# AltiVec/SSE Migration Guide

Performance > Vector Engines

2005-09-08

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

**Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.**

**IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.**

**THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.**

**Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.**

# Contents

# Figures and Tables

# Introduction to AltiVec/SSE Migration Guide

*AltiVec/SSE Migration Guide* will assist experienced developers who need to migrate their vector-oriented code from the PowerPC AltiVec extensions to the Intel x86 SSE extensions. Both of these are sets of SIMD (single instruction, multiple data) instructions, accessible through C intrinsics. The instructions operate on special sets of 128-bit registers that can be used to hold vectors of smaller-sized data, to be operated on in parallel.

The two sets of instructions serve the same purposes, but are implemented differently; porting of algorithms from one to the other must be done carefully.

Most work involving vector-oriented calculations can be done via Apple's Accelerate frameworks, which provide higher-level functions for image processing, signal processing, linear algebra, vector math, and operations on large numbers. The advantage of using these frameworks is that the hardware dependencies are abstracted away by highly optimized library code that will be maintained not only for PowerPC and Apple's initial Intel processors, but also for future processors.

Developers who have already written AltiVec code should consider adopting the Accelerate frameworks, instead of porting to SSE. However, some developers will need to port their code, or to write new AltiVec and SSE versions of new algorithms. Similarly, those who are porting Windows applications to Mac OS X may need to port existing SSE code to AltiVec.

**Important:** This is a preliminary document for an application binary interface (ABI) in development. Although this document has been reviewed for technical accuracy, it is not final. Apple Computer is supplying this information to help developers plan for the adoption of the technologies and programming interfaces described herein. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the ABI. For information about updates to this and other developer documentation, view the New & Updated sidebars in subsequent seeds of the ADC Reference Library.

## Who Should Read This Document

Any developer who needs to port existing AltiVec code to SSE or vice versa, or who needs to write custom-optimized code for both architectures.

## Organization of This Document

This document is organized into the following chapters:

- "AltiVec to SSE Migration Overview" (page 9) This chapter introduces basic information on migrating vector-oriented code from the PowerPC AltiVec extensions to the Intel x86 SSE extensions.

- "Programming SSE in C" (page 15) This chapter describes the intrinsics and data types provided for programming SSE in C.
- "Translating Altivec to SSE" (page 19) This chapter provides in-depth tutorial information on translating AltiVec to to SSE code.

## Assumptions

The document assumes the following:

- Your application runs in Mac OS X.

  Your application can use any of the Mac OS X development environments: Carbon, Cocoa, Java, or BSD UNIX.

  If your application runs in a version of the Mac OS that is earlier than Mac OS X version 10.0, you should first read Carbon Porting Guide and Technical Note TN2003 Moving Your Code to Mac OS X.

  If your application runs in the UNIX operating system but not specifically in Mac OS X, you should first read Porting UNIX/Linux Applications to Mac OS X.

  If your application runs only in the Windows operating system, you should first read Porting to Mac OS X from Windows Win32 API.

- You know how to use Xcode.

  Currently Xcode is the only GUI tool available that compiles code to run universally.

  If you are unfamiliar with Xcode, you might want to take a look at Xcode 2.1 User Guide.

  If you have been using CodeWarrior, you should read Moving Projects from CodeWarrior to Xcode.

## Conventions

The term **x86** is a generic term used in some parts of this book to refer to the class of microprocessors manufactured by Intel. This book uses the term x86 as a synonym for IA-32 (Intel Architecture 32-bit).

# AltiVec to SSE Migration Overview

Intel's Streaming SIMD Extensions, or "SSE" is a 128-bit SIMD vector extension to the x86 ISA that is quite similar to AltiVec. Most of the good practices for AltiVec apply. These include enabling full compiler optimizations, function call inlining, proper alignment and organization of data, attention to pipeline latencies, dispatch limitations, etc. As always, the largest opportunities for performance improvement comes from high level optimization techniques, most importantly choosing the right algorithm. The same goes for PowerPC vs. x86 in general.

However, there are some key differences between the two. For a broad overview of general tips and techniques for writing universal binaries, please see:Universal Binary Programming Guidelines.

A good source of x86 specific tuning advice and architectural documentation is Intel's web site. In particular, please see the processor optimization reference manual and accompanying software developers manuals: Intel Pentium References

There are also a number of very interesting (though in many cases highly speculative) resources available on the web to help you better understand Pentium behavior.

This document is intended to be an addendum to the above sources with information specifically relevant to tuning for SSE and high performance programming in general. It is targeted specifically towards the segment of the developer population that is already knowledgeable about high performance programming using AltiVec, especially those people with a substantial investment in AltiVec who would like to leverage that investment moving forward onto the Intel architecture.

Before we begin, we would like to strongly urge developers who are starting the process of porting AltiVec code to SSE to look to see if this work has already been done for you in Accelerate.framework. There has been a large body of work added to Accelerate.framework in recent years that you may not have been able to take advantage of previously, for reasons that may no longer exist. We recommend taking a few minutes to take a look. Accelerate.framework does signal processing (vDSP.h), image processing (vImage.h), linear algebra (BLAS/LAPACK), vector math library (vMathLib), and large integer computation (vBasicOps.h, vBigNum.h). The framework will transparently select the best code for the appropriate CPU, be that G3, G4, G5 or Pentium. In many cases, you don't have to know anything about vector programming to use Accelerate.framework.

## AltiVec and SSE

What we are calling SSE in this document was actually delivered as three separate vector extensions to the IA-32 ISA, which appeared (in order over time) under the names SSE, SSE2 and SSE3. Each builds on the extension that went before it. The first two are defined to be part of the baseline hardware requirement for MacOS X for Intel. SSE3 has been recently introduced (first in the Prescott family of Pentium 4 processors) and may or may not be available on a machine running MacOS X for Intel. In addition, another vector extension, MMX, was available before SSE was introduced. It does packed integer arithmetic in a separate 64-bit register file that aliases to the x87 FPU register set, the scalar floating point unit (used only for long double on MacOS X for Intel.) It is also a defined part of the MacOS X for Intel, but for reasons explained later does not get as much use. All of these vector extensions are also defined for EM64T and AMD64.

AltiVec and SSE are quite similar at the highest levels. They are SIMD vector units with the same vector size (128-bits) and a similar general design. SSE adds several important new features compared to AltiVec. The single and double precision floating point engines are fully IEEE-754 compliant, which means that all four rounding modes, exceptions and flags are available. Misaligned loads and stores are handled in hardware. There is hardware support for floating point division and square root. There is a Sum of Absolute Differences instruction for video encoding. All of the floating point operations provided are available in both scalar and packed variants. These features will be described in more detail in later sections.

## Hardware Overview

*Registers*

The Streaming SIMD Extensions define a set of 8 named 128-bit wide vector registers, called XMM registers. These are a flat register file like AltiVec. It is not stack based like the x87 register file. It has no special purpose registers like the x86 integer register file. On our ABI, all eight registers are volatile. For EM64T, the register file grows to 16 registers. (Note: Apple has not yet defined a ABI for 64-bit programming on MacOS X for Intel. 06/24/05)

In addition, there is a parallel set of 64-bit MMX registers that are used by the MMX extension to x86. The MMX register file aliases the x87 floating point register stack. Use of MMX causes an automatic x87 state save. The x87 unit will not function properly until you issue a EMMS instruction. (Use _mm_empty() for this.) Thus, MMX and x87 are mutually exclusive and may not be used at the same time. There is however no piece of hardware or software that is in place to prevent you from making this mistake. Unsurprisingly, failure to call _mm_empty() or use of MMX concurrently with x87 floating point code is a common mistake for people new to MMX. In certain cases, the paranoid may choose to use compiler devices like `-mno-mmx` flag to prevent unintentional MMX use, although such measures do not provide complete automatic protection. The flag does nothing to prevent use of those segments of SSE or SSE2 that use the MMX register file.

*Pipelines, Latencies and Unrolling*

There is quite a bit of variability between implementations of x86 based processors. Small parts of the design get regular tweaking even in minor updates to the processor. It is difficult to make sweeping generalization about the exact operation of various stages of the x86 pipelines: fetch, decode, dispatch, issue, execution and completion. Please see processor specific Intel documentation for a more complete description of the particular performance characteristics of each processor that you are targeting.

Generally speaking, the smaller register file on the x86 architecture compared to PowerPC is backed by a much larger reorder buffer, to reorder the execution of instructions to keep pipelines full. From the perspective of a developer experienced with AltiVec, it may initially appear difficult to keep pipelines full with eight registers. While this would be true of a strictly in-order architecture, the large reorder window allows the processor to pull future instructions forward to fill gaps in the pipelines to help make sure that the processor stays full. The processor may pull instructions forward from the next loop iteration. Indeed, in some cores it may not be uncommon to see several loop iterations unrolled in hardware in the reorder buffers. This process occurs transparently to the developer and may perform differently on different cores.

Utilizing a heavily out-of-order core may mean that your approach to unrolling your code may need to be different. Whereas in AltiVec it may have been a good idea to unroll up to eight-way in parallel, on SSE this will most likely overflow the register file. That will cause the compiler to spill temporary data onto the stack, introducing a large number of extra loads and stores into the critical code path, likely slowing things down dramatically.

Here is a code example unrolled two-way in parallel:

```
for( i = 0; i < N - 1; i+= 2 )
{
    float a0 = in[0]; float a1 = in[1]; in += 2;
    float b0 = in2[0]; float b1 = in2[1]; in2 += 2;
    a0 += b0; a1 += b1;
    a0 *= 3.14159f; a1 *= 3.14159f;
    out[0] = a0; out[1] = a1; out += 2;
}
```

It is important to minimize register spillage on x86. The right thing to do on x86 is usually to either not unroll at all (cores with a trace cache) or unroll serially (cores without a trace cache). Either approach should keep the pipelines full, presuming that the core of the loop is not so large that the distance that the processor needs to look ahead to find parallel calculation streams exceeds the size of the reorder buffer. Serial unrolling is a way to eliminate a few test and branch instructions. However, if the processor core has a trace cache, this advantage will often be more than offset by the cost of flushing more microcode out of the cache to make room for the unrolled loop.

Here is a code example unrolled two-way serially:

```
for( i = 0; i < N - 1; i+= 2 )
{
    //First loop iteration
    float a = in[0]; float b = in2[0]; a += b;
    a *= 3.14159f;
    out[0] = a;
    //second iteration
    a = in[1]; b = in2[1]; a += b;
    a *= 3.14159f;
    out[1] = a;
    in += 2; in2 += 2; out += 2;
}
```

For many SSE instructions, the second (non-destination) instruction argument may be a direct reference to memory instead of a register. Direct memory references are a good way to save registers, since they allow you to make use of data without first needing to load it into a named register. Make no mistake, the load still happens. The out-of-order processor core is probably doing a load behind the scenes. The key difference is that you don't need to sacrifice a named register to hold the loaded data. Also, the processor doesn't have to then get the data back out of the named register, a process which is more expensive on Intel than PowerPC, and which can actually cause processor stalls on Intel.

