
4 128-Bit Media and Scientific Programming

This chapter describes the 128-bit media and scientific programming model. This model includes all instructions that access the 128-bit XMM registers—called the *128-bit media instructions*. These instructions perform integer and floating-point operations primarily on vector operands (a few of the instructions take scalar operands). They can speed up certain types of procedures—typically high-performance media and scientific procedures—by substantial factors, depending on data-element size and the regularity and locality of data accesses to memory.

4.1 Overview

4.1.1 Origins

The 128-bit media instruction set includes instructions originally introduced as the streaming SIMD extension (SSE) and SSE-2 instructions. For details on the extension-set origin of each instruction, see Appendix C, “Instruction Subsets and CPUID Feature Sets”, on page 1203 of Volume 3.

4.1.2 Compatibility

128-bit media instructions can be executed in any of the architecture’s operating modes. Existing SSE and SSE-2 binary programs run in legacy and compatibility modes without modification. The support provided by the x86-64 architecture for such binaries is identical to that provided by legacy x86 architectures.

To run in 64-bit mode, legacy 128-bit media programs must be recompiled. The recompilation has no side effects on such programs, other than to provide access to the following additional resources:

- Access to the eight extended XMM registers (for a total of 16 XMM registers).
- Access to the eight extended general-purpose registers (for a total of 16 GPRs).
- Access to the extended 64-bit width of all GPRs.
- Access to the 64-bit virtual address space.
- Access to the RIP-relative addressing mode.

The 128-bit media instructions use data registers, a control and status register (MXCSR), rounding control, and an exception reporting and response mechanism that are distinct from and functionally independent of those used by the x87 floating-point instructions. Because of this, 128-bit media programming support usually requires exception handlers that are distinct from those used for x87 exceptions. This support is provided by virtually all legacy operating systems for the x86 architecture.

4.2 Capabilities

The 128-bit media instructions are designed to support media and scientific applications. The vector operands used by these instructions allow applications to operate in parallel on multiple elements of vectors. The elements can be integers (from bytes to quadwords) or floating-point (either single-precision or double-precision). Arithmetic operations produce signed, unsigned, and/or saturating results.

The availability of several types of vector move instructions and (in 64-bit mode) twice the legacy number of XMM registers (a total of 16 such registers) can eliminate substantial memory-access overhead, making a substantial difference in performance.

4.2.1 Types of Applications

Typical media applications well-suited to the 128-bit media programming model include a broad range of audio, video, and graphics programs. For example, music synthesis, speech synthesis, speech recognition, audio and video compression (encoding) and decompression (decoding), 2D and 3D graphics, streaming video (up to high-definition TV), and digital signal processing (DSP) kernels are all likely to experience higher performance using 128-bit media instructions than using other types of instructions in x86-64 architecture.

Such applications commonly use small-sized integer or single-precision floating-point data elements in repetitive loops, in which the typical operations are inherently parallel. For example, 8-bit and 16-bit data elements are commonly used for pixel information in graphics applications, in which each of the RGB pixel components (red, green, blue, and alpha) are represented by an 8-bit or 16-bit integer. 16-bit data elements are also commonly used for audio sampling.

The 128-bit media instructions allow multiple data elements like these to be packed into 128-bit vector operands located in XMM registers or memory. The instructions operate in parallel on each of the elements in these vectors. For example, 16 elements of 8-bit data can be packed into a 128-bit vector operand, so that all 16 byte elements are operated on simultaneously, and in pairs of source operands, by a single instruction.

The 128-bit media instructions also support a broad spectrum of scientific applications. For example, their ability to operate in parallel on double-precision floating-point vector elements makes them well-suited to computations like dense systems of linear equations, including matrix and vector-space operations with real and complex numbers. In professional CAD applications, for example, high-performance physical-modeling algorithms can be implemented to simulate processes such as heat transfer or fluid dynamics.

4.2.2 Integer Vector Operations

Most of the 128-bit media arithmetic instructions perform parallel operations on pairs of vectors. *Vector* operations are also called *packed* or *SIMD* (single-instruction, multiple-data) operations. They take vector operands consisting of multiple elements, and all elements are operated on in parallel. Figure 4-1 shows an example of parallel operations on pairs of 16 byte-sized integers in the source operands. The result of the operation replaces the first source operand. There are also instructions that operate on vectors of words, doublewords, or quadwords.

