

MORTON COORDINATE ADJUSTMENT PROCESSORS, METHODS, SYSTEMS, AND INSTRUCTIONS

BACKGROUND

Technical Field

5 Embodiments described herein generally relate to processors. In particular, embodiments described herein generally relate to the use of Morton coordinates in processors.

Background Information

Computer systems and other electronic devices often utilize data that is organized in data structures. The data structures may represent particular arrangements or organizations of the data. One common type of data structure often used in computer systems is an array, such as a two-dimensional (2D) array.

Figure 1 illustrates an example of a two-dimensional (2D) array 100 in which data items (e.g., the values 7, 16, 24, 27, etc.) are arranged in two dimensions. Representatively, the 2D data structure may represent a table, matrix, or the like. In the illustration, the two dimensions are defined by a first dimension or x-axis 102, and a second dimension or y-axis 104. The x-axis and y-axis are mutually perpendicular and define a 2D grid in which the data items are arranged. The data items in the 2D array may be identified by the values of the x and y indices or coordinates along the two axis. The x-coordinate represents the distance and/or relative position along the x-axis measured from the origin where the two axis intersect, whereas the y-coordinate represents the distance and/or relative position along the y-axis measured from the origin. In the illustrated example, the x-coordinates and the y-coordinates have the values of 0, 1, 2, and 3. Representatively, the coordinates or indices may represent row and column numbers. By way of example, the data item having the value of 14 may be identified by the x,y-coordinates (1,2) which may indicate the data item in column 2, row 3. Similarly, the data item having the value of 24 may be identified by decrementing the y-coordinate by one unit to get the x,y-coordinates (1,1). Other examples are contemplated where the 2D data structure represents a Cartesian coordinate system, and the coordinates may represent locations of points in the Cartesian coordinate system.

Within computer systems and other electronic devices, such arrays and other data structures may be stored in memory or other linear storage. Different ways are possible for storing the 2D and other multi-dimensional arrays in memory. For example, the 2D arrays may be stored in row-major order. In row-major order, the rows of the array are contiguous in the memory. For example, the data items may be stored in the memory in the order 27, 3, 8, 11, 9, 24, 7, 1, 8, 14, 16, 2, 7, 16, 4, and 20. Alternatively, the 2D arrays may be stored in the memory

in column-major order. In column-major order, the columns of the array are contiguous in the memory. For example, the data items may be stored in the memory in the order 27, 9, 8, 7, 3, 24, 14, 16, 8, 7, 16, 4, 11, 1, 2, and 20.

BRIEF DESCRIPTION OF THE DRAWINGS

5 The invention may best be understood by referring to the following description and accompanying drawings that are used to illustrate embodiments. In the drawings:

Figure 1 illustrates an example of a two-dimensional (2D) array.

Figure 2A illustrates an example of a Morton order curve mapped to the 2D array of **Figure 1**.

10 **Figure 2B** is a block diagram illustrating example 2D, 3D, and 4D Morton coordinates.

Figure 3 is a block diagram of an embodiment of a processor that is operable to perform an embodiment of a Morton coordinate adjustment instruction.

Figure 4 is a block flow diagram of an embodiment of a method of performing an embodiment of a Morton coordinate adjustment instruction.

15 **Figure 5** is a block diagram of an example embodiment of adjusting a source 3D Morton coordinate to a result/adjusted 3D Morton coordinate to reflect a decrement of a given or indicated dimension by one.

Figure 6 is a block diagram illustrating an example embodiment of a 2D Morton coordinate adjustment operation to adjust 2D Morton coordinates in a source packed data operand into corresponding adjusted 2D Morton coordinates in a result packed data operand to reflect a decrement in a given indicated dimension.

Figure 7 is a block diagram illustrating an example embodiment of a 3D Morton coordinate adjustment operation to adjust 3D Morton coordinates in a source packed data operand into corresponding adjusted 3D Morton coordinates in a result packed data operand to reflect a decrement in a given indicated dimension.

Figure 8 is a block diagram of a particular example embodiment of an immediate that is suitable for one or more embodiments of a Morton coordinate adjustment instruction.

Figure 9 is a block diagram of a more detailed example embodiment of a suitable processor that is operable to perform an embodiment of a Morton coordinate adjustment instruction.

Figure 10 is a block diagram of an embodiment of a Morton coordinate adjustment instruction.

Figures 11A-11C are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof, according to embodiments of the invention.

Figure 12A-B is a block diagram illustrating an exemplary specific vector friendly instruction format and an opcode field, according to embodiments of the invention.

Figure 13A-D is a block diagram illustrating an exemplary specific vector friendly instruction format and fields thereof, according to embodiments of the invention.

5 **Figure 14** is a block diagram of an embodiment of a register architecture.

Figure 15A is a block diagram illustrating an embodiment of an in-order pipeline and an embodiment of a register renaming out-of-order issue/execution pipeline.

Figure 15B is a block diagram of an embodiment of processor core including a front end unit coupled to an execution engine unit and both coupled to a memory unit.

10 **Figure 16A** is a block diagram of an embodiment of a single processor core, along with its connection to the on-die interconnect network, and with its local subset of the Level 2 (L2) cache.

Figure 16B is a block diagram of an embodiment of an expanded view of part of the processor core of **Figure 16A**.

15 **Figure 17** is a block diagram of an embodiment of a processor that may have more than one core, may have an integrated memory controller, and may have integrated graphics.

Figure 18 is a block diagram of a first embodiment of a computer architecture.

Figure 19 is a block diagram of a second embodiment of a computer architecture.

Figure 20 is a block diagram of a third embodiment of a computer architecture.

20 **Figure 21** is a block diagram of a fourth embodiment of a computer architecture.

Figure 22 is a block diagram of use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set, according to embodiments of the invention.

DETAILED DESCRIPTION OF EMBODIMENTS

25 Disclosed herein are Morton coordinate adjustment instructions, processors to execute the instructions, methods performed by the processors when processing or executing the instructions, and systems incorporating one or more processors to process or execute the instructions. In the following description, numerous specific details are set forth (e.g., specific instruction operations, data formats, processor configurations, microarchitectural details, sequences of operations, etc.). However, embodiments may be practiced without these specific details. In
30 other instances, well-known circuits, structures and techniques have not been shown in detail to avoid obscuring the understanding of the description.

One challenge is that the way data items are stored in memory or linear storage may significantly affect the performance of algorithms that use the data items. For one thing, the data

items generally need to be read into a processor from the memory, but only a limited number of bits can be read into the processor from the memory at one time. For example, commonly processors are only able to read 512-bits of contiguous data from the memory at a time (e.g., in a single read operation). Different data items will be read in contiguous order depending upon whether the data items are stored in row-major order, column-major order, or some other order. For example, if the data items are stored in row-major order, then the data items may be read in the order 27, 3, 8, 11, 9, 24, and so on, up to a maximum of 512-bits. Conversely, if the data items are stored in row-major order, then the data items may be read in the order 27, 9, 8, 7, 3, 24, and so on, up to a maximum of 512-bits. Generally, each read operation from the memory may only be able to obtain some of the data items of interest from the array. By way of example, especially when the arrays are large (e.g., have at least more than 512-bits), if the data items are stored in row-major order, it may not even be possible to obtain neighboring data in different rows but the same column (e.g., data items 27 and 9) from the first column of the array in the same read operation, even though these data items are adjacent to one another in the array. Analogous situations may be encountered when the data are in column-major order, and for storage of data for 3D and 4D arrays.

The way the data items are arranged in the memory also generally affects the ability to efficiently cache the data items in one or more caches of the processor. Accesses to data items in the cache(s) typically have lower latencies than accesses to data items in the memory. However, even if data items of interest are in the cache(s), poor cache utilization may tend to result if the data items are scattered among many different cache lines. Commonly, each cache line stores 512-bits of contiguous data that has been read from the memory. If the data items are not arranged in the memory in way that is efficient for the relevant algorithm using the data, then the data items may be sparse in the cache lines. In a severe scenario, each cache line may hold only a single data item of interest. Conversely, if the data items were to be arranged in the memory in a way that is highly efficient for the particular algorithm, then the data items of interest may be more densely packed into the cache lines with each cache line containing from multiple to many data items of interest. This may help to improve the effectiveness of the cache(s). Similarly, if the data items were to be arranged in the memory in a way that is highly efficient for the particular algorithm, then more data items of interest could be read into the processor in each read operation at least on average. Accordingly, approaches that would allow the data to be arranged in the memory in a way that is efficient for the particular algorithm may help to improve performance.

There are various different types of algorithms that tend to process data that has multi-

dimensional locality or proximity relative to other data. As one example, image processing algorithms (e.g., red eye reduction, compression, etc.) often tend to process data for groups of adjacent, neighboring, or otherwise proximate pixels together or concurrently. The algorithms may be relatively more interested in data for a block of neighboring pixels, rather than data for all the pixels in a single row or column (e.g., as may be the case in a row-major or column-major arrangement). Similarly, in many video processing algorithms (e.g., compression, video surveillance analysis, robotic vision, etc.) it is common to process the data for groups of neighboring pixels and/or the data in corresponding pixels of sequential video frames together or concurrently. For example, compression is often achieved by storing differences between such pixels rather than absolute pixel values. Examples of other applications or algorithms that also tend to utilize data with multi-dimensional locality include, but are not limited to, tomographic analysis, seismic analysis, geometric modeling, matrix operations (e.g., matrix multiply and/or transpose), finite element analysis, ray tracing, Fourier transforms, parallel data construction applications, and graphics applications, to name just a few. However, as discussed above, especially when relatively large arrays are involved, row-major order, column major order, and various other arrangements of the data often do not provide efficient arrangements of the data for applications that heavily utilize data with multi-dimensional locality. As a result, other ways of organizing the data that preserve multi-dimensional locality would tend to offer certain advantages for certain applications.

A Z-order curve, also known as a Morton order curve, is a continuous space-filling curve or function that is able to map multi-dimensional data or space (e.g., a Cartesian coordinate system data or space and/or an n-dimensional Euclidian data or space) to a single dimension while preserving the multi-dimensional locality or proximity of the data. That is, the Morton order curve may map data in a 2D, 3D, 4D, or other multi-dimensional space, onto a linear list or arrangement of data in a way that preserves the multi-dimensional locality of the data (e.g., data with locality in the multi-dimensional space also has locality in the linear list or arrangement provided by the Morton curve). The order of the data along the Morton order curve is referred to as Z-curve order or Morton order. Morton order is reflected in each point's Morton code or Morton coordinate or Morton index. The Z-order curve has as a basic unit a Z-shaped curve that linearly connects four points. The overall Z-order space-filling curve is formed by connecting multiple or many of these Z-shaped curves or units together to fill a 2D, 3D, 4D, or other multi-dimensional space.

Figure 2A illustrates an example of a Z-order curve or Morton order curve 206 mapped to the 2D array 100 of **Figure 1**. As shown, a number of the Z-shaped curves or units (in this

example four) may be connected together in a linear arrangement to traverse or fill all the points in the 2D array. In this example, since there are sixteen data items in the 2D array, four Z-shaped units, each having four points, entirely traverse the sixteen data item 2D array. In the illustration, the coordinates are shown in decimal notation (e.g., 0, 1, 2, 3, etc.). Equivalent binary representations of the coordinates are also shown in parenthesis (e.g., 000, 001, 010, 011, etc.) alongside the decimal coordinates. By way of example, the decimal coordinate value 2 is equivalent to the binary coordinate value “010”.

Mapping the Morton order curve to the array involves determining the Morton codes or coordinates of points or data items of the array. The Morton coordinates of the individual points along the Morton order curve may be calculated by interleaving the bits of the binary representations of the multiple multi-dimensional coordinates in a fixed pattern. For example, if the first, second, and third bits of the x-coordinate are represented respectively as x_1 , x_2 , and x_3 , and if the first, second, and third bits of the y-coordinate are represented respectively as y_1 , y_2 , y_3 then the Morton coordinate for a point may be calculated by interleaving or “shuffling” the bits into the order $x_1y_1x_2y_2x_3y_3$. To further illustrate, the Morton coordinate for the point ($x=1$, $y=0$) may be found by interleaving the bits of the binary representations for these coordinates (i.e., 001, 000) to achieve the Morton coordinate of value “000001”. From right to left, the first, third, and fifth bits correspond to the x-coordinate, whereas the second, fourth, and sixth bits correspond to the y-coordinate. In the illustration, the least significant or lowest order bits are shown on the right. Similarly, in 3D and 4D, the 3D or 4D Morton coordinates may be found by interleaving the bits of the binary representations for three and four coordinates, respectively. Higher dimensions may be done analogously. In the opposite direction, calculating the binary representations of the multiple multi-dimensional coordinates involves the reverse fixed de-interleaving or “de-shuffling” the bits of the Morton coordinates into the separate coordinates. For example, the Morton coordinate $x_1y_1x_2y_2x_3y_3$ may be converted into the binary representations of the x and y-coordinates by de-interleaving the bits $x_1y_1x_2y_2$ to generate the x-coordinate as $x_1x_2x_3$ and the y-coordinate as $y_1y_2y_3$. To further illustrate, the Morton coordinate “000001” may be converted into the binary representation of the x-coordinate “001” and y-coordinate “000”. Similarly, in 3D and 4D, the binary representations of three or four different coordinates may be found by de-interleaving or de-shuffling the bits of the 3D or 4D Morton coordinates.