The good news is that these changes make life easy for you, the software developer. You may find that you don't need to unroll by hand at all. It is very easy for the compiler to unroll code serially, since it can do so without worrying about aliasing problems. Direct memory references reduce the work involved with making use of constants.

The latencies and throughputs for various instructions are listed in the Intel Pentium Processor Optimization Reference Manuals (Appendix C, see link at the top of this page). At the time of the announcement of MacOS X for Intel (June, 2005), a student of comparative architecture between PowerPC and x86 would observe that pipeline lengths are generally shorter on x86. Lower latencies make it possible to more easily fill pipelines with a modestly sized reorder window. In addition, then current architectures commonly have a vector throughput of one instruction per two cycles on vector execution units. This has the effect of halving the amount of instruction level parallelism required to saturate pipelines, at the cost of decreased throughput. All AltiVec instructions proceed with a throughput of one instruction per cycle.

# Instruction Overview

The instruction set architecture (ISA) for SSE is similar to other parts of the x86 ISA. No operations take more than two register operands. (Sometimes a third argument is present as an immediate operand set at compile/link time.) Typically, one of the register operands is used for both input and output data, which is to say that one of the two operands is destroyed and replaced with the instruction results. It is frequently necessary to copy data that is needed later to avoid having it destroyed. (If you are using a C compiler, the compiler will do this for you and provide the illusion of non-destructive operations.) The other argument frequently may be either a register or a direct memory reference, that takes its data straight from memory.

There are three major classes of data on the SSE vector unit: integer, single precision floating point and double precision floating point vectors, each of which may be serviced by separate parts of the processor, akin to the AltiVec VSIU, VCIU, VFPU, but for int, float and double. The three data types share the same XMM register file, so you can do one type operation directly on the result of an another type of operation (for example do a vector floating point add of the result of a vector integer computation). This is exactly like AltiVec. No conversions are done. The bits are just passed around unmodified. If you want to convert between types (e.g. convert an int to a float) with retention of value (e.g. 0x00000001 ? 1.0f), there are special instructions for that.

However, unlike AltiVec, passing data back and forth between the three parts of the vector unit in this manner is frowned upon. In many cases, you will discover up to three seemingly redundant instructions that all do the same thing, one each for integer, single precision floating-point and double precision floating-point. Typical examples are vector loads and stores, certain permutes, and Boolean operations. There may be performance penalties for inter-unit data passing. It is recommended that, where possible, you use the appropriate instruction for the appropriate data type.

The Intel SIMD vector architecture was deployed over time as a series of four vector extensions to the x86 ISA. The first was MMX, followed by SSE, SSE2, and SSE3. SSE3 is the most recent, and is an optional feature of machines supported by MacOS X for Intel. The other three are guaranteed to be there, so you need only worry about SSE3. Details on each follow.

*MMX*

MMX, the first of the vector extensions provides a series of packed integer operators that utilize eight 64-bit registers described above. We do not describe MMX at length here because the operations defined by MMX are, generally speaking, also available in a 128-bit format in SSE2. Their use on SSE2 does not collide with the x87 unit, making SSE2 the generally preferred way to do these sorts of operations. MMX remains useful in a limited number of cases, especially those involving small data sets (particularly those 64 bits in size) and for some difficult to parallelize operations such as large-precision integer addition, but these cases are rare. MMX is sometimes used as a source of additional register storage area. However, since the vector ALU is shared with SSE2, there is likely no throughput advantage to using the two in parallel. Likewise since the cost of moving data to and from the MMX register file from XMM is likely to be larger than a simple aligned 128-bit load or store, such uses should be justified by real performance improvements.

MMX is enabled using the GCC compiler flag `-mmmx`. MMX is enabled by default on gcc-4.0. If MMX is enabled, the C preprocessor symbol `__MMX__` is defined. MMX is disabled using the `-mno-mmx` flag on GCC.

*SSE*

SSE adds a series of packed and scalar single precision floating point operations, and some conversions between single precision and integer. SSE uses the XMM register file, which is distinct from the MMX register file and does not alias the x87 floating point stack.

All operations under SSE are done under the control of the MXCSR, a special purpose control register that contains IEEE-754 flags and mask bits. SSE is enabled using the GCC compiler flag `-msse`. SSE is enabled by default on gcc-4.0. If SSE is enabled, the C preprocessor symbol __`SSE`__ is defined.

*SSE2*

SSE2 adds a series of packed and scalar double precision floating point operations. Like SSE, SSE2 uses the XMM register file. All floating point operations under SSE2 are also done under the control of the MXCSR to set rounding modes, flags and exception masks. In addition, SSE2 replicates most of the integer operations in MMX, except modified appropriately to fit the 128-bit XMM register size. In addition, SSE2 adds a large number of data type conversion instructions.

SSE2 is enabled using the GCC compiler flag -msse2. SSE2 is enabled by default on gcc-4.0. If SSE2 is enabled, the C preprocessor symbol __SSE2__ is defined.

*SSE3*

SSE3 adds a small series of instructions mostly geared to making complex floating point arithmetic work better in some data layouts. However, since it is possible to get the same or better performance by repacking data as uniform vectors rather than non-uniform vectors ahead of time, it is not expected that most developers will need to rely on this feature. Finally, it adds a small set of additional permutes and some horizontal floating point adds and subtracts that may be of use to some developers. Further details on SSE3 can be found in the Intel's documentation.

SSE3 is enabled using the GCC compiler flag -msse3. SSE3 is an optional hardware feature on MacOS X for Intel and is not enabled by default on gcc-4.0. If SSE3 is turned on, the C preprocessor symbol __`SSE3`__ is defined.

# Programming SSE in C

This chapter describes the C data types and intrinsics for use in programming SSE. It also shows how to detect the availability of SSE3 at run time.

## Data Types and Intrinsics

Like AltiVec, there is a C Programming Interface for SSE. The two follow the same general design:

- The SIMD vector register is described in C as a special 128 bit data type.
- A series of function-like intrinsics are used to do SIMD style operations on those variables.

A notable difference is that many more intrinsics in the Intel C programming extensions do not correspond 1:1 with instructions in the ISA. Some developers may choose to limit their use of intrinsics to those that map 1:1 with ISA, so as not to introduce hidden expensive calculations.

*Data Types*

Intel defines three basic data types for SSE programming in C:

**Table 2-1** Basic SSE Data Types

| Any Packed Integer | float[4] | double[2] |
|---|---|---|
| __m128i | __m128 | __m128d |

These types are portable across the Gnu C Compiler, the Intel C Compiler and various x86 C compilers targeted towards the Windows™ operating system.

One shortcoming of this set of data types is that the __m128i type does not adequately describe the type and number of integer elements in the __m128i vector. Both Intel and Microsoft defined extensions to this subset to build in this information, and Apple is no exception. The Accelerate.framework defines a series of vector types that may be used for both AltiVec and SSE programming. It is recommended that you use these, since the extra information will make it easier to read your own code and make it possible for gdb and xcode to properly format vector data. In addition, it will allow you to share data types with AltiVec, which may simplify some programming tasks. To use the types described below, use the following `#include` line:

```
#include <Accelerate/Accelerate.h>
```

**Table 2-2** Vector Data Types for Both AltiVec and SSE

| | 8-bit | 16-bit | 32-bit | 64-bit |
|---|---|---|---|---|
| signed | vSInt8 | vSInt16 | vSInt32 | vSInt64 |

| | 8-bit | 16-bit | 32-bit | 64-bit |
|---|---|---|---|---|
| unsigned | vUInt8 | vUInt16 | vUInt32 | vUInt64 |
| floating point | - | - | vFloat | vDouble |

Please note that while the 64-bit types are indeed defined for AltiVec by Accelerate.framework (and do work in the sense that you can load and store vectors full of 64-bit data types in and out of AltiVec register), there are no intrinsics (or instructions) defined by AltiVec itself to do SIMD style operations on elements of this size. The Accelerate.framework vBasicOps.h header declares some functions to allow you to do packed 64-bit integer operations. (These function using AltiVec intrinsics for smaller element sizes to build up larger operations — see available source code for vBasicOpsavailable source code for vBasicOps.) Certain C language operators (e.g. +, -, *, /) may function with the vDouble type on GCC-4.0 and later on PowerPC. However these simply map the vector type to the scalar FPU and do standard arithmetic on the data using scalar code.

*Intrinsics*

Intel also defines a set of function-like intrinsics for programming SSE in C. These are similar to those provided by AltiVec, with some small differences. The Intel intrinsics use `_mm_-` instead of `vec_-` as the operator prefix. In addition, where AltiVec relies on C++ style function overloading to decide based on argument type which particular flavor of add to use among many, Intel has encoded this information as a suffix on the intrinsic:

**Table 2-3** Suffixes of SSE Intrinsics

| AltiVec | SSE |
|---|---|
| vec_add( vSInt8, vSInt8 ); | _mm_add_epi8( vSInt8, vSInt8 ); |
| vec_add( vSInt16, vSInt16 ); | _mm_add_epi16( vSInt16, vSInt16 ); |
| vec_add( vSInt32, vSInt32 ); | _mm_add_epi32( vSInt32, vSInt32 ); |
| vec_add( vFloat, vFloat ); | _mm_add_ps( vFloat, vFloat ); |
| - | _mm_add_epi64( vSInt64, vSInt64 ); |
| - | _mm_add_pd( vDouble, vDouble ); |
| - | _mm_add_ss( vFloat, vFloat ); |
| - | _mm_add_sd( vDouble, vDouble ); |

The suffixes are defined as follows:

**Table 2-4** SSE Intrinsics Suffix Definitions

| suffix | description |
|---|---|
| -pi# | MMX (64-bit) vector containing packed #-bit integers |
| -pu# | MMX (64-bit) vector containing packed #-bit unsigned integers |
| -epi# | XMM (128-bit) vector containing packed #-bit integers |

| suffix | description |
|---|---|
| -epu# | XMM (128-bit) vector containing packed #-bit unsigned integers |
| -ps | XMM (128-bit) vector containing packed single precision floating point values |
| -ss | XMM (128-bit) vector containing one single precision floating point value |
| -pd | XMM (128-bit) vector containing packed double precision floating point values |
| -sd | XMM (128-bit) vector containing one double precision floating point value |
| -si64 | MMX (64-bit) vector containing a single 64-bit int |
| -si128 | XMM (128-bit) vector |

The various intrinsics are available in one of four headers, one each for MMX, SSE, SSE2, and SSE3, when the corresponding ISA appeared:

**Table 2-5** Headers for SSE Intrinsics

| | |
|---|---|
| MMX | mmintrin.h |
| SSE | xmmintrin.h |
| SSE2 | emmintrin.h |
| SSE3 | pmmintrin.h |

The complete set of operations available for the Intel architecture is detailed in the Intel Architecture Software Developer's Manual (Volume 2, see link in the Introduction at top of page). There is a partial AltiVec to SSE translation table in the Universal Binary Programming Guide, Appendix B. More thorough conversion tables appear in various segments entitled Algorithms/Conversions in the part of this document to follow.

