VMX128 Instruction Set Summary

by Bruce Dawson
Software Design Engineer
Advanced Technology Group (ATG)

Published: October 10, 2008

VMX is a powerful vector processing instruction set for operating on 16-byte registers containing floating-point or integer data. To improve VMX for game programming, some changes were made for Xbox 360 processors. The Xbox 360 version of VMX is called VMX128. Writing VMX128 code takes some effort, but can lead to significant performance improvements that can push your game to new levels of fidelity, realism, and fun.

Changes for VMX128 include:

- Adding 96 registers (for a total of 128) to allow greater loop unrolling and less spilling of variables to memory.
- Adjusting most instructions to allow access to the extra registers.
- Adding new instructions:
  - **vmulfp128** – vector multiply
  - **vpermwi128** – permute word immediate
  - **vrlimi128** – rotate left and mask insert
  - **lvlx128** and **lvrx128** – load vector left and right
  - **stvlx128** and **stvrx128** – store vector left and right
  - **vpkd3d128** and **vupkd3d128** – D3D pack and unpack
  - **vmsum3fp128** and **vmsum4fp128** – dot product
- Removing some instructions; primarily the vector integer multiply instructions.

This white paper summarizes all of the VMX128 instructions by category to make it easier to understand the capabilities of VMX128. For more detailed information about the instructions, refer to the XDK documentation. For instructions that are new to VMX128, see New Instructions in VMX128, a white paper.

This white paper has the following sections:

VMX Register Overview

In order to understand the VMX128 instructions, it is helpful to have a clear understanding of 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, is stored in the most significant bits of the register, and is stored in the lowest address in memory. VMX registers can be interpreted as words, halfwords, bytes, or bits.

Intrinsics

The recommended way to use VMX128 is with compiler intrinsics. This lets the compiler handle register allocation, instruction scheduling, and integrating VMX code with global variables, classes, and other C/C++ constructs. See PPCIntrinsics.h and VectorIntrinsics.h for a complete list of intrinsics. The **__vector4** type is used in these intrinsics to represent a VMX128 register.

Although it also is possible to use VMX128 by writing inline assembly language, it generally is not recommended as the extra complication is rarely justified. In most cases, if the compiler is not generating efficient code—perhaps causing excessive load-hit-stores—then this can be addressed by changing the C/C++ source code rather than dropping in to assembly language. Because the compiler knows the latencies of all of the VMX128 instructions, it is difficult to improve on its instruction scheduling.

For example, assume we want to do a vector-multiply-add and then to choose, for each result, between that result and the original parameter based on a mask. With intrinsics we write:

```
__vector4 IntrinsicsTest( __vector4 mask, __vector4 a,
                          __vector4 b, __vector4 c )
{
    return __vsel( a, __vmaddfp( a, b, c ), mask );
}
```

This generates the following optimal code:

```
vmaddfp      vr0,vr2,vr3,vr4
vsel         vr1,vr2,vr0,vr1
blr
```

Note that the intrinsic for an instruction is just the assembly language mnemonic with a double underscore prefix.

The return value of an intrinsic represents the destination register of the assembly language instruction. Each intrinsic expands to one instruction, plus extra instructions when needed to shuffle registers or to load referenced data from memory. For instance, if a global variable is used as input to an intrinsic, then the compiler typically generates three instructions to load the value if it was not previously loaded. If we change the above code to make "a" a global variable, then the compiler automatically generates code to load it. The compiler recognizes that "a" cannot be changed in this function so it loads it only once, even though it is used twice. This code:

```
__vector4 a;
__vector4 IntrinsicsTest( __vector4 mask, __vector4 b,
                          __vector4 c )
{
    return __vsel( a, __vmaddfp( a, b, c ), mask );
}
```

Generates the following optimal code:

```
lis         r11,a
addi        r10,r11,a
lvx128      vr0,r0,r10
vmaddfp     vr13,vr0,vr2,vr3
vsel        vr1,vr0,vr13,vr1
blr
```

Conventions

A few conventions are used throughout this white paper to simplify the information. All VMX128 intrinsics are listed, grouped by function. Intrinsics are listed instead of instructions because it is expected that this is how programmers will access VMX128 functionality, and because it simplifies the lists. In some cases, there are different instructions that the compiler chooses based on which registers are used, and using intrinsics avoids this complication.