Figure 2B is a block diagram illustrating an example 6-bit 2D Morton coordinate 208-1, an example 9-bit 3D Morton coordinate 208-2, and an example 12-bit 4D Morton coordinate 208-3. In this example, each of the x, y, and z-coordinates is represented with only three bits. In

other embodiments, each of the x, y, and z-coordinates in the 3D Morton coordinate may have other numbers of bits. For example, for a 2D Morton coordinate each of the x, y, and z-coordinates may have from about seven to sixteen bits to allow the 2D Morton coordinate with twice as many bits to fit in a 32-bit data element, or may have from about seven to thirty-two bits to allow the 3D Morton coordinate with twice as many bits to fit in a 64-bit data element. Likewise, for a 3D Morton coordinate each of the x, y, and z-coordinates may have from about seven to ten bits to allow the 3D Morton coordinate with three times as many bits to fit in a 32-bit data element, or may have from seven to twenty-one bits to allow the 3D Morton coordinate to fit in a 64-bit data element. Similarly, for a 4D Morton coordinate each of the x, y, and z-coordinates may have from six to eight bits to allow the 4D Morton coordinate with four times as many bits to fit in a 32-bit data element, or may have from six to sixteen bits to allow the 4D Morton coordinate to fit in a 64-bit data element.

Due in part to its ability to represent multi-dimensional locality in data, 2D, 3D, 4D, and/or other multi-dimensional arrays may be rearranged to corresponding 2D, 3D, 4D, and/or other multi-dimensional Morton order representations in order to help improve the performance of certain types of applications. For example, before an application processes data it may be rearranged in the memory from a 2D, 3D, or 4D array to a corresponding 2D, 3D, or 4D Morton order representation. After that application has processed the data utilizing the Morton order, in some instances, it may be desirable to convert the 2D, 3D, or 4D Morton order representation back to the 2D, 3D, or 4D arrays. In other scenarios, the data may initially be organized in a 2D, 3D, or 4D Morton order representation and rearranged to a 2D, 3D, or 4D array in order to improve performance, or for other reasons. In any event, there are times when it is appropriate to convert between Morton order arrangements and multi-dimensional arrays and/or multi-dimensional spaces. However, such conversions generally tend to be computationally intensive (e.g., generally tend to involve interleaving or de-interleaving the bits of many different coordinates).

In other instances, while data is ordered according to a multi-dimensional Morton order arrangement, it may be appropriate to move around in the multi-dimensional Morton coordinates as moving around in the multi-dimensional space coordinates. For example, it may be appropriate to move between a pair of multi-dimensional Morton order coordinates that respectively correspond to points in the multi-dimensional space that have different coordinates. For example, as shown in **Figure 2A** at reference number 203, it may be desired to move from the 2D Morton coordinate "000110" and the 2D Morton coordinate "000011" which respectively correspond to points in the 2D array in which the x-coordinate is decremented by one from $x=2$

to $x=1$, while all the other coordinates in the multi-dimensional space (in this case only the y-coordinate) stay the same. As another example, as shown in **Figure 2A** at reference number 205, it may be desired to move from the 2D Morton coordinate “001101” and the 2D Morton coordinate “000111” which respectively correspond to points in the 2D array in which the y-coordinate is decremented by one from $y=2$ to $y=1$ while all the other coordinates in the multi-dimensional space (in this case only the x-coordinate) stay the same. Notice that in both of these examples, the initial and final 2D Morton coordinates are not sequential in Morton order (e.g., they change from “6” to “3” and “13” to “7” respectively), although in other cases the initial and final 2D Morton coordinates may be sequential in Morton order.

There are various different reasons in which it might be desirable or appropriate to move around in the multi-dimensional Morton coordinates in this way. The scope of the invention is not limited to any known such reason. Representatively, as one possible example, it may be desirable in a particular algorithm to decrement a row index and/or a column index of a 2D matrix or array, to decrement the pixel number in a given row or column of a 2D image while processing the image, etc. In some cases, it may be more useful (e.g., worth the computational overhead and time) to convert out of the multi-dimensional Morton order arrangement entirely into a multi-dimensional space arrangement so that moving around in the Morton order in this way is not needed. However, in other cases it may not be worth the computational overhead and/or time to entirely convert out of the multi-dimensional Morton order arrangement, but still may be useful to be able to move between Morton coordinates corresponding to differences (e.g., unit one decrements) of one of the coordinates in a multi-dimensional space (e.g., a decrement of one of the x, y, z, or t-coordinates in a 4D space).

Since the multi-dimensional Morton coordinates are formed by interleaving or shuffling the bits of the individual multi-dimensional coordinates, one possible approach would be to use one or more instructions to de-interleave or de-shuffle the bits into the separate multi-dimensional coordinates, use one or more instructions to decrement one of the multi-dimensional coordinates (e.g., decrement the x-coordinate), and then use one or more instructions to re-interleave or re-shuffle the bits of the separate multi-dimensional coordinates (including those of the decremented x-coordinate) to determine the new corresponding multi-dimensional Morton coordinate. However, such de-interleaving and re-interleaving and/or such a need to use multiple instructions may tend to take more time and/or consume more processing resources and/or consume more power. Disclosed herein are single instructions that may be used to accelerate determinations of new multi-dimensional Morton coordinates when changing (e.g., decrementing) a coordinate of a multi-dimensional space (e.g., decrementing an x-coordinate,

decrementing a y-coordinate, decrementing a z-coordinate, or decrementing a t-coordinate, etc.).

Figure 3 is a block diagram of an embodiment of a processor 310 that is operable to perform an embodiment of a Morton coordinate adjustment instruction 312. In some embodiments, the processor may be a general-purpose processor (e.g., a general-purpose microprocessor or central processing unit (CPU) of the type used in desktop, laptop, or other computers). Alternatively, the processor may be a special-purpose processor. Examples of suitable special-purpose processors include, but are not limited to, network processors, communications processors, cryptographic processors, graphics processors, co-processors, embedded processors, digital signal processors (DSPs), and controllers (e.g., microcontrollers). The processor may have any of various complex instruction set computing (CISC) architectures, reduced instruction set computing (RISC) architectures, very long instruction word (VLIW) architectures, hybrid architectures, other types of architectures, or have a combination of different architectures (e.g., different cores may have different architectures).

During operation, the processor 310 may receive the Morton coordinate adjustment instruction 312. For example, the instruction may be fetched or otherwise received from memory on an interconnect. The instruction may represent a macroinstruction, assembly language instruction, machine code instruction, or other instruction or control signal of an instruction set of the processor.

Referring again to **Figure 3**, the processor includes a decode unit or decoder 314. The decode unit may receive and decode the Morton coordinate adjustment instruction. The Morton coordinate adjustment instruction may be part of an instruction set of the processor. The decode unit may output one or more relatively lower-level instructions or control signals (e.g., one or more microinstructions, micro-operations, micro-code entry points, decoded instructions or control signals, etc.), which reflect, represent, and/or are derived from the relatively higher-level Morton coordinate adjustment instruction. The decode unit may be implemented using various different mechanisms including, but not limited to, microcode read only memories (ROMs), look-up tables, hardware implementations, programmable logic arrays (PLAs), and other mechanisms suitable to implement decode units.

In some embodiments, instead of the Morton coordinate adjustment instruction being provided directly to the decode unit, an instruction emulator, translator, morpher, interpreter, or other instruction conversion module may optionally be used. Various types of suitable instruction conversion modules may be implemented in software, hardware, firmware, or a combination thereof. In some embodiments, the instruction conversion module may be located outside the processor, such as, for example, on a separate die and/or in a memory (e.g., as a

static, dynamic, or runtime emulation module). By way of example, the instruction conversion module may receive the Morton coordinate adjustment instruction, which may be of a first instruction set, and may emulate, translate, morph, interpret, or otherwise convert the Morton coordinate adjustment instruction into one or more corresponding intermediate instructions or control signals, which may be of a second different instruction set. The one or more intermediate instructions or control signals of the second instruction set may be provided to the decode unit, which may decode them into one or more lower-level instructions or control signals executable by native hardware of the processor (e.g., one or more execution units).

Referring again to **Figure 3**, the processor also includes a set of packed data registers 318. Each of the packed data registers may represent an on-die storage location that is operable to store packed data, vector data, or Single instruction, multiple data (SIMD) data. In SIMD architectures, a packed data instruction, vector instruction, or SIMD instruction may operate on multiple data elements or multiple pairs of data elements simultaneously or in parallel. The processor may have parallel execution hardware responsive to the packed data instruction to perform the multiple operations simultaneously or in parallel. Multiple data elements may be packed within one register or memory location as packed data or vector data. In packed data, the bits of the register or other storage location may be logically divided into a sequence of data elements. For example, a 256-bit wide packed data register may have four 64-bit wide data elements, eight 32-bit data elements, sixteen 16-bit data elements, etc. Each of the data elements may represent a separate individual piece of data (e.g., a pixel color, a coordinate, etc.), which may be operated upon separately and/or independently of the others. The packed data registers may represent architecturally-visible or architectural registers that are visible to software and/or a programmer and/or are the registers indicated by instructions of the instruction set of the processor to identify operands. These architectural registers are contrasted to other non-architectural registers in a given microarchitecture (e.g., temporary registers, reorder buffers, retirement registers, etc.). The packed data registers may be implemented in different ways in different microarchitectures using known techniques and are not limited to any particular type of design. Examples of suitable types of registers include, but are not limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, and combinations thereof.

In some embodiments, the instruction may explicitly specify (e.g., through one or more fields or a set of bits), or otherwise indicate (e.g., implicitly indicate), a source packed data operand 320. The source packed data operand may include a plurality of Morton coordinates. The instruction may also explicitly specify (e.g., through one or more fields or a set of bits), or

otherwise indicate a destination storage location where a result packed data operand 324 is to be stored. As one example, the instruction may have operand specification fields to specify registers, memory locations, or other storage locations for one or more of the source packed data operand and the destination storage location. Alternatively, one or more of the source packed data operand and the destination storage location may optionally be implicit to the instruction (e.g., implicit to an opcode of the instruction). As another option, a storage location used for the source packed data operand may also be reused as a destination storage location for the result packed data operand (e.g., it may be implicit to the instruction to use the same storage location initially for the source operand and later for the result operand). As shown, in some embodiments, the source packed data operand 320 may optionally be stored in a first packed data register and the result packed data operand may optionally be stored in a second packed data register, although this is not required. Alternatively, memory locations, or other storage locations, may optionally be used.

In some embodiments, the instruction may also specify or otherwise indicate a dimensionality of a multi-dimensional space having points that the Morton coordinates of the source packed data operand are mapped to. In some embodiments, the instruction may be operative to indicate the dimensionality of the multi-dimensional space as being any one of at least two different dimensions (e.g., any one of two dimensional and three dimensional), or in some embodiments as being any one of at least three different dimensions (e.g., any one of two dimensional, three dimensional, and four dimensional). In some embodiments, the instruction may have an immediate to indicate the dimensionality (e.g., 1-bit may indicate any one of at least two different dimensionalities, or 2-bits may indicate any one of at least three or at least four different dimensionalities). Alternatively, an instruction (e.g., an opcode) may be dedicated to a given dimensionality and the instruction (e.g., the opcode) may implicitly indicate the dimensionality. For example, an instruction set may include different instructions with different opcodes for different dimensionalities (e.g., two dimensional, three dimensional, four dimensional, etc.).

In some embodiments, the instruction may also specify or otherwise indicate a given dimension of the multi-dimensional space. The given dimension may represent the dimension in which there is to be a fixed change in coordinate position, such as, for example, a coordinate unit decrement in the given dimension. For example, the x-coordinate (or the y-coordinate) of a 2D, 3D, or 4D space or coordinate system may be decremented by one. In some embodiments, the instruction may have one or more bits (e.g., an immediate) to indicate the given dimension. For example, in 2D space 1-bit may be used to indicate the given dimension as either of the two

dimensions. As another example, in 3D or 4D space, 2-bits may be used to indicate the dimension as any one of the two or three respective dimensions.

Referring again to **Figure 3**, the execution unit 316 is coupled with the decode unit 314 and the packed data registers 318. The execution unit may receive the one or more decoded or
5 otherwise converted instructions or control signals that represent and/or are derived from the Morton coordinate adjustment instruction 312. The execution unit may also receive the source packed data operand 320 that is to include the Morton coordinates. The execution unit is operable in response to and/or as a result of the instruction (e.g., in response to one or more instructions or control signals decoded from the instruction) to store the result packed data
10 operand 324 in the destination storage location indicated by the instruction. The result packed data operand may include a plurality of Morton coordinates that are each to correspond to a different one of the Morton coordinates of the source packed data operand. In some embodiments, the Morton coordinates of the result packed data operand may be mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton
15 coordinates of the source packed data operand are mapped to by a fixed change in the given dimension indicated by the instruction. In some embodiments, the fixed change may be a unit decrement in coordinates in the given dimension (e.g., the x-coordinates of the points may be decremented by one (e.g., from x=7 to x=6)). In some embodiments, the Morton coordinates of the result packed data operand may be generated without de-interleaving the bits of the
20 corresponding Morton coordinates of the source packed data operand that correspond to different dimensions (e.g., without de-interleaving the different interleaved bits that correspond to the x and y-dimensions, or x, y, and z-dimensions). That is, the given coordinate may be decremented *in situ* or in place while in the Morton coordinate. In some embodiments, at least one Morton coordinate in the result packed data operand may not be in sequential Morton order with a
25 corresponding Morton coordinate of the source packed data operand (e.g., in Figure 2A the transition may be from the Morton coordinate at relative position 13 to the Morton coordinate at relative position 7 as shown at reference 205). In some embodiments, the result may be any of those shown and described for **Figures 5-7**, although the scope of the invention is not so limited.

