



**NAS**software Limited  
Incorporating InfoSAR

# Translation of PowerPC/AltiVec SIMD Macros to IA32/SSE

Peter Cromwell, Ian McConnell\*  
N.A. Software Ltd

Revision 1.0; Released 16th March 2009

## 1 Introduction

This document is aimed at experienced developers who needs to migrate their existing vector-oriented C/C++ PowerPC AltiVec code to the Intel x86 (Intel Architecture 32-bit) SSE (Streaming SIMD Extensions) extensions.

The SIMD facilities of the PowerPC/AltiVec chip can be accessed from C code using the `altivec.h` header file. This makes the interface defined in the Motorola specification *AltiVec Technology Programming Interface Manual* available through a set of macros that target the SIMD assembly instructions.

This document accompanies a modified version of the `altivec.h` file (`altivec2sse.h`) which uses the same Motorola interface but targets Intel processors with the SSE2 level of SIMD support.

This can be done because, at the conceptual level, AltiVec and SSE are quite similar. They are single instruction/multiple data (SIMD) vector units with  $4 \times 32$  bit vectors that prefer to be 16 byte aligned. The vectors are accessed through a C programming interface which treats them as a special 128 bit data type with a set of function-like intrinsics. Intel used the `_mm_` prefix whereas AltiVec has `vec_`. Generally, there is a good correlation between the two instruction sets with about two thirds of the most commonly used functions directly translatable. For the remaining AltiVec instructions, many can be emulated with a few SSE calls.

Places that are likely to require a programmer's attention include:

- code for handling misaligned data
- use of instructions that saturate the result on overflow
- use of instructions in which the position of individual elements within a SIMD vector matters (*e.g.* `pack`, `perm`).

In the first case, reading misaligned data is much simpler in SSE — you no longer need to load two adjacent aligned vectors and extract the required elements, you just load an unaligned vector with `_mm_loadu_si128` or `_mm_loadu_ps` for integer or float data. The last case is a problem caused by the change in endian format (see §??).

Note that the kind of translation provided (direct mapping, inline function, none) may vary with the data type. If the instruction you want is not translated, check to see whether switching between signed and unsigned, or changing the width of an integer type will help.

For more information on porting code between AltiVec and SSE see:

*AltiVec/SSE Migration Guide*,

[http://developer.apple.com/documentation/performance/Conceptual/Accelerate\\_sse\\_migration/SSE\\_Performance\\_Programming](http://developer.apple.com/documentation/performance/Conceptual/Accelerate_sse_migration/SSE_Performance_Programming),

<http://developer.apple.com/hardwaredrivers/ve/sse.html>

## 2 Requirements and use

You will need to have the standard header files `xmmintrin.h` and `emmintrin.h` installed on your system to provide access to the SSE and SSE2 instructions.

---

This work was funded by Intel Corporation with whom copyright exists

Replace the original `altivec.h` header file you use for compiling on a PowerPC with the file `altivec2sse.h` containing SSE translation macros.

Recompile your code on an Intel machine using the appropriate flag to enable the SSE extensions, typically `-msse2`. Under VxWorks you may also need the flag `-flax-vector-conversions` to remove superfluous type conversion errors.

### 3 Compiler dependency

In the Motorola specification, the same high-level function name is used with multiple data types and the compiler selects the correct instruction based on the types of the arguments the function is called with. The `altivec.h` header file has been derived from one that was supplied with `gcc`. It makes use of some internal compiler macros to fake the function overloading in the C interface. This is probably not portable between compilers so you may need access to `gcc`. Fortunately, the compilers supplied for use on real-time systems or for building DSP applications are often based on `gcc`.

### 4 Endian issues

AltiVec is big-endian and SSE is little-endian. Both units represent their data with IEEE-754 floating point format, but the Intel architecture stores the results in little-endian order. So, for the array

```
float data[4]= { 10.f, 20.f, 30.f, 40.f };
vFloat v = _mm_loadu_ps(&data[0]);
```

the data in `v` will look like this:

```
v = {40.f, 30.f, 20.f, 10.f}
```

On the PowerPC values stored in memory and values stored in a processor register are represented in the same way. On Intel processors, the order of the bytes is reversed when data are transferred between registers and memory.

If your code is pure SIMD so that you always apply identical operations to all elements of a vector then this will not affect you. When the results are saved to memory, the Intel architecture restores the byte order to what would be expected.

