New Instructions in VMX128

Collapse All

By Bruce Dawson, Software Design Engineer

Advanced Technology Group (ATG)

Last updated: October 2008

VMX is a powerful vector-processing instruction set that provides a solid base for Xbox 360. To improve VMX for game programming, however, we made some changes for the Xbox 360 processors. The Xbox 360 version of VMX is called VMX128.

Here are the changes to VMX128:

- Added 96 registers to allow greater loop unrolling and less spilling of variables to memory.
- Adjusted most instructions to allow access to the extra registers. All of the new instructions discussed here can reference all 128 registers without restrictions.
- Added new instructions:
  - vmulfp128 – vector multiply floating point
  - vpermwi128 – vector permute word immediate
  - vrlimi128 – vector rotate left immediate and mask insert
  - lvlx128, lvrx128 – load vector left indexed and right indexed
  - stvlx128, stvrx128 – store vector left indexed and right indexed
  - vpkd3d128, vupkd3d128 – vector pack D3D and vector unpack D3D
  - vmsum3fp128, vmsum4fp128 – dot product
- Removed some instructions – primarily the vector integer multiply instructions

This white paper discusses the new VMX128 instructions in detail.

VMX Register Overview

To understand the VMX128 instructions, it helps to understand how the elements of a VMX register are numbered and labeled, and how they are stored in memory.

Figure 1. Elements in a VMX Register

Most significant bits... ...Least significant bits

Low address... ...High address

B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11	B12	B13	B14	B15
Half 0		Half 1		Half 2		Half 3		Half 4		Half 5		Half 6		Half 7	
Word 0				Word 1				Word 2				Word 3			
X				Y				Z				W			

As the diagram shows, the X element is also known as word zero. It is stored in the most-significant bits of the register and in the lowest address in memory. VMX registers can be interpreted as words, half-words, bytes, or bits.

vmulfp128

Name: Vector Multiply Floating-Point

Intrinsics syntax: __vector4 result = __vmulfp(V0, V1);

Assembler syntax: vmulfp128 VRT, VRA, VRB

Latency: 12 cycles

Pipeline: Vector float pipeline

Regular VMX doesn't have a vector floating-point multiply instruction. Instead, the vmaddfp instruction must be used, with a zero for the addend. The addition of **vmulfp** to VMX128 doesn't make the multiplication faster — the latency of **vmulfp** is identical to vmaddfp. It just makes vector multiplies more convenient by not requiring an extra instruction to load a zero, and an extra register to store the zero.

Example

```
// Scale the elements of V by the elements of Scale, and store the
// result in ScaledV
__vector4 ScaledV = __vmulfp( V, Scale );
```

vpermwi128

Name: Vector Permute Word Immediate

Intrinsics syntax: __vector4 result = __vpermwi(V0, permute);

Assembler syntax: vpermwi128 VRT, VRB, permute

Latency: 4 cycles

Pipeline: Permute pipeline

The **vpermwi** instruction does an arbitrary swizzle of the words of a register and copies the results to the destination. For each of the four words in the destination register, the mask specifies which of the four source words should be placed there. The high two bits of the permute constant specify which word should go into the X field, the next two bits of the permute constant specify which word should go into the Y field, and so on. The following table specifies the values used to select a particular input field.

Value	Vector Element
0	X
1	Y
2	Z
3	W

The permute value must be a compile-time constant. To create appropriate constants, use the following macro, from vectorintrinsics.h.

```
#define VPERMWI_CONST(x,y,z,w)
((((x)&3) << 6) | (((y)&3) << 4) | \
                                (((z)&3) << 2) | ((w)&3))
```

The four parameters identify which words to copy to the x, y, z, and w words of the destination.

Example

```
// Copy V0 to resultYXZZ while rearranging the xyzw elements
// into the order yxzz.
// In binary, the permute constant will look like 01 00 10 10
// in order to select y, x, z, and then z again. In hex, it is 0x4A.
__vector4 resultYXZZ = __vpermwi( V0, VPERMWI_CONST( 1, 0, 2, 2 ) );
```

vrlimi128

Name: Vector Rotate Left Immediate and Mask Insert

Intrinsics syntax: result = __vrlimi(result, V1, mask, shift);

Assembler syntax: vrlimi128 VRT, VRB, mask, shift

Latency: 4 cycles

Pipeline: Permute pipeline

The **vrlimi** instruction does a word-based rotate and mask insert of one register into another. The source register (the second source register in the intrinsic form) is rotated zero to three words to the left. A rotation amount of one moves x to w, y to x, z to y, and w to z. Then, for each of the four words of a VMX register, a mask bit selects whether to insert the shifted word or leave the destination alone. A set bit in the mask means that the shifted input is inserted into the destination. The mask bits for the vector elements are as follows:

Bitmask	Vector Element
8	X
4	Y
2	Z
1	W

The reason the bitmask for X is eight is because X is word zero, so X is represented by bit zero in the bitmask. Bits on PowerPC are numbered starting from the most-significant bit, which in a four-bit field has a value of eight.