Instructions with useful variations for different element sizes such as **__stvebx**, **__stvehx**, and **__stvewx** (store vector element byte, word, or half) are summarized as **__stve(b|h|w)x** in order to reduce redundancy.

Instructions with identical (and, thus, useless) variations (**lvxl** and **lve(b|h|w)x** are identical to **lvx** on Xbox 360) have the identical variations omitted completely.

Intrinsics that represent instructions unique to VMX128 are annotated with an * (asterisk).

The tables for instructions list the latency and the pipeline set (upper or lower) used by the instructions in the table, to indicate how expensive the instructions are and how well instructions pair. For more details about upper and lower pipeline sets, see A Detailed Examination of the Xbox 360 CPU Pipelines, a white paper.

Load/Store Intrinsics

Table 1. Load intrinsics, upper pipeline set, 2-cycle latency

Instruction	Description
__lvx	Load Vector Indexed – load 16-byte aligned vector
__lvlx*	Load Vector Left Indexed – load left part of arbitrarily aligned vector
__lvrx*	Load Vector Right Indexed – load right part of arbitrarily aligned vector
__lvsl	Load Vector for Shift Left – creates a permute vector – rarely used
__lvsr	Load Vector for Shift Right – creates a permute vector – rarely used

The workhorse intrinsic VMX load intrinsic is **__lvx**, which always loads 16-byte aligned data, ignoring the bottom four address bits. This instruction, however, is generated automatically any time vector data that is in memory is referenced, so in most cases there is no need to explicitly use **__lvx**. Allow the compiler to worry about generating **lvx** instructions like this:

```
__vector4 AddTwoValues( const __vector4* pInput)
{
    return pInput[ 0 ] + gVector;
}
```

The other two most common VMX load intrinsics are **__lvlx** and **__lvrx**. The **__lvlx** intrinsic loads from the specified address up to the next 16-byte boundary, loading from 1 byte to 16 bytes and left aligning them. The intrinsic **__lvrx** loads from the previous 16-byte boundary up to the specified address, loading from 0 bytes to 15 bytes and right aligning them. These intrinsics can be used together as shown in the **__loadunalignedvector** define in order to load an arbitrarily aligned **__vector4**.

```
__vor( __lvlx( ptr, 0 ), __lvrx( ptr, 16 ) )
```

The **__lvlx** intrinsic also is quite useful for loading scalar values into a vector register. If you use **__lvlx(&s, 0)** on a naturally aligned scalar value (char, short, int, or float) it loads it into the most significant bits of the vector register—the X component for 32-bit values.

Intrinsics **__lvsl** and **__lvsr** are used to create a permute vector that can be used with a pair of **__lvx** loads and a permute instruction to load misaligned data, but with VMX128 this job usually is done instead with **__lvlx** and **__lvrx**.

Table 2. Store intrinsics, upper pipeline set, latency not applicable

Instruction	Description
__stvx	Store Vector Indexed – store vector to 16-byte aligned address
__stve(b|h|w)x	Store Vector Element Byte|Halfword|Word Indexed
__stvlx*	Store Vector Left Indexed – store left part of arbitrarily aligned vector
__stvrx*	Store Vector Right Indexed – store right part of arbitrarily aligned vector

The **__stvx** intrinsic is the most commonly used VMX store intrinsic. This instruction, however, is generated automatically by the compiler and, thus, rarely needs to be used explicitly. As with **__lvx,** this instruction ignores the bottom four address bits.

The **__stve(b|h|w)x** intrinsics are useful for writing out scalar or other partial vector data. They store whichever element is lined up with the destination address, so the element to be stored usually needs to be splatted to all elements before storing.

The **__stvlx** and **__stvrx** intrinsics store up to and from a 16-byte boundary, respectively, and are the reverse of **__lvlx** and **__lvrx**. They are used by the **__storeunalignedvector** macro to store to an arbitrarily aligned 16-byte destination.

The load and store instructions all have versions with **_volatile** appended to the intrinsic name. These variations exist to support volatile pointers and map to the same underlying instructions.

Because VMX loads and stores always use the indexed addressing mode, and because PowerPC requires two instructions to generate an arbitrary 32-bit address, loading and storing of VMX data can use a significant number of instructions. Be sure to look at the code generated to see when the compiler is generating unexpected loads and stores.