The execution unit and/or the processor may include specific or particular logic (e.g.,
30 transistors, integrated circuitry, or other hardware potentially combined with firmware (e.g., instructions stored in non-volatile memory) and/or software) that is operable to perform the Morton coordinate adjustment instruction and/or store the result in response to and/or as a result of the Morton coordinate adjustment instruction (e.g., in response to one or more instructions or control signals decoded from the Morton coordinate adjustment instruction). By way of

example, the execution unit may include a logic unit, an arithmetic logic unit, or the like. In some embodiments, the execution unit may utilize one or more masks, one or more shift circuits, and one or more logical circuits (e.g., AND, NOT, etc.) to perform the instruction. Alternatively, other bit manipulation circuitry known in the arts may optionally be used.

5 To avoid obscuring the description, a relatively simple processor 310 has been shown and described. However, the processor may optionally include other components. Possible examples of such components include, but are not limited to, the components shown and described for any of **Figures 9** and/or any of **16-19**. Various different embodiments may include various different combinations and configurations of such components. Such
10 components may be coupled with one another in order to allow them to operate according to their operation. In some embodiments, all of the components may be included in at least one core, some cores, a subset of the cores, or all of the cores of the processor. In various embodiments, the processor may have at least one, two, four, eight, sixteen, thirty-two, or more cores.

15 **Figure 4** is a block flow diagram of an embodiment of a method 430 of performing an embodiment of a Morton coordinate adjustment instruction. In various embodiments, the method may be performed by a processor, instruction processing apparatus, or other digital logic device. In some embodiments, the method of **Figure 4** may be performed by and/or within the processor of **Figure 3**. The components, features, and specific optional details described herein
20 for the processor of **Figure 3**, also optionally apply to the method of **Figure 4**. Alternatively, the method of **Figure 4** may be performed by and/or within a similar or different processor or apparatus. Moreover, the processor of **Figure 3** may perform methods the same as, similar to, or different than those of **Figure 4**.

The method includes receiving the Morton coordinate adjustment instruction, at block 432.
25 In various aspects, the instruction may be received at a processor or a portion thereof (e.g., an instruction fetch unit, a decode unit, a bus interface unit, etc.). In various aspects, the instruction may be received from an off-processor and/or off-die source (e.g., from memory, interconnect, etc.), or from an on-processor and/or on-die source (e.g., from an instruction cache, instruction queue, etc.). The instruction may specify or otherwise indicate a source packed data operand
30 that includes a plurality of Morton coordinates, may specify or otherwise indicate a dimensionality of a multi-dimensional space having points that the Morton coordinates are mapped to, may specify or otherwise indicate a given dimension of the multi-dimensional space, and may specify or otherwise indicate a destination storage location.

A result packed data operand may be stored in the destination storage location, in response

to and/or as a result of the instruction (e.g., responsive to performing the instruction), at block 434. In some embodiments, the result packed data operand may include a plurality of Morton coordinates, which each correspond to a different one of the Morton coordinates of the source packed data operand (e.g., in a same bit position or relative position within the source and result operands). In some embodiments, the Morton coordinates of the result packed data operand may be mapped to points in the multi-dimensional space that may differ from the points that the corresponding Morton coordinates of the source packed data operand are mapped to by a fixed change in the given dimension that is indicated by the instruction.

In some embodiments, the fixed change may be a unit decrement (e.g., the x-coordinates of the points may be decremented by one). In some embodiments, the instruction may be operative to indicate the dimensionality of the multi-dimensional space as being any one of at least two different dimensions (e.g., two dimensional and three dimensional), or in some cases as being any one of at least three different dimensions (e.g., two dimensional, three dimensional, and four dimensional). In some embodiments, the instruction may have an immediate to indicate the dimensionality (e.g., 1-bit may indicate any one of at least two different dimensionalities, or 2-bits may indicate any one of at least three different dimensionalities). In some embodiments, the instruction may have one or more bits (e.g., an immediate) to indicate the given dimension (e.g., as being any one of at least two, three, or four different possible dimensions in various embodiments). In some embodiments, the Morton coordinates of the result packed data operand may be generated without de-interleaving the bits of the corresponding Morton coordinates of the source packed data operand that correspond to different dimensions (e.g., without de-interleaving the different interleaved bits that correspond to the x and y-dimensions, or x, y, and z-dimensions). In some embodiments, the source packed data operand and one or more result packed data operands may be any of those of **Figures 5-7**, although the scope of the invention is not so limited.

The illustrated method involves architectural operations (e.g., those visible from a software perspective). In other embodiments, the method may optionally include one or more microarchitectural operations. By way of example, the instruction may be prefetched, stored in an instruction cache, fetched by an instruction fetch unit, decoded, scheduled, source operands may be accessed, executed out-of-order with respect to other instructions, an execution unit may perform microarchitectural operations to implement the instruction, etc.

Figure 5 is a block diagram of an example embodiment of adjusting a source 3D Morton coordinate 508 to a result/adjusted 3D Morton coordinate 508* to reflect a decrement of a given or indicated dimension by one. The source 3D Morton coordinate has 9-bits with 3-bits used for

each of the x, y, and z-coordinates. As used herein, x and y, and z are used broadly to represent any desired coordinates or dimensions. They are not limited to any known quantities such as spatial coordinates in the familiar Cartesian coordinate system, but rather may be used for any desired quantities including potentially space, time, pressure, temperature, intensity, voltage, and the like. Specifically, x1 in bit-0, x2 in bit-3, and x3 in bit-6 correspond to the x-coordinate or dimension. Likewise, y1 in bit-1, y2 in bit-4, and y3 in bit-7 correspond to the y-dimension. Similarly, z1 in bit-2, z2 in bit-5, and z3 in bit-8 correspond to the z-dimension. Other embodiments may use other numbers of bits to represent the coordinates. For example, from 7-bits to 10-bits may be used if the 3D Morton coordinate is to be contained in a 32-bit data element. As another example, from 7-bits to 21-bits or from 12-bits to 21-bits may be used if the 3D Morton coordinate is to be contained in a 64-bit data element.

In the illustrated embodiment, the z-dimension is decremented. In other embodiments, either the x-dimension, or the y-dimension, may be decremented instead. Since the z-dimension is decremented, only bits of the source Morton coordinate corresponding to the z-dimension may need to be adjusted in order to effect the decrement in the z-dimension. The bits of the y-dimension and z-dimension remain unchanged.

In some embodiments, as shown at reference 540, to adjust the Morton coordinate to account for decrementing the given or indicated dimension (e.g., in this case the z-dimension), all least significant zeroed bits (i.e., those which are binary zero) corresponding to the given or indicated dimension, which are less significant than a least significant set bit corresponding to the given or indicated dimension, may be set. In other words, these trailing zeroed bits may be changed from initially being binary zero to being binary one. Additionally, as shown at reference 542, the least significant set bit corresponding to the given or indicated dimension may be cleared or zeroed (i.e., changed from initially being binary one to being binary zero). This effectively adjusts the Morton coordinate to reflect the given or indicated dimension being decremented by one.

Considering the specific example bit values shown in the illustration, in the source 3D Morton coordinate 508 z3 in bit-8 is the least significant set bit in the z-dimension, since z2 in bit-5 and z1 in bit-2 are both cleared or zeroed. Accordingly, as shown by reference 542, and by bit-8 of the result/adjusted 3D Morton coordinate 508*, z3 may be cleared or zeroed. Moreover, as shown by reference 540, and by bit-5 and bit-2 of the result/adjusted 3D Morton coordinate 508*, both z2 and z1 may be set to binary one. This is just one illustrative example, but analogous changes may be used for other bit strings and to represent decrements in other dimensions (e.g., x or y-dimensions).

Notice that the Morton coordinate may be adjusted or altered without needing to de-interleave, change, and re-interleave the bits of the various different dimensions or coordinates. Rather, the bits may be manipulated in situ or in place within the Morton coordinates. Advantageously, this may help to reduce the amount of computation and/or instructions needed and/or power consumption.

Figure 6 is a block diagram illustrating an example embodiment of a 2D Morton coordinate adjustment operation 650 to adjust 2D Morton coordinates (m) in a source packed data operand 620 into corresponding 2D Morton coordinates in a result packed data operand to reflect a decrement in a given indicated dimension. The operation may be performed in response to an example embodiment of a Morton coordinate adjustment instruction.

The instruction may specify or otherwise indicate the source packed data operand 620. The source packed data operand has a plurality of 2D Morton coordinates (m). In the particular illustrated embodiment, the source packed data operand has a number N 2D Morton coordinates, labeled m_1 through m_N . Commonly, each of the Morton coordinates may be stored in a different data element of the source packed data operand. In various embodiments, the width of the source packed data operand may be 64-bits, 128-bits, 256-bits, 512-bits, or 1024-bits, although the scope of the invention is not so limited. In some embodiments, the data elements of the source packed data operand may be 32-bit data elements or 64-bit data elements, although the scope of the invention is not so limited. The number of data elements and/or Morton coordinates in the source packed data operand may be the width in bits of the source packed data operand divided by the width in bits of each of the data elements. In various embodiments, there may be at least two, at least four, at least eight, at least sixteen, at least thirty-two, or more than thirty-two data elements and/or Morton coordinates in the source packed data operand.

In the illustration, example values for 6-bit 2D Morton coordinates are shown. Specifically, m_1 has a 6-bit 2D Morton coordinate of value "000100", m_2 has a 6-bit 2D Morton coordinate of value "000110", m_3 has a 6-bit 2D Morton coordinate of value "001100", and m_N has a 6-bit 2D Morton coordinate of value "001110". Each of these values is written in the format "y3x3y2x2y1x1" in which the least significant bits are on the right, and in which the least significant bit of the x-dimension is the first bit on the right.

In some embodiments, the instruction may also specify or otherwise indicate a dimensionality 646 of a multi-dimensional space having points that the Morton coordinates of the source packed data operand are mapped to. In this example, the instruction indicates a dimensionality of two dimensions (2D). In other words, the 2D Morton coordinates of the source packed data operand are mapped to points in a 2D space. Other embodiments may

indicate other dimensionalities (e.g., 3D, 4D, etc.). In various embodiments, the dimensionality may be flexibly indicated or specified by flexible fields or bits of the instruction, or may be implicit to the instruction (e.g., implicit to an opcode).

In some embodiments, the instruction may also specify or otherwise indicate a given dimension 648 that is to be decremented or changed. In this example, the instruction indicates that the x-dimension is to be decremented by one. In an alternate embodiment, the instruction may instead indicate that the y-dimension is to be decremented by one.

A result packed data operand 624 may be generated (e.g., by an execution unit 616) and stored in response to the instruction. The result packed data operand may be stored in a destination storage location that may be specified or otherwise indicated by the instruction. In various embodiments, the destination storage location may be a packed data register, memory location, or other storage location.

The result packed data operand may include a plurality of adjusted 2D Morton coordinates (m^*). In the particular illustrated embodiment, the source packed data operand has a number N adjusted 2D Morton coordinates, labeled m_1^* through m_N^* . Each of the adjusted 2D Morton coordinates (m^*) of the result packed data operand corresponds to a different one of the 2D Morton coordinates (m) of the source packed data operand (e.g., in corresponding bit positions and/or relative positions within the operands). In some embodiments, the adjusted 2D Morton coordinates (m^*) of the result packed data operand may be mapped to points in the multi-dimensional space that differ from the points that the corresponding 2D Morton coordinates (m) of the source packed data operand are to be mapped to by a fixed change (e.g., a decrement by one) in the given dimension (e.g., the x-dimension) indicated by the instruction.

As previously mentioned, in some embodiments, to adjust the Morton coordinate to account for decrementing the given or indicated dimension (e.g., in this case the x-dimension), all least significant zeroed bits (i.e., those which are binary zero) corresponding to the given or indicated dimension, which are less significant than a least significant set bit corresponding to the given or indicated dimension, may be set. In other words, these trailing zeroed bits may be changed from initially being binary zero to being binary one. Additionally, the least significant set bit corresponding to the given or indicated dimension may be cleared or zeroed (i.e., changed from initially being binary one to being binary zero). This effectively adjusts the 2D Morton coordinate to reflect the given or indicated dimension being decremented by one.

In the illustration, corresponding example values for the adjusted 6-bit 2D Morton coordinates (m^*) are shown. Specifically, m_1^* has a 6-bit 2D Morton coordinate of value "000001", m_2^* has a 6-bit 2D Morton coordinate of value "000011", m_3^* has a 6-bit 2D Morton

coordinate of value “001001”, and m_N^* has a 6-bit 2D Morton coordinate of value “001011”. These values are only examples, but illustrate how the bits corresponding to a given indicated dimension (e.g., in this example the x-dimension) may be selectively adjusted while keeping the bits corresponding to the other non-decremented dimensions the same.