The source register is not modified.

Both the rotate amount and the insert mask must be compile-time constants.

The **vrlimi** instruction does a mask insert into the target register, so only one source register is specified; the other is implicitly the destination. However, the C intrinsic lets you specify two source values. Generating code with **__vrlimi** will be most efficient if you assign the result to the register specified for the first input.

Example

```
// Shift OutDataN so the x component is in the w component, and then
// mask insert OutDataN with OutDataV so that the w component is
// replaced with the shifted value from OutDataN.
```

```
// The shift value of one shifts OutDataN left one position, wrapping
// x around to w. The mask value of 1 selects the w value for
// insertion from OutDataN.
// OutDataN and OutDataV are not modified.
// Because the destination and first source registers are different,
// the compiler will insert an extra vector move instruction.
__vector4 packed = __vrlimi( OutDataV, OutDataN, 1, 1 );
```

lvlx128, lvrx128

Name: Load Vector Left Indexed and Load Vector Right Indexed

Intrinsics syntax: __vector4 result = __lvlx(pointer, offset);

Intrinsics syntax: __vector4 result = __lvrx(pointer, offset);

Assembler syntax: lvlx128 VRT, RA, RB

Assembler syntax: lvrx128 VRT, RA, RB

Latency: 2 cycles

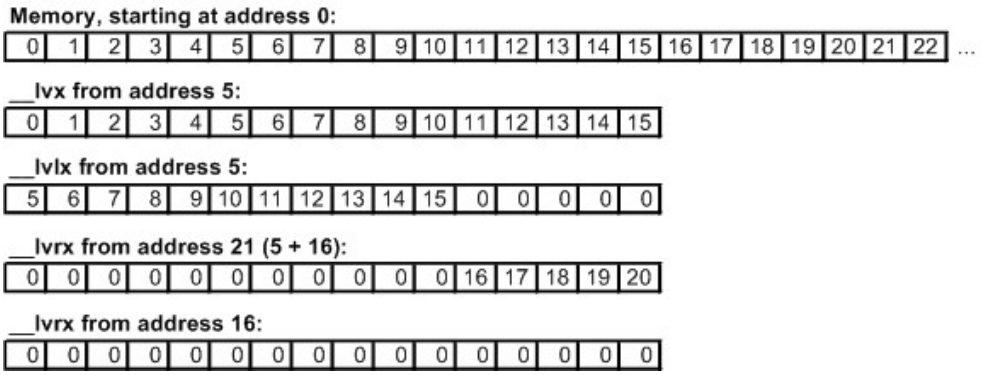
Pipeline: Vector/Scalar Load Pipeline

The lvlx and lvrx instructions are designed to support loading misaligned vector data. lvlx also allows single-instruction loading of naturally aligned bytes, shorts, floats, or 8-byte data.

- lvlx uses indexed addressing, so it calculates the load address by adding the contents of the two integer registers specified. As usual, if RA is R0, the value zero is used instead of the register contents. lvlx then loads data from the specified address up to the next 16-byte boundary. The loaded data is left aligned in the register, starting at byte zero, and the remainder of the register is zeroed. Thus, lvlx fills the left portion of the register, starting with element x, with 1 to 16 bytes of data. If the address is 16-byte aligned, lvlx will load 16 bytes of data.
- lvrx is designed to load the right portion of a vector register after being passed an address 16 bytes beyond the vector's location in memory. lvrx loads from the previous 16-byte boundary to the specified address. If the specified address is a 16-byte boundary, no data is loaded. The loaded data is right-aligned in the register.

The following chart shows a block of memory and the data loaded from it by lvx, lvlx, and lvrx.

Figure 2. Block of Memory



The following macro, from vectorintrinsics.h, demonstrates how to use the intrinsics **__lvlx** and **__lvrx** to load a 16-byte vector from an unaligned address.

```
#define __loadunalignedvector(ptr) \
                __vor(__lvlx(ptr, 0), __lvrx(ptr,16))
```

If you have a byte, short, float, or 8-bit block of data that you want to load into a VMX register, as long as the data doesn't cross a 16-byte boundary, you can load it with a single lvlx instruction. As long as your data is natively aligned, it is guaranteed not to cross a 16-byte boundary.