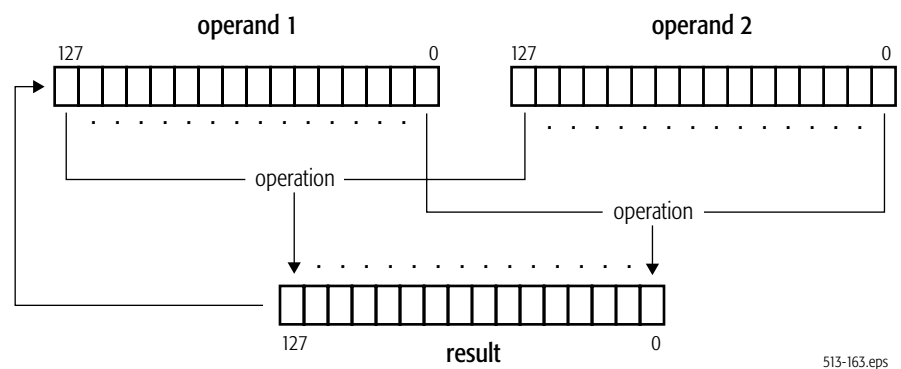
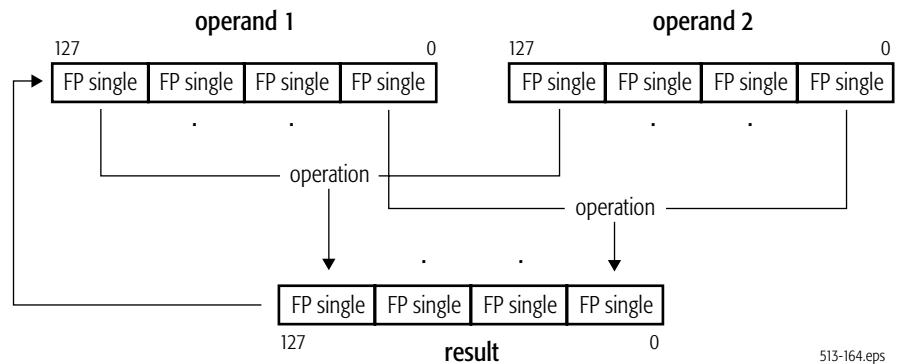


Figure 4-1. Parallel Operations on Vectors of Integer Elements

4.2.3 Floating-Point Vector Operations

There are almost as many 128-bit floating-point instructions as integer instructions. Figure 4-2 shows an example of parallel operations on vectors containing four 32-bit single-precision floating-point values. There are also instructions that operate on vectors containing two 64-bit double-precision floating-point values.



513-164.eps

Figure 4-2. Parallel Operations on Vectors of Floating-Point Elements

Integer and floating-point instructions can be freely intermixed in the same procedure. The floating-point instructions allow media applications such as 3D graphics to accelerate geometry, clipping, and lighting calculations. Pixel data are typically integer-based, although both integer and floating-point instructions are often required to operate completely on the data. For example, software can change the viewing perspective of a 3D scene through transformation matrices by using floating-point instructions in the same procedure that contains integer operations on other aspects of the graphics data.

128-bit media programs using floating-point instructions are typically much easier to write and of higher performance than x87 floating-point programs, because the XMM register file is flat rather than stack-oriented, there are twice as many registers (in 64-bit mode), and 128-bit media instructions can operate on two or four times the number of floating-point operands as compared with x87 instructions. This ability to operate in parallel on multiple pairs of floating-point elements often makes it possible to remove local temporary variables that would otherwise be needed in x87 floating-point code.

4.2.4 Data Conversion and Reordering

There are instructions that support data conversion of vector elements, including conversions between integer and floating-point data types—located in XMM registers, MMX registers, GPR registers, or memory—and conversions of element-ordering or precision. For example, the unpack instructions take two vector operands and interleave their low or high elements. Figure 4-3 shows an unpack and interleave operation on word-sized elements (PUNCKLWD). If the left-hand source operand has elements whose value is zero, the operation converts each element in the low half of the right-hand operand to a data type of twice its original precision—useful, for example, in multiply operations in which results may overflow or underflow.