Floating-Point Math Intrinsics

Table 3. Floating-point math intrinsics, lower pipeline set, 12-cycle to 14-cycle latency

Instruction	Description
__vaddfp	Vector Add Floating Point
__vmaddfp	Vector Multiply-Add Floating Point
__vmulfp*	Vector Multiply Floating Point
__vnmsubfp	Vector Negative Multiply-Subtract Floating Point
__vsubfp	Vector Subtract Floating Point
__vmsum3fp*/__vdot3fp	Vector Multiply-Sum 3-way Floating Point

Instruction	Description
__vmsum4fp*/__vdot4fp	Vector Multiply-Sum 4-way Floating Point
__vexptefp	Vector 2 Raised to the Exponent Estimate Floating Point
__vlogefp	Vector Log Base 2 Estimate Floating Point
__vrefp	Vector Reciprocal Estimate Floating Point
__vrsqrtefp	Vector Reciprocal Square Root Estimate Floating Point

Table 4. Floating-point math intrinsics, lower pipeline set, 4-cycle latency

Instruction	Description
__vmaxfp	Vector Maximum Floating Point
__vminfp	Vector Minimum Floating Point

All floating-point intrinsics work on single-precision floating-point numbers. Double precision is not supported.

Because the latencies of all of these instructions are nearly identical, it is much better to do a **__vmaddfp** rather than a **__vmulfp** followed by a **__vaddfp**. However, because there is no rounding after the multiply in a **__vmaddfp** instruction, the results are different from doing a separate multiply and add.

Similarly to how the compiler automatically generates vector load and store instructions, it also automatically generates vector multiply, add, and multiply-add instructions when using regular multiplication and addition operators on **__vector4** types.

```
__vector4 OperatorTest1(__vector4 a, __vector4 b)
{
    // This generates vaddfp
    return a + b;
}

__vector4 OperatorTest2(__vector4 a, __vector4 b, __vector4 c)
{
    // This generates vmaddfp
    return a * b + c;
}
```

The automatic generation of **__vmaddfp** from **__vmulfp** and **__vmaddfp** can be controlled with **#pragma fp_contract**.

The dot-product instructions, in conjunction with reciprocal-square-root estimate, are particularly useful for normalizing vectors, calculating vector lengths, matrix multiplication, and other common tasks. The algorithm used for the dot product instructions is optimized for speed rather than accuracy; for details, see New Instructions in VMX128.

The estimate instructions give 6 bits to 12 bits of precision. For greater accuracy, they can be followed with one or more Newton-Raphson refinement steps. For

instance, **__vrsqrtefp** gives a ~12-bit accurate estimate of the reciprocal square roots of the four floats in a **__vector4**. If greater accuracy is needed then this estimate can be refined with the function below, which gives a ~24-bit accurate estimate. Proper handling of special values such as zero, infinity, negative numbers, and NANs (not a numbers) requires extra code, but for many purposes (such as normalizing vectors) these extra checks are not needed.

```
__vector4 vHalf = { 0.5, 0.5, 0.5, 0.5 };

__vector4 vrsqrt24( __vector4 v )
{
    __vector4 vResult = __vrsqrtefp( v );

    // Now refine the estimate with Newton's method
    __vector4 vHalfV = __vmulfp( v, vHalf );
    __vector4 vRecip = __vmulfp( vResult, vResult );
    __vector4 vScale = __vnmsubfp( vHalfV, vRecip, vHalf );
    vResult = __vmaddfp( vResult, vScale, vResult );

    // Refinement and multiplication by v means that
    // zero and infinity aren't handled - they cause
    // zero by infinity multiplication, and a NAN result.

    return vResult;
}
```