5 **Figure 7** is a block diagram illustrating an example embodiment of a 3D Morton coordinate adjustment operation 752 to adjust 3D Morton coordinates (m) in a source packed data operand 720 into corresponding 3D Morton coordinates in a result packed data operand to reflect a decrement in a given indicated dimension. The operation may be performed in response to an example embodiment of a Morton coordinate adjustment instruction.

10 The operation of **Figure 7** has certain similarities to the operation of **Figure 6**, and differs primarily in that the coordinates are 3D instead of 2D. To avoid obscuring the description, the different and/or additional characteristics for the operation of **Figure 7** will primarily be described, without repeating all the optionally similar or common characteristics and details relative to the operation of **Figure 6**. However, it is to be appreciated that the previously
15 described characteristics and details of the operation of **Figure 6**, including the variations and alternatives, may also optionally apply to the operation of **Figure 7**, unless stated otherwise or otherwise clearly apparent.

As before, the instruction may specify or otherwise indicate the source packed data operand 720. The source packed data operand has a plurality of 3D Morton coordinates (m).
20 Aside from having 3D instead of 2D Morton coordinates, the source packed data operand may be similar to or the same as that of **Figure 6**. In the illustration, example values for 9-bit 3D Morton coordinates are shown. Specifically, m_1 has a 9-bit 3D Morton coordinate of value “000101000”, and m_N has a 9-bit 3D Morton coordinate of value “100111010”. As before, each of these values is written in the format “y3x3y2x2y1x1”.

25 In some embodiments, the instruction may also specify or otherwise indicate a dimensionality 746 of a multi-dimensional space having points that the Morton coordinates of the source packed data operand are mapped to. In this example, the instruction indicates a dimensionality of three dimensions (3D). In other words, the 3D Morton coordinates of the source packed data operand are mapped to points in a 3D space (e.g., a 3D Cartesian coordinate
30 system and/or 3D Euclidian space). Other embodiments may indicate other dimensionalities (e.g., 2D, 4D, 5D, etc.). In various embodiments, the dimensionality may be flexibly indicated or specified by flexible fields or bits of the instruction, or may be implicit to the instruction (e.g., implicit to an opcode).

In some embodiments, the instruction may also specify or otherwise indicate a given

dimension 748 that is to be decremented or changed. In this example, the instruction indicates that the x-dimension is to be decremented by one. In an alternate embodiment, the instruction may instead indicate that the y-dimension is to be decremented by one.

A result packed data operand 724 may be generated (e.g., by an execution unit 716) and stored in response to the instruction. The result packed data operand may include a plurality of adjusted 3D Morton coordinates m_1^* through m_N^* that each correspond to a different one of the 3D Morton coordinates (m) of the source packed data operand. In some embodiments, the adjusted 3D Morton coordinates (m^*) of the result packed data operand may be mapped to points in the multi-dimensional space that differ from the points that the corresponding 3D Morton coordinates (m) of the source packed data operand are to be mapped to by a fixed change (e.g., a decrement by one) in the given indicated dimension (e.g., the x-dimension).

The Morton coordinates may be adjusted as previously described. In the illustration, corresponding example values for the adjusted 9-bit 3D Morton coordinates (m^*) are shown. Specifically, m_1^* has a 9-bit 2D Morton coordinate of value "000100001" and m_N^* has a 9-bit 3D Morton coordinate of value "100110011". These values are only examples, but illustrate how the bits corresponding to a given indicated dimension (e.g., in this example the x-dimension) may be selectively adjusted while keeping the bits corresponding to the other non-decremented dimensions the same.

One possible example of pseudocode to implement an embodiment of a Morton coordinate adjustment instruction named `zorderprev` is shown below. In some embodiments, hardware or logic of an execution unit may be designed to implement the instruction using shifting logic, masking logic, and logical to implement logical operations (e.g., AND, NOT, etc.) operations similar to or the same as those shown in the pseudocode. Alternatively, those skilled in the art will appreciate other designs to implement the instruction based on the present disclosure.

```

25   zorderprev dst, src1, imm8
      dst=src1;
      unsigned long mask=1<<imm8[2:3];
      while (mask && !(mask&dst)) { //find trailing zeros
      dst ^= mask; //turn them into ones
30   mask = mask << imm8[0:1]; } //shift mask to next coord value
      if(mask) dst &= ~mask; //first non-zero becomes zero (borrow)

```

Figure 8 is a block diagram of a particular example embodiment of an immediate 854 that is suitable for one or more embodiments of a Morton coordinate adjustment instruction. One or more bits of the immediate may represent a dimensionality specifier 846. For example, in the

illustration two bits of the immediate, for example bits [1:0], may represent the dimensionality specifier 846. The dimensionality specifier may specify a dimensionality of a multi-dimensional space having points that Morton coordinates associated with the instruction are mapped to. In one embodiment, these two bits may have a first binary value (e.g., "10") to specify or indicate a dimensionality of 2D, a second different binary value (e.g., "11") to specify or indicate a dimensionality of 3D. In some embodiments, these two bits may optionally have a third still different binary value (e.g., "00") to specify or indicate 4D. In some embodiments, these two bits may optionally have a fourth still different binary value (e.g., "01") to specify or indicate a still different dimensionality (e.g., 5D). In other embodiments, the instruction may be operative to allow only one dimensionality, or only two different dimensionalities. These different dimensionalities of the multi-dimensional space may also be considered dimensionalities of the Morton coordinates (e.g., whether the Morton coordinates are 2D, 3D, 4D, etc.).

One or more bits of the immediate may represent a dimension specifier 848. For example, in the illustration two bits of the immediate, for example bits [3:2], may represent the dimension specifier 848. The dimension specifier may specify or indicate a dimension that is to be changed (e.g., decremented). In one embodiment, these two bits may have a first binary value (e.g., "00") to specify or indicate a first dimension (e.g., an x-dimension), may have a second different binary value (e.g., "01") to specify or indicate a second dimension (e.g., a y-dimension), and may have a third still different binary value (e.g., "10") to specify or indicate a third dimension (e.g., a z-dimension). Alternatively, if the instruction is only to support two different dimensions a single bit may optionally be used to indicate either of these two different dimensions. In some embodiments (e.g., in the case of 4D), these two bits may have a fourth still different binary value (e.g., "11") to specify or indicate a fourth dimension (e.g., a t-dimension).

The illustrated immediate is an 8-bit immediate, although the scope of the invention is not so limited. Alternate embodiments may use 2-bit, 4-bit, 6-bit, 16-bit, or other immediates. In other embodiments, either fewer or more bits may be used for each of the dimensionality specifier and/or the dimension specifier. Moreover, different bits than those shown may optionally be used.

Figure 9 is a block diagram of a more detailed example embodiment of a suitable processor that is operable to perform an embodiment of a Morton coordinate adjustment instruction. The processor includes a core 980. The processor may optionally have multiple cores (e.g. at least two, at least four, at least eight, at least thirty, etc.). In some cases, all of the cores may be the same as the core shown, or in other cases some of the cores may be different (e.g., have different components or instruction sets). In some cases, all of the cores may be able

to execute a Morton coordinate adjustment instruction as disclosed herein, or in other cases only some of the cores may. The core 980 includes a branch prediction unit 981 to predict branches program flow. The branch prediction unit is coupled with an instruction prefetch unit 982. The instruction prefetch unit may prefetch or otherwise receive instructions, including the Morton coordinate adjustment instruction, from external memory through a memory unit 989 coupled therewith. A level 1 (L1) instruction cache is coupled with the instruction prefetch unit. The L1 instruction cache may cache or otherwise store prefetched or otherwise received instructions including the Morton coordinate adjustment instruction. An instruction fetch unit 984 is coupled with the L1 instruction cache and a decode unit 914. The instruction fetch unit may fetch or otherwise receive instructions, including the Morton coordinate adjustment instruction, from the L1 instruction cache and provide the instructions, including the Morton coordinate adjustment instruction, to the decode unit. The decode unit may be the same as, or similar to, the other decode units described herein.

The processor includes one or more register file units 918. Commonly, the register file unit(s) may include various different types of registers, such as, for example, packed data registers, general-purpose registers, a status or flags register, control or configuration registers, etc. In an embodiment using out-of-order (OOO) execution, the processor may also optionally include a register rename and/or allocator unit coupled with the register file unit(s) to allocate resources and perform register renaming on registers (e.g., packed data registers associated with the Morton coordinate adjustment instruction). Also, in the case of OOO execution, the processor may optionally include one or more scheduler units 986 coupled with the decode unit, the rename/allocation unit, and one or more execution units 916. The scheduler unit(s) may schedule operations on the execution units. At least one of the execution units may be the same as, or similar to, other execution units disclosed herein. Commonly, the processor may optionally have multiple different types of execution units, such as, for example integer execution units, floating point execution units, vector execution units, one or more memory access units 987 or bus interface units, and the like. In an embodiment using out-of-order (OOO) execution, the processor may also optionally include a retirement or commit unit 991 coupled with the register file unit(s) and the rename/allocator unit 985 to retire or commit instructions. The processor may include an L1 data cache unit 974 to cache or otherwise store data including data elements and/or operands for instructions including the Morton coordinate adjustment instruction. A level 2 (L2) cache unit 990 may optionally be included and may optionally be shared by multiple cores. The L2 cache unit may store data and instructions including the Morton coordinate adjustment instruction. The processor may also optionally

include one or more translation lookaside buffers (not shown) to cache address translation data. In an embodiment using out-of-order (OOO) execution, the processor may also optionally include a reorder buffer (not shown) to reorder execution results and/or one or more reservation stations (not shown). Various embodiments of processors may include various different combinations and configurations of some or all of these components. Embodiments are not limited to any known such combination or configuration.

Figure 10 is a block diagram of an embodiment of a Morton coordinate adjustment instruction 1012. The instruction includes an operation code or opcode 1080. The opcode may represent a plurality of bits or one or more fields that are operable to identify the instruction and/or the operation to be performed (e.g., to adjust a Morton coordinate based on a unit decrement of a given coordinate in a multi-dimensional space or multi-dimensional coordinate system).

The instruction also includes a source operand specifier 1082 to explicitly specify a register, memory location, or other storage location used to store a source packed data operand. The instruction also includes an optional destination storage location specifier 1084 to explicitly specify a destination register or other destination storage location to store a result packed data operand. By way of example, each of these specifiers may include a set of bits or one or more fields to explicitly specify an address of a register, memory location, or other storage location. Alternatively, instead of the instruction having explicit specifiers for each of these storage locations, the instruction may optionally have one or more implicit storage locations (e.g., implicit to an opcode of the instruction) for one or more of the source operand and the destination storage locations. For example, it may be implicit to an opcode of the instruction to use a given fixed register for one of the source operand and the destination storage location so that the given fixed register doesn't need to be explicitly specified. As another example, it may be implicit to reuse the same register or other storage location (e.g., explicitly specified once by the instruction) for both the source operand and as a destination storage location to store the result packed data operand (e.g., an implicit source/destination register).

In some embodiments, the instruction may optionally include a coordinate specifier 1086. The coordinate specifier may specify a particular coordinate of a 2D, 3D, 4D, or other multi-dimensional space that is to be decremented or reduced. For example, the coordinate specifier may specify that the x-coordinate is to be reduced, or that the y-coordinate is to be reduced. In some embodiments, the instruction may optionally include a dimensionality specifier 1088. The dimensionality specifier may specify a dimensionality of the Morton curve or coordinates (e.g., whether the Morton coordinates in the source packed data operand are 2D, 3D, 4D, or other

multi-dimensional Morton coordinates. In some embodiments, the coordinate specifier 1086 and the dimensionality specifier 1088 may optionally be included in an immediate of the instruction (e.g., a 4-bit, 6-bit, or 8-bit immediate). Alternatively, one or more of them may be other non-immediate bits of the instruction. In yet another alternate embodiment, one or more of the coordinate and/or the dimensionality may optionally be fixed for the instruction (e.g., fixed for an opcode). For example, one instruction (e.g., one opcode) may be provided for one dimensionality (e.g., 2D), another instruction (e.g., another opcode) may be provided for another dimensionality (e.g., 3D), and so on. As another example, one instruction (e.g., one opcode) may be provided for one coordinate (e.g., the x-coordinate), another instruction (e.g., another opcode) may be provided for another coordinate (e.g., the y-coordinate), and so on.

It is to be appreciated that this is just one illustrative example of a suitable instruction. Alternate embodiments may include a subset of the illustrated fields/specifiers, may add additional fields/specifiers, may overlap certain fields/specifiers, etc. For example, the instruction may optionally have other fields or bits, such as, for example, a bit to indicate a 32-bit or 64-bit execution state or mode. In addition, the illustrated order and arrangement of the fields/specifiers is not required. The fields/specifiers may be rearranged variously. In addition, fields/specifiers need not include contiguous sequences of bits, but rather may include non-contiguous or separated bits. In some embodiments, the instruction format may have a VEX or EVEX encoding or instruction format and attributes as disclosed elsewhere herein, although the scope of the invention is not so limited. Further details on VEX and EVEX encodings and formats are discussed further below.

An instruction set includes one or more instruction formats. A given instruction format defines various fields (number of bits, location of bits) to specify, among other things, the operation to be performed (opcode) and the operand(s) on which that operation is to be performed. Some instruction formats are further broken down though the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of

this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands. A set of SIMD extensions referred to the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme, has been , has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developers Manual, October 2011; and see Intel® Advanced Vector Extensions Programming Reference, June 2011).

