New “Bulldozer” and “Piledriver” Instructions

A step forward for high performance software development

Brent Hollingsworth
Software Program Manager
Advanced Micro Devices, Inc.
October 2012

"Bulldozer" and "Piledriver" Software Instructions

In 2011 AMD made a significant step for high performance software development with the release of the AMD FX™, AMD Opteron™ 6200 and AMD Opteron™ 4200 processors. Powered by the "Bulldozer" core, these processors added support for a large number of instruction sets including SSE4.1/4.2, AES, CLMUL and AVX. They also introduced two new instruction sets - FMA4 and XOP.

AMD is now taking another step forward with new processors powered by the "Piledriver" core. These processors add support for an additional four instruction sets. Support for the new "Bulldozer" and "Piledriver" instructions is shown below.

		"Bulldozer"	"Piledriver"
FMA	Fused Multiply Accumulate	FMA3	FMA3, FMA4
XOP	eXtended Operations	✓	✓
CVT16	Convert 16		✓
BMI	Bit Manipulation Instructions		✓
TBM	Trailing Bit Manipulation		✓

This article will provide a brief introduction to the new instructions. Additional information about each instruction can be obtained through your compiler vendor and the manuals listed in the Referernces section at the end.

Intrinsics

When available, the examples and documentation below favor the C function wrappers provided by compiler vendors called *intrinsics* over assembly. Intrinsic functions are available in most popular compilers including GCC and Visual Studio. To use the newest instructions, you may need to update your compiler. The FMA4 instructions for example are first available in Visual Studio in Visual Studio 10 SP1. The examples below were developed on Visual Studio 11 beta.

XMM / YMM Registers

While the BMI and TBM instructions operate on general purpose registers, the other new instructions operate on 128-bit XMM and 256-bit YMM registers. These registers are most often subdivided into packed data fields representing 8, 16, 32 and 64-bit data. Vector operations process these multiple values in parallel. Scalar operations perform calculations only on the value in the lowest order position. The figure below represents a 128 bit register with eight 16-bit values.

A	B	C	D	E	F	G	H

Fused Multiply Accumulate

Fused Multiply Accumulate instructions help provide performance and accuracy improvements for multiply-add tasks commonly performed in scientific computing. Hardware support for these operations was first added by AMD with the FMA4 instructions in the "Bulldozer" core. Future AMD processors will add support for FMA3 instructions (see FMA3/FMA4 below).

Multiply Accumulate

Multiply-accumulate operations are calculations of the form

```
d = a + (b * c)
```

For example the dot product of vectors ***odd*** = [1, 3, 5] and ***even*** = [2, 4, 6] is calculated as

```
odd·even =1*2 + 3*4 + 5*6 = 44
```

This would traditionally be computed in five steps

```
X = 1 * 2
Y = 3 * 4
Z = 5 * 6
X = X + Y
X = X + Z
```

Multiply accumulate instructions condense the execution to three steps

```
X = 0 + (1 * 2)
X = X + (3 * 4)
X = X + (5 * 6)
```

Multiply Accumulate vs. Fused Multiply Accumulate

In addition to reducing the number of operations, FMA instructions can also help improve precision. In the first execution stream above, the hardware floating point unit would perform rounding on each computation individually.

In a *fused* multiply accumulate instruction; the result is rounded only after completion of both the multiplication and addition. FMA operations on AMD hardware are accurate within ½ bit in the least significant bit.

FMA3/ FMA4

FMA3 and FMA4 differ only in their usage of memory registers. In the multiply-accumulate operation

d = a + (b * c)

FMA3 requires that the target ***d*** be either ***a***, ***b***, or ***c***. FMA4 allows the target to be a fourth register.

Usage

Support for FMA4 is indicated by the value in bit 16 in ECX when calling CPUID function 0x8000_0001.

Code Example – Dot Product

The code below calculates the dot product of ***odd*** = [1, 3, 5] and ***even*** = [2, 4, 6] as shown in the example above. It calls the _mm_macc_ss intrinsic which takes three arguments [src1, src2, src3] and returns src3 + (src1 * src2). It is a scalar operation, processing only the lowest order 32-bit value in the 128-bit register.

```
#include <iostream>
#include <intrin.h>

int main(){
   __m128 odd [3];                 // Odd values
   odd[0].m128_f32[0] = 1;         // Store data in the low words
   odd[1].m128_f32[0] = 3;         // for scalar calculation
   odd[2].m128_f32[0] = 5;

   __m128 even[3];                 // Even values
   even[0].m128_f32[0] = 2;        // Store data in the low words
   even[1].m128_f32[0] = 4;        // for scalar calculation
   even[2].m128_f32[0] = 6;

   __m128 result = _mm_setzero_ps();
   result = _mm_macc_ss( odd[0], even[0], result );
   result = _mm_macc_ss( odd[1], even[1], result );
   result = _mm_macc_ss( odd[2], even[2], result );

   std::cout << "Result: " << result.m128_f32[0] << std::endl;
   return 0;
}
```

FMA4 Instructions

The family of FMA4 intrinsic functions is provided below. The first table contains functions which operate on 128-bit XMM registers.