If you refer to left/right, high/low or even/odd locations within a vector and you are getting incorrect results, this is very likely to be the source of the problem.

In the case of a vector register, the entire vector is byte swapped: not only are the bytes representing each element reversed, but also the order of the elements within a vector. The first is essentially invisible to the programmer but the second affects the indexing in the vector. Any instruction that addresses a particular element in a vector will access the wrong location. For example, it interchanges left and right shifts when permuting elements.

In general, a macro does not have sufficient context to determine whether the elements of a vector register should be indexed in register order or memory order. The following instructions have been implemented with both register order and memory order versions. By default you will get the register order implementation, which is a literal translation of the AltiVec macro. If you define the macro `MEMORY_ORDER` before reading in `altivec.h` you will get the memory order implementation, which reverses the indexing order.

- splat: `vec_splat`
- merge: `vec_merge`, `vec_mergel`
- shift octet: `vec_slo`, `vec_sro`
- multiply even/odd: `vec_mule`, `vec_mulo`
- unpack: `vec_unpackh`, `vec_unpackl`.

Fortunately, one of the main uses of the AltiVec permute unit is in loading and storing misaligned data but, as mentioned above, SSE provides easier ways to do this.

## 5 Efficiency issues

Do not expect your highly tuned AltiVec code to be translated into high-performance SSE. You will get a quick and easy first cut.

In AltiVec you can gain a speed advantage by unrolling up to eight-way in parallel. On x86, there are fewer registers and unrolling may overflow the register file causing a large number of extra loads. In this case returning to the simpler, unrolled code may give a performance advantage.

If you want high-performance SSE, you may need to re-examine your algorithm to account for the out-of-order execution and fewer registers. Some of the issues are discussed below.

Whether an AltiVec instruction maps onto a single SIMD instruction, a sequence of SIMD instructions, or a serial implementation can depend on the datatype. You may be able to switch to a faster instruction by changing the width or signed/unsigned condition of an integer.

AltiVec logical shift and arithmetic shift instructions operate independently on each element of a SIMD vector, but the SSE equivalents shift all elements by the same amount. The AltiVec behaviour is simulated with serial implementations. If you do not need this generality, you will get better performance by changing your code to call the SSE functions.

## 6 Accuracy and rounding

Any algorithm using floating point calculations being ported from the AltiVec to SSE should be tested for numerical accuracy. While both processors store their data with IEEE-754 floating point format, the AltiVec design is based upon a fused multiply add. The Intel processor separates these into a multiply and an add operation and so may incur an extra rounding step.

The Intel processor does, however, support more rounding modes (nearest, zero, Inf and -Inf) compared to the *round to nearest* of the AltiVec.

## 7 Instructions that are simulated

Each of the following AltiVec instructions has no direct SSE equivalent but its effect has been produced by combining sequences of SIMD instructions, or with a serial implementation that processes each element of the vector in turn. Whether simulation is necessary may

depend on the data type: some AltiVec instructions have direct translations for some but not all of `signed/unsigned`, `char`, `short`, and `int`.

### 7.1 Instructions with SIMD simulations

- absolute value: `vec_abs`, `vec_abss`
- average: `vavguw`, `vavgusb`, `vavgsh`, `vavgsw`
- rounding: `vrfiz`, `vrfigp`, `vrfigm`, `vrfign`
- type conversion: `vcfsx`, `vctsx`
- unsigned comparison: `vcmpgtub`, `vcmpgtuh`, `vcmpgtuw`
- bounds checking: `vcmpbfp`
- logical nor: `vnor`
- maximum: `vmaxsb`, `vmaxuh`, `vmaxuw`, `vmaxsw`, `vmaxfp`
- minimum: `vminsb`, `vminuh`, `vminuw`, `vminsw`, `vminfp`
- select: `vec_sel`
- shift octet: `vslo`, `vsro`
- splat: `vspltb`, `vsplth`, `vspltw`, `vspltf`.
- approximation: `vec_expte`, `vec_loge`
- unsigned type conversion: `vec_vcfux`, `vec_vctux`

Cache hints are also ignored. The LRU variants `vec_ldl` and `vec_stl` of the load and store instructions that mark cache lines as ‘least recently used’ are `#defined` to the ordinary `vec_ld` and `vec_st`.