Exemplary Instruction Formats

Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

VEX Instruction Format

VEX encoding allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 128 bits. The use of a VEX prefix provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as $A = A + B$, which overwrites a source operand. The use of a VEX prefix enables operands to perform nondestructive operations such as $A = B + C$.

Figure 11A illustrates an exemplary AVX instruction format including a VEX prefix 1102, real opcode field 1130, Mod R/M byte 1140, SIB byte 1150, displacement field 1162, and IMM8 1172. **Figure 11B** illustrates which fields from **Figure 11A** make up a full opcode field 1174 and a base operation field 1142. **Figure 11C** illustrates which fields from **Figure 11A** make up a register index field 1144.

VEX Prefix (Bytes 0-2) 1102 is encoded in a three-byte form. The first byte is the Format Field 1140 (VEX Byte 0, bits [7:0]), which contains an explicit C4 byte value (the unique value used for distinguishing the C4 instruction format). The second-third bytes (VEX Bytes 1-2) include a number of bit fields providing specific capability. Specifically, REX field 1105 (VEX Byte 1, bits [7-5]) consists of a VEX.R bit field (VEX Byte 1, bit [7] – R), VEX.X bit field (VEX byte 1, bit [6] – X), and VEX.B bit field (VEX byte 1, bit[5] – B). Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding VEX.R, VEX.X, and VEX.B. Opcode map field 1115 (VEX byte 1, bits [4:0] – mmmmm) includes content to encode an implied leading opcode byte. W Field 1164 (VEX byte 2, bit [7] – W) - is represented by the notation VEX.W, and provides different functions depending on the instruction. The role of VEX.vvvv 1120 (VEX Byte 2, bits [6:3]-vvvv) may include the following: 1) VEX.vvvv

encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) VEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. If VEX.L 1168 Size field (VEX byte 2, bit [2]-L) = 0, it indicates 128 bit vector; if VEX.L = 1, it indicates 256 bit vector. Prefix encoding field 125 (VEX byte 2, bits [1:0]-pp) provides additional bits for the base operation field.

Real Opcode Field 1130 (Byte 3) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field 1140 (Byte 4) includes MOD field 1142 (bits [7-6]), Reg field 1144 (bits [5-3]), and R/M field 1146 (bits [2-0]). The role of Reg field 1144 may include the following: encoding either the destination register operand or a source register operand (the rrr of Rrrr), or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 1146 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) - The content of Scale field 1150 (Byte 5) includes SS1152 (bits [7-6]), which is used for memory address generation. The contents of SIB.xxx 1154 (bits [5-3]) and SIB.bbb 1156 (bits [2-0]) have been previously referred to with regard to the register indexes Xxxx and Bbbb.

The Displacement Field 1162 and the immediate field (IMM8) 1172 contain address data.

Generic Vector Friendly Instruction Format

A vector friendly instruction format is an instruction format that is suited for vector instructions (e.g., there are certain fields specific to vector operations). While embodiments are described in which both vector and scalar operations are supported through the vector friendly instruction format, alternative embodiments use only vector operations the vector friendly instruction format.

Figures 12A-12B are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. **Figure 12A** is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention; while **Figure 12B** is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention. Specifically, a generic vector friendly instruction format 1200 for which are defined class A and class B instruction templates, both of which include no memory access 1205 instruction templates and memory access 1220

instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

While embodiments of the invention will be described in which the vector friendly instruction format supports the following: a 64 byte vector operand length (or size) with 32 bit (4
 5 byte) or 64 bit (8 byte) data element widths (or sizes) (and thus, a 64 byte vector consists of either 16 doubleword-size elements or alternatively, 8 quadword-size elements); a 64 byte vector operand length (or size) with 16 bit (2 byte) or 8 bit (1 byte) data element widths (or sizes); a 32
 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit
 10 (1 byte) data element widths (or sizes); and a 16 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); alternative embodiments may support more, less and/or different vector operand sizes (e.g., 256
 byte vector operands) with more, less, or different data element widths (e.g., 128 bit (16 byte)
 data element widths).

The class A instruction templates in **Figure 12A** include: 1) within the no memory access
 15 1205 instruction templates there is shown a no memory access, full round control type operation
 1210 instruction template and a no memory access, data transform type operation 1215
 instruction template; and 2) within the memory access 1220 instruction templates there is shown
 a memory access, temporal 1225 instruction template and a memory access, non-temporal 1230
 instruction template. The class B instruction templates in **Figure 12B** include: 1) within the no
 20 memory access 1205 instruction templates there is shown a no memory access, write mask
 control, partial round control type operation 1212 instruction template and a no memory access,
 write mask control, vsize type operation 1217 instruction template; and 2) within the memory
 access 1220 instruction templates there is shown a memory access, write mask control 1227
 instruction template.

25 The generic vector friendly instruction format 1200 includes the following fields listed
 below in the order illustrated in **Figures 12A-12B**.

Format field 1240 – a specific value (an instruction format identifier value) in this field
 uniquely identifies the vector friendly instruction format, and thus occurrences of instructions in
 the vector friendly instruction format in instruction streams. As such, this field is optional in the
 30 sense that it is not needed for an instruction set that has only the generic vector friendly
 instruction format.

Base operation field 1242 – its content distinguishes different base operations.

Register index field 1244 – its content, directly or through address generation, specifies the
 locations of the source and destination operands, be they in registers or in memory. These

include a sufficient number of bits to select N registers from a PxQ (e.g. 32x512, 16x128, 32x1024, 64x1024) register file. While in one embodiment N may be up to three sources and one destination register, alternative embodiments may support more or less sources and destination registers (e.g., may support up to two sources where one of these sources also acts as the destination, may support up to three sources where one of these sources also acts as the destination, may support up to two sources and one destination).

Modifier field 1246 – its content distinguishes occurrences of instructions in the generic vector instruction format that specify memory access from those that do not; that is, between no memory access 1205 instruction templates and memory access 1220 instruction templates. Memory access operations read and/or write to the memory hierarchy (in some cases specifying the source and/or destination addresses using values in registers), while non-memory access operations do not (e.g., the source and destinations are registers). While in one embodiment this field also selects between three different ways to perform memory address calculations, alternative embodiments may support more, less, or different ways to perform memory address calculations.

Augmentation operation field 1250 – its content distinguishes which one of a variety of different operations to be performed in addition to the base operation. This field is context specific. In one embodiment of the invention, this field is divided into a class field 1268, an alpha field 1252, and a beta field 1254. The augmentation operation field 1250 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

Scale field 1260 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base}$).

Displacement Field 1262A– its content is used as part of memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{displacement}$).

Displacement Factor Field 1262B (note that the juxtaposition of displacement field 1262A directly over displacement factor field 1262B indicates one or the other is used) – its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N) – where N is the number of bytes in the memory access (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{scaled displacement}$). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field 1274 (described later herein) and the data manipulation field 1254C. The displacement field 1262A and the displacement factor field 1262B are optional in the sense that

they are not used for the no memory access 1205 instruction templates and/or different embodiments may implement only one or none of the two.

Data element width field 1264 – its content distinguishes which one of a number of data element widths is to be used (in some embodiments for all instructions; in other embodiments for only some of the instructions). This field is optional in the sense that it is not needed if only one data element width is supported and/or data element widths are supported using some aspect of the opcodes.

Write mask field 1270 – its content controls, on a per data element position basis, whether that data element position in the destination vector operand reflects the result of the base operation and augmentation operation. Class A instruction templates support merging-writemasking, while class B instruction templates support both merging- and zeroing-writemasking. When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the write mask field 1270 allows for partial vector operations, including loads, stores, arithmetic, logical, etc. While embodiments of the invention are described in which the write mask field's 1270 content selects one of a number of write mask registers that contains the write mask to be used (and thus the write mask field's 1270 content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's 1270 content to directly specify the masking to be performed.

Immediate field 1272 – its content allows for the specification of an immediate. This field is optional in the sense that is it not present in an implementation of the generic vector friendly format that does not support immediate and it is not present in instructions that do not use an immediate.

Class field 1268 – its content distinguishes between different classes of instructions. With reference to **Figures 12A-B**, the contents of this field select between class A and class B instructions. In **Figures 12A-B**, rounded corner squares are used to indicate a specific value is

present in a field (e.g., class A 1268A and class B 1268B for the class field 1268 respectively in **Figures 12A-B**).

Instruction Templates of Class A

In the case of the non-memory access 1205 instruction templates of class A, the alpha field 1252 is interpreted as an RS field 1252A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 1252A.1 and data transform 1252A.2 are respectively specified for the no memory access, round type operation 1210 and the no memory access, data transform type operation 1215 instruction templates), while the beta field 1254 distinguishes which of the operations of the specified type is to be performed. In the no memory access 1205 instruction templates, the scale field 1260, the displacement field 1262A, and the displacement scale field 1262B are not present.

No-Memory Access Instruction Templates – Full Round Control Type Operation

In the no memory access full round control type operation 1210 instruction template, the beta field 1254 is interpreted as a round control field 1254A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field 1254A includes a suppress all floating point exceptions (SAE) field 1256 and a round operation control field 1258, alternative embodiments may support may encode both these concepts into the same field or only have one or the other of these concepts/fields (e.g., may have only the round operation control field 1258).

SAE field 1256 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 1256 content indicates suppression is enabled, a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler.

Round operation control field 1258 – its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 1258 allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 1250 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

In the no memory access data transform type operation 1215 instruction template, the beta field 1254 is interpreted as a data transform field 1254B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

In the case of a memory access 1220 instruction template of class A, the alpha field 1252 is

interpreted as an eviction hint field 1252B, whose content distinguishes which one of the eviction hints is to be used (in **Figure 12A**, temporal 1252B.1 and non-temporal 1252B.2 are respectively specified for the memory access, temporal 1225 instruction template and the memory access, non-temporal 1230 instruction template), while the beta field 1254 is interpreted
 5 as a data manipulation field 1254C, whose content distinguishes which one of a number of data manipulation operations (also known as primitives) is to be performed (e.g., no manipulation; broadcast; up conversion of a source; and down conversion of a destination). The memory access 1220 instruction templates include the scale field 1260, and optionally the displacement field 1262A or the displacement scale field 1262B.

10 Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

Memory Access Instruction Templates – Temporal

15 Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Memory Access Instruction Templates – Non-Temporal

20 Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Instruction Templates of Class B

25 In the case of the instruction templates of class B, the alpha field 1252 is interpreted as a write mask control (Z) field 1252C, whose content distinguishes whether the write masking controlled by the write mask field 1270 should be a merging or a zeroing.

30 In the case of the non-memory access 1205 instruction templates of class B, part of the beta field 1254 is interpreted as an RL field 1257A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 1257A.1 and vector length (VSIZE) 1257A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 1212 instruction template and the no memory access, write mask control, VSIZE type operation 1217 instruction template), while the rest of the beta field 1254 distinguishes which of the operations of the specified type is to be performed. In the no memory access 1205 instruction templates, the scale field 1260, the displacement field 1262A, and the displacement scale field 1262B are not present.

In the no memory access, write mask control, partial round control type operation 1210 instruction template, the rest of the beta field 1254 is interpreted as a round operation field 1259A and exception event reporting is disabled (a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler).

5 Round operation control field 1259A – just as round operation control field 1258, its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 1259A allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying
10 rounding modes, the round operation control field's 1250 content overrides that register value.

In the no memory access, write mask control, VSIZE type operation 1217 instruction template, the rest of the beta field 1254 is interpreted as a vector length field 1259B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

15 In the case of a memory access 1220 instruction template of class B, part of the beta field 1254 is interpreted as a broadcast field 1257B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 1254 is interpreted the vector length field 1259B. The memory access 1220 instruction templates include the scale field 1260, and optionally the displacement field 1262A or the
20 displacement scale field 1262B.

With regard to the generic vector friendly instruction format 1200, a full opcode field 1274 is shown including the format field 1240, the base operation field 1242, and the data element width field 1264. While one embodiment is shown where the full opcode field 1274 includes all of these fields, the full opcode field 1274 includes less than all of these fields in embodiments
25 that do not support all of them. The full opcode field 1274 provides the operation code (opcode).

The augmentation operation field 1250, the data element width field 1264, and the write mask field 1270 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

The combination of write mask field and data element width field create typed instructions
30 in that they allow the mask to be applied based on different data element widths.

The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing

may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor
5 may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance general purpose cores with out of order execution and register renaming intended for general-
10 purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and class B. Of course, features from one class may also be implement in the other class in different embodiments of the invention. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into an variety of different
15 executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently executing the code.

20 Exemplary Specific Vector Friendly Instruction Format

Figure 13 is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. **Figure 13** shows a specific vector friendly instruction format 1300 that is specific in the sense that it specifies the location, size, interpretation, and order of the fields, as well as values for some of those fields. The specific
25 vector friendly instruction format 1300 may be used to extend the x86 instruction set, and thus some of the fields are similar or the same as those used in the existing x86 instruction set and extension thereof (e.g., AVX). This format remains consistent with the prefix encoding field, real opcode byte field, MOD R/M field, SIB field, displacement field, and immediate fields of the existing x86 instruction set with extensions. The fields from **Figure 12** into which the fields
30 from **Figure 13** map are illustrated.