In addition, GCC has a set of GCC native non-portable intrinsics, described here. Please note that these are subject to change. GCC can and does regularly remove `__builtins` from the programming environment.

*Sample function*

Here is a function that calculates the distances from the origin {0,0} of a set of 4 {x,y} pairs in AltiVec:

```
#include <Accelerate/Accelerate.h> //contains data types used
vFloat Distance( vFloat x, vFloat y )
{
    vFloat x2 = vec_madd( x, x, (vFloat) (-0.0f) ); //x * x
    vFloat distance2 = vec_madd( y, y, x2 ); // x*x + y*y
    return vsqrtf( distance2 ); //from Accelerate.framework
}
```

and here is the same thing in SSE:

```
#include <Accelerate/Accelerate.h> //contains data types used
#include <xmmintrin.h> //declares _mm_* intrinsics
vFloat Distance( vFloat x, vFloat y )
{
    vFloat x2 = _mm_mul_ps( x, x); //x * x
    vFloat distance2 = _mm_add_ps(_mm_mul_ps( y, y), x2); // x*x + y*y
    return vsqrtf( distance2 ); //from Accelerate.framework
}
```

If you wish to tie yourself to GCC specific features, you may investigate GCC's unified vector programming interfaces. That would allow you to write the following and compile for both platforms:

```
#include <Accelerate/Accelerate.h>
//Not portable to other compilers!
vFloat Distance( vFloat x, vFloat y )
{
    return vsqrtf( x*x + y*y ); //from Accelerate.framework
}
```

Since this is a new feature, it is suggested that you inspect generated code thoroughly. In addition, there are clearly other ways to do the same thing, using some inline functions or macros using more traditional interfaces, that may preserve your compiler independence.

# Detecting SSE3

SSE3 is an optional hardware feature on MacOS X for Intel. If you wish to use SSE3 features, you must detect them first, similar to how you are required to check for AltiVec. The same interfaces are used, just a different `sysctlbyname()` selector:

```
#include <sys/sysctl.h>
int IsSSE3Present( void )
{
    int hasSSE3 = 0;
    size_t length = sizeof( hasSSE3 );
    int error = sysctlbyname("hw.optional.sse3", &hasSSE3, &length, NULL, 0);
    if( 0 != error ) return 0;
    return hasSSE3;
}
```

Similar selectors exist for MMX, SSE and SSE2, but since those are required features for MacOS X for Intel, it is not required that you test them before using those vector extensions, in software intended solely for MacOS X for Intel. (SSE is not available in any format for MacOS X for PowerPC and AltiVec is not available for MacOS X for Intel. When writing code for Universal Binaries to run on MacOS X, you should conditionalize your code using appropriate symbols like __VEC__ and __SSE2__ to prevent the compiler from seeing vector code for unsupported architectures for each fork of the universal binary.)

# Translating Altivec to SSE

Translating AltiVec to SSE is not especially difficult to do. There is "pretty good" instruction parity between the two. AltiVec has more operations, but generally speaking, the operations that SSE provides mostly match up 1:1 with AltiVec equivalents. So, for example, where AltiVec has a `vadduwm` (vector add, unsigned word modulo — 32-bit int modulo add), SSE2 has a `PADDD` (Packed ADD Doubleword). Similar parity exists over the 60 or 70% of the AltiVec ISA that is the most commonly used part of AltiVec. In many cases, where the SSE ISA comes up short, there is a 2-3 instruction work around to deliver the same results. However, in some especially difficult cases, a new algorithm may be required.

Because both architectures share the same fundamental design (128-bit SIMD that prefers 16 byte aligned data), the work required beyond simple coding of intrinsics to make use of the two vector architectures is quite similar. Principally, these are development of parallel algorithms, changing data layouts, and dealing with misalignment. In our experience translating AltiVec to SSE for Accelerate.framework, this was by far the most time consuming part of writing the AltiVec segment. All of this work is directly reusable without further effort for the SSE version. As a result, translating AltiVec to SSE has taken perhaps 10-20% of the time that it took to vectorize for AltiVec in the first place for Accelerate.framework. This allows us to support both architectures in Accelerate.framework with a minimum of extra effort. Hopefully your experience will be similar.

## Translating Floating Point Operations

Both AltiVec and SSE do single precision floating point arithmetic. SSE2 also does double precision floating point arithmetic. Finally, for each packed vector floating point operation on SSE or SSE2, there is also a scalar version that can be done by the vector unit that operates on only one element in the vector:

**Table 3-1** Floating Point Capabilities of AltiVec and SSE

| | packed vector | scalar on vector |
|---|---|---|
| float | AltiVec + SSE | SSE |
| double | SSE2 | SSE2 |

The scalar-on-vector feature is used by MacOS X on Intel to do most scalar floating point arithmetic. So, if you write a normal floating point expression, such as `float a = 2.0f;` that will be done on XMM. (For compiler illuminati, the GCC compiler flag, `-mfpmath=sse`, is on by default.) Single and double precision scalar floating point arithmetic is done on the SSE unit both for speed and also so as to deliver computational results much more like those obtained from PowerPC. The legacy x87 scalar floating point unit is still used for long double, because of its enhanced precision.

Please note that the results of floating point calculations will likely not be exactly the same between PowerPC and Intel, because the PowerPC scalar and vector FPU cores are designed around a fused multiply add operation. The Intel chips have separate multiplier and adder, meaning that those operations must be done separately. This means that for some steps in a calculation, the Intel CPU may incur an extra rounding step, which may introduce 1/2 ulp errors at the multiplication stage in the calculation. Please note that in cases involving catastrophic cancellation, this may give results that are vastly different after the addition or subtraction has completed.

*SSE Floating Point Environment*

The floating point environments on SSE and AltiVec are very similar. Both vector floating point units are heavily influenced by IEEE-754. Both units store their data in IEEE-754 floating point format, though, in memory, the Intel architecture stores the bytes in little endian order. Both deliver nearly the same feature set of correctly rounded basic operations, such as addition, subtraction and multiplication. (Intel is slightly richer.) For more complicated functions such as the vector versions of the standard libm transcendental operations (`sin()`, `cos()`, `pow()`, etc.), look to Accelerate.framework. (`#include <Accelerate/Accelerate.h>`). Accelerate.framework actually provides this class of operations in two different flavors:

- For simple long array computation, look to vForce.h, new for MacOS X.4.
- For vector transcendentals involving 128-bit SIMD vectors, look to vfp.h (available on all MacOS X versions).

When it comes to other aspects of IEEE-754 compliance, SSE is a bit of a step up. While AltiVec delivers the Java subset of IEEE-754, the Intel vector unit is a fully IEEE-754 compliant machine, delivering full rounding modes, exceptions and flags.

A feature comparison chart follows:

**Table 3-2** Features: AltiVec vs. SSE

| | **AltiVec** | **SSE** |
|---|---|---|
| rounding modes | round to nearest only | all four (nearest, zero, Inf, -Inf) |
| exceptions | denormals | all IEEE-754 + denormals |
| flags | none | all IEEE-754 + denormals |
| square root | software | hardware |
| divide | software | hardware |
| compare bounds | yes | no |
| log2(x) and 2^x estimates | yes | no |
| reciprocal estimate | yes | yes |
| reciprocal sqrt estimate | yes | yes |
| scalar on vector | no | yes |

All hardware supported operations (that aren't estimates) are correctly rounded to 24 bits (float) or 53 bits (double) of precision. (The other 9/12 bits are used for exponent and sign information, exactly like PowerPC.) The accuracy of the estimates is very close to that of AltiVec, approximately 12 bits for reciprocal estimate and reciprocal square root estimate.

*Denormal Handling*

Neither AltiVec or SSE currently provide fast hardware support for denormals. In each case, a vector status and control register is available, with some bits that can be changed to turn on and off denormal handling. If denormal handling is turned on on a PowerPC machine and a denormal is encountered, the calculation is handled in a fast kernel trap. On a SSE enabled Intel chip the denormal is handled in hardware. As shown below, the denormal is expensive to handle in both cases:

**Table 3-3** Costs of Denormal Handling

| | G5 | Pentium 4 (P4 660) |
|---|---|---|
| By default, denormals are: | OFF | ON |
| Cost for handling a denormal: | 1100 cycles | 1550 cycles |

In the table above, OFF means denormals are not handled; they are flushed to zero. ON means denormals are handled, with a 1100-1550 cycle penalty.

If one turns off denormal handling, then the two machines flush the denormals to zero and proceed as if the data they are operating on is zero. This path operates at the same cost as arithmetic on normalized numbers, at the expense of incorrect results for denormalized inputs or outputs (which are flushed to zero).

Historical use of denormals has been varied. Under MacOS 9, operating system handling of denormals was on by default, meaning that ever time you hit a denormal on MacOS 9, the machine would take a large stall while the correct result was calculated for the vector unit by the operating system. MacOS X for PowerPC ships with denormals off for AltiVec by default. (We did explore turning them on briefly in the safety and comfort of our own test labs in order to deliver more correct results for MacOS X.3, but found we were breaking some 3rd party audio code with real time data delivery needs.) However, denormal handling is on by default for PowerPC scalar floating point, where denormals are handled in hardware at no additional cost.

Under MacOS X for Intel, denormal handling is back ON by default. This is required for standards compliant operation of normal scalar floating point code, which, if you will recall, is being done by the vector unit. Since the SSE vector status and control register (MXCSR) does not differentiate between scalar and vector operations done on the vector engine, this means that the denormal handling is on by default for packed vector arithmetic too.

If you are writing code with real time delivery needs, especially audio code, you may consider turning denormals off. Please be aware that if you do so, your code, both scalar and vector (except long double) will flush denormals to zero, meaning that strictly speaking, the results will be incorrect for the set of denormalized numbers. For certain classes of computations, particularly audio, this is generally not a problem — listeners are likely unable to hear the difference between the range of denormalized numbers (0 < x < 2^126) and zero. For others, it is a problem. *Proceed wisely*. We recommend leaving denormal handling enabled unless you actually have a problem. Generally speaking, denormals do not happen often enough to cause trouble. However, certain classes of algorithms (most notably IIR filters) may produce nothing but denormals in certain situations (input gain goes to zero). If that occurs in a real time thread, system responsiveness may be adversely affected. Results may not be delivered on time.