128-bit operations

	Vector		Scalar	
	Double	**Single**	**Double**	**Single**
Multiply Add	_mm_macc_pd	_mm_macc_ps	_mm_macc_sd	_mm_macc_ss
Multiply Add Subtract	_mm_maddsub_pd	_mm_maddsub_ps		
Multiply Subtract	_mm_msub_pd	_mm_msub_ps	_mm_msub_sd	_mm_msub_ss
Multiply Subtract Add	_mm_msubadd_pd	_mm_msubadd_ps		
Negative Multiply Add	_mm_nmacc_pd	_mm_nmacc_ps	_mm_nmacc_sd	_mm_nmacc_ss
Negative Multiply Subtract	_mm_nmsub_pd	_mm_nmsub_ps	_mm_nmsub_sd	_mm_nmsub_ss

The second table lists functions which operate on 256-bit YMM registers.

256-bit Operations

	Vector	
	Double	**Single**
Multiply Add	_mm256_macc_pd	_mm256_macc_ps
Multiply Add Subtract	_mm256_maddsub_pd	_mm256_maddsub_ps
Multiply Subtract	_mm256_msub_pd	_mm256_msub_ps
Multiply Subtract Add	_mm256_msubadd_pd	_mm256_msubadd_ps
Negative Multiply Add	_mm256_nmacc_pd	_mm256_nmacc_ps
Negative Multiply Subtract	_mm256_nmsub_pd	_mm256_nmsub_ps

XOP

The XOP instructions are valuable operations that were not included in the AVX specification. They can be organized into four groups

1. Integer vector operations
   - compare
   - horizontal addition and subtraction
   - multiply accumulate and multiply add accumulate
   - shift and rotate
2. Vector byte permutation
3. Vector conditional move
4. Floating point fraction extraction

Usage

Support for XOP instructions is indicated by the value in bit 11 in ECX when calling CPUID function 0x8000_0001.

1. *Integer Vector Operations*

Compare

The compare XOP operations return a calculated value based on the comparison of two source registers. The comparison operation (less than, greater than, etc.) is selected by a third parameter. See the AMD64 Architecture Programmer's Manual for more information.

	Input & Output Width			
	8	**16**	**32**	**64**
Compare (Signed)	`_mm_com_epi8`	`_mm_com_epi16`	`_mm_com_epi32`	`_mm_com_epi64`
Compare (Unsigned)	`_mm_com_epu8`	`_mm_com_epu16`	`_mm_com_epu32`	`_mm_com_epu64`

Horizontal Addition and Subtract

XOP Horizontal Addition operations perform addition across the values in a single 128-bit packed integer register. The number of added values depends on the specific instruction. The example below describes the operation of the xop intrinsic **_mm_haddq_epi8** with the sixteen 8-bit input values on top, and two 64-bit outputs on bottom.

A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10	A11	A12	A13	A14	A15
A0 + A1 + A2 + A3 + A4 + A5 + A6 + A7								A8 + A9 + A10 + A11 + A12 + A13 + A14 + A15							

The horizontal add XOP operations are given in the table below

	Output Width	Input Width		
		8	16	32
Horizontal Add (Signed)	**16**	_mm_haddw_epi8		
	32	_mm_haddd_epi8	_mm_haddd_epi16	
	64	_mm_haddq_epi8	_mm_haddq_epi16	_mm_haddq_epi32
Horizontal Add (Unsigned)	**16**	_mm_haddw_epu8		
	32	_mm_haddd_epu8	_mm_haddd_epu16	
	64	_mm_haddq_epu8	_mm_haddq_epu16	_mm_haddq_epu32
Horizontal Subtract (Signed)	**16**	_mm_hsubw_epi8		
	32		_mm_hsubd_epi16	
	64			_mm_hsubq_epi32

Multiply (Add) Accumulate

The Integer Multiply Accumulate performs a Multiply Accumulate operation as described in the FMA documentation above. No rounding is performed on the integer data.