It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 1300 in the context of the generic vector friendly instruction format 1200 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 1300 except where claimed. For example, the

generic vector friendly instruction format 1200 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 1300 is shown as having fields of specific sizes. By way of specific example, while the data element width field 1264 is illustrated as a one bit field in the specific vector friendly instruction format 1300, the invention is not so limited (that is, the generic vector friendly instruction format 1200 contemplates other sizes of the data element width field 1264).

The generic vector friendly instruction format 1200 includes the following fields listed below in the order illustrated in **Figure 13A**.

EVEX Prefix (Bytes 0-3) 1302 - is encoded in a four-byte form.

Format Field 1240 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 1240 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

REX field 1305 (EVEX Byte 1, bits [7-5]) – consists of a EVEX.R bit field (EVEX Byte 1, bit [7] – R), EVEX.X bit field (EVEX byte 1, bit [6] – X), and 1257BEX byte 1, bit[5] – B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 1111B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

REX' field 1210 – this is the first part of the REX' field 1210 and is the EVEX.R' bit field (EVEX Byte 1, bit [4] - R') that is used to encode either the upper 16 or lower 16 of the extended 32 register set. In one embodiment of the invention, this bit, along with others as indicated below, is stored in bit inverted format to distinguish (in the well-known x86 32-bit mode) from the BOUND instruction, whose real opcode byte is 62, but does not accept in the MOD R/M field (described below) the value of 11 in the MOD field; alternative embodiments of the invention do not store this and the other indicated bits below in the inverted format. A value of 1 is used to encode the lower 16 registers. In other words, R'Rrrr is formed by combining EVEX.R', EVEX.R, and the other RRR from other fields.

Opcode map field 1315 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

Data element width field 1264 (EVEX byte 2, bit [7] – W) - is represented by the notation EVEX.W. EVEX.W is used to define the granularity (size) of the datatype (either 32-bit data

elements or 64-bit data elements).

EVEX.vvvv 1320 (EVEX Byte 2, bits [6:3]-vvvv)- the role of EVEX.vvvv may include the following: 1) EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) EVEX.vvvv
 5 encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) EVEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. Thus, EVEX.vvvv field 1320 encodes the 4 low-order bits of the first source register specifier stored in inverted (1s complement) form. Depending on the instruction, an extra different EVEX bit field is used to extend the specifier size to 32 registers.

10 EVEX.U 1268 Class field (EVEX byte 2, bit [2]-U) – If EVEX.U = 0, it indicates class A or EVEX.U0; if EVEX.U = 1, it indicates class B or EVEX.U1.

Prefix encoding field 1325 (EVEX byte 2, bits [1:0]-pp) – provides additional bits for the base operation field. In addition to providing support for the legacy SSE instructions in the EVEX prefix format, this also has the benefit of compacting the SIMD prefix (rather than
 15 requiring a byte to express the SIMD prefix, the EVEX prefix requires only 2 bits). In one embodiment, to support legacy SSE instructions that use a SIMD prefix (66H, F2H, F3H) in both the legacy format and in the EVEX prefix format, these legacy SIMD prefixes are encoded into the SIMD prefix encoding field; and at runtime are expanded into the legacy SIMD prefix prior to being provided to the decoder's PLA (so the PLA can execute both the legacy and EVEX
 20 format of these legacy instructions without modification). Although newer instructions could use the EVEX prefix encoding field's content directly as an opcode extension, certain embodiments expand in a similar fashion for consistency but allow for different meanings to be specified by these legacy SIMD prefixes. An alternative embodiment may redesign the PLA to support the 2 bit SIMD prefix encodings, and thus not require the expansion.

25 Alpha field 1252 (EVEX byte 3, bit [7] – EH; also known as EVEX.EH, EVEX.rs, EVEX.RL, EVEX.write mask control, and EVEX.N; also illustrated with α) – as previously described, this field is context specific.

Beta field 1254 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s₂₋₀, EVEX.r₂₋₀, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with $\beta\beta\beta$) – as previously described, this
 30 field is context specific.

REX' field 1210 – this is the remainder of the REX' field and is the EVEX.V' bit field (EVEX Byte 3, bit [3] - V') that may be used to encode either the upper 16 or lower 16 of the extended 32 register set. This bit is stored in bit inverted format. A value of 1 is used to encode the lower 16 registers. In other words, V'VVVV is formed by combining EVEX.V',

EVEX.vvvv.

Write mask field 1270 (EVEX byte 3, bits [2:0]-kkk) – its content specifies the index of a register in the write mask registers as previously described. In one embodiment of the invention, the specific value EVEX.kkk=000 has a special behavior implying no write mask is used for the particular instruction (this may be implemented in a variety of ways including the use of a write mask hardwired to all ones or hardware that bypasses the masking hardware).

Real Opcode Field 1330 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field 1340 (Byte 5) includes MOD field 1342, Reg field 1344, and R/M field 1346. As previously described, the MOD field's 1342 content distinguishes between memory access and non-memory access operations. The role of Reg field 1344 can be summarized to two situations: encoding either the destination register operand or a source register operand, or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 1346 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 1250 content is used for memory address generation. SIB.xxx 1354 and SIB.bbb 1356 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

Displacement field 1262A (Bytes 7-10) – when MOD field 1342 contains 10, bytes 7-10 are the displacement field 1262A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

Displacement factor field 1262B (Byte 7) – when MOD field 1342 contains 01, byte 7 is the displacement factor field 1262B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since disp8 is sign extended, it can only address between -128 and 127 bytes offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values -128, -64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field 1262B is a reinterpretation of disp8; when using displacement factor field 1262B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is based on the assumption that the effective displacement is multiple of the granularity of the

memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field 1262B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 1262B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

Immediate field 1272 operates as previously described.

10 **Full Opcode Field**

Figure 13B is a block diagram illustrating the fields of the specific vector friendly instruction format 1300 that make up the full opcode field 1274 according to one embodiment of the invention. Specifically, the full opcode field 1274 includes the format field 1240, the base operation field 1242, and the data element width (W) field 1264. The base operation field 1242 includes the prefix encoding field 1325, the opcode map field 1315, and the real opcode field 1330.

Register Index Field

Figure 13C is a block diagram illustrating the fields of the specific vector friendly instruction format 1300 that make up the register index field 1244 according to one embodiment of the invention. Specifically, the register index field 1244 includes the REX field 1305, the REX' field 1310, the MODR/M.reg field 1344, the MODR/M.r/m field 1346, the VVVV field 1320, xxx field 1354, and the bbb field 1356.

Augmentation Operation Field

Figure 13D is a block diagram illustrating the fields of the specific vector friendly instruction format 1300 that make up the augmentation operation field 1250 according to one embodiment of the invention. When the class (U) field 1268 contains 0, it signifies EVEX.U0 (class A 1268A); when it contains 1, it signifies EVEX.U1 (class B 1268B). When U=0 and the MOD field 1342 contains 11 (signifying a no memory access operation), the alpha field 1252 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 1252A. When the rs field 1252A contains a 1 (round 1252A.1), the beta field 1254 (EVEX byte 3, bits [6:4]– SSS) is interpreted as the round control field 1254A. The round control field 1254A includes a one bit SAE field 1256 and a two bit round operation field 1258. When the rs field 1252A contains a 0 (data transform 1252A.2), the beta field 1254 (EVEX byte 3, bits [6:4]– SSS) is interpreted as a three bit data transform field 1254B. When U=0 and the MOD field 1342 contains 00, 01, or 10

(signifying a memory access operation), the alpha field 1252 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 1252B and the beta field 1254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 1254C.

When U=1, the alpha field 1252 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 1252C. When U=1 and the MOD field 1342 contains 11 (signifying a no memory access operation), part of the beta field 1254 (EVEX byte 3, bit [4]- S₀) is interpreted as the RL field 1257A; when it contains a 1 (round 1257A.1) the rest of the beta field 1254 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the round operation field 1259A, while when the RL field 1257A contains a 0 (VSIZE 1257.A2) the rest of the beta field 1254 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the vector length field 1259B (EVEX byte 3, bit [6-5]- L₁₋₀). When U=1 and the MOD field 1342 contains 00, 01, or 10 (signifying a memory access operation), the beta field 1254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 1259B (EVEX byte 3, bit [6-5]- L₁₋₀) and the broadcast field 1257B (EVEX byte 3, bit [4]- B).

Exemplary Register Architecture

Figure 14 is a block diagram of a register architecture 1400 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 1410 that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format 1300 operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 1259B	A (Figure 12A; U=0)	1210, 1215, 1225, 1230	zmm registers (the vector length is 64 byte)
	B (Figure 12B; U=1)	1212	zmm registers (the vector length is 64 byte)
Instruction templates that do include the vector length field 1259B	B (Figure 12B; U=1)	1217, 1227	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 1259B

In other words, the vector length field 1259B selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding length; and instructions templates without the vector length field 1259B operate on the maximum

vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 1300 operate on packed or scalar single/double-precision floating point data and packed or scalar integer data. Scalar operations are operations performed on the lowest order data element position in an zmm/ymm/xmm register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

Write mask registers 1415 - in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers 1415 are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

General-purpose registers 1425 - in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

Scalar floating point stack register file (x87 stack) 1445, on which is aliased the MMX packed integer flat register file 1450 - in the embodiment illustrated, the x87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Exemplary Core Architectures, Processors, and Computer Architectures

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special

purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by descriptions of exemplary processors and computer architectures.

Exemplary Core Architectures

In-order and out-of-order core block diagram

Figure 15A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. **Figure 15B** is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in **Figures 15A-B** illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

In **Figure 15A**, a processor pipeline 1500 includes a fetch stage 1502, a length decode stage 1504, a decode stage 1506, an allocation stage 1508, a renaming stage 1510, a scheduling (also known as a dispatch or issue) stage 1512, a register read/memory read stage 1514, an execute stage 1516, a write back/memory write stage 1518, an exception handling stage 1522, and a commit stage 1524.

Figure 15B shows processor core 1590 including a front end unit 1530 coupled to an execution engine unit 1550, and both are coupled to a memory unit 1570. The core 1590 may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core 1590 may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit 1530 includes a branch prediction unit 1532 coupled to an instruction

cache unit 1534, which is coupled to an instruction translation lookaside buffer (TLB) 1536, which is coupled to an instruction fetch unit 1538, which is coupled to a decode unit 1540. The decode unit 1540 (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control
5 signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit 1540 may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core 1590 includes a microcode ROM or other medium that stores
10 microcode for certain macroinstructions (e.g., in decode unit 1540 or otherwise within the front end unit 1530). The decode unit 1540 is coupled to a rename/allocator unit 1552 in the execution engine unit 1550.

The execution engine unit 1550 includes the rename/allocator unit 1552 coupled to a retirement unit 1554 and a set of one or more scheduler unit(s) 1556. The scheduler unit(s) 1556
15 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 1556 is coupled to the physical register file(s) unit(s) 1558. Each of the physical register file(s) units 1558 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point,, status
20 (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. In one embodiment, the physical register file(s) unit 1558 comprises a vector registers unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) 1558 is overlapped by the retirement unit 1554 to illustrate various ways in which
25 register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit 1554 and the physical register file(s) unit(s) 1558 are coupled to the execution cluster(s) 1560. The execution cluster(s) 1560 includes a set of one or more execution units 1562 and a set of one or
30 more memory access units 1564. The execution units 1562 may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or

multiple execution units that all perform all functions. The scheduler unit(s) 1556, physical register file(s) unit(s) 1558, and execution cluster(s) 1560 are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster – and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) 1564). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

The set of memory access units 1564 is coupled to the memory unit 1570, which includes a data TLB unit 1572 coupled to a data cache unit 1574 coupled to a level 2 (L2) cache unit 1576. In one exemplary embodiment, the memory access units 1564 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 1572 in the memory unit 1570. The instruction cache unit 1534 is further coupled to a level 2 (L2) cache unit 1576 in the memory unit 1570. The L2 cache unit 1576 is coupled to one or more other levels of cache and eventually to a main memory.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 1500 as follows: 1) the instruction fetch 1538 performs the fetch and length decoding stages 1502 and 1504; 2) the decode unit 1540 performs the decode stage 1506; 3) the rename/allocator unit 1552 performs the allocation stage 1508 and renaming stage 1510; 4) the scheduler unit(s) 1556 performs the schedule stage 1512; 5) the physical register file(s) unit(s) 1558 and the memory unit 1570 perform the register read/memory read stage 1514; the execution cluster 1560 perform the execute stage 1516; 6) the memory unit 1570 and the physical register file(s) unit(s) 1558 perform the write back/memory write stage 1518; 7) various units may be involved in the exception handling stage 1522; and 8) the retirement unit 1554 and the physical register file(s) unit(s) 1558 perform the commit stage 1524.

The core 1590 may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 1590 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units 1534/1574 and a shared L2 cache unit 1576, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

Specific Exemplary In-Order Core Architecture

Figures 16A-B illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip. The logic blocks communicate through a high-bandwidth interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

Figure 16A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1602 and with its local subset of the Level 2 (L2) cache 1604, according to embodiments of the invention. In one embodiment, an instruction decoder 1600 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 1606 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 1608 and a vector unit 1610 use separate register sets (respectively, scalar registers 1612 and vector registers 1614) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 1606, alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

The local subset of the L2 cache 1604 is part of a global L2 cache that is divided into separate local subsets, one per processor core. Each processor core has a direct access path to its own local subset of the L2 cache 1604. Data read by a processor core is stored in its L2 cache

subset 1604 and can be accessed quickly, in parallel with other processor cores accessing their own local L2 cache subsets. Data written by a processor core is stored in its own L2 cache subset 1604 and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1012-bits wide per direction.