To turn denormals off on AltiVec, set the Non-Java bit in the AltiVec VSCR. To turn denormals off on SSE, turn on the Denormals Are Zero and Flush to Zero (DAZ and FZ) bits in the MXCSR:

```
#include <xmmintrin.h>
int oldMXCSR = _mm_getcsr(); //read the old MXCSR setting
int newMXCSR = oldMXCSR | 0x8040; // set DAZ and FZ bits
_mm_setcsr( newMXCSR ); //write the new MXCSR setting to the MXCSR
... // do your work with denormals off here
```

```
//restore old MXCSR settings to turn denormals back on if they were on
_mm_setcsr( oldMXCSR );
```

You may also use the C99 Standard fenv.h, with the Mac OS X for Intel specific default denormals-off floating point environment.

```
#include <fenv.h>
#pragma STDC FENV_ACCESS ON
fenv_t oldEnv;
//Read the old environment and set the new environment using default flags and
 denormals off
fegetenv( &oldEnv );
fesetenv( FE_DFL_DISABLE_SSE_DENORMS_ENV );
... //do work here
//Restore old floating point environment
fesetenv( &oldEnv );
```

**Note:** Both of the above code examples lose track of floating point status flag changes that occur while denormals are turned off. Rather than simply swapping floating point environments, it is possible to preserve floating point state across this series of operations by doing various bitwise boolean operations to copy information between states. This may be required if you are relying on floating point state flags for diagnostic information or are using SIGFPE.

Setting and checking other bits in the MXCSR will allow you to take an exception if you hit a denormal (DM) and check to see whether you have previously hit a denormal (DE) in the vector unit, in addition to the typical IEEE-754 exceptions and flags.

Denormals also cause large stalls on the x87 scalar floating point unit. Simply loading and storing denormals in and out of the x87 unit may cause a stall. The processor has to convert them to 80-bit extended format and back during these operations. There is no way to disable denormal handling on x87.

*Algorithms and Conversions*

Here is a table of standard conversions for floating point operations in AltiVec and SSE:

**Table 3-4** Converting Floating Point Operations from AltiVec to SSE

| AltiVec | SSE |
|---|---|
| vec_add(a, b) | _mm_add_ps(a, b) |
| vec_and(a, b) | _mm_and_ps(a, b) |
| vec_andc(a, b) | _mm_andnot_ps(b, a) |
| vec_ceil(a) | see below |
| vec_expte(a) | none (use vexpf in Accelerate.framework, vfp.h) |
| vec_floor(a) | see below |
| vec_loge(a) | none (use vlogf in Accelerate.framework, vfp.h) |
| vec_madd(a, b, c) | _mm_add_ps( _mm_mul_ps(a,b), c) |

| AltiVec | SSE |
|---|---|
| vec_max(a, b) | _mm_max_ps(a, b) |
| vec_min(a, b) | _mm_min_ps(a, b) |
| vec_nmsub(a, b, c) | _mm_sub_ps(c, _mm_mul_ps(a, b)) |
| vec_nor(a, b) | _mm_xor_ps( _mm_or_ps(a, b), 0xFFFFFFFF ) |
| vec_or(a,b) | _mm_or_ps(a, b) |
| vec_re(a) | _mm_rcp_ps(a) |
| vec_round(a) | see below |
| vec_rsqrte(a) | _mm_rsqrt_ps(a) |
| vec_sel(a,b,c) | _mm_or_ps(_mm_andnot_ps(c,a), _mm_and_ps(c,b)) |
| vec_sub(a,b) | _mm_sub_ps(a,b) |
| vec_trunc(a) | see below |
| vec_xor(a,b) | _mm_xor_ps(a,b) |

Don't forget to convert all of those vec_madd( a, b, -0.0f) calls to _mm_mul_ps(a, b). It will save an instruction.

The most notable missing conversion in the table above is explicit floating point rounding to integer. In many cases, this can be solved by setting the appropriate rounding mode in the MXCSR and converting the vFloat to a vSInt32. This process is covered more in depth in the conversions section below. However, that only works if the floating point value is representable as a 32-bit integer. Since many do not fit into the 32-bit signed integer range, it may be necessary to use a full precision floor function. The basic operation involves adding a large magic number to the vFloat, then subtracting it away again. The number is chosen such that the unit in the last place of the magic number corresponds to the 1's binary digit. This value is 2^23, 0x1.0p23f. This causes rounding at that position according to the processor's rounding mode after the addition. When 2^23 is subtracted away again, the value will be restored, but correctly rounded to integer value.

There are some tricks to this process. Negative numbers may require that you reverse the order of the add and subtract, or use 2^24. With some clever programming you may be able to avoid toggling the MXCSR to set rounding modes and come up with an algorithm that works for all four rounding modes. Depending on your specific application, you may be able to avoid some or all of these steps. In the simplest case, it is just an add and a subtract. Here is some sample code for `floor` and `trunc`.

```
static inline vFloat _mm_floor_ps( vFloat v ) __attribute__ ((always_inline));
static inline vFloat _mm_floor_ps( vFloat v )
{
    static const vFloat twoTo23 = (vFloat){ 0x1.0p23f, 0x1.0p23f, 0x1.0p23f,
0x1.0p23f };
    vFloat b = (vFloat) _mm_srli_epi32( _mm_slli_epi32( (vUInt32) v, 1 ), 1 );
 //fabs(v)
    vFloat d = _mm_sub_ps( _mm_add_ps( _mm_add_ps( _mm_sub_ps( v, twoTo23 ),
twoTo23 ), twoTo23 ), twoTo23 ); //the meat of floor
    vFloat largeMaskE = (vFloat) _mm_cmpgt_ps( b, twoTo23 ); //-1 if v >= 2**23
    vFloat g = (vFloat) _mm_cmplt_ps( v, d ); //check for possible off by one 
error
    vFloat h = _mm_cvtepi32_ps( (vUInt32) g ); //convert positive check result
 to -1.0, negative to 0.0
    vFloat t = _mm_add_ps( d, h ); //add in the error if there is one
    //Select between output result and input value based on v >= 2**23
    v = _mm_and_ps( v, largeMaskE );
    t = _mm_andnot_ps( largeMaskE, t );
    return _mm_or_ps( t, v );
}
static inline vFloat _mm_trunc_ps( vFloat v ) __attribute__ ((always_inline));
static inline vFloat _mm_trunc_ps( vFloat v )
{
    static const vFloat twoTo23 = (vFloat){ 0x1.0p23f, 0x1.0p23f, 0x1.0p23f, 
0x1.0p23f };
    vFloat b = (vFloat) _mm_srli_epi32( _mm_slli_epi32( (vUInt32) v, 1 ), 1 );
 //fabs(v)
    vFloat d = _mm_sub_ps( _mm_add_ps( b, twoTo23 ), twoTo23 ); //the meat of 
floor
    vFloat largeMaskE = (vFloat) _mm_cmpgt_ps( b, twoTo23 ); //-1 if v >= 2**23
    vFloat g = (vFloat) _mm_cmplt_ps( b, d ); //check for possible off by one 
error
    vFloat h = _mm_cvtepi32_ps( (vUInt32) g ); //convert positive check result
 to -1.0, negative to 0.0
    vFloat t = _mm_add_ps( d, h ); //add in the error if there is one
    //put the sign bit back
    vFloat sign = (vFloat) _mm_slli_epi31( _mm_srli128( (vUInt32) v, 31), 31 );
    t = _mm_or_ps( t, sign );
    //Select between output result and input value based on fabs(v) >= 2**23
    v = _mm_and_ps( v, largeMaskE );
    t = _mm_andnot_ps( largeMaskE, t );
    return _mm_or_ps( t, v );
}
```

## Translating Integer Operations

Most integer operations on SSE are in the SSE2 segment of the vector extensions. Packed vector integer arithmetic first debuted on the Intel platform in MMX. The same operations were later redeployed on the XMM register file in SSE2. All vector integer instructions generally start with the letter P (for packed). Most integer instructions come in two flavors with the same name, one for MMX and one for XMM. For a complete list, please see the Intel Architecture Software Developer's Manual, Volumes 2. (Link available at the top of this page.) Because the two share the same name and use of MMX can damage x87 floating point state, it may be advisable in certain circumstances to employ GCC compiler flags such as `-mno-mmx`, to avoid inadvertently using MMX.

*Integer Add / Subtract / Min / Max*

You will find the full complement of modulo adds and subtracts on SSE2. In addition, SSE2 also does 64-bit modulo addition and subtraction. The AltiVec `vec_addc` and `vec_subc` for large-precision unsigned integer addition and subtraction do not have SSE counterparts, however. It is suggested that you use the 64-bit adder to handle your extended integer precision.

SSE2 supports saturated addition for 8- and 16-bit element sizes only. Min and Max functions are available for vUInt8 and vSInt16, only.

*Integer Multiplication*

One of the more difficult problems to solve when translating AltiVec to SSE is what to do about integer multiplication. There is almost no overlap between AltiVec and SSE for integer multiplication. The AltiVec `vec_mladd` operation is a little bit like `_mm_mullo_epi16`, and `vec_msum` is a little bit like `_mm_madd_epi16`, but they are by no means a close match. There are 5 integer multipliers on SSE2:

**Table 3-5** SSE Integer Multiplication Operations

| | | |
|---|---|---|
| _mm_mullo_epi16(a,b) | vSInt16 or vUInt16 | low 16 bits of the product of two 16-bit integers |
| _mm_mulhi_epi16(a,b) | vSInt16 | high 16 bits of the product of two 16-bit signed integers |
| _mm_mulhi_epu16(a,b) | vUInt16 | high 16 bits of the product of two 16-bit unsigned integers |
| _mm_mul_epu32(a,b) | vUInt32 | 64-bit product of two vUInt32's (odd elements) |
| _mm_madd_epi16(a,b) | vSInt16 | sum of adjacent signed 32-bit products of int16_t |

One shared calculation motif that works reasonably well between AltiVec and SSE is the concept of full precision multiplies, where two vectors with element of size N, multiply to create two product vectors with element size 2N. On AltiVec, this is done with `vec_mule` and `vec_mulo`, followed by `vec_merge` to interleave even and odd results. On SSE, you can use the low and high 16 bit multiplies, with a merge operation (see `_mm_unpacklo_epi16` and `_mm_unpackhi_epi16`).

The other shared calculation motif that works well is to exploit commonalities between `_mm_madd_epi16` and `vec_msum( vSInt16, vSInt16, vSInt32 )`. Finally, in a very small number of cases, you can use `_mm_mullo_epi16(a,b)` interchangeably with `vec_mladd(a,b,0)`.