Integer Intrinsics

Table 5. Integer intrinsics, lower pipeline set, 4-cycle latency

Instruction	Description
__vaddcuw	Vector Add and Write Carry-Out Unsigned Word
__vadds(b|h|w)s	Vector Add Signed Byte|Halfword|Word Saturate
__vaddu(b|h|w)m	Vector Add Unsigned Byte|Halfword|Word Modulo
__vaddu(b|h|w)s	Vector Add Unsigned Byte|Halfword|Word Saturate
__vavgs(b|h|w)	Vector Average Signed Byte|Halfword|Word
__vavgu(b|h|w)	Vector Average Unsigned Byte|Halfword|Word
__vsubcuw	Vector Subtract and Write Carry-Out Unsigned Word
__vsubs(b|h|w)s	Vector Subtract Signed Byte|Halfword|Word Saturate
__vsubu(b|h|w)m	Vector Subtract Unsigned Byte|Halfword|Word Modulo
__vsubu(b|h|w)s	Vector Subtract Unsigned Byte|Halfword|Word Saturate
__vmaxs(b|h|w)	Vector Maximum Signed Byte|Halfword|Word
__vmaxu(b|h|w)	Vector Maximum Unsigned Byte|Halfword|Word
__vmins(b|h|w)	Vector Minimum Signed Byte|Halfword|Word
__vminu(b|h|w)	Vector Minimum Unsigned Byte|Halfword|Word

VMX128 can do integer operations: add, average, subtract, maximum, and minimum on signed and unsigned values of various sizes, with both saturated and modulo

math. There is no performance penalty for switching between integer and floating-point math.

The integer instructions can reference only the first 32 VMX registers. The compiler schedules around this, but this imposes additional code-generation restrictions when they are used and may force the compiler to shuffle data between registers.

Compare Intrinsics

Table 6. Compare intrinsics, lower pipeline set, 4-cycle latency

Instruction	Description
__vcmpbfp(R)	Vector Compare Bounds Floating Point
__vcmpeqfp(R)	Vector Compare Equal To Floating Point
__vcmpgefp(R)	Vector Compare Greater Than or Equal To Floating Point
__vcmpgtfp(R)	Vector Compare Greater Than Floating Point
__vcmpequ(b|h|w)(R)	Vector Compare Equal To Unsigned Byte|Halfword|Word
__vcmpgts(b|h|w)(R)	Vector Compare Greater Than Signed Byte|Halfword|Word
__vcmpgtu(b|h|w)(R)	Vector Compare Greater Than Unsigned Byte|Halfword|Word

The compare intrinsics create a bit mask to indicate which components passed the test, and which components failed. Whenever possible this bit mask should be used to select between different results, rather than branching based on the result. The different intrinsics are for floating point compares and integer compares of various precisions, both signed and unsigned.

If there is a function that does different calculations based on the input value, it is often more efficient to do both calculations and then to select the result based on a compare. If the calculations are inlined, then this avoids expensive branches and allows the two calculations to be overlapped. The **__vsel** intrinsic selects the appropriate results on a per-component basis.

```
// if (v < 0) return NegCalc(v);
// else return PosCalc(v);
// Select result per component.
__vector4 PosNegCalc(__vector4 v)
{
    // Calculate both possible values – these functions are
    // assumed to be inlined.
    __vector4 vNeg = NegCalc(v);
    __vector4 vPos = PosCalc(v);
    // Mask will be set for components where
    // v is greater than zero.
    __vector4 vGreaterMask = __vcmpgefp(v, __vzero());
    __vector4 vResult = __vsel(vNeg, vPos, vGreaterMask);
    return vResult;
}
```

When branching on the results is needed, the recording form (indicated with an "R" at the end of the intrinsic) can be used to put a two-bit result code into an unsigned int. If the bit at 0x80 is set, it means all components passed the test. If the bit at 0x20 is set, it means that none of the components passed the test. Be aware that branching on vector compares generally causes a pipeline flush, similar to branching on scalar floating-point compares. Sample code might look something like this:

```
float CosOrSin( __vector4 v, float f )
{
    unsigned int cr;
    __vcmpgefpR( v, __vzero(), &cr );
    if ( cr & 0x80 )
        return sin( f );
    else
        return cos( f );
}
```

When applicable, compare and select should be preferred over branching, and the **__vmaxfp** and **__vminfp** intrinsics should be preferred over using compare and select.

Bitwise Intrinsics

Table 7. Bitwise intrinsics, lower pipeline set, 4-cycle latency

Instruction	Description
__vand	Vector Logical AND
__vandc	Vector Logical AND with Complement
__vnor	Vector Logical NOR
__vor	Vector Logical OR
__vsel	Vector Select
__vxor	Vector Logical XOR

Bitwise operations can be performed on vectors containing integer or floating-point data and can be used to combine vectors based on compare results. This usually is done with **__vsel**, but can sometimes be done with **__vand** or other intrinsics. Bitwise operations also can be used to modify the sign bit of floating-point numbers.