Saturation

The saturating versions of the instructions perform an additional saturating step. If the value of the resulting calculation is greater than the maximum or less than the minimum possible integer value, the result is set to the min or max value respectively. On the non-saturating versions, the overflow is ignored.

Accumulate Hi / Lo

Some versions of the function take only the odd (hi) or even (lo) indexed words as input.

	Output Width	Input Width	
		16	32
Multiply Accumulate	**16**	_mm_macc_epi16	
	32	_mm_maccd_epi16	_mm_macc_epi32
Multiply Accumulate Hi	**64**		_mm_macchi_epi32
Multiply Accumulate Lo	**64**		_mm_macclo_epi32
Saturating Integer Multiply Add	**16**	_mm_maccs_epi16	
	32	_mm_maccsd_epi16	_mm_maccs_epi32
Saturating Integer Multiply Add Hi	**64**		_mm_maccshi_epi32
Saturating Integer Multiply Add Lo	**64**		_mm_maccslo_epi32

Multiply Add Accumulate

XOP Multiply Add Accumulate operations perform multiply add accumulate calculations on inputs from three source registers. The example below describes the operation of the xop intrinsic _mm_maddd_epi16 with the three input registers on top and the output register on bottom.

A0	A1	A2	A3	A4	A5	A6	A7
B0	B1	B2	B3	B4	B5	B6	B7
C0		C1		C2		C3	
A0*B0 + A1*B1 + C0		A2*B2 + A3*B3 + C1		A4*B4 + A5*B5 + C2		A6*B6 + A7*B7 + C3	

The multiply add accumulate XOP operations are given in the table below

	Output Width	Input Width
		16, 32
Multiply Add Accumulate	32	_mm_maddd_epi16
Saturating Multiply Add Accumulate	32	_mm_maddsd_epi16

Shift and Rotate

The shift XOP operations perform bit shifts similar to the SSE2 shift operations, but allow the user to specify a different shift count for each data field rather than a single immediate. If the shift count is positive, the data shifts to the left. If the count is negative, the data shifts to the right.

The rotate XOP instructions behave similarly to the shift above, but as bits are shifted off one end of a data field they are appended to the other.

Arithmetic vs. Logical Shift

Arithmetic shift operations preserve the sign of integer numbers when shifting. Logical shift operations reposition the bits without regard to sign.

Rotate by Immediate

In the rotate XOP functions, each of the packed values in the first source register is shifted by the count packed in the second source register. In the rotate by immediate version, all the packed values are rotated by the same count passed in the second immediate parameter.

	Input and Output Width			
	8	16	32	64
Arithmetic Shift	_mm_sha_epi8	_mm_sha_epi16	_mm_sha_epi32	_mm_sha_epi64
Logical Shift	_mm_shl_epi8	_mm_shl_epi16	_mm_shl_epi32	_mm_shl_epi64
Rotate	_mm_rot_epi8	_mm_rot_epi16	_mm_rot_epi32	_mm_rot_epi64
Rotate by Immediate	_mm_roti_epi8	_mm_roti_epi16	_mm_roti_epi32	_mm_roti_epi64

2. Vector Byte Permutation

These operations perform selective permutation between two source packed registers. They use additional parameters to specify complex transformations on the input data. See the function documentation for more information.

Permute

The resulting packed integer register contains data selected from the two source registers and transformed as controlled by the selector parameter.

Permute2

The resulting packed register contains either zero or the corresponding value from either of the sources as controlled by the passed selector and control data.

	Input and Output Width		
	8	**32**	**64**
Permute Byte	`_mm_perm_epi8`		
Permute2 (Floating Point, 128)		`_mm_permute2_ps`	`_mm_permute2_pd`
Permute2 (Floating Point, 256)		`_mm256_permute2_ps`	`_mm256_permute2_pd`

3. *Vector Conditional Move*

This operation performs a bitwise select between two 128-bit inputs. The selection is specified in a third 128-bit register.

	Input & Output Width	
	128	**256**
Floating Point Extraction (Vector)	`_mm_cmov_si128`	`_mm256_cmov_si256`

4. *Floating point fraction extraction*

These operations extract the fraction portion (part to the right of the decimal) of the packed vector or scalar value.

	Input & Output Width	
	32	**64**
Floating Point Extraction (Vector, 128)	`_mm_frcz_ps`	`_mm_frcz_pd`
Floating Point Extraction (Vector 256)	`_mm256_frcz_ps`	`_mm256_frcz_pd`
Floating Point Extraction (Scalar)	`_mm_frcz_ss`	`_mm_frcz_sd`

F16C / CVT16

Most scientific calculations are performed using 32 and 64-bit floating point data. When high precision and large data range are desired, 64-bit double precision data is generally preferred over 32-bit single precision.