*Integer Algorithms and Conversions*

Here is a table of simple AltiVec to SSE translations for integer arithmetic:

**Table 3-6** Converting Integer Arithmetic Operations from AltiVec to SSE

| AltiVec | Type | SSE2 |
|---|---|---|
| vec_abs(a) | vSInt8 | vSInt8 t = _mm_cmpgt_epi8(0,a); return _mm_sub_epi8(_mm_xor_si128( a, t), t); |
| vec_abs(a) | vSInt16 | _mm_max_epi16( a, _mm_sub_epi16( 0, a ) ) |
| vec_abs(a) | vSInt32 | vSInt8 t = _mm_srai_epi32(a,31);return _mm_sub_epi32(_mm_xor_si128( a, t), t); |
| vec_abss(a) | vSInt8 | vSInt8 t = _mm_cmpgt_epi8(0,a);return _mm_subs_epi8(_mm_xor_si128( a, t), t); |
| vec_abss(a) | vSInt16 | _mm_max_epi16( a, _mm_subs_epi16( 0, a ) ) |
| vec_abss(a) | vSInt32 | none |
| vec_add(a,b) | vSInt8 | _mm_add_epi8(a,b) |
| vec_add(a,b) | vUInt8 | _mm_add_epi8(a,b) |

| AltiVec | Type | SSE2 |
|---|---|---|
| vec_add(a,b) | vSInt16 | _mm_add_epi16(a,b) |
| vec_add(a,b) | vUInt16 | _mm_add_epi16(a,b) |
| vec_add(a,b) | vSInt32 | _mm_add_epi32(a,b) |
| vec_add(a,b) | vUInt32 | _mm_add_epi32(a,b) |
| vec_adds(a,b) | vSInt8 | _mm_adds_epi8(a,b) |
| vec_adds(a,b) | vUInt8 | _mm_adds_epu8(a,b) |
| vec_adds(a,b) | vSInt16 | _mm_adds_epi16(a,b) |
| vec_adds(a,b) | vUInt16 | _mm_adds_epu16(a,b) |
| vec_adds(a,b) | vSInt32 | none |
| vec_adds(a,b) | vUInt32 | none |
| vec_and(a,b) | any int type | _mm_and_si128(a,b) |
| vec_andc(a,b) | any int type | _mm_andnot_si128(b,a) |
| vec_avg(a,b) | vSInt8 | none |
| vec_avg(a,b) | vUInt8 | _mm_avg_epu8(a,b) |
| vec_avg(a,b) | vSInt16 | none |
| vec_avg(a,b) | vUInt16 | _mm_avg_epu16(a,b) |
| vec_avg(a,b) | vSInt32 | none |
| vec_avg(a,b) | vUInt32 | none |
| vec_madds(a,b,c) | vSInt16 | none (see note 1 below) |
| vec_max(a,b) | vSInt8 | vSInt8 t = _mm_cmpgt_epi8(a,b);return _mm_or_si128( _mm_andnot_si128(t,b),_mm_and_si128(t,a)); |
| vec_max(a,b) | vUInt8 | _mm_max_epu8(a,b) |
| vec_max(a,b) | vSInt16 | _mm_max_epi16(a,b) |
| vec_max(a,b) | vUInt16 | none |
| vec_max(a,b) | vSInt32 | vSInt32 t = _mm_cmpgt_epi32(a,b);return _mm_or_si128( _mm_andnot_si128(t,b),_mm_and_si128(t,a)); |
| vec_max(a,b) | vUInt32 | none |

| AltiVec | Type | SSE2 |
|---|---|---|
| vec_min(a,b) | vSInt8 | vSInt8 t = _mm_cmpgt_epi8(a,b);return _mm_or_si128( _mm_and_si128(t,b),_mm_andnot_si128(t,a)); |
| vec_min(a,b) | vUInt8 | _mm_min_epu8(a,b) |
| vec_min(a,b) | vSInt16 | _mm_min_epi16(a,b) |
| vec_min(a,b) | vUInt16 | none |
| vec_min(a,b) | vSInt32 | vSInt32 t = _mm_cmpgt_epi32(a,b);return _mm_or_si128( _mm_and_si128(t,b),_mm_andnot_si128(t,a)); |
| vec_min(a,b) | vUInt32 | none |
| vec_mladd(a,b,c) | vSInt16 or vUInt16 | _mm_add_epi16( _mm_mullo_epi16(a,b),c) |
| vec_mradds(a,b,c) | vSInt16 | none (see note 1 below) |
| vec_msum(a,b,c) | vSInt16 or vUInt16 | none |
| vec_msum(a,b,c) | vSInt16 | _mm_add_epi32( _mm_madd_epi16(a,b), c ) |
| vec_msum(a,b,c) | vUInt16 | none |
| vec_msums(a,b,c) | vSInt16 or vUInt16 | none (saturated 32-bit add missing) |
| vec_mule(a,b) | any | none (see note 2 below) |
| vec_mulo(a,b) | any | none (see note 2 below) |
| vec_nor(a,b) | any | _mm_xor_si128( _mm_or_si128(a,b), -1) |
| vec_or(a,b) | any | _mm_or_si128(a,b) |
| vec_sel(a,b,c) | any | _mm_or_si128( _mm_and_si128(c,b), _mm_andnot_si128(c,a)) |
| vec_sub(a,b) | vSInt8 or vUInt8 | _mm_sub_epi8(a,b) |
| vec_sub(a,b) | vSInt16 or vUInt16 | _mm_sub_epi16(a,b) |
| vec_sub(a,b) | vSInt32 or vUInt32 | _mm_sub_epi32(a,b) |
| vec_subs(a,b) | vSInt8 | _mm_subs_epi8(a,b) |
| vec_subs(a,b) | vUInt8 | _mm_subs_epu8(a,b) |
| vec_subs(a,b) | vSInt16 | _mm_subs_epi16(a,b) |
| vec_subs(a,b) | vUInt16 | _mm_subs_epu16(a,b) |
| vec_subs(a,b) | vSInt32 | none |

| AltiVec | Type | SSE2 |
|---|---|---|
| vec_subs(a,b) | vUInt32 | none |
| vec_sum4s(a,b) | any | none |
| vec_sum2s(a,b) | vSInt32 | none |
| vec_sums(a,b) | vSInt32 | none |
| vec_xor(a,b) | any | _mm_xor_si128(a,b) |

Note 1: Something similar can be done with `_mm_mulhi_epi16` and `vec_add(s)_epi16`. However, `_mm_mulhi_epi16` shifts right by 16, and the AltiVec instruction shifts right by 15, so some change in fixed point format will be required.

```
//AltiVec: multiply a * b and return double wide result in high and low result
void vec_mul_full( vSInt32 *highResult, vSInt32 *lowResult, vSInt16 a, vSInt16
 b)
{
    vSInt32 even = vec_mule( a, b );
    vSInt32 odd = vec_mulo( a, b );
    *highResult = vec_mergeh( even, odd );
    *lowResult = vec_mergel( even, odd );
}
//SSE2: multiply a * b and return double wide result in high and low result
void _mm_mul_full( vSInt32 *highResult, vSInt32 *lowResult, vSInt16 a, vSInt16
 b)
{
    vSInt32 hi = _mm_mulhi_epi16( a, b );
    vSInt32 low = _mm_mullo_epi16( a, b );
    *highResult = _mm_unpacklo_epi16( hi, low );
    *lowResult = _mm_unpackhi_epi16( hi, low );
}
```

# Translating Compare Operations

*Testing Inequalities*

Vector compares are done on SSE in substantially the same way as for AltiVec. The same basic set of compare instructions (similar to `vec_cmp*`) are available. They return a vector containing like sized elements with -1 for a true result and 0 for a false result in the corresponding element. The floating point compares provide the full set that AltiVec provides (except `vec_cmpb`) and in addition provide ordered and unordered compares and the != test. In addition, all vector floating point compares come in both scalar and packed versions.

The integer compares test for equality and inequality. The inequality test are for signed integers only. There are no unsigned compare greater than instruction. There are no compare instructions for 64-bit types.

*Conditional Execution*

Branching based on the result of a compare is handled differently from AltiVec, however. The AltiVec compares set some bits in the condition register, upon which the processor can branch directly. SSE compares set no analogous bits. Instead, use MOVMSKPD, MOVMSKPS or PMOVMSKB instruction to copy the top bit out of each element, crunch them together into a 2- ,4- or 16-bit int for double, float and integer data respectively, and copy to an integer register. You may then test that bit field to decide whether or not to branch. This example implements the SSE version of AltiVec's vec_any_eq intrinsic for vFloat:

```
int _mm_any_eq( vFloat a, vFloat b )
{
    //test a==b for each float in a & b
    vFloat mask = _mm_cmpeq_ps( a, b );
    //copy top bit of each result to maskbits
    int maskBits = _mm_movemask_ps( mask );
    return maskBits != 0;
}
```

If you are branching based on the result of a compare of one element only, then you can do the whole thing in one instruction using either UCOMISD/UCOMISS or COMISD/COMISS.

*Select*

Branching is expensive on Intel, just as it is on PowerPC. Most of the time that a test is done, the developer on either platform will elect not to do conditional execution, but instead evaluate both sides of the branch and select the correct result based on the value of the test. In AltiVec, this would look like this:

```
// if (a > 0 ) a += a;
vUInt32 mask = vec_cmpgt( a, zero );
vFloat twoA = vec_add( a, a);
a = vec_sel( a, twoA, mask );
```

In SSE, the same algorithm is used. However, SSE has no select instruction. One must use AND, ANDNOT,, OR instead:

```
vFloat _mm_sel_ps( vFloat a, vFloat b, vFloat mask )
{
    b = _mm_and_ps( b, mask );
    a = _mm_andnot_ps( mask, a );
    return _mm_or_ps( a, b );
}
```

Then, the SSE version of the above AltiVec code may be written:

```
// if (a > 0 ) a += a
vFloat mask = _mm_cmpgt_ps( a, zero );
vFloat twoA = _mm_add_ps( a, a);
a = _mm_sel_ps( a, twoA, mask );
```

We have found that in practice, it is sometimes possible to cleverly replace select with simpler Boolean operators like a single AND, OR or XOR, especially in vector floating point code. While not a performance win for AltiVec (it's a wash), for SSE this replaces three instructions with one, and can be a large win for code that uses select frequently. Very infrequently, sleepy AltiVec programmers may momentarily forget about vec_min and vec_max, and use compare / select instead. Those are a nice win too, when you can find them.