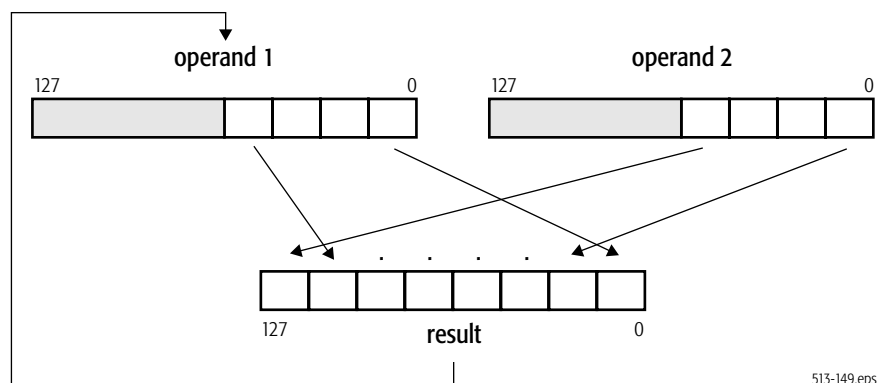
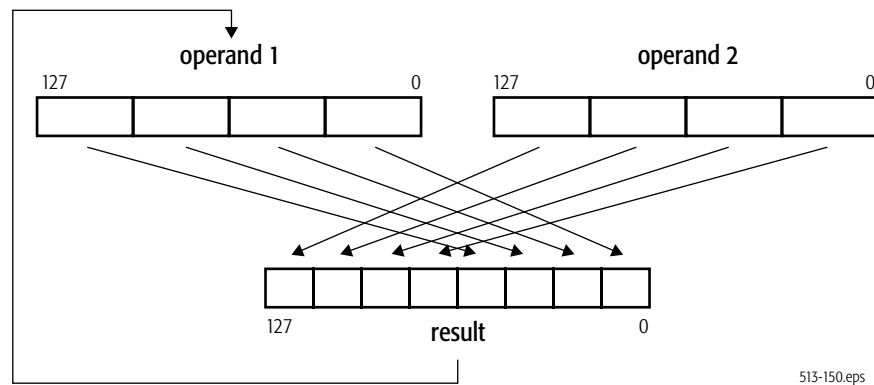


Figure 4-3. Unpack and Interleave Operation

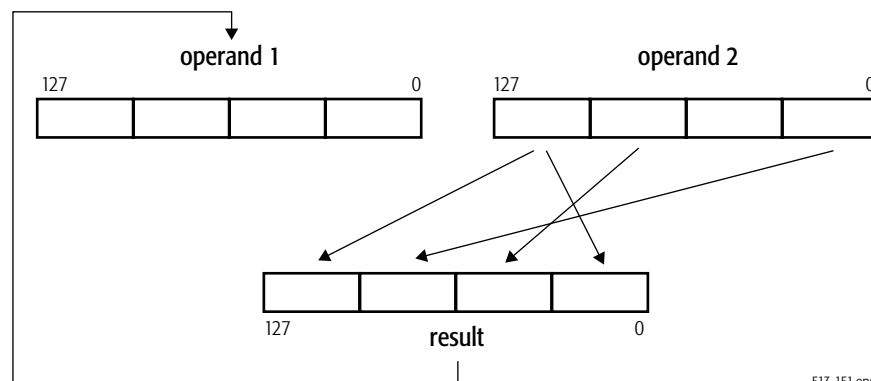
There are also pack instructions, such as PACKSSDW shown in Figure 4-4, that convert each element in a pair of vectors to lower precision by selecting the elements in the low half of each vector. Vector-shift instructions are also supported. They can scale each element in a vector to higher or lower values.



513-150.eps

Figure 4-4. Pack Operation

Figure 4-5 shows one of many types of shuffle operation (PSHUFD). Here, the second operand is a vector containing doubleword elements, and an immediate byte provides shuffle control for up to 256 permutations of the elements. Shuffles are useful, for example, in color imaging when computing alpha saturation of RGB values. In this case, a shuffle instruction can replicate an alpha value in a register so that parallel comparisons with three RGB values can be performed.