Figure 16B is an expanded view of part of the processor core in **Figure 16A** according to embodiments of the invention. **Figure 16B** includes an L1 data cache 1606A part of the L1 cache 1604, as well as more detail regarding the vector unit 1610 and the vector registers 1614. Specifically, the vector unit 1610 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 1628), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit 1620, numeric conversion with numeric convert units 1622A-B, and replication with replication unit 1624 on the memory input. Write mask registers 1626 allow predicating resulting vector writes.

Processor with integrated memory controller and graphics

Figure 17 is a block diagram of a processor 1700 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in **Figure 17** illustrate a processor 1700 with a single core 1702A, a system agent 1710, a set of one or more bus controller units 1716, while the optional addition of the dashed lined boxes illustrates an alternative processor 1700 with multiple cores 1702A-N, a set of one or more integrated memory controller unit(s) 1714 in the system agent unit 1710, and special purpose logic 1708.

Thus, different implementations of the processor 1700 may include: 1) a CPU with the special purpose logic 1708 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 1702A-N being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores 1702A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 1702A-N being a large number of general purpose in-order cores. Thus, the processor 1700 may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor 1700 may be a part of and/or may be

implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 1706, and external memory (not shown) coupled to the set of integrated memory controller units 1714. The set of shared cache units 1706 may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based interconnect unit 1712 interconnects the integrated graphics logic 1708, the set of shared cache units 1706, and the system agent unit 1710/integrated memory controller unit(s) 1714, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units 1706 and cores 1702-A-N.

In some embodiments, one or more of the cores 1702A-N are capable of multi-threading. The system agent 1710 includes those components coordinating and operating cores 1702A-N. The system agent unit 1710 may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores 1702A-N and the integrated graphics logic 1708. The display unit is for driving one or more externally connected displays.

The cores 1702A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 1702A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Computer Architectures

Figures 18-21 are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

Referring now to **Figure 18**, shown is a block diagram of a system 1800 in accordance with one embodiment of the present invention. The system 1800 may include one or more processors 1810, 1815, which are coupled to a controller hub 1820. In one embodiment the

controller hub 1820 includes a graphics memory controller hub (GMCH) 1890 and an Input/Output Hub (IOH) 1850 (which may be on separate chips); the GMCH 1890 includes memory and graphics controllers to which are coupled memory 1840 and a coprocessor 1845; the IOH 1850 is coupled to input/output (I/O) devices 1860 to the GMCH 1890. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 1840 and the coprocessor 1845 are coupled directly to the processor 1810, and the controller hub 1820 in a single chip with the IOH 1850.

The optional nature of additional processors 1815 is denoted in **Figure 18** with broken lines. Each processor 1810, 1815 may include one or more of the processing cores described herein and may be some version of the processor 1700.

The memory 1840 may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment, the controller hub 1820 communicates with the processor(s) 1810, 1815 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 1895.

In one embodiment, the coprocessor 1845 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like. In one embodiment, controller hub 1820 may include an integrated graphics accelerator.

There can be a variety of differences between the physical resources 1810, 1815 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

In one embodiment, the processor 1810 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 1810 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 1845. Accordingly, the processor 1810 issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 1845. Coprocessor(s) 1845 accept and execute the received coprocessor instructions.

Referring now to **Figure 19**, shown is a block diagram of a first more specific exemplary system 1900 in accordance with an embodiment of the present invention. As shown in **Figure 19**, multiprocessor system 1900 is a point-to-point interconnect system, and includes a first processor 1970 and a second processor 1980 coupled via a point-to-point interconnect 1950. Each of processors 1970 and 1980 may be some version of the processor 1700. In one

embodiment of the invention, processors 1970 and 1980 are respectively processors 1810 and 1815, while coprocessor 1938 is coprocessor 1845. In another embodiment, processors 1970 and 1980 are respectively processor 1810 coprocessor 1845.

Processors 1970 and 1980 are shown including integrated memory controller (IMC) units
5 1972 and 1982, respectively. Processor 1970 also includes as part of its bus controller units point-to-point (P-P) interfaces 1976 and 1978; similarly, second processor 1980 includes P-P interfaces 1986 and 1988. Processors 1970, 1980 may exchange information via a point-to-point (P-P) interface 1950 using P-P interface circuits 1978, 1988. As shown in **Figure 19**, IMCs 1972 and 1982 couple the processors to respective memories, namely a memory 1932 and a
10 memory 1934, which may be portions of main memory locally attached to the respective processors.

Processors 1970, 1980 may each exchange information with a chipset 1990 via individual P-P interfaces 1952, 1954 using point to point interface circuits 1976, 1994, 1986, 1998. Chipset 1990 may optionally exchange information with the coprocessor 1938 via a high-performance
15 interface 1939. In one embodiment, the coprocessor 1938 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both
20 processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

Chipset 1990 may be coupled to a first bus 1916 via an interface 1996. In one embodiment, first bus 1916 may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of
25 the present invention is not so limited.

As shown in **Figure 19**, various I/O devices 1914 may be coupled to first bus 1916, along with a bus bridge 1918 which couples first bus 1916 to a second bus 1920. In one embodiment, one or more additional processor(s) 1915, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal
30 processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus 1916. In one embodiment, second bus 1920 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 1920 including, for example, a keyboard and/or mouse 1922, communication devices 1927 and a storage unit 1928 such as a disk drive or other mass storage device which may include instructions/code and data 1930, in one embodiment.

Further, an audio I/O 1924 may be coupled to the second bus 1920. Note that other architectures are possible. For example, instead of the point-to-point architecture of **Figure 19**, a system may implement a multi-drop bus or other such architecture.

Referring now to **Figure 20**, shown is a block diagram of a second more specific exemplary system 2000 in accordance with an embodiment of the present invention. Like elements in **Figures 19 and 20** bear like reference numerals, and certain aspects of **Figure 19** have been omitted from **Figure 20** in order to avoid obscuring other aspects of **Figure 20**.

Figure 20 illustrates that the processors 1970, 1980 may include integrated memory and I/O control logic ("CL") 1972 and 1982, respectively. Thus, the CL 1972, 1982 include integrated memory controller units and include I/O control logic. **Figure 20** illustrates that not only are the memories 1932, 1934 coupled to the CL 1972, 1982, but also that I/O devices 2014 are also coupled to the control logic 1972, 1982. Legacy I/O devices 2015 are coupled to the chipset 1990.

Referring now to **Figure 21**, shown is a block diagram of a SoC 2100 in accordance with an embodiment of the present invention. Similar elements in **Figure 17** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 21**, an interconnect unit(s) 2102 is coupled to: an application processor 2110 which includes a set of one or more cores 202A-N and shared cache unit(s) 1706; a system agent unit 1710; a bus controller unit(s) 1716; an integrated memory controller unit(s) 1714; a set of one or more coprocessors 2120 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 2130; a direct memory access (DMA) unit 2132; and a display unit 2140 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 2120 include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the invention may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

Program code, such as code 1930 illustrated in **Figure 19**, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For

purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as “IP cores” may be stored on a tangible, machine readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable’s (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

Emulation (including binary translation, code morphing, etc.)

In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware,

firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

Figure 22 is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. **Figure 22** shows a program in a high level language 2202 may be compiled using an x86 compiler 2204 to generate x86 binary code 2206 that may be natively executed by a processor with at least one x86 instruction set core 2216. The processor with at least one x86 instruction set core 2216 represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler 2204 represents a compiler that is operable to generate x86 binary code 2206 (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core 2216. Similarly, **Figure 22** shows the program in the high level language 2202 may be compiled using an alternative instruction set compiler 2208 to generate alternative instruction set binary code 2210 that may be natively executed by a processor without at least one x86 instruction set core 2214 (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, CA). The instruction converter 2212 is used to convert the x86 binary code 2206 into code that may be natively executed by the processor without an x86 instruction set core 2214. This converted code is not likely to be the same as the alternative instruction set binary code 2210 because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter 2212 represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute the x86 binary code 2206.

Components, features, and details described for any of **Figures 5-10** may also optionally apply to any of **Figures 3-4**. Moreover, components, features, and details described for any of

the apparatus may also optionally apply to any of the methods, which in embodiments may be performed by and/or with such apparatus. Any of the processors described herein may be included in any of the systems-on-chips (SoCs) and/or computer systems disclosed herein and/or in other computer systems (e.g., desktops or cell phones) and/or other SoCs. In some
5 embodiments, the computer system may include an interconnect, a processor coupled with the interconnect, and a dynamic random access memory (DRAM) coupled with the interconnect. Alternatively, instead of DRAM, other types of volatile memory that don't need to be refreshed may be used, or flash memory may be used. Examples of other components that may optionally be integrated or included on-die or on-chip with the processor in a SoC include, but are not
10 limited to, a memory controller, memory (e.g., read only memory (ROM), random access memory (RAM), and/or flash memory), a graphics device (e.g., a graphics processing unit (GPU), graphics core, or graphics processor, etc.), a digital signal processor, an image signal processor, a liquid crystal display (LCD) or other display controller, and one or more input and/or output (I/O) controllers to provide control for one or more external interfaces (e.g.,
15 universal serial bus (USB), universal asynchronous receiver/transmitter (UART), FireWire, Ethernet, Bluetooth, etc.). SoC for cellular phones may also optionally include a long term evolution (LTE) modem, a sensor engine, and a satellite navigation system controller (e.g., global positioning satellite (GPS), Galileo, and/or Beidou). In some embodiments, the instructions may have features or details of the instruction formats disclosed herein, although this
20 is not required.

In the description and claims, the terms "coupled" and/or "connected," along with their derivatives, may have be used. These terms are not intended as synonyms for each other. Rather, in embodiments, "connected" may be used to indicate that two or more elements are in direct physical and/or electrical contact with each other. "Coupled" may mean that two or more
25 elements are in direct physical and/or electrical contact with each other. However, "coupled" may also mean that two or more elements are not in direct contact with each other, but yet still co-operate or interact with each other. For example, an execution unit may be coupled with a register and/or a decode unit through one or more intervening components. In the figures, arrows are used to show connections and couplings.

30 The term "and/or" may have been used. As used herein, the term "and/or" means one or the other or both (e.g., A and/or B means A or B or both A and B).

In the description above, specific details have been set forth in order to provide a thorough understanding of the embodiments. However, other embodiments may be practiced without some of these specific details. The scope of the invention is not to be determined by the specific

examples provided above, but only by the claims below. In other instances, well-known circuits, structures, devices, and operations have been shown in block diagram form and/or without detail in order to avoid obscuring the understanding of the description. Where considered appropriate, reference numerals, or terminal portions of reference numerals, have been repeated among the
5 figures to indicate corresponding or analogous elements, which may optionally have similar or the same characteristics, unless specified or clearly apparent otherwise.

Certain operations may be performed by hardware components, or may be embodied in machine-executable or circuit-executable instructions, that may be used to cause and/or result in a machine, circuit, or hardware component (e.g., a processor, portion of a processor, circuit, etc.)
10 programmed with the instructions performing the operations. The operations may also optionally be performed by a combination of hardware and software. A processor, machine, circuit, or hardware may include specific or particular circuitry or other logic (e.g., hardware potentially combined with firmware and/or software) is operable to execute and/or process the instruction and store a result in response to the instruction.

Some embodiments include an article of manufacture (e.g., a computer program product) that includes a machine-readable medium. The medium may include a mechanism that provides, for example stores, information in a form that is readable by the machine. The machine-readable medium may provide, or have stored thereon, an instruction or sequence of instructions, that if and/or when executed by a machine are operable to cause the machine to perform and/or result in
20 the machine performing one or operations, methods, or techniques disclosed herein.

In some embodiments, the machine-readable medium may include a non-transitory machine-readable storage medium. For example, the non-transitory machine-readable storage medium may include a floppy diskette, an optical storage medium, an optical disk, an optical data storage device, a CD-ROM, a magnetic disk, a magneto-optical disk, a read only memory (ROM), a programmable ROM (PROM), an erasable-and-programmable ROM (EPROM), an electrically-erasable-and-programmable ROM (EEPROM), a random access memory (RAM), a static-RAM (SRAM), a dynamic-RAM (DRAM), a Flash memory, a phase-change memory, a phase-change data storage material, a non-volatile memory, a non-volatile data storage device, a non-transitory memory, a non-transitory data storage device, or the like. The non-transitory
25 machine-readable storage medium does not consist of a transitory propagated signal. In some embodiments, the storage medium may include a tangible medium that includes solid matter.

Examples of suitable machines include, but are not limited to, a general-purpose processor, a special-purpose processor, a digital logic circuit, an integrated circuit, a system-on-chip (SoC), or the like. Still other examples of suitable machines include a computer system or other

electronic device that includes a processor, a digital logic circuit, or an integrated circuit. Examples of such computer systems or electronic devices include, but are not limited to, desktop computers, laptop computers, notebook computers, tablet computers, netbooks, cellular phones (e.g., smartphones), servers, network devices (e.g., routers and switches.), Mobile Internet
5 devices (MIDs), media players, smart televisions, nettops, set-top boxes, and video game controllers.