*Algorithms and Conversions*

Here is a conversion table for AltiVec to SSE translation for vector compares and select:

**Table 3-7** Converting Vector Compare and Select Operations from AltiVec to SSE

| AltiVec | Type | SSE |
|---|---|---|
| vec_cmpeq(a,b)) | vSInt8 | _mm_cmpeq_epi8(a,b) |
| vec_cmpeq(a,b) | vUInt8 | _mm_cmpeq_epi8(a,b) |
| vec_cmpeq(a,b) | vSInt16 | _mm_cmpeq_epi16(a,b) |
| vec_cmpeq(a,b) | vUInt16 | _mm_cmpeq_epi16(a,b) |
| vec_cmpeq(a,b) | vSInt32 | _mm_cmpeq_epi32(a,b) |
| vec_cmpeq(a,b) | vUInt32 | _mm_cmpeq_epi32(a,b) |
| vec_cmpeq(a,b) | vFloat | _mm_cmpeq_ps(a,b) |
| vec_cmpge(a,b) | vFloat | _mm_cmpge_ps(a,b) |
| vec_cmpgt(a,b) | vSInt8 | _mm_cmpgt_epi8(a,b) |
| vec_cmpgt(a,b) | vUInt8 | _mm_max_epu8(a,b) != b |
| vec_cmpgt(a,b) | vSInt16 | _mm_cmpgt_epi16(a,b) |
| vec_cmpgt(a,b) | vUInt16 | _mm_cmpgt_epi16(a+0x8000, b+0x8000) |
| vec_cmpgt(a,b) | vSInt32 | _mm_cmpgt_epi32(a,b) |
| vec_cmpgt(a,b) | vUInt32 | _mm_cmpgt_epi32(a+0x80000000, b+0x80000000) |
| vec_cmpgt(a,b) | vFloat | _mm_cmpgt_ps(a,b) |
| vec_cmple(a,b) | vFloat | _mm_cmple_ps(a,b) |
| vec_cmplt(a,b) | vSInt8 | _mm_cmpgt_epi8(b,a) |
| vec_cmplt(a,b) | vUInt8 | _mm_min_epu8(a,b) != b |
| vec_cmplt(a,b) | vSInt16 | _mm_cmpgt_epi16(b,a) |
| vec_cmplt(a,b) | vUInt16 | _mm_cmpgt_epi16(b+0x8000, a+0x8000) |
| vec_cmplt(a,b) | vSInt32 | _mm_cmpgt_epi32(b,a) |
| vec_cmplt(a,b) | vUInt32 | _mm_cmpgt_epi32(b+0x80000000, a+0x80000000) |
| vec_cmplt(a,b) | vFloat | _mm_cmplt_ps(a,b) |

# Translating Conversion Operations

SSE has a wide variety of data type conversions. Like AltiVec, if you wish to simply use a vector of one type (e.g. vFloat) as a vector of another type (e.g. vSInt32) without changing the bits, you can do that with a simple typecast:

```
vFloat one = (vFloat) {1.0f, 1.0f, 1.0f, 1.0f };
vSInt32 oneBits = (vSInt32) one;
```

The variable oneBits will now contain {0x3f800000, 0x3f800000, 0x3f800000, 0x3f800000}, the bit pattern for a vector full of 1.0f. This is a free operation, requiring at most one instruction to copy the data between registers, but in the optimum case no work needs to be done. (Note: please see caution about moving data between vector int, float and double types, under "MMX" in the"Instruction Overview" (page 12) section.)

However, if you wish to convert one type of vector to another with retention of numerical value (instead of bit pattern) then you will wish to use the appropriate conversion instruction. Conversions among different types generally follow the same pathway as for AltiVec, except that 16 bit pixels are not really a native data type for SSE. There is no hardware conversion between 16-bit pixel and vUInt8. The rest of the conversions are described below:

*Float - Int Conversions*

Conversions between floating point and integer types are similar to AltiVec with a few differences:

- The vec_ctf, vec_ctu and vec_cts instructions take a second parameter, an immediate to be used to adjust the power of two of the result. The SSE conversion functions take no second parameter. To do this power of 2 scaling on SSE, multiply the floating point input or output by the appropriate power of 2.
- In the float-to-int direction, floating point input values larger than the largest representable int result in 0x80000000 (a very negative number) rather than the largest representable int on PowerPC.
- There are no unsigned conversions between int and float
- All four rounding modes are available directly through the MXCSR. You won't need vec_floor, vec_trunc, vec_ceil, vec_round to round before you do the conversion to int. There are two different flavors of float-to-int conversion: _mm_cvtps_epi32 and _mm_cvttps_epi32. The first rounds according to the MXCSR rounding bits. The second one always uses round towards zero.
- Conversions between vDouble and vSInt32 are also available.

Here is how to fix the overflow saturation difference for vFloat to vSInt32 conversions:

```
const vFloat two31 = (const vFloat) {0x1.0p31f,0x1.0p31f,0x1.0p31f,0x1.0p31f};
//Convert float to signed int, with AltiVec style overflow
//(i.e. large float -> 0x7fffffff instead of 0x80000000)
vSInt32 _mm_cts( vFloat v )
{
    vFloat overflow = _mm_cmpge_ps( v, two31);
    vSInt32 result = _mm_cvtps_epi32( v );
    return _mm_xor_ps( result, overflow );
}
```

Here is a function that does vFloat to vUInt32 conversion, that gives the correct results, with AltiVec saturation for out of range inputs. You can write faster functions if you are willing to sacrifice correctness or saturate differently:

```
static inline vUInt32 _mm_ctu_ps( vFloat f )
{
    vFloat two32 = _mm_add_ps( two31, two31);
    vFloat zero = _mm_xor_ps(f,f);
    //check for overflow before conversion to int
    vFloat overflow = _mm_cmpge_ps( f, two31 );
    vFloat overflow2 = _mm_cmpge_ps( f, two32 );
    vFloat subval = _mm_and_ps( overflow, two31 );
    vUInt32 addval = _mm_slli_epi32((vUInt32)overflow, 31);
    vUInt32 result;
    //bias the value to signed space if it is >= 2**31
    f = _mm_sub_ps( f, subval );
    //clip at zero
    f = _mm_max_ps( f, zero );
    //convert to int with saturation
    result = _mm_cvtps_epi32( f ); //rounding mode should be round to nearest
    //unbias
    result = _mm_add_epi32( result, addval );
    //patch up the overflow case
    result = _mm_or_si128( result, (vUInt32)overflow2 );
    return result;
}
```

Some special case short-cuts for `float`-to-`unsigned int` conversion:

- If you do not need the complete unsigned range, you may consider just using the float to signed conversion, with some possible preclipping using `_mm_min_ps` and `_mm_max_ps`.
- If you do not mind throwing away the least significant (up to) 8 bits of your result for values in the range -2^24 < f < 2^24, this can be done more quickly by subtracting 0x1.0p31f from your floating point input, doing the signed conversion, then subtracting 0x80000000 from the result.

Finally, the vUInt32 to vFloat conversion can be done using the signed conversion, 16-bits at a time:

```
const vFloat two16 = (const vFloat) {0x1.0p16f,0x1.0p16f,0x1.0p16f,0x1.0p16f};
//Convert vUInt32 to vFloat according to the current rounding mode
static inline vFloat _mm_ctf_epu32( vUInt32 v )
{
    // Avoid double rounding by doing two exact conversions
    //of high and low 16-bit segments
    vSInt32 hi = _mm_srli_epi32( (vSInt32) v, 16 );
    vSInt32 lo = _mm_srli_epi32( _mm_slli_epi32( (vSInt32) v, 16 ), 16 );
    vFloat fHi = _mm_mul_ps( _mm_cvtepi32_ps( hi ), two16);
    vFloat fLo = _mm_cvtepi32_ps( lo );
    // do single rounding according to current rounding mode
    // note that AltiVec always uses round to nearest. We use current
    // rounding mode here, which is round to nearest by default.
    return _mm_add_ps( fHi, fLo );
}
```

Once again, if you don't care about the last few bits of precision and correctly rounded results or the high half of the unsigned int range, then you can probably speed things up a bit.

### *Int - Int Conversions*

Int - Int conversions change the size of vector elements. This in turn changes the number of vectors required to hold the data to either twice as many or half as many, depending on whether the elements are getting larger or smaller. The basic method by which data conversions are done is the same between AltiVec and SSE. A few details differ.

### *Int - Int Conversions (Large to Small)*

Conversion of larger int types into smaller int types will mean converting two vectors to one. Formally, these come in saturating and non-saturating variants, to take care of the case where the value of the integer input exceeds the value representable in the smaller result integer. AltiVec provides both styes. SSE provides only the saturating variety. To do unsaturated pack on SSE, use a left and right shift to truncate the data into the appropriate range. (For signed data, use a right algebraic shift. For unsigned data, use a right logical shift.) This will prevent the saturated pack instructions from doing any saturation. Example, pack two vUInt16's down into a vUInt8 without saturation:

```
vUInt8 vec_pack_epu16( vUInt16 hi, vUInt16 lo );
```

We would like to use `_mm_packus_epi16` for this. Unfortunately, values outside the range [0,255] will pack with saturation yielding 0 or 255 as the result. What is more, since the instruction takes signed input, and we have unsigned inputs, values larger than 32768 will get truncated to 0 instead of 255. To fix that, we whack off the high bits. This can be done by `AND`-ing with (`vUInt16`)(0x00FF). :

```
vUInt8 vec_pack_epu16( vUInt16 hi, vUInt16 lo )
{
    const vUInt16 mask = (const vUInt16){0x00ff, 0x00ff, 0x00ff, 0x00ff, 0x00ff,
 0x00ff, 0x00ff, 0x00ff };
    // mask off high byte
    hi = _mm_and_si128( hi, mask );
    lo = _mm_and_si128( lo, mask );
    return _mm_packus_epi16( hi, lo );
}
```

If you need to return a signed unsaturated result, then use a right algebraic shift instead, and the appropriate signed saturated pack. In this case, we have to use the shift. The AND won't do the appropriate sign extension:

```
vSInt8 vec_pack_epu16( vUInt16 hi, vUInt16 lo )
{
    // shift hi and lo left by 8 to chop off high byte
    hi = _mm_slli_epi16( hi, 8 );
    lo = _mm_slli_epi16( lo, 8 );
    // shift hi and lo back right again (algebraic)
    hi = _mm_srai_epi16( hi, 8 );
    lo = _mm_srai_epi16( lo, 8 );
    return _mm_packs_epi16( hi, lo );
}
```

A number of saturated packing instructions are missing, such as `vSInt32` to `vUInt16`. In such cases, it may be required that you add / subtract small biases from the value so that the pack operation works correctly, then subtract / add them back out after the pack is complete. In some circumstances, this may be further complicated by the lack of a 32-bit saturated add.

### *Int - Int Conversions (Small to Large)*

Conversion of smaller int types to larger int types will mean converting one vector into two. SSE handles this in the same way AltiVec does, using high and low flavors of the conversion to handle the high and low halves of the vector. While AltiVec provides both signed and unsigned unpack primitives (the unsigned ones are `vec_merge(0,v)`), SSE provides only the unsigned variety.

To convert unsigned ints to larger unsigned ints, simply unpack with zero:

```
//SSE translation of vec_mergeh( 0, v )
vUInt32 vec_unpackhi_epu16( vUInt16 v )
{
    vUInt16 zero = _mm_xor_si128( v, v );
    return (vUInt32) _mm_unpackhi_epi16( v, zero );
}
```

Observe that the argument order for the unpack instruction is backwards from AltiVec. As discussed later, this may become further confused by byte swapping.

