Using VMX128 Instructions

⊟ Collapse All

The Xbox 360 CPU incorporates the VMX128 vector processing architecture, which is a custom enhancement of VMX (also known as AltiVec or Velocity Engine). The VMX128 technology provides a set of 128 vector registers, each of which are 128 bits wide, and a powerful set of instructions for high-performance vector math operations.

Of these three methods, the compiler intrinsics will always produce the most efficient code. This allows you to use all of the optimization features of the compiler. The compiler will allocate the use of vector registers on your behalf in order to minimize CPU stalls.

By contrast, coding directly in assembly language, using the assembler instead of the compiler, typically produces the least efficient code, the kind of code that relies entirely on the programmer to ensure optimum execution. For this reason, you should consider these instructions as informational, as a supplement to the documentation about the compiler intrinsics.

Many of the math functions defined in xboxmath.h make use of the compiler intrinsics, and the header file is a good source for examples of their use.

You can find more information about the VMX128 enhancements to the Xbox 360 CPU in the New Instructions in VMX128 white paper.

VMX vs. VMX128

VMX128 instructions support access to all 128 vector registers available on the the Xbox 360 CPU.

Unlike VMX instructions, you can access VMX128 functionality only through the inline assembler of the Visual C++ compiler, or through Xbox 360 compiler intrinsics. The stand-alone Xbox 360 assembler does not support the VMX128 instructions.

If you invoke any of the vector intrinsics specific to VMX128, the compiler generates code that emulates these instructions. This means it is possible to develop code that uses the vector intrinsics specific to Xbox 360, even though the performance gains expected from their use will not be seen until the release of beta hardware.

New instructions for VMX128

Most of the instructions supported only by VMX128 use the "128" suffix in the instruction name. Many of these are functionally redundant to their still-supported VMX counterparts. The only difference is their ability to address 128 registers instead of only 32.

You can find a list of completely new instructions for VMX128 under Listing of New VMX128 Instructions.

Unsupported VMX Instructions

A number of VMX architecture instructions have been dropped from VMX128.

These dropped instructions dealt primarily with the multiplication of integer vectors. You can find a complete list under Listing of Dropped VMX Instructions.

Loads and Stores

All load and store instructions accept a pair of general-purpose registers that are added together to produce an effective address (EA). Typically, one register provides the base address of the memory region and the other contains a byte offset.

In general, the EA must be aligned on a 16-byte (128-bit) boundary. If the EA is not so aligned, the EA will be masked so that the fetch or write will occur beginning at the nearest (lesser) 16-byte boundary. Memory operations on misaligned addresses require special coding. You should use the supporting instructions for this purpose.

The functions lvebx, lvehx, and lvewx/lvewx128, for example, load their byte/half-word/word values into the position of the vector register that aligns with its 16-byte boundary in memory. Officially, the remaining bytes are "undefined." In practice, however, it loads the surrounding bytes from memory. This makes these instructions effectively the same as the lvx operation.

You can use the lvsl and lvsr instructions, which construct a permute control vector to help complete the load, to access the loads from addresses not aligned on 16-byte boundaries. See those instruction descriptions for examples.

The "last" instruction forms (such as lvxl) provide the CPU with information that should improve the functioning of the cache. On Xbox 360, however, this is of no value, and there is no need to use these forms.

Latency for all load instructions is two cycles. For store instructions, latency is determined by external factors such as memory latency and cache hit. Thus, no figure is given.

Dot-Product Instructions

The VMX128 instructions vmsum3fp128 and vmsum4fp128 perform a dot-product operation ($V_1 \bullet V_2$) on 3-element and 4-element vector pairs, respectively.

These instructions are comparable to the mulps/haddps/haddps operations on the PC.

- Dot-Product Performance
- Dot-Product Precision
- Dot-Product Uses

Dot-Product Performance

The vector dot-product instructions on Xbox 360 have a high latency (14 cycles), but they are still considerably faster than other methods. For example, dot product vertex transforms typically are faster than transforms that use vmaddfp. Back-to-back dot product operations can occur in parallel as long as no registers are in common, whereas a vmaddfp transform must also use **vmadd**s that are dependent on the same register.

As a point of comparison, the equivalent operation on the PC has a 33-cycle latency, with a throughput of one every eight cycles.

Compiler intrinsics provide these dot-product operations. When compiled for alpha development consoles, the compiler provides code that emulates the instructions. This means results and performance will not be identical to that of the final hardware.

Dot-Product Precision

The dot-product instructions operate in 28-bit intermediate workspace. They give you more consistent and precise results than they would if the same operation were performed with separate multiply and sum operations.

