Companion CD

The companion CD contains two file types: SIT files typically used by Macintosh and ZIP files used by other platforms. Each processor type is supported by its own file.

MIPS	vmp_mips.zip	vmp_mips.sit
SH4	vmp_sh4.zip	vmp_sh4.sit
PowerPC	vmp_ppc.zip	vmp_ppc.sit
X86	vmp_x86.zip	vmp_x86.sit

By unzipping a file into a working folder such as \Bench, all the files for that processor type will be unpacked into that root folder. If all files are unpacked, the basic root structure will be similar to the following:

\Bench\vmp_mips\	Supported Chapters: {2-4, 6-11, 16}
\incmips	
\chap02	
:	
\chap16	
\vmp_sh4\chap12	Supported Chapters: {12}
\vmp_ppc\	Supported Chapters: {3-4, 6-11, 14}
\incppc	
\chap03	
:	
\chap14	
\vmp_x86\	Supported Chapters: {2-4, 6-14, 16, 18}
\incx86	
\chap02	
:	
\chap18	

You have the choice of unpacking all the processors or only the ones you desire. As mentioned in the book, please note that the X86 was the primary processor in discussion and so not all processor types have supported code shipped with the book. This was primarily due to time constraints of writing the book and the accompanying code. If some code used with other processor types is of interest to you, check out the book's web page at http://www.leiterman.com/books.html. Or e-mail the author at books@leiterman.com for requests.

Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement.

Additionally, opening the CD package makes this book nonreturnable.

CD/Source Code Usage License Agreement

Please read the following CD/Source Code usage license agreement before opening the CD and using the contents therein:

1. By opening the accompanying software package, you are indicating that you have read and agree to be bound by all terms and conditions of this CD/Source Code usage license agreement.
2. The compilation of code and utilities contained on the CD and in the book are copyrighted and protected by both U.S. copyright law and international copyright treaties, and is owned by Wordware Publishing, Inc. Individual source code, example programs, help files, freeware, shareware, utilities, and evaluation packages, including their copyrights, are owned by the respective authors.
3. No part of the enclosed CD or this book, including all source code, help files, shareware, freeware, utilities, example programs, or evaluation programs, may be made available on a public forum (such as a World Wide Web page, FTP site, bulletin board, or Internet news group) without the express written permission of Wordware Publishing, Inc. or the author of the respective source code, help files, shareware, freeware, utilities, example programs, or evaluation programs.
4. You may not decompile, reverse engineer, disassemble, create a derivative work, or otherwise use the enclosed programs, help files, freeware, shareware, utilities, or evaluation programs except as stated in this agreement.
5. The software, contained on the CD and/or as source code in this book, is sold without warranty of any kind. Wordware Publishing, Inc. and the authors specifically disclaim all other warranties, express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose with respect to defects in the disk, the program, source code, sample files, help files, freeware, shareware, utilities, and evaluation programs contained therein, and/or the techniques described in the book and implemented in the example programs. In no event shall Wordware Publishing, Inc., its dealers, its distributors, or the authors be liable or held responsible for any loss of profit or any other alleged or actual private or commercial damage, including but not limited to special, incidental, consequential, or other damages.
6. One (1) copy of the CD or any source code therein may be created for backup purposes. The CD and all accompanying source code, sample files, help files, freeware, shareware, utilities, and evaluation programs may be copied to your hard drive. With the exception of freeware and shareware programs, at no time can any part of the contents of this CD reside on more than one computer at one time. The contents of the CD can be copied to another computer, as long as the contents of the CD contained on the original computer are deleted.
7. You may not include any part of the CD contents, including all source code, example programs, shareware, freeware, help files, utilities, or evaluation programs in any compilation of source code, utilities, help files, example programs, freeware, shareware, or evaluation programs on any media, including but not limited to CD, disk, or Internet distribution, without the express written permission of Wordware Publishing, Inc. or the owner of the individual source code, utilities, help files, example programs, freeware, shareware, or evaluation programs.
8. You may use the source code, techniques, and example programs in your own commercial or private applications unless otherwise noted by additional usage agreements as found on the CD.

Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement.

Additionally, opening the CD package makes this book nonreturnable.