To convert signed ints into larger signed ints, simply merge with itself, then right algebraic shift to do the sign extension:

```
//SSE translation of vec_unpackh( 0, v )
vSInt32 vec_unpackhi_epi16( vSInt16 v )
{
    //depending on your view of the world, you may want
    //_mm_unpacklo_epi16 here instead
    vSInt32 t = (vSInt32) _mm_unpackhi_epi16( v,v );
    return _mm_srai_epi32( t, 16 );
}
```

*Algorithms and Conversions*

Here is a conversion table for AltiVec to SSE translation for data type conversions:

**Table 3-8** Converting Data Types

| AltiVec | Type | SSE |
|---|---|---|
| vec_ctf(a,0) | vSInt32 | _mm_cvtepi32_ps(a) |
| vec_ctf(a,0) | vUInt32 | see _mm_ctf (Note 1) |
| vec_cts(a,0) | vFloat | see _mm_cts (Note 1) |
| vec_ctu(a,0) | vFloat | see _mm_ctu (Note 1) |
| vec_mergeh(0,a) | vUInt8 | _mm_unpackhi_epi8( a, 0 ) |
| vec_mergeh(0,a) | vUInt16 | _mm_unpackhi_epi16( a, 0) |
| vec_mergel(0,a) | vUInt8 | _mm_unpacklo_epi8( a, 0 ) |
| vec_mergel(0,a) | vUInt16 | _mm_unpacklo_epi16( a, 0 ) |
| vec_pack(a,b) | vSInt16 | _mm_packs_epi16( _mm_srai_epi16( _mm_slli_epi16( a, 8), 8),_mm_srai_epi16( _mm_slli_epi16( b, 8), 8) ); |

| AltiVec | Type | SSE |
|---|---|---|
| vec_pack(a,b) | vUInt16 | _mm_packs_epi16( _mm_and_si128( a, 0x00FF),_mm_and_si128( b, 0x00FF) ); |
| vec_pack(a,b) | vSInt32 | _mm_packs_epi32( _mm_srai_epi32( _mm_slli_epi16( a, 16), 16),_mm_srai_epi32( _mm_slli_epi32( a, 16), 16) ); |
| vec_pack(a,b) | vUInt32 | _mm_packs_epi32( _mm_srai_epi32( _mm_slli_epi32( a, 16), 16),_mm_srai_epi32( _mm_slli_epi32( a, 16), 16) ); |
| vec_packs(a,b) | vSInt16 | _mm_packs_epi16(a,b) |
| vec_packs(a,b) | vUInt16 | none |
| vec_packs(a,b) | vSInt32 | _mm_packs_epi32(a,b) |
| vec_packs(a,b) | vUInt32 | none |
| vec_packsu(a,b) | vSInt16 | _mm_packus_epi16(a,b) |
| vec_packsu(a,b) | vUInt16 | none |
| vec_packsu(a,b) | vSInt32 | none |
| vec_packsu(a,b) | vUInt32 | none |
| vec_unpackh(a) | vSInt8 | _mm_srai_epi16( _mm_unpackhi_epi8(a,a), 8) |
| vec_unpackh(a) | vSInt16 | _mm_srai_epi32( _mm_unpackhi_epi16(a,a), 16) |
| vec_unpackl(a) | vSInt8 | _mm_srai_epi16( _mm_unpacklo_epi8(a,a), 8) |
| vec_unpackl(a) | vSInt16 | _mm_srai_epi32( _mm_unpacklo_epi16(a,a), 16) |

Note 1: Sample code appears under “Float - Int Conversions”

It is likely that a number of the `vec_packs` and `vec_packsu` translations above reported as "none" do exist. However, we haven't found any that simultaneously work for all possible inputs and which also perform satisfactorily. Where possible, the first choice is to find some other format to pack the data into that is supported well by the Intel vector ISA. In other cases, you may be aware that certain classes of inputs do not happen in your particular function. This may reduce the problem space a bit and allow for a much more efficient solution.

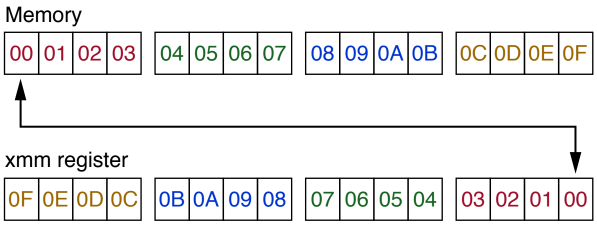
## Translating Permute Operations

As we shall describe in the `vec_perm` and `shuffle` section to follow, the Intel permute capability isn't as flexible as AltiVec. Generally speaking, it is not possible to permute data in a data dependent way — that is, except for self-modifying code, the order of the reshuffling must be known at compile time. This means that the Intel permute unit (as defined by the series of instructions in MMX, SSE, SSE2, and SSE3) cannot be used for lookup tables, to select pivot elements in register, to do misalignment handling, etc., unless the exact nature of the permute is known at compile time.

Things are not quite so bleak as they may appear at first. It is frequently true that there is a workaround for this sort of untranslatable functionality. Left or right 128-bit octet shifts which used to be handled by `lvsl` and `vperm` might instead be handled with some clever misaligned loads. MMX has arbitrary left and right shifts on its 64-bit registers. Lookup tables can still be done the old fashioned way, with separate loads for each element. (This is a bit easier under Intel, because scalar loads go to a defined place in the vector. Loading and splatting a scalar on AltiVec is perhaps unnecessarily unwieldy.) Finally, certain transformations (e.g. byte swapping) can be accomplished in a few vector instructions, in place of one permute.

Caution: Would be users of the Intel permute unit should be aware that the x86 memory architecture is little endian. Data is byte-swapped on load and store in and out of the vector unit. The swap occurs over the entire 16-byte vector, like this:

**Figure 3-1** Vector elements in memory order compared to register order



As described more fully in the loads and stores segment below, this means that both the ordering of bytes within each elements and the order of elements within the vector are reversed. This can make permutes confusing. If your left shifts go right, and your right shifts go left, and all your attempts at permute do the wrong thing, you may have forgotten that you are working on a little endian machine.

*Merge*

AltiVec's `vec_mergeh` and `vec_mergel` intrinsics translate directly to `_mm_unpackhi` and `_mm_unpacklo`. Vector unpacks are available for 8-, 16-, 32- and 64-bit data varieties.

Which flavor (high or low) to use and what order to place the arguments in is complicated by the little endian storage format. Under AltiVec, `vec_mergeh(even,odd)` could be used for a wide variety of purposes. On a big endian system, these are all degenerate. On a little endian system, they fall into a couple of classes for interleaving and unpacking data, which to further complicate things can be viewed based on the order of data as it appears in register, or following a store to memory:

- Interleaving data — let's say you start with left and right audio channels, each in its own vector, and you need to make an interleaved audio stream consisting of data in the order {left, right, left, right, ...}. On AltiVec, you'd just use `vec_merge(left, right)` and be done with it. On SSE you must first take into account the fact that this is a little endian system and memory order is the important one! This means that you actually want { ..., right2, left2, right1, left1, right0, left0} in register, so that you get {left0, right0, left1, right1, left2, right3, ...} when you store it out. That means you will be using `_mm_unpacklo` to replace `vec_mergeh`. In addition, the first argument of `_mm_unpacklo` is the one that goes in the odd position, whereas for `vec_mergeh`, the first argument would go into the even position. This means that `vec_mergeh( even, odd)` translates to `_mm_unpacklo(even,odd)` for data viewed in memory order, and `_mm_unpackhi( odd, even )` to replace `vec_mergeh` if the data is viewed in register order.
- Enlarging data — if you are using `vec_mergeh` to convert ints to larger ints (e.g. the SSE equivalent of converting a `vector unsigned short` to a pair of `vector unsigned ints`), then everything changes. In this case, one wants a different set of swaps to occur on storage to memory, so as to preserve the

high / low order of the two elements. To be brief, `vec_mergeh( high, low )` maps to `_mm_unpacklo( low, high )` for data viewed in memory order, and `_mm_unpackhi( low, high )` for data viewed in register order.

*Shifts and Rotations*

SSE provides a series of shift operations for most vector types, including 64-bit shifts and 128-bit octet shifts. The exception is 8-bit vector types, for which no shifts are available. For the rest, you may shift left or right. Right shifts come in the familiar logical (zero fill) and algebraic (sign extend) formats. Algebraic shifts are only available for 16- and 32-bit element sizes. A feature table follows:

**Table 3-9** Vector Shift Operations

| | 8-bit | 16-bit | 32-bit | 64-bit | 128-bit |
|---|---|---|---|---|---|
| left logical | none | yes | yes | yes | by octet |
| right logical | none | yes | yes | yes | by octet |
| right algebraic | none | yes | yes | none | none |

For all supported types except 128-bit shifts, you may shift either by an immediate value, known at compile time, or by a value present in a XMM vector. 128-bit shifts are by immediate only. The shift by value capability is different from AltiVec, however, in that while AltiVec allows you to shift each element in the vector a different amount from its fellows, on SSE all elements must be shifted by the same amount, the quantity held in the right-most element (register element order).

SSE has no rotate instructions. If you need to rotate a M-bit element by N-bits, you'll need to shift left by N bits, shift another copy right by M-N bits and OR the results together.

*vec_perm and shuffle*

In certain cases, it is possible to translate `vec_perm` to `SHUFPS`, `SHUFPD`, `PSHUFHW`, `PSHUFLW` or `PSHUFD`. The permute map must be known at compile time, and the data movement pattern must be supported by one of the above instructions. There are no shuffles capable of data organization at the byte level (apart from `_mm_unpack`). They all operate on 16-, 32- or 64-bit elements.

Many uses of `vec_perm` are not supported by SSE. It is frequently necessary to abandon permute based algorithms, when moving to SSE. In some cases, it may even be necessary to abandon SSE altogether and fall back on scalar code. However, in most cases this is not necessary. Most of the tough permute cases are linked to misalignment handling or scatter loading. Typically the best approach for these sorts of problems is to use the misaligned vector loads or scalar loads in SSE to do the work, rather than rely on the permute unit. Since scalar loads place data in a defined place in the register, it is typically easier on SSE to do scatter loading.

## Loads and Stores

A point should probably be made at the outset of this discussion, because it is one that is underemphasized in discussions about SIMD vector units in general. The Load / Store Units (LSUs) underlying most SIMD architectures (including both AltiVec and SSE) are not in themselves SIMD units. That is, you can't load or store to multiple addresses in parallel in a single instruction (unless they are contiguous, and therefore representable by a single address). Each LSU operates on only one address at a time. Your only opportunity to increase apparent parallelism is to make your single load or store do more work by loading or storing more bytes at a time. Apart from that, there is no SIMD-style parallelism in the LSU.

Why is this relevant? It is important to understand that while the vector unit is highly efficient for arithmetic, there should be no expectation of enhanced speed from the LSU portion of the AltiVec or SSE vector hardware compared to scalar code, except where loads or stores of large (up to 128-bit) chunks do the work of multiple smaller scalar loads or stores. Since every bit of data that you do arithmetic on must be first loaded into register, the LSU is potentially a bottleneck. If you want enhanced parallelism from the LSU, the only way to do that is to arrange your data in a contiguous format so that you can load in as much data as possible in as large a chunk as possible using a single address, do the calculation, then store the result out in a single big contiguous chunk. If your data is scattered throughout memory, this is not possible. Your vector code will spend a lot of time doing lots of little loads trying to coalesce scattered data into vectors and then even more time trying to scatter the results back out to memory using lots of little stores. If that isn't enough, there are also profound cache inefficiencies to accessing your data that way. Poor data layouts can nullify the vector advantage and even make vector code run slower than scalar code in some cases.