### 7.2 Instructions with serial simulations

- load/store by element: `vec_lde`, `vec_ste`
- multiply even/odd: `vec_mule`, `vec_mulo`
- multiply-sum: `vec_msum`
- pack/unpack: `vec_pack`, `vupkhsb`, `vupkhsh`, `vupklhsb`, `vupklsh`
- logical shift: `vslb`, `vslh`, `vslw`, `vsrb`, `vsrh`, `vsrw`
- arithmetic shift: `vsrab`, `vsrah`, `vsraw`
- rotate: `vrlb`, `vrlh`, `vrlw`
- long shift: `vec_sll`, `vec_srl`.

## 8 Instructions that are ignored

The following AltiVec instructions are `#defined` to be empty macros, or zero if they return a value. You will not get an error by using them — they will just be ignored.

- cache touches: `vec_dss`, `vec_dssall`, `vec_dst`, `vec_dstst`, `vec_dstt`, `vec_dststt`
- status register: `vec_mfvscr`, `vec_mtvscr`.

## 9 Instructions with no translation

The following AltiVec instructions do not have direct SSE equivalents and have not been given alternative implementations — you will get an ‘undefined reference’ error if you attempt to link code that uses them.

For most of the omitted instructions the position of individual elements within the vector matters (see §??) or the AltiVec instructions saturate the result on overflow. In the first case a programmer should analyse the code to ensure the correct order of elements is used; in the second case it may be possible to substitute the non-saturated version of an instruction.

- include carry: `vec_addc`, `vec_subc`
- saturate result: `vec_vaddsws`, `vec_vadduws`, `vec_madds`, `vec_mradds`, `vec_msums`, `vec_vsubsws`, `vec_vsubuws`
- saturated pack: `vec_vpkshus`, `vec_vpkswus`, `vec_vpkuwus`
- saturated sum across vector: `vec_sums`, `vec_sum2s`, `vec_sum4s`
- pack/unpack pixels: `vec_packpx`, `vec_vupkhp`, `vec_vupklp`
- multiply low and add: `vec_mladd`
- create shift vectors for unaligned data: `vec_lvsl`, `vec_lvslr`
- permutation: `vec_perm`,
- double shift: `vec_sld`.

## 10 Benchmarks

We give here examples of PowerPC/AltiVec vector codes as run on PowerPC and then on Intel using the `altivec2sse.h` include file. The systems used to provide timings are:

Linux/Intel: an Intel Core 2 (T7200), 2GHz, 4Mb cache.

Linux/PPC: a MPC8641HPCN board (7448), 1.5GHz.

These have different clock speeds; we factor this out by giving the number of clock cycles used per vector element.

### 10.1 Vector Add

The AltiVec code to sum two arrays of floating point vectors

```
float *block_out, *block_in1, *block_in2;
for (i = 0; i < v_length; i += 4) {
    vector float Avf32, Bvf32, Rvf32;
    Avf32 = vec_ld(0, &block_in1[i]);
    Bvf32 = vec_ld(0, &block_in2[i]);
    Rvf32 = vec_add(Avf32, Bvf32);
    vec_st(Rvf32, 0, &block_out[i]);
}
```

will run on SSE with the inclusion of `altivec2sse.h`.

```
#define MEMORY_ORDER
#include "altivec2sse.h"
```

Timings in instruction cycles per array element are given below.

Vector length	Linux/Intel	Linux/PPC
000256	1.1	1.5
001024	1.2	1.5
004096	1.8	3.4
016384	1.8	3.4
065536	1.9	6.0
131072	1.8	21.3

Timings in  $\mu$ seconds:

Vector length	Linux/Intel	Linux/PPC
000256	0.1	0.3
001024	0.6	1.0
004096	3.6	9.2
016384	14.7	36.7
065536	61.8	261.2
131072	120.2	1859.0

## 10.2 Vector Sine

Model AltiVec code to calculate the sine of an array of floating point numbers while taking account of the vector length of four might look like:

```
float *block_out, *block_in;
for (i = 0; i < v_length; i += 4) {
    vector float Avf32, Rvf32;
    Avf32 = vec_ld(0, &block_in[i]);
    Rvf32 = vsin(Avf32);
    vec_st(Rvf32, 0, &block_out[i]);
}
```