The compiler automatically generates an appropriate constant and a **__vxor** for unary negate of floating-point vectors.

The compiler uses **__vor** when it needs to move data from one register to another.

Operator overloading can be used to access some of the bitwise intrinsics in a more readable way, as the following code demonstrates.

```
// Implement an xor operator
__vector4 operator^( __vector4 a, __vector4 b )
{
    return __vxor(a, b);
}

// Use the newly defined xor operator.
__vector4 OperatorTest3( __vector4 a, __vector4 b )
{
    return a ^ b;
}
```

While **__vxor** can be used to create a zero, there are better ways to accomplish this, as demonstrated in the **__vzero** pseudo-intrinsic.

Because **__vsel** has three vector inputs, there are some register restrictions on this instruction. The compiler automatically takes care of these restrictions, but it does mean that **__vsel** can be slightly more expensive than it might seem, so be wary of excessive usage, and watch for code-generation inefficiencies.

Shift Intrinsics

Table 8. Shift intrinsics, mixed pipeline sets, 4-cycle latency

Instruction	Pipeline Set	Description
__vrl(b|h|w)	Lower	Vector Rotate Left Byte|Halfword|Word – rotate elements left
__vsl(b|h|w)	Lower	Vector Shift Left Byte|Halfword|Word – shift elements left
__vsr(b|h|w)	Lower	Vector Shift Right Byte|Halfword|Word – shift elements right
__vsra(b|h|w)	Lower	Vector Shift Right Algebraic Byte|Halfword|Word – shift elements right algebraic
__vsl	Lower	Vector Shift Left-shift vector left 0-7 bits – shift entire vector left
__vsr	Lower	Vector Shift Right-shift vector left 0-7 bits – shift entire vector right
__vslo	*Upper*	Vector Shift Left Octet – shift entire vector left by a multiple of eight
__vsro	*Upper*	Vector Shift Right Octet – shift entire vector right by a multiple of eight
__vsldoi	*Upper*	Vector Shift Left Double Octet Immediate – shift vector pair left by a multiple of eight
__vrlimi*	Lower	Vector Rotate Left Immediate & Mask Insert – word-based rotate and insert

The first four intrinsics shift the individual elements separately; bits do not move from one element to the next. The next four intrinsics shift the entire register, with

__vsl and **__vsr** shifting short distances, and **__vslo** and **__vsro** shifting long distances.

Normally, the shift amount is encoded in another register, which requires an extra step to load it. Both **__vsldoi** and **__vrlimi**, however, take immediate shift amounts. **__vrlimi** is particularly useful as it allows a shift while simultaneously inserting words into another register. This can be used together with **__lvlx** to insert a scalar value from memory into any element of a register, as demonstrated in **__loadunalignedvectorelement**.

Permute/Splat/Immediate Load Intrinsics

Table 9. Permute/splat intrinsics, upper pipeline set, 4-cycle latency

Instruction	Description
__vmrgh(b|h|w)	Vector Merge High Byte|Halfword|Word – merge two vectors
__vmrgl(b|h|w)	Vector Merge Low Byte|Halfword|Word – merge two vectors
__vperm	Vector Permute – permute two vectors, controlled by another
__vsplt(b|h|w)	Vector Splat Byte|Halfword|Word – splat one element everywhere
__vpermwi*	Vector Permute Word Immediate – permute a single vector

The first three permute instructions allow combining of two registers in various ways. The **__vsplt** intrinsic copies one element of a vector to all elements of the destination. The **__vpermwi** intrinsic allows an arbitrary word-based permuting of a VMX128 value and, thus, can do anything that **__vsplt** can do, just not as readably. The **VPERMWI_CONST** macro can be used to generate the permute control byte for **__vpermwi**.

The **__vpermwi** intrinsic can be used as an alternate way of moving data from one register to another, if an upper pipeline set instruction is wanted instead of the default **__vor**.

The most flexible of these instructions is **__vperm**, as it allows fully general byte-wise permutation of two source vectors. This flexibility, however, comes at a cost as a permute control vector must be loaded or created, and the large number of input registers forces register restrictions to allow instruction encoding. Therefore, whenever possible, it is preferable to use alternate instructions such as immediate shift, permute, or splat instructions.