513-151.eps

Figure 4-5. Shuffle Operation

There is an instruction that inserts a single word from a general-purpose register or memory into an XMM register, at a specified

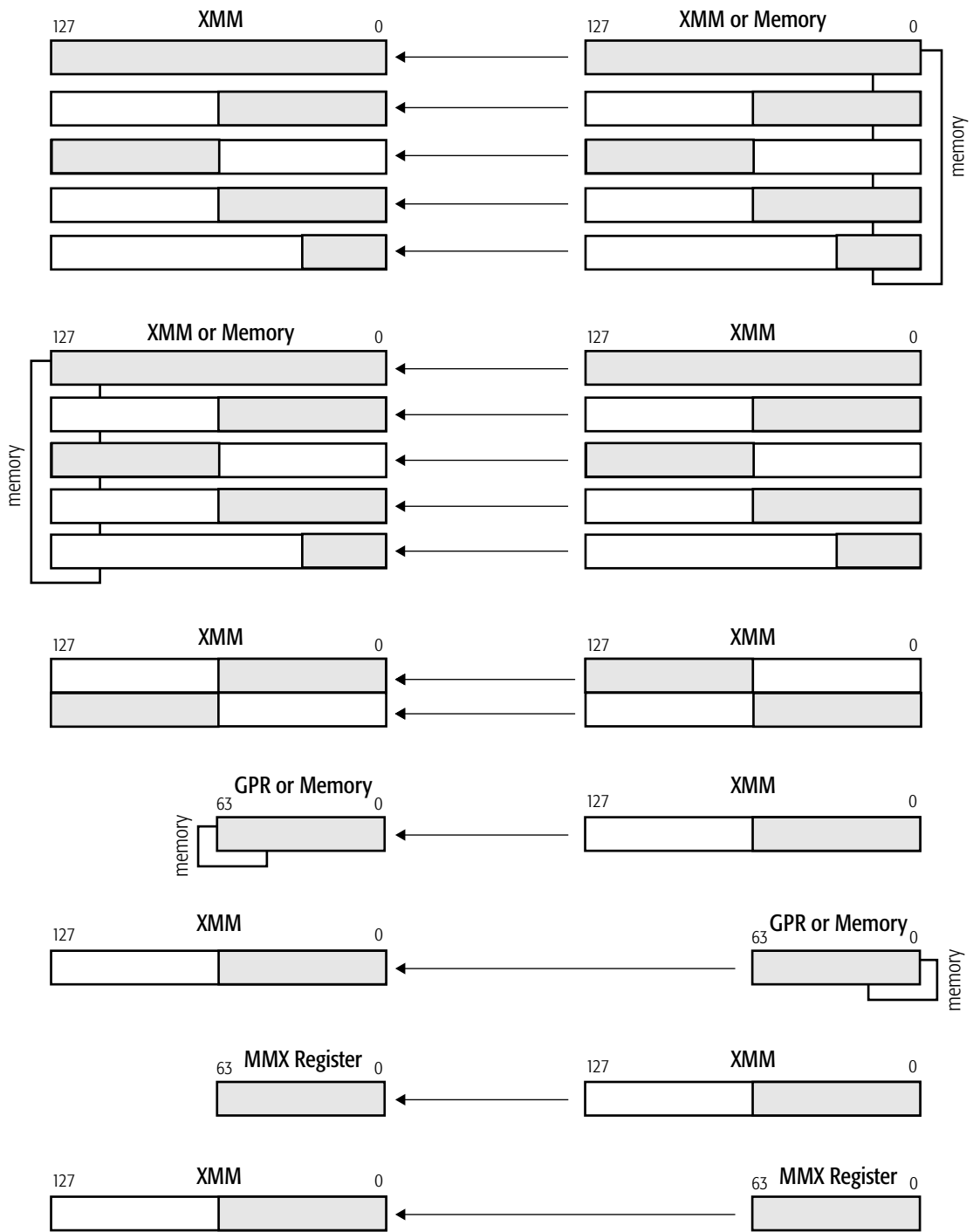
location, leaving the other words in the XMM register unmodified.

4.2.5 **Block Operations**

Move instructions—along with unpack instructions—are among the most frequently used instructions in 128-bit media procedures. Figure 4-6 shows the combined set of move operations supported by the integer and floating-point move instructions. These instructions provide a fast way to copy large amounts of data between registers or between registers and memory. They support block copies and sequential processing of contiguous data.