Example

```
// Load a scalar 32-bit value (must be 4-byte aligned) from
// address ptr+offset into the x element of scalarFloat.
// The other elements of scalarFloat will either be loaded from
// memory or zeroed, depending on the 16-byte relative alignment
// of the address.
__vector4 scalarFloat = __lvlx( ptr, offset );
```

Example

```
// Load 16 bytes of data from an arbitrarily aligned address.
// The offset for __lvrx must be 16 bytes greater than the
// offset for __lvlx.
__vector4 result = __vor( __lvlx( ptr, 0 ), __lvrx( ptr, 16 ) );
```

If you use lvlx or lvrx on non-cacheable memory—memory allocated with PAGE_NOCACHE or PAGE_WRITECOMBINE—you get an alignment exception crash.

stvlx128, stvrx128

Name: Store Vector Left Indexed and Store Vector Right Indexed

Intrinsics syntax: __stvlx(V0, pointer, offset);

Intrinsics syntax: __stvrx(V0, pointer, offset);

Assembler syntax: stvlx128 VRS, RA, RB

Assembler syntax: stvrx128 VRS, RA, RB

Latency: not applicable

Pipeline: Vector/Scalar Store Pipeline

The stvlx and stvrx instructions are the store equivalents of lvlx and lvrx. They store arbitrarily aligned vector registers.

- stvlx stores the left portion of the vector register to the specified address, up to the next 16-byte boundary. If the address is a multiple of 16 bytes, stvlx will store 16 bytes.
- stvrx stores the right portion of the vector register up to the specified address, from the previous 16-byte boundary. If the address is a multiple of 16 bytes, stvrx does nothing.

The following macro, from vectorintrinsics.h, demonstrates how to use the intrinsics __stvlx and __stvrx to store a 16-byte vector to an unaligned address.

```
#define __storeunalignedvector(v, ptr) \
               (__stvlx(v, ptr, 0), __stvrx(v, ptr,16))
```

Example

```
// Store 16 bytes to an arbitrarily aligned address.
__stvlx( V0, ptr, 0 );
__stvrx( V0, ptr, 16 );
```

The instructions stvlx and stvrx work well for misaligned full-register stores, but they do not work well for doing partial register stores. However the instructions stvebx, stvehx, and stvewx from regular VMX can be used for this purpose.

If you use stvlx or stvrx on non-cacheable memory—memory allocated with PAGE_NOCACHE or PAGE_WRITECOMBINE—you get an alignment exception crash.

vpkd3d128, vupkd3d128

Name: Vector Pack D3D and Vector Unpack D3D

Intrinsics syntax: __vector4 result = __vupkd3d(packed, DT);

Intrinsics syntax: result = __vpkd3d(result, result, DT, MS, SHW);

Assembler syntax: vupkd3d128 VRT, VRB, DT

Assembler syntax: vpkd3d128 VRT, VRB, DT, MS, SHW

Latency: 4 cycles

Pipeline: Permute pipeline

The **vpkd3d** and **vupkd3d128** instructions are designed to pack and unpack between 32-bit float data and various packed formats. Using packed data can conserve memory, conserve memory bandwidth, and reduce cache usage. Although packing and unpacking data requires extra instructions, the memory bandwidth saved often makes these extra instructions worthwhile.

The packed formats are also supported by the GPU.

The DT, MS, and SHW parameters must all be compile time constants because they are embedded in the instruction opcode.

The **vpkd3d** instruction does a mask insert into the target register, so only one source register is specified; the other is implicitly the destination. However, the C intrinsic lets you specify two source values. Code generation with the **__vpkd3d** intrinsic will be most efficient if the result is assigned to the same register that was specified for the first input.

DT: Data Type, the Format of the Packed Data

There are seven different types of data that can be packed and unpacked, specified by a data-type code. Each data type packs two or four floats into a 32-bit or 64-bit result. The individual items in the packed data can be unsigned integers, signed integers, or 16-bit floating-point numbers. The following table lists these data types and the types of data that are produced. The packed fields information identifies which floating-point elements (X, Y, Z, or W) are packed, how many bits are in each packed field, and the packed type ("u" for unsigned, "s" for signed, and "f" for float). "W to 8u" means that the floating-point value in the W field is packed to an 8-bit unsigned integer, or vice versa when unpacking data.