Table 10. Immediate load intrinsic, upper pipeline set, 4-cycle latency

Instruction	Description
__vspltis(b|h|w)	Vector Splat Immediate Signed Byte|Halfword|Word

The **__vspltis** intrinsic allows loading immediate byte, half, or word values into all elements of a VMX128 variable. This is the most efficient way to generate a zero,

and various other values. With various combinations of **__vspltis**, convert, unpack, and bitwise operations, it is possible to generate many useful constants.

Pack and Unpack Intrinsics

Table 11. Pack intrinsics, upper pipeline set, 4-cycle latency

Instruction	Description
__vpkd3d*	Vector Pack D3Dtype, Rotate Left Immediate & Mask Insert
__vpkpx	Vector Pack Pixel – pack eight 32-bit pixels into 16-bit pixels
__vpks(h|w)ss	Vector Pack Signed Halfword|Word Signed Saturate
__vpks(h|w)us	Vector Pack Signed Halfword|Word Unsigned Saturate
__vpku(h|w)um	Vector Pack Unsigned Halfword|Word Unsigned Modulo
__vpku(h|w)us	Vector Pack Unsigned Halfword|Word Unsigned Saturate

Table 12. Unpack intrinsics, upper pipeline set, 4-cycle latency

Instruction	Description
__vupkd3d*	Vector Unpack D3Dtype
__vupkhpx	Vector Unpack High Pixel
__vupkhs(b|h)	Vector Unpack High Signed Byte|Halfword
__vupklpx	Vector Unpack Low Pixel
__vupkls(b|h)	Vector Unpack Low Signed Byte|Halfword

The pack and unpack intrinsics often are used when loading data from memory, operating on it, and then storing it back into memory. This helps to reduce memory and memory-bandwidth usage. The d3d pack and unpack instructions are designed specifically to work with the Xbox 360 GPU and support conversions between float and float-16, and others. For more information about **__vpkd3d** and **__vupkd3d**, see New Instructions in VMX128.

Convert and Miscellaneous

Table 4. Convert intrinsics, lower pipeline set, 12-cycle latency

Instruction	Description
__vcfpsxws/__vctsxs	Vector Convert from Floating Point to Signed Fixed-point Word Saturate
__vcfpuxws/__vctuxs	Vector Convert from Floating Point to Unsigned Fixed-point Word Saturate
__vcsxwfp/__vcfsx	Vector Convert from Signed Fixed-point Word to Floating Point
__vcuxwfp/__vcfux	Vector Convert from Unsigned Fixed-point Word to Floating Point

Instruction	Description
__vrfim	Vector Round to Floating-Point Integer toward Minus infinity
__vrfin	Vector Round to Floating-Point Integer to Nearest
__vrfip	Vector Round to Floating-Point Integer toward Plus infinity
__vrfiz	Vector Round to Floating-Point Integer toward Zero

VMX128 supports a range of instructions for converting between integers—signed and unsigned—and floating-point values. The convert instructions allow specifying a scaling factor, a value to be added or subtracted from the exponent.

VMX128 also supports four different rounding instructions. These instructions round a floating-point number to an integral value, but the result is still in floating-point format.

Table 5. Miscellaneous intrinsics, lower pipeline set, 4-cycle latency

Instruction	Description
__mfvscr	Move From Vector Status and Control Register
__mtvscr	Move To Vector Status and Control Register (flushes pipelines)

Moving to and from the vector status and control register should not be needed in normal VMX code and these instructions are listed purely for completeness.

VMX128 Tips and Tricks

When porting scalar code to VMX128 it is often possible to get a four-times speedup —even more if the algorithm maps well to the capabilities of VMX128. If a port to VMX128 gives less of a speedup than expected, then the problem usually is a mixing of scalar and vector math. Many math libraries "helpfully" convert some vector results, such as dot products, from **__vector4** to float. Unfortunately, scalar float values are stored in different registers, forcing the compiler to store the result to memory so that the result can be loaded into a scalar floating-point register for further math. Frequently moving data back and forth between scalar floating-point and vector registers is the biggest source of VMX128 slowdowns, and avoiding it, by implementing related scalar operations in VMX128, is the most important performance trick. The **__vector4** type has x, y, z, and w components to allow scalar float access, and contains an array of floats and an array of integers to allow many ways of referencing the components of the **__vector4**. However, when these elements are referenced individually, the compiler is forced to generate scalar code, which leads to load-hit-stores and other inefficiencies. For this reason, never reference the internal components of a **__vector4** in time-critical code.