Vector data should be kept together, preferably in aligned, uniform arrays so that it can be accessed in as big a chunk as possible. This is doubly important on SSE, where misaligned loads and stores cannot reach the same peak theoretical throughput as aligned loads and stores, and where the permute unit is much less capable at reordering data. If you are vectorizing a body of code for the first time, you should give serious thought to how your data is organized into memory. If you already have AltiVec code, then translating to SSE should be a snap, because you probably already did that work when writing the AltiVec code.

*Misalignment*

SSE provides aligned and misaligned loads and stores in three different flavors: integer, single precision floating point and double precision floating point. It is suggested that you use the appropriate load and store for the data that you are working on. The aligned and misaligned loads and stores are simple and easy to use and shouldn't require too much explanation, except for the caution that the aligned variants will trigger an illegal instruction exception if they are passed a misaligned address.

**Table 3-10** Misaligned Load and Store Instructions

| AltiVec | Type | SSE |
|---|---|---|
| vec_ld(0,p) | vSInt8 | _mm_load_si128((__m128i*)p) or _mm_loadu_si128((__m128i*)p) |
| vec_ld(0,p) | vUInt8 | _mm_load_si128((__m128i*)p) or _mm_loadu_si128((__m128i*)p) |
| vec_ld(0,p) | vSInt16 | _mm_load_si128((__m128i*)p) or _mm_loadu_si128((__m128i*)p) |
| vec_ld(0,p) | vUInt16 | _mm_load_si128((__m128i*)p) or _mm_loadu_si128((__m128i*)p) |
| vec_ld(0,p) | vSInt32 | _mm_load_si128((__m128i*)p) or _mm_loadu_si128((__m128i*)p) |
| vec_ld(0,p) | vUInt32 | _mm_load_si128((__m128i*)p) or _mm_loadu_si128((__m128i*)p) |
| vec_ld(0,p) | vFloat | _mm_load_ps(p) or _mm_loadu_ps(p) |
| vec_st(v,0,p) | vSInt8 | _mm_store_si128((__m128i*)p, v) or _mm_storeu_si128((__m128i*)p, v) |
| vec_st(v,0,p) | vUInt8 | _mm_store_si128((__m128i*)p, v) or _mm_storeu_si128((__m128i*)p, v) |
| vec_st(v,0,p) | vSInt16 | _mm_store_si128((__m128i*)p, v) or _mm_storeu_si128((__m128i*)p, v) |

| AltiVec | Type | SSE |
|---|---|---|
| vec_st(v,0,p) | vUInt16 | _mm_store_si128((__m128i*)p, v) or _mm_storeu_si128((__m128i*)p, v) |
| vec_st(v,0,p) | vSInt32 | _mm_store_si128((__m128i*)p, v) or _mm_storeu_si128((__m128i*)p, v) |
| vec_st(v,0,p) | vUInt32 | _mm_store_si128((__m128i*)p, v) or _mm_storeu_si128((__m128i*)p, v) |
| vec_st(v,0,p) | vFloat | _mm_store_ps(p, v) or _mm_storeu_ps(p, v) |

There are no Least Recently Used variants on the loads and stores. (Note: the G5 ignores the LRU amendment and treats `lvxl` and `stvxl` as `lvx` and `stvx`.) There are however non-temporal stores. These cause the store to be written directly to memory. If the address maps to entries in the cache, the cache data is flushed out with the store. These can provide large performance increases, but should be used with caution. You only want to use them with data that you aren't going to need again for a while. Non-temporal stores require use of a SFENCE synchronization primitive before the data may be loaded back in again to ensure a coherent memory state.

The method of handling misaligned 128-bit vector loads and stores is nearly orthogonal between AltiVec and SSE. While you can do aligned loads on SSE like AltiVec, SSE lacks the concatenate-and-shift-by-variable capability that AltiVec has (done with `lvsl`, `vperm`). Though SSE has 128-bit shifts (by octet), they take immediates which must be known at compile time, preventing their use for misalignment. In most cases, it is required to use the misaligned load and store instructions when one needs to access data of unknown alignment. This means that your AltiVec misalignment handling code is probably not directly translatable to SSE. You'll likely need to rewrite that segment of the function with entirely new code to handle misalignment the SSE way.

Misaligned stores are much slower than misaligned loads. They should be avoided whenever possible. Typically the right thing to do with misaligned arrays is have a small scalar loop that iterates until it reaches a store address that is aligned, then skip to vector code to do as many aligned stores as possible, then do a bit more scalar calculation at the end. With some clever code design, it is also possible to use misaligned stores at either end of the array and aligned stores for the middle. This can be a little complicated if the function operates in place.

Keep in mind that the different misalignment handling strategies carry along with them different rules about when it is safe to do what. For example, it is always safe to load an aligned vector as long as at least one byte in the vector is valid data. However, it is only safe to load a misaligned vector if all the bytes in the misaligned vector are valid data. Thus, while you may have frequently read a few bytes past the end of misaligned arrays with AltiVec (which only supports aligned vector loads — misalignment is handled in software), you may not do that safely using SSE, where misaligned loads are directly supported in hardware and using aligned loads to access misaligned data is generally not done because the requisite shift instruction is missing.

The Intel hardware prefetchers are generally much more agile and able than similar hardware on PowerPC. If you do need to prefetch, you may use the GCC extension `__builtin_prefetch( ptr )`. This works on PowerPC, as well. It fetches a cacheline containing the address pointed to by *`ptr`*.

*Scalar Loads and Stores*

SSE provides a rich set of scalar loads and stores. `MOVQ` and `MOVD` can be used to move 8- and 4-byte integers to and from the low element on the XMM vector. These instructions can also be used to move the same amount of data to and from the MMX and r32 registers, which is a feature unknown to PowerPC. So, while there are no 16-bit and 8-bit element loads and stores, one can do a byte or 16-bit word load or store using the scalar integer registers, and use `MOVD` to move data between the integer registers and the vector unit.

Similarly, there are scalar floating point move instructions for single and double precision floating point, MOVSS and MOVSD. These likewise place or use the data in the low element of the XMM register. They can be used to move data between XMM registers as well.

Be aware that when the destination operand is an XMM register, the move element instructions will zero the rest of the destination register. Element loads and stores do not have alignment restrictions. Because alignment is handled so differently between AltiVec vector element loads and stores and SSE vector element loads and stores, segments of code that rely on these operations will in many cases need to be rewritten.

SSE element loads and stores are very important to the SSE architecture. More so than for AltiVec, element loads and stores are frequently the solution to difficult permute problems.

*x86 is Little Endian!*

As mentioned before (see Figure 4-1 (page 36)), elements of a vector and the bytes within the elements are reversed when stored in register. If your data looks like this in code:

```
float f[4]= { 0.0f, 1.0f, 2.0f, 3.0f };
vFloat v = _mm_loadu_ps( f );
```

The data in v will look like this:

```
v = { 3.0f, 2.0f, 1.0f, 0.0f }
```

Don't worry! If you store the data back out, it will be swapped again and appear in the original order shown by f[4]. The order is only backwards in register. (The bytes inside the elements themselves are in big endian byte order in register. The swap on store, makes the bytes in the elements little endian and restores the order of the elements to the expected order.)

If your permutes all seem to be broken, and left shifts go right and right shifts go left, it is likely you've forgotten about this element ordering reversal.

## Performance Tips

- Shark it! Shark is still the best way to identify performance problems. This will help you determine what to vectorize. The system trace facility will show you what other problems need to fixed to make vectorization the win it should be. There is no cycle accurate simulator available for Intel at this time.
- Unroll Different. You may have unrolled N-way in parallel on PowerPC. The Intel architecture is much narrower, and tolerates serial data dependencies better. Indeed, serial data dependencies get better throughput for some stages of the pipeline (register allocation) than does simple access of named registers. Generally speaking, the compiler can handle this form of unrolling for you, saving you time. There are no aliasing problems to worry about.
- Register spillage is expensive. Don't believe everything you read about really fast loads and stores. They are fast, but they still take time. If you are spilling data out on the stack, those loads and stores usually are taking significant time that could have been used for something else.
- Reduce or eliminate your need for the permute unit. It is not as strong as on AltiVec. You could find yourself spending all your CPU time solving permute problems rather than doing actual work. (This was already a problem for AltiVec!) Reorganize your data in memory so that it doesn't need to be reorganized in the permute unit. Align data whenever possible. Many permute problems can be solved a different way by loading in data differently, taking advantage of Intel's many different MOV instructions.

- Don't bother synthesizing constants on the fly like you did for AltiVec. Most of the time, you won't have register space to keep those constants in register. You also don't have `vec_splat_*`, so synthesizing constants takes a lot longer.
- See if you can replace vec_sel with simpler Boolean operations like `AND`, `ANDNOT`, `XOR` or `OR`. You can save two instructions and maybe a register or two every time you manage to do that.
- It is still just as worthwhile to pay attention to cache usage. You can prefetch data using the GCC extension, __builtin_prefetch(). If you need to store out data and won't need to use it again for a while, the non-temporal stores might be a large win. These flush the cash, so be careful. They aren't always a win!
- If you store out multiple small pieces of data (e.g. four floats in float[4]) and then load in a large piece of data (e.g. a vFloat) that covers that same area, this causes a large stall, because the floating point stores need to flush out all the way to the caches before the data is available. Store forwarding doesn't work in that case. While unions are a handy way to do data transfer, they can get you into trouble too.
- Just like AltiVec, denormal stalls can be very expensive. Unlike AltiVec, you are much more likely to encounter them.
- While translating code from AltiVec to SSE, pay attention to the expense of each translation. Some AltiVec instructions translate directly to a single SSE equivalent, while another potentially very similar instruction may take a dozen SSE instructions to do. Sometimes, it is better to be flexible about which one you use, rather than translate verbatim — convert floats to signed ints, rather than unsigned, if you don't need the extra range. Other times, you might want to rewrite the core logic of an algorithm to emphasize the strengths of both vector architectures, not just one.
- AltiVec is a rich ISA. This gives you a lot of freedom. There are frequently three ways to do anything, one of which is highly unintuitive but delivers a miracle in two instructions. SSE is smaller. Usually, the obvious way to do something is the best way. Keep it simple.
- SSE involves destructive instructions most of the time. If you can phrase your algorithm in terms of destructive logic, you can probably save some unnecessary copies, and possibly some register spillage. (This will probably preclude software pipelining. However, software pipelining may not be necessary because the Intel processors are highly out-of-order.)
- Heed cautions and tips in the Intel Processor Optimization Reference Manual.

# Document Revision History

This table describes the changes to *AltiVec/SSE Migration Guide*.

| Date | Notes |
|---|---|
| 2005-09-08 | New document that explains how to convert PowerPC AltiVec code to Intel SSE code. |