Data Type Identifier	Packed Fields			
VPACK_D3DCOLOR	W to 8u	X to 8u	Y to 8u	Z to 8u
VPACK_NORMSHORT2	X to 16s	Y to 16s	0.0 on unpacking	3.0 on unpacking
VPACK_NORMPACKED32	W to 2u	Z to 10s	Y to 10s	X to 10s

VPACK_FLOAT16_2	X to 16f	Y to 16f	0.0 on unpacking	1.0 on unpacking
VPACK_NORMSHORT4	X to 16s	Y to 16s	Z to 16s	W to 16s
VPACK_FLOAT16_4	X to 16f	Y to 16f	Z to 16f	W to 16f
VPACK_NORMPACKED64	W to 4u	Z to 20s	Y to 20s	X to 20s

In all cases, the right-most field goes in the least-significant bits of the vector register, with the other fields packed beside it. The packed data is always 32-bits or 64-bits total, filling either the w element or the z and w elements. For instance, in the VPACK_FLOAT16_4 case, the X and Y components of the floating-point vector are packed into the Z component, and the Z and W components are packed into the W component.

It's worth pointing out that VPACK_D3DCOLOR, VPACK_NORMPACKED32, and VPACK_NORMPACKED64 all modify the order of X, Y, Z, and W, as can be seen in the chart above.

When packing data, the floating-point input values are clamped to the maximum range of the destination format—no clamping is needed before using vpkd3d128.

When unpacking, the data to unpack always comes from the least-significant bits of the vector register — the w element or the z and w elements.

MS: Mask Select

You can specify a mask to control how the packed data is inserted into the destination register. There are three different mask select options:

1. VPACK_32: insert the least-significant 32 bits of the packed data. This is typically used with the data types that pack to 32 bits.
2. VPACK_64LO: insert 64 bits of packed data. This is typically used with the data types that pack to 64 bits. If the shift amount is three, only the low word is inserted (the w word, to the most significant 32-bits of the destination).
3. VPACK_64HI: This is the same as VPACK_64LO, except in the case where the shift amount is three. In that case, only the high word is inserted (the z word, to the least-significant 32-bits of the destination).

SHW: Shift Word amount

When packing data with **vpkd3d**, the packed data is mask-inserted into the destination register. The packed data defaults to being mask-inserted into the least-significant bits of the destination, but you can specify a shift amount from zero to three in order to shift the packed data left that many 32-bit words before inserting.

Conversions

The conversions between 32-bit floating-point and packed fields vary depending on whether the packed data is signed, unsigned, or floating-point.

Signed Packed

When unpacking from one of the signed integer formats to 32-bit floats, the hardware does a simple bit operation. It takes the signed number, sign extends it, and then does an integer add to the floating-point representation of 3.0. If x is the signed integer being unpacked, the result is a floating-point value of $3.0 + x * 2^{-22}$. After unpacking, use vmaddfp to convert the value to the range you want to work in.

If the range you want is −1.0 to 1.0, to get the fractional part into range you need to multiply by (1 << 22) / ((1 << (numBits−1)) − 1). Then you need to subtract the 3.0. Since the 3.0 has now been scaled up, you need to subtract 3.0 times your multiplier. To maintain precision, you must do this in a single vmaddfp operation. If you are unpacking a 16-bit signed type, the code would look like this:

Example

```
#define UnpackMul16S  ((1 << 22) / (float)((1 << 15) - 1))
#define UnpackAdd16S (-UnpackMul16S * 3.0)
static const __vector4 vUnpackMul16161616 = { UnpackMul16S,
                UnpackMul16S, UnpackMul16S, UnpackMul16S };
static const __vector4 vUnpackAdd16161616 = { UnpackAdd16S,
                UnpackAdd16S, UnpackAdd16S, UnpackAdd16S };
// Unpack the least-significant 16-bits of input, the z and
// w elements.
__vector4 result = __vupkd3d( V0, VPACK_NORMSHORT4 );
// Convert the unpacked data from 3.0 + x * 2^-22 range to
// 0.0 to 1.0 range.
__vector4 normalResult = __vmaddfp( result,
                vUnpackMul16161616, vUnpackAdd16161616 );
```

There is one special case. If the signed value is the most negative value that can be represented, a NAN (*Not A Number*) is created instead. This gives the same number of valid numbers above and below zero.

When packing, you must follow the reverse process. Use vmaddfp to get your floating-point data into the correct format — $3.0 + x * 2^{-22}$ — and then use **vpkd3d**. The multiplier constant is the reciprocal of the constant used for unpacking, and the addition constant is simply 3.0.