A related recommendation is that it is important to give the VMX unit a large enough piece of work. Writing tiny functions that do a little bit of VMX may give modest speedups, but writing an entire streaming algorithm in VMX can give much bigger wins. Good VMX code processes many items in order to hide initialization costs, and tries to process several items in parallel in order to hide instruction latency. VMX128

code can easily be dominated by loading and storing values. Every load and store uses from one to three instructions, so reducing these to a bare minimum can be important. Some constants can easily be created using combinations of **__vspltis(b|h|w)** and various other instructions. When that is impractical, it is worth combining multiple constants into one vector in memory, and using **__vspltw** and other inexpensive operations to generate the needed constants, as in this example:

```
void SpltVectorInit()
{
    static const __vector4 cData = { 1.0f, 2.0f, 0.5f, -0.0f };

    const __vector4 vOne = __vspltw( cData, 0 );
    const __vector4 vTwo = __vspltw( cData, 1 );
    const __vector4 vHalf = __vspltw( cData, 2 );
    const __vector4 vHalfAndTwo = __vpermwi( cData,
                VPERMWI_CONST( 1, 2, 1, 2) );
    const __vector4 vSignMask = __vspltw( cData, 3 );
    const __vector4 vNegTwo = __vxor( vTwo, vSignMask );

    DoStuff( vOne, vTwo, vHalf, vHalfAndTwo, vNegTwo );
}
```

In order to initialize integer VMX constants—sometimes needed for masks or permute constants—you can use the **__vector4i** data type. For example:

```
static const __vector4i g_vI = { 1, 2, 0xFFFFFFFF, 3 };
```

The **__vector4i** type is just a typedef for a 16-byte aligned array of 32-bit unsigned integers. To use this data type with the VMX128 intrinsics, cast it to a **__vector4** pointer and dereference it, like this:

```
__vector4 ApplyOddMask( __vector4 v)
{
    return __vand( v, *(__vector4*)g_vI );
}
```

When loading multiple pieces of data from a structure, the best technique often is to keep the structure 16-byte aligned, to load all of the data into a few VMX registers, and then to adjust the data after loading based on the structure's actual layout. This may include decompressing the data from, for example, 16-bit floats to 32-bit floats, as well as breaking the loaded data apart into two and three component parts using **__vsldoi** and other shift and permute instructions. This lets you trade off memory bandwidth and data size against the number of instructions executed.

Because VMX allows integer, bitwise, and floating-point operations on the same registers, there are tricks that can be performed by using knowledge of the IEEE floating-point math format. For instance, given a sign-bit mask it is possible to implement negate and absolute value with **__vxor** and **__vandc**. In fact, this technique is required since no instructions are supplied for negate and absolute value, and the compiler automatically implements unary floating-point negate.

It also is possible to implement floating-point multiplication and division through integer addition and subtraction from the floating-point exponent. This trick,

however, should be used with extreme care since overflow and underflow leads to quite unexpected results.

When writing performance critical VMX code, consider using the **/QXSTALLS** compiler option to analyze your code's performance. This useful technique can help you find performance problems and then address them, either by avoiding load-hit-stores or by reducing dependencies in order to allow better scheduling.

Math Library

In addition to using VMX through intrinsics, it is possible to use VMX through the Xbox 360 math library. This is a header only library of functions for dealing with quaternions, matrices, vectors, and more. Using the Xbox 360 math library allows more readable code to be written—**XMConvertVectorFloatToUInt** instead of **__vcfpuxws**—and provides access to pre-written functions like **XMVectorCos**. The Xbox 360 math library also serves as a demonstration of best practices for VMX128 and can be used as a source of example code to be incorporated into your own math library.

Summary

Getting maximum performance from the Xbox 360 console requires using VMX128, which requires knowing what instructions are available and how they can be combined. This white paper provided an introduction to many of the instructions that can be used to pump up the processor. Learning these basic instructions, and learning how to effectively combine instructions, can lead to greater performance.

References

New Instructions in VMX128 (white paper)

Compiler switch **/QXSTALLS**

Xbox 360 Math Library

IBM AltiVec™ Documentation

VMX Optimization – Taking It Up A Level