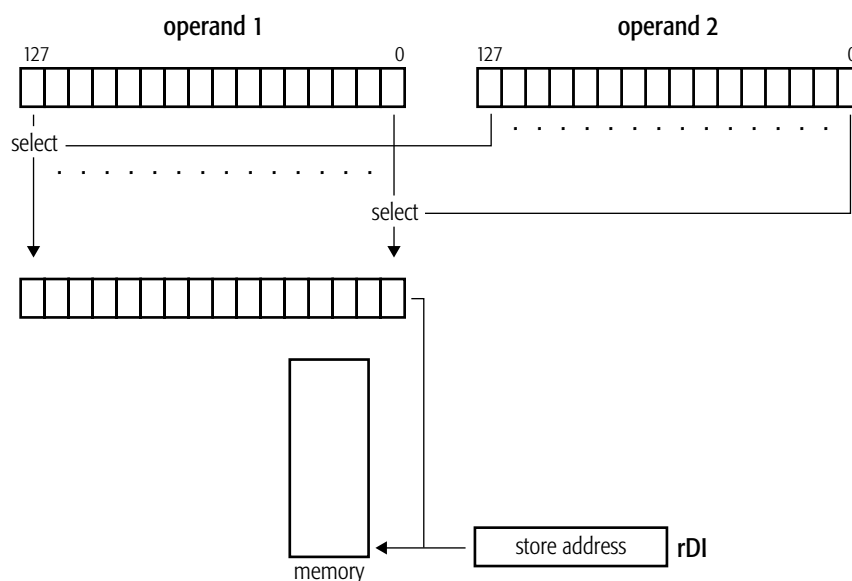
When moving between XMM registers, or between an XMM register and memory, each integer move instruction can copy up to 16 bytes of data. When moving between an XMM register and an MMX or GPR register, an integer move instruction can move 8 bytes of data. The floating-point move instructions can copy vectors of four single-precision or two double-precision floating-point operands in parallel.

Streaming-store versions of the move instructions permit bypassing the cache when storing data that is accessed only once. This maximizes memory-bus utilization and minimizes cache pollution. There is also a streaming-store integer move-mask instruction that stores bytes from one vector, as selected by mask values in a second vector. Figure 4-7 shows the MASKMOVDQU operation. It can be used, for example, to handle end cases in block copies and block fills based on streaming stores.



513-171.eps

Figure 4-6. Move Operations



513-148.eps

Figure 4-7. Move Mask Operation

4.2.6 Matrix and Special Arithmetic Operations

The instruction set provides a broad assortment of vector add, subtract, multiply, divide, and square-root operations for use on matrices and other data structures common to media and scientific applications. It also provides special arithmetic operations including multiply-add, average, sum-of-absolute differences, reciprocal square-root, and reciprocal estimation.

Media applications often multiply and accumulate vector and matrix data. In 3D-graphics geometry, for example, objects are typically represented by triangles, each of whose vertices are located in 3D space by a matrix of coordinate values, and matrix transforms are performed to simulate object movement.

128-bit media integer and floating-point instructions can perform several types of matrix-vector or matrix-matrix operations, such as addition, subtraction, multiplication, and accumulation, to effect 3D transforms of vertices. Efficient matrix multiplication is further supported with instructions that can first transpose the elements of matrix rows and columns. These transpositions can make subsequent accesses to memory or cache more efficient when performing arithmetic matrix operations.

Figure 4-8 shows a vector multiply-add instruction (PMADDWD) that multiplies vectors of 16-bit integer elements to yield intermediate results of 32-bit elements, which are then summed pair-wise to yield four 32-bit elements. This operation can be used with one source operand (for example, a coefficient) taken from memory and the other source operand (for example, the data to be multiplied by that coefficient) taken from an XMM register. It can also be used together with a vector-add operation to accumulate *dot product* results (also called *inner* or *scalar products*), which are used in many media algorithms such as those required for finite impulse response (FIR) filters, one of the commonly used DSP algorithms.