Example

```
#define PackMul16S     ((float)((1 << 15) - 1) / (1 << 22))
static const __vector4 vPackMul16161616 = { PackMul16S, PackMul16S,
                PackMul16S, PackMul16S };
static const __vector4 vPackAdd = { 3.0, 3.0, 3.0, 3.0 };
// Convert the floats to be packed from 0.0 to 1.0 range to
// 3.0 + x * 2^-22 range.
__vector4 result = __vmaddfp( V0, vPackMul16161616, vPackAdd );
result = __vpkd3d( result, result, VPACK_NORMSHORT4, VPACK_64LO, 0 );
```

Unsigned Packed

Unsigned numbers are dealt with in a way similar to signed numbers, but with a few important differences.

When unpacking unsigned data, the result is in the form 1.0 + x * 2^{-23}. The base value is 1.0 instead of 3.0, which means that the least-significant bit of the float, where the packed data is placed, is now 2^{-23} instead of 2^{-22}. The multiplication constant for unpacking unsigned numbers is of the form ((1 << 23) / ((1 << numBits) − 1). The code to unpack unsigned 8-bit values to the range 0.0 to 1.0 looks like this:

Example

```
#define UnpackMul8U  ((1 << 23) / 255.0)
static const __vector4 vUnpackMul8888 = { UnpackMul8U,
                UnpackMul8U, UnpackMul8U, UnpackMul8U };
static const __vector4 vUnpackAdd8888 = { -UnpackMul8U,
                -UnpackMul8U, -UnpackMul8U, -UnpackMul8U };
// The data to be unpacked must be in the least-significant bits,
// in the w element.
// Unpack the data into 1.0 + x*(2^-23) range.
__vector4 result = __vupkd3d( input, VPACK_D3DCOLOR );
// Restore the data to the range 0.0 to 1.0.
__vector4 normalResult = __vmaddfp( result,
                vUnpackMul8888, vUnpackAdd8888 );
```

When packing unsigned data, the data needs to be in the 3.0 + x * 2^{-22} format.

Example

```
#define PackMul8     (255.0 / (1 << 22))
static const __vector4 vPackMul8888 = { PackMul8, PackMul8,
                PackMul8, PackMul8 };
static const __vector4 vPackAdd = { 3.0, 3.0, 3.0, 3.0 };
// Assume the four floats in "input" are in the range
// 0.0 to 1.0. We need to get them into the range
// 3.0 + x*(2^-22), where x is a number from 0 to 255
__vector4 result = __vmaddfp( input, vPackMul8888, vPackAdd );
// This places the 32-bit result in the least-significant bits
// of result - the w element.
result = __vpkd3d( result, result, VPACK_D3DCOLOR, VPACK_32, 0 );
```

Some pack types, such as NORMPACKED32 and NORMPACKED64, contain both signed and unsigned elements, so your unpacking code must use a mixed set of multipliers and addends.

Floating-Point Packed

The 16-bit floating-point format used by Xbox 360 differs slightly from the 16-bit floating-point format used elsewhere. 16-bit floats consist of a sign bit, a 5-bit exponent, and a 10-bit mantissa. The format works similarly to 32-bit floats (including an implied leading 1 in the mantissa) and denormals. However, Xbox 360 16-bit floats do not support infinities and NANs, and the denormals are not used by the pack and unpack instructions. Instead, the maximum exponent value is used to double the maximum representable value. Xbox 360 16-bit floats have a maximum exponent of +16, which includes an exponent bias of 15, giving them a maximum value of 131,008.

The conversion from 16-bit float to 32-bit float is exact, with one exception: A 16-bit float that is a denormal is converted to a 32-bit float zero.

When packing from a 32-bit float to a 16-bit float, some accuracy and range is lost, but the value will be retained without requiring extra scaling steps. The conversion to 16-bit float rounds toward zero rather than toward nearest. 32-bit floats greater than or equal to 131,008 will be truncated to 131,008. 32-bit floats that are too small to be represented as a normalized 16-bit float will be converted to zero.