In applications such as image processing however, application developers sometimes trade the range and accuracy of larger data formats for the increased speed of processing 16-bit data.

AMD "Piledriver" core processors include support for new instructions which accelerate the conversion between 16 and 32-bit floating point values. These instructions provide performance advantages over the bit-manipulation traditionally required to perform these conversions.

Floating Point

Unlike simple integer formats, floating point makes use of sub divided data fields and complex rules to derive values from data. When working with floating point data, the data is divided into three distinct parts: **sign bit**, **exponent** and **fraction.** The fraction part is also called the **mantissa** or **significand**.

The table below gives the size of the bit fields as well as the exponent bias associated with each format. For more detailed information refer to the IEEE 754 standard which defines the various formats.

	Sign Bits	Exponent Bits	Fraction Bits	Exponent Bias
16-bit (half precision)	1	5	10	15
32-bit (single precision)	1	8	23	127
64-bit (double precision)	1	11	52	1023

Floating point values are calculated using the formula

$$\text{Value} = -1^{\text{sign}} * 1.\mathit{fraction}_2 * 2^{(\text{exponent} - \text{bias})}$$

The example below shows the decimal conversion of the 16-bit float [1110010011010010].

Sign	Exponent	Fraction
1 [negative]	11001 [25]	**0011010010**

The value is separated into its component parts above. Calculation of the resulting value follows. Note that multiplying by 2^{10} in the third step below is equivalent to shifting the decimal 10 places to the right.

[negative] 1.**0011010010** * $2^{(25-15)}$
[negative] 1.**0011010010** * 2^{10}
[negative] 10011010010
[negative] 1234

Usage

Support for F16C instructions is indicated by the value in bit 29 in ECX when calling the CPUID function 0x0000_0001.

F16C

Two new F16C instructions are supported by the "Piledriver" core processor

	Output Width	Input Width 16	Input Width 32
Floating Point Convert (Vector 128)	**16**		`_mm_cvtps_ph`
	32	`_mm_cvtph_ps`	
Floating Point Convert (Vector 256)	**16**		`_mm256_cvtps_ph`
	32	`_mm256_cvtph_ps`	

The figure below shows the input and output packed 128-bit register when calling _mm_cvtps_ph. The top register contains four 32-bit values as input. The bottom register shows the resulting four 16-bit values packed in the lower 64-bits

A passed immediate controls the rounding mode (round up, truncate, etc). See the documentation of the VCVTPH2PS and VCVTPS2PH instruction for more information.

BMI and TBM

BMI and TBM are optimized version of common bit manipulation instructions. Unlike the previous instructions that operate on 128 and 256-bit XMM and YMM registers, these instructions operate on general purpose registers.

For more information about an instruction, see the AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions.

BMI

Support for BMI instructions is indicated by bit 3 in EBX when calling the CPUID function 0x0000_0007.

Instruction	Description
ANDN	Logical And-Not
BEXTR (reg)	Bit Field Extract
BLSI	Isolate Lowest Set Bit
BLSMSK	Mask From Lowest Set Bit
BLSR	Reset Lowest Set Bit
TZCNT	Count Trailing Zeros

TBM

Support for TBM instructions is indicated by bit 21 in ECX when calling the CPUID function 0x8000_0001.

Instruction	Description
BEXTR (imm)	Bit Field Extract
BLCFILL	Fill From Lowest Clear Bit
BLCI	Isolate Lowest Clear Bit
BLCIC	Isolate Lowest Clear Bit and Complement
BLCMSK	Mask From Lowest Clear Bit
BLCS	Set Lowest Clear Bit
BLSFILL	Fill From Lowest Set Bit
BLSIC	Isolate Lowest Set Bit and Complement
T1MSKC	Inverse Mask from Trailing Ones
TZMSK	Mask From Trailing Zeros

References

AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions
http://support.amd.com/us/Processor_TechDocs/24594_APM_v3.pdf

AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP and FMA4 Instructions
http://support.amd.com/us/Embedded_TechDocs/43479.pdf

AMD CPUID Specification
http://support.amd.com/us/Embedded_TechDocs/25481.pdf

XOP Intrinsics
http://msdn.microsoft.com/en-us/library/gg466493(v=vs.100).aspx

Software Optimization Guide for AMD Family 15h Processors
http://support.amd.com/us/Processor_TechDocs/47414_15h_sw_opt_guide.pdf

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.