The greater intermediate precision also allows results to be order invariant, regardless of vector direction. In typical implementations, the result varies depending on the order of summing, as in the following example:

- 1 + -1 + 1e-8 = 1e-8
- 1 + 1e-8 + -1 = 0

On Xbox 360, the multiplies are performed and the 28-bit intermediate results are retained. The operation then aligns all products to the one with the largest exponent, then performs the sum. Despite this overall greater accuracy, note that simplifications in the rounding and adding will sometimes give one-bit errors compared to the expected result. The maximum error will be 1 part in 2^{23} compared to the largest product.

Also note that an overflow results in a NaN, rather than an infinity.

Dot-Product Uses

- When you multiply matrices, note that instead of using vmaddfp to produce A×B, you can multiply B×A with dot products without changing matrix format.
- You can use the dot product operation to calculate vector length squared in a single instruction.
- You can use the dot product operation with estimate instructions for fast normalizing and length estimation of vectors.

D3D Pack/Unpack

The operations vpkd3d128 and vupkd3d128 are provided to facilitate Direct3D vector packing to reduce memory footprint and memory bandwidth for data.

The operation results are shifted to the desired position, then inserted into the vector register using a mask specified by the instruction operand. In this way, you can build a vector from packed results. Another way to do it is to use stvewx to pack and store a single item.

Typically, the pack instruction, vpkd3d128, is preceded by a vmaddfp instruction to put the floating-point values in the required range.

Typically, the unpack instruction, vupkd3d128, is followed by a vmaddfp instruction to put the floating-point values in the desired range.

The following code samples (expressed in compiler intrinsics) suggest how these operations are used.

- Float16 Pack/Unpack
- D3DCOLOR Pack/Unpack

Float16 Pack/Unpack

```
__vector4 PackVector4sToFloat16s( __vector4 a, __vector4 b ) {
    // Convert four floats to four HALFs and mask insert them into 'a'
    a = __vpkd3d( a, a, VPACK_FLOAT16_4, VPACK_64LO, 0 );
    // Convert four more floats to four HALFs and mask insert them
    // into 'a', this time shifting them up.
    a = __vpkd3d( a, b, VPACK_FLOAT16_4, VPACK_64LO, 2 );
    return a;
}


void UnpackFloat16sFromVector4s( __vector4 packed, __vector4& a,
            __vector4& b ) {
    // Unpack first half of 'packed' into 'a'
    a = __vupkd3d( packed, VPACK_FLOAT16_4 );
    // Shift the other half of 'packed' down
    __vector4 packed2 = __vsldoi( packed, packed, 8 );
    // Now unpack the other half of 'packed' into 'b'
    b = __vupkd3d( packed2, VPACK_FLOAT16_4 );
}
```

D3DCOLOR Pack/Unpack

```
#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 };
void PackVector4ToD3DCOLOR( __vector4 input, DWORD* pDest )
{
    // 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 );
    result = __vpkd3d( result, result, VPACK_D3DCOLOR, VPACK_32, 0 );
    // Now we need to store the 32-bit result, using __stvewx. We
    // have to splat the result from element 3 to all four words so
    // that the correct value will be stored regardless of alignment.
    result = __vspltw( result, 3 );
    // Store the result. pDest must be DWORD aligned.
    __stvewx( result, pDest, 0 );
}


#define UnpackMul8U  ((1 << 23) / 255.0)
static const __vector4 vUnpackMul8888 = { UnpackMul8U,
                UnpackMul8U, UnpackMul8U, UnpackMul8U };
static const __vector4 vUnpackAdd8888 = { -UnpackMul8U,
                -UnpackMul8U, -UnpackMul8U, -UnpackMul8U };
__vector4 UnpackVector4FromD3DCOLOR( const DWORD* pSrc ) {
    // Load the left portion of the vector. If pSrc is DWORD
    // aligned this will load all the data we need to the input.x.
    __vector4 input = __lvlx( pSrc, 0 );
    // __lvlx loads to input.x, but __vupkd3d looks in input.w
    input = __vspltw( input, 0 );
    // Unpack the data into 1.0 + x*(2^-23) format
    __vector4 result = __vupkd3d( input, VPACK_D3DCOLOR );
    // Restore the data to 0.0 to 1.0 range.
    __vector4 normalResult = __vmaddfp( result,
                vUnpackMul8888, vUnpackAdd8888 );
    return normalResult; }
```

Notes on Reference Documentation

1. Most of the individual pages that describe the VMX128 instructions provide a pseudocode example of the operation. This example is meant to quickly convey the effect of the instruction. All pages have a text description of the operation performed. Where pseudocode and text appear to disagree, the text should be assumed to be definitive.
2. The pseudocode uses the symbol '::' to express the concatenation of two 128-bit registers into a 256-bit sequence (always an interim value).
3. Each page provides latency information for each instruction that includes the processing pipeline used and the number of cycles.