Example

```
// Convert four floats to four HALFs and mask insert them into
// the least-significant 64-bits of "a" (the z and w words)
a = __vpkd3d( a, source, VPACK_FLOAT16_4, VPACK_64LO, 0 );
```

Example

```
// Unpack the least-significant 64 bits of packed into four
// floats in "a".
```

```
a = __vupkd3d( packed, VPACK_FLOAT16_4 );
```

vmsum3fp128, vmsum4fp128

Name: Vector Multiply Sum Three Floating-Point and Vector Multiply Sum Four Floating-Point

Intrinsics syntax: __vector4 result = __vmsum3fp(V0, V1);

Intrinsics syntax: __vector4 result = __vmsum4fp(V0, V1);

Assembler syntax: vmsum3fp128 VRT, VRA, VRB

Assembler syntax: vmsum4fp128 VRT, VRA, VRB

Latency: 14 cycles

Pipeline: Dot-product pipeline

The **vmsum** instructions do a dot product — that is, they perform an element-by-element multiply and then sum the results of the multiplication. The series of dependent operations and the rearrangements necessary to add different elements make dot product an expensive operation to write in most vector architectures. Because of this, and because dot product is so useful in computer graphics, it was important to add this instruction to VMX128.

The result of the dot product is written to all elements of the result vector. **vmsum3fp** ignores the w elements of the source vectors, but the latency is the same and the result is still written to all four elements.

The dot-product instructions will sometimes give surprising 1-bit errors in the least-significant bits of the results. For instance, **vmsum4fp((1, 1, 1, 1), (1, -1, 1, -1))** will not actually produce zero. These errors should be at the limits of floating-point precision and can usually be ignored. The next section describes how the dot-product instructions work and why these 1-bit errors occur.

Dot-Product Details

The preceding sections provide what you need to know about dot-product instructions — for most purposes. To understand expected accuracy and the occasional 1-bit error, however, you might want to know more about implementation details.

The first step is multiplying the elements of the source vectors. The source mantissas are 24 bits long (23 bits plus an implied 1), so the results are 48 bits long; however, the bottom 20 bits are discarded. No rounding is done when going from 48 bits to 28 bits, so there could be almost a 1-bit error in the last place. However, 28 bits offers more precision than a regular float, so in most cases the results will be more accurate.

Before adding the results, the processor must align them. It finds the product with the largest exponent and shifts all the other results as needed. Normally, when adding three or four numbers, the result will depend on the order in which the numbers are added, but the VMX128 dot product is order invariant, because it aligns all the addends to the largest exponent.

The products are unsigned numbers with the sign stored explicitly in a sign bit. Before the addition is performed, the products must be put into twos-complement format, which may require negating some of them, if the products are not all the same sign. The processor tries to minimize the number of products it negates. Whichever sign — positive or negative — is most common will be left alone, and the products of the opposite sign are negated. In case of a tie — only possible with **vmsum4fp** — the processor negates the negative products. Minimizing the number of negations is necessary because the processor doesn't do a true negation. Instead, it complements the bits, whereas a true negation requires a complement and an addition of one. This simplified negation will introduce 1-bit errors into the least-significant bit, but the 28-bit precision of the additions should minimize the effect.

After aligning and negating, the adding is done. If the sum is negative, the processor negates it and adjusts the sign bit. Again, the negation is not a true twos-complement negation. The processor shifts the sum left or right, as needed, to normalize it, truncates it to 23 bits and an implied 1, and then writes the result to all elements of the result vector.

Most special case inputs — NANs and infinities — are handled as you would expect.

If the result of the dot product overflows, the correct response is to set the result to infinity. However, the VMX128 dot product will instead generate a NAN.

Precision Guarantees

Floating-point math that involves multiple additions might yield large percentage errors compared to the mathematically correct result, because most or all of the significant digits might cancel out. Therefore, it is usually more useful to compare the maximum error relative to the largest number added. For a four-element dot product implemented with a **vmsum4fp**, the maximum error is approximately one part in 2^{22} compared to the largest product. The maximum error for the VMX128 dot-product instructions is one part in 2^{23} compared to the largest product.

Unexpected Behavior

Ideally, **vmsum4fp ((1, 1, 1, 1), (1, -1, 1, -1))** should produce a zero, but because of simplifications in the negation logic it will produce a result of 2^{-28}, or about 3.7e-9. This is the most common type of case where these one-bit errors will be noticed. However, this error still falls within the guaranteed accuracy of one part in 2^{23} compared to the largest product. Similar errors can happen when using fused multiply-add operations to implement dot product.

Example

```
// Calculate the length squared of a vector.
__vector4 result = __vmsum3fp ( vector, vector );
```

Summary

Getting maximum performance from the Xbox 360 console requires using VMX128. The new VMX128 instructions are some of the most powerful instructions for game programmers. Understanding them will help you write more efficient code.