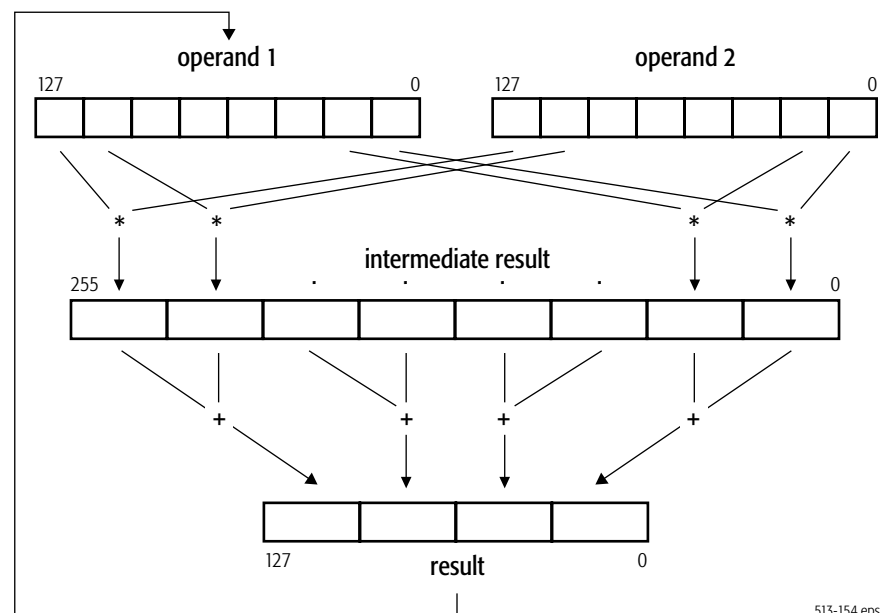
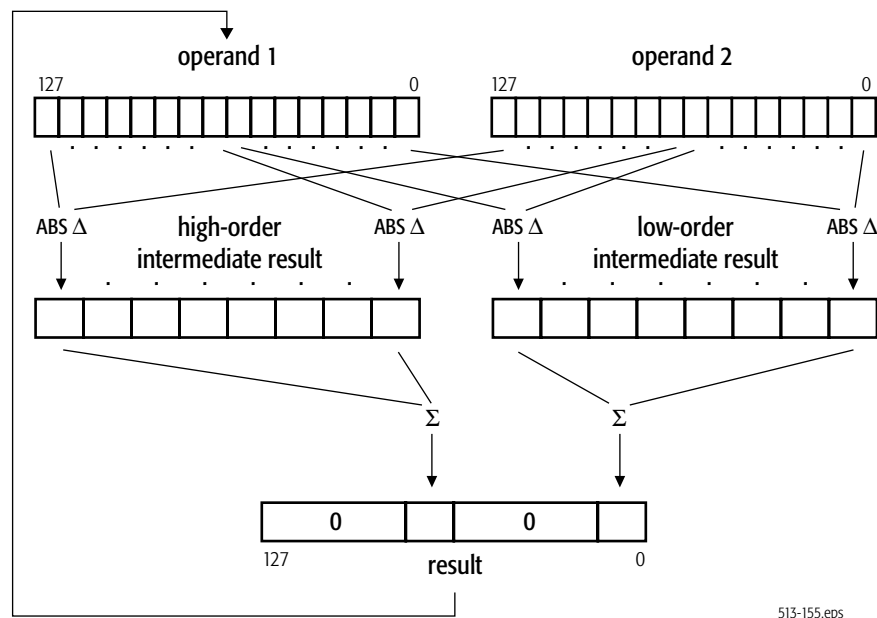


Figure 4-8. Multiply-Add Operation

There is also a sum-of-absolute-differences instruction (PSADBW), shown in Figure 4-9. This is useful, for example, in computing motion-estimation algorithms for video compression.



513-155.eps

Figure 4-9. Sum-of-Absolute-Differences Operation

There is an instruction for computing the average of unsigned bytes or words. The instruction is useful for MPEG decoding, in which motion compensation involves many byte-averaging operations between and within macroblocks. In addition to speeding up these operations, the instruction also frees up registers and make it possible to unroll the averaging loops.

Some of the arithmetic and pack instructions produce vector results in which each element *saturates* independently of the other elements in the result vector. Such results are clamped (limited) to the maximum or minimum value representable by the destination data type when the true result exceeds that maximum or minimum representable value. Saturating data is useful for representing physical-world data, such as sound and color. It is used, for example, when combining values for pixel coloring.