A fast vector sine algorithm is given in "A Fast, Vectorizable Algorithm for Producing Single-Precision Sine-Cosine Pairs" available from <http://arxiv.org/pdf/cs.MS/0406049>

The code is reproduced below

```
vector float vsin(vector float v)
{
    vector float s1, s2, c1, c2, fixmag1;
    vector float vzero = VEC_CONST(0.0);
    vector float vone = VEC_CONST(1.0);
    vector float vtwo = VEC_CONST(2.0);
    vector float vhalfpi = VEC_CONST(1.0/(2.0*3.1415926536));
    vector float v_ss1 = VEC_CONST( 1.5707963268);
```

```

vector float v_ss2 = VEC_CONST(-0.6466386396);
vector float v_ss3 = VEC_CONST( 0.0679105987);
vector float v_ss4 = VEC_CONST(-0.0011573807);
vector float v_cc1 = VEC_CONST(-1.2341299769);
vector float v_cc2 = VEC_CONST( 0.2465220241);
vector float v_cc3 = VEC_CONST(-0.0123926179);

vector float x1 = vec_madd(v, vhalfpi, vzero);
/* q1=x/2pi reduced onto (-0.5,0.5), q2=q1**2 */
vector float q1 = vec_nmsub(vec_round(x1), vone, x1);
vector float q2 = vec_madd(q1, q1, vzero);
s1= vec_madd(q1,
             vec_madd(q2,
                     vec_madd(q2, vec_madd(q2, v_ss4, v_ss3), v_ss2),
                     v_ss1),
             vzero);
c1= vec_madd(q2,
             vec_madd(q2, vec_madd(q2, v_cc3, v_cc2), v_cc1),
             vone);
/* now, do one out of two angle-doublings to get sin & cos theta/2 */
c2 = vec_nmsub(s1, s1, vec_madd(c1, c1, vzero));
s2 = vec_madd(vtwo, vec_madd(s1, c1, vzero), vzero);
/* now, cheat on the correction for magnitude drift...
   if the pair has drifted to (1+e)*(cos, sin),
   the next iteration will be (1+e)**2*(cos, sin)
   which is, for small e, (1+2e)*(cos,sin).
   However, on the (1+e) error iteration,
   sin**2+cos**2=(1+e)**2=1+2e also,
   so the error in the square of this term
   will be exactly the error in the magnitude of the next term.
   Then, multiply final result by (1-e) to correct */
/* must use this method with un-normalized series, since magnitude error is large */
fixmag1 = Reciprocal(vec_madd(s2,s2,vec_madd(c2,c2,vzero)));
c1 = vec_nmsub(s2, s2, vec_madd(c2, c2, vzero));
s1 = vec_madd(vtwo, vec_madd(s2, c2, vzero),
             vzero);
return vec_madd(s1, fixmag1, vzero);
}

```

When compiling on a PowerPC machine, the `-faltivec` option to `gcc` allows the construct

```
vector float vzero = (vector float)(0.);
```

This shorthand is not supported by Intel/SSE version of `gcc` so it may be necessary to manually edit the code to use the more general form

```
vector float vzero = {0., 0., 0., 0.};
```

The general form should work on both AltiVec and SSE versions of gcc, but in the above code a macro has been used to switch notations:

```
#ifdef ALTIVEC
#define VEC_CONST(x) (vector float)(x)
#else
#define VEC_CONST(x) {(x), (x), (x), (x)}
#endif
```

The benchmark results for this code, in machine cycles per array element are

Vector length	Linux/Intel	Linux/PPC
000256	29.7	27.1
001024	30.2	27.0
004096	30.3	27.1
016384	30.3	28.5
065536	31.9	28.5
131072	30.6	31.8

The timings in  $\mu$ seconds:

Vector length	Linux/Intel	Linux/PPC
000256	3.9	4.6
001024	17.5	18.5
004096	61.7	74.1
016384	246.5	311.7
065536	983.7	1247.0
131072	1961.3	2779.5

### 10.3 Fourier Transform

To illustrate the conversion process on a typical FFT routine we use a 1024 point FFT module (a Stockham algorithm) which forms part of the N.A. Software multi-algorithm FFT suite. Results for this routine are:

	Linux/Intel	Linux/PPC
Cycles/elt	30.0	29.5
$\mu$ seconds	30.7	20.1