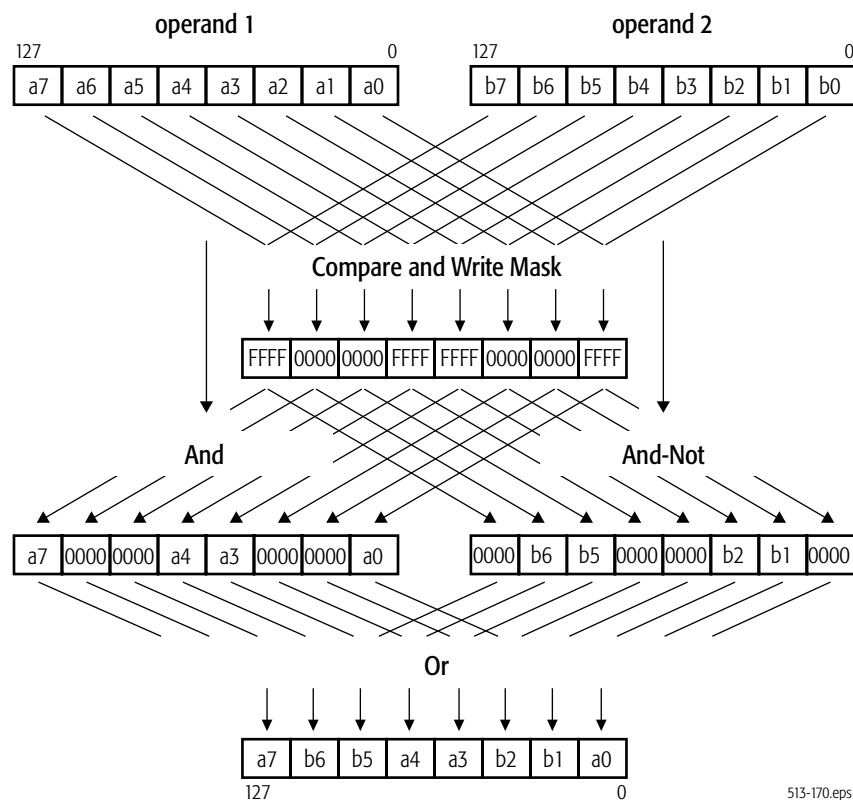
4.2.7 Branch Removal

Branching is a time-consuming operation that, unlike most 128-bit media vector operations, does not exhibit parallel behavior (there is only one branch target, not multiple targets, per branch instruction). In many media applications, a branch involves selecting between only a few (often only two) cases. Such branches can be replaced with 128-bit media vector

compare and vector logical instructions that simulate predicated execution or conditional moves.

Figure 4-10 shows an example of a non-branching sequence that implements a two-way multiplexer—one that is equivalent to the ternary operator “?:” in C and C++. The comparable code sequence is explained in “Compare and Write Mask” on page 187.

The sequence in Figure 4-10 begins with a vector compare instruction that compares the elements of two source operands in parallel and produces a mask vector containing elements of all 1s or 0s. This mask vector is ANDed with one source operand and ANDed-Not with the other source operand to isolate the desired elements of both operands. These results are then ORed to select the relevant elements from each operand. A similar branch-removal operation can be done using floating-point source operands.



513-170.eps

Figure 4-10. Branch-Removal Sequence

The min/max compare instructions, for example, are useful for clamping, such as color clamping in 3D graphics, without the need for branching. Figure 4-11 illustrates a move-mask instruction (PMOVMASKB) that copies sign bits to a general-purpose register (GPR). The instruction can extract bits from mask patterns, or zero values from quantized data, or sign bits—resulting in a byte that can be used for data-dependent branching.

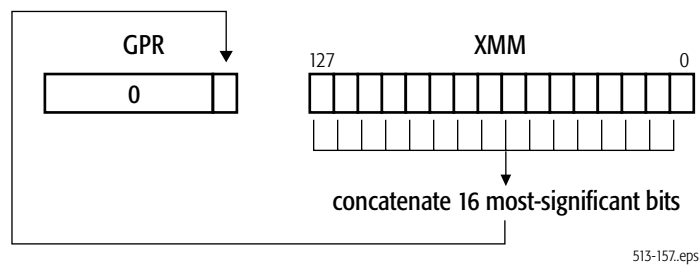


Figure 4-11. Move Mask Operation