Reference throughout this specification to "one embodiment," "an embodiment," "one or more embodiments," "some embodiments," for example, indicates that a particular feature may be included in the practice of the invention but is not necessarily required to be. Similarly, in the
10 description various features are sometimes grouped together in a single embodiment, Figure, or description thereof for the purpose of streamlining the disclosure and aiding in the understanding of various inventive aspects. This method of disclosure, however, is not to be interpreted as reflecting an intention that the invention requires more features than are expressly recited in each claim. Rather, as the following claims reflect, inventive aspects lie in less than all features of a
15 single disclosed embodiment. Thus, the claims following the Detailed Description are hereby expressly incorporated into this Detailed Description, with each claim standing on its own as a separate embodiment of the invention.

EXAMPLE EMBODIMENTS

The following examples pertain to further embodiments. Specifics in the examples may be
20 used anywhere in one or more embodiments.

Example 1 is a processor that includes a plurality of packed data registers, and a decode unit to decode an instruction. The instruction is to indicate a source packed data operand that is to include a plurality of Morton coordinates, optionally a dimensionality of a multi-dimensional space having points that the Morton coordinates are to be mapped to, and a given dimension of
25 the multi-dimensional space, and a destination storage location. The processor also includes an execution unit coupled with the packed data registers and the decode unit. The execution unit, in response to the decode unit decoding the instruction, is to store a result packed data operand in the destination storage location. The result packed data operand is to include a plurality of Morton coordinates that are each to correspond to a different one of the Morton coordinates of
30 the source packed data operand. The Morton coordinates of the result packed data operand are to be mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the source packed data operand are to be mapped to by a fixed change in the given dimension indicated by the instruction.

Example 2 includes the processor of Example 1, in which the execution unit is to store the

result packed data operand in which the fixed change in the given dimension is a unit decrement in the given dimension.

Example 3 includes the processor of Example 1, in which the execution unit is to generate each Morton coordinate of the result packed data operand by setting all lowest order (or least significant) zeroed bits corresponding to the given dimension of each corresponding Morton coordinate of the source packed data operand, which are lower order (or less significant) than each lowest order set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand. Also, clearing the lowest order (or least significant) set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand.

Example 4 includes the processor of any one of Examples 1 to 3, in which the decode unit is to decode the instruction which is to indicate the dimensionality of the multi-dimensional space as being any one of at least two dimensional and three dimensional.

Example 5 includes the processor of Example 4, in which the decode unit is to decode the instruction which is to indicate the dimensionality of the multi-dimensional space as being any one of said two dimensional, said three dimensional, and optionally four dimensional.

Example 6 includes the processor of any one of Examples 1 to 3, in which the decode unit is to decode the instruction which is to have an immediate to indicate the dimensionality of the multi-dimensional space.

Example 7 includes the processor of any one of Examples 1 to 3, in which the decode unit is to decode the instruction which is to have an opcode that is to indicate the dimensionality of the multi-dimensional space.

Example 8 includes the processor of any one of Examples 1 to 3, in which the decode unit is to decode the instruction which is optionally to have an immediate or other set of bits that is to include at least two bits to indicate the given dimension as being any one of at least two different dimensions.

Example 9 includes the processor of any one of Examples 1 to 3, in which the decode unit is to decode the instruction which is to have at least two bits to indicate the given dimension as being any one of at least three or in some cases optionally four different dimensions.

Example 10 includes the processor of any one of Examples 1 to 9, in which the execution unit is to generate the Morton coordinates of the result packed data operand without de-interleaving bits of the Morton coordinates of the source packed data operand that correspond to different dimensions.

Example 11 includes the processor of any one of Examples 1 to 10, in which the execution

unit is to store the result packed data operand having at least one Morton coordinate that is not to be in sequential Morton order with a corresponding Morton coordinate of the source packed data operand.

Example 12 includes the processor of any one of Examples 1 to 11, further including an optional branch prediction unit to predict branches. An optional instruction prefetch unit coupled with the branch prediction unit. The instruction prefetch unit to prefetch instructions including the instruction. An optional level 1 (L1) instruction cache coupled with the instruction prefetch unit. The L1 instruction cache to store instructions including the instruction. An optional L1 data cache to store data. An optional level 2 (L2) cache coupled with the L1 instruction cache and the L1 data cache. The L2 cache to store data and instructions. An optional instruction fetch unit coupled with the L1 instruction cache and the decode unit to fetch the instruction from the L1 instruction cache and provide the instruction to the decode unit. An optional register rename unit coupled with the packed data registers to rename the packed data registers.

Example 13 is a method in a processor that includes receiving an instruction. The instruction indicating a source packed data operand that includes a plurality of Morton coordinates, a dimensionality of a multi-dimensional space having points that the Morton coordinates are mapped to, a given dimension of the multi-dimensional space, and a destination storage location. The method also includes storing a result packed data operand in the destination storage location responsive to performing the instruction. The result packed data operand includes a plurality of Morton coordinates that each correspond to a different one of the Morton coordinates of the source packed data operand. The Morton coordinates of the result packed data operand are mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the source packed data operand are mapped to by a fixed change in the given dimension indicated by the instruction.

Example 14 includes the method of Example 13, in which storing the result packed data operand includes storing the result packed data operand in which the fixed change is a unit decrement in the given dimension.

Example 15 includes the method of Example 13, further including generating each Morton coordinate of the result packed data operand by setting all lowest order zeroed bits corresponding to the given dimension of each corresponding Morton coordinate of the source packed data operand, which are lower order than each lowest order set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand. Also, clearing the lowest order set bit corresponding to the given dimension of each Morton coordinate of the

source packed data operand.

Example 16 includes the method of Example 13, in which receiving includes receiving the instruction that is able to indicate the dimensionality of the multi-dimensional space as being any one of at least two dimensional and three dimensional.

5 Example 17 includes the method of Example 13, in which receiving includes receiving the instruction having an immediate that indicates the dimensionality of the multi-dimensional space.

Example 18 includes the method of Example 13, in which receiving includes receiving the instruction having an opcode that indicates the dimensionality of the multi-dimensional space.

10 Example 19 includes the method of Example 13, in which receiving includes receiving the instruction having at least two bits that are able to indicate the given dimension as being any one of at least three different dimensions.

Example 20 includes the method of any one of Examples 13 to 19, further including generating the Morton coordinates of the result packed data operand without de-interleaving bits of the Morton coordinates of the source packed data operand that correspond to different dimensions.

Example 21 is a system to process instructions including an interconnect, and a processor coupled with the interconnect. The processor is to receive an instruction that is to indicate a source packed data operand that is to include a plurality of Morton coordinates, a given dimension of a multi-dimensional space having points that the Morton coordinates are to be mapped to, and a destination packed data register. The processor, in response to the instruction, is to store a result packed data operand in the destination packed data register. The result packed data operand is to include a plurality of Morton coordinates, which are each to correspond to a different one of the Morton coordinates of the source packed data operand. The Morton coordinates of the result packed data operand are to be mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the source packed data operand are to be mapped to by a decrement of one in the given dimension. The system also includes a dynamic random access memory (DRAM) coupled with the interconnect.

30 Example 22 includes the system of Example 19, in which the processor is responsive to the instruction which allows a dimensionality of the multi-dimensional space to be any one of a plurality of different dimensionalities.

Example 23 is an article of manufacture including a non-transitory machine-readable storage medium. The non-transitory machine-readable storage medium storing an instruction.

The instruction is to indicate a source packed data operand that is to include a plurality of Morton coordinates. A given dimension of a multi-dimensional space having points that the Morton coordinates are to be mapped to, and a destination storage location. The instruction if executed by a machine is to cause the machine to perform operations including storing a result packed data operand in the destination storage location. The result packed data operand is to include a plurality of Morton coordinates that are each to correspond to a different one of the Morton coordinates of the source packed data operand. The Morton coordinates of the result packed data operand are to be mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the source packed data operand are to be mapped to by a decrement of one in the given dimension indicated by the instruction.

Example 24 includes the article of manufacture of Example 21, in which an opcode of the instruction allows a plurality of different dimensionalities of the multi-dimensional space, and in which the instruction has one or more bits to indicate a dimensionality of the multi-dimensional space.

Example 25 is a processor or other apparatus that is operative to perform the method of any one of Examples 13 to 20.

Example 26 is a processor or other apparatus that includes means for performing the method of any one of Examples 13 to 20.

Example 27 is a processor or other apparatus that includes any combination of modules and/or units and/or logic and/or circuitry and/or means for performing the method of any one of Examples 13 to 20.

Example 28 is an article of manufacture that includes an optionally non-transitory machine-readable medium, which optionally stores or otherwise provides an instruction, which if and/or when executed by a processor, computer system, electronic device, or other machine, is operative to cause the machine to perform the method of any one of Examples 13 to 20.

Example 29 is a computer system or other electronic device including an interconnect, the processor of any one of claims 1 to 12 coupled with the interconnect, an optional dynamic random access memory (DRAM) coupled with the interconnect, an optional wireless transceiver coupled with the interconnect, an optional image sensor coupled with the interconnect, and an optional touchscreen controller coupled with the interconnect.

Example 30 is a processor or other apparatus substantially as described herein.

Example 31 is a processor or other apparatus that is operative to perform any method substantially as described herein.

1. A processor comprising:
 - a plurality of packed data registers;
 - a decode unit to decode an instruction, wherein the instruction is to indicate a source packed data operand that is to include a plurality of Morton coordinates, a dimensionality of a multi-dimensional space having points that the Morton coordinates are to be mapped to, a given dimension of the multi-dimensional space, and a destination storage location; and
 - an execution unit coupled with the packed data registers and the decode unit, the execution unit, in response to the decode unit decoding the instruction, to store a result packed data operand in the destination storage location, wherein the result packed data operand is to include a plurality of Morton coordinates that are each to correspond to a different one of the Morton coordinates of the source packed data operand, wherein the Morton coordinates of the result packed data operand are to be mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the source packed data operand are to be mapped to by a fixed change in the given dimension indicated by the instruction.
2. The processor of claim 1, wherein the execution unit is to store the result packed data operand in which the fixed change in the given dimension is a unit decrement in the given dimension.
3. The processor of claim 1, wherein the execution unit is to generate each Morton coordinate of the result packed data operand by:
 - setting all lowest order zeroed bits corresponding to the given dimension of each corresponding Morton coordinate of the source packed data operand, which are lower order than each lowest order set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand; and
 - clearing the lowest order set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand.
4. The processor of any one of claims 1 to 3, wherein the decode unit is to decode the instruction which is to indicate the dimensionality of the multi-dimensional space as being any one of at least two dimensional and three dimensional.
5. The processor of claim 4, wherein the decode unit is to decode the instruction which is to indicate the dimensionality of the multi-dimensional space as being any one of said two dimensional, said three dimensional, and four dimensional.
6. The processor of any one of claims 1 to 3, wherein the decode unit is to decode the instruction which is to have an immediate to indicate the dimensionality of the multi-dimensional space.

7. The processor of any one of claims 1 to 3, wherein the decode unit is to decode the instruction which is to have an opcode that is to indicate the dimensionality of the multi-dimensional space.

8. The processor of any one of claims 1 to 3, wherein the decode unit is to decode the instruction which is to have an immediate that is to include at least two bits to indicate the given dimension as being any one of at least two different dimensions.

9. The processor of any one of claims 1 to 3, wherein the decode unit is to decode the instruction which is to have at least two bits to indicate the given dimension as being any one of at least three different dimensions.

10. The processor of any one of claims 1 to 3, wherein the execution unit is to generate the Morton coordinates of the result packed data operand without de-interleaving bits of the Morton coordinates of the source packed data operand that correspond to different dimensions.

11. The processor of any one of claims 1 to 3, wherein the execution unit is to store the result packed data operand having at least one Morton coordinate that is not to be in sequential Morton order with a corresponding Morton coordinate of the source packed data operand.

12. The processor of any one of claims 1 to 3, further comprising:
 a branch prediction unit to predict branches;
 an instruction prefetch unit coupled with the branch prediction unit, the instruction prefetch unit to prefetch instructions including the instruction;
 a level 1 (L1) instruction cache coupled with the instruction prefetch unit, the L1 instruction cache to store instructions including the instruction;
 an L1 data cache to store data;
 a level 2 (L2) cache coupled with the L1 instruction cache and the L1 data cache, the L2 cache to store data and instructions;
 an instruction fetch unit coupled with the L1 instruction cache and the decode unit to fetch the instruction from the L1 instruction cache and provide the instruction to the decode unit; and
 a register rename unit coupled with the packed data registers to rename the packed data registers.

13. A method in a processor comprising:
 receiving a instruction, the instruction indicating a source packed data operand that includes a plurality of Morton coordinates, a dimensionality of a multi-dimensional space having points that the Morton coordinates are mapped to, a given dimension of the multi-dimensional

space, and a destination storage location; and

storing a result packed data operand in the destination storage location responsive to performing the instruction, the result packed data operand including a plurality of Morton coordinates that each correspond to a different one of the Morton coordinates of the source packed data operand, wherein the Morton coordinates of the result packed data operand are mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the source packed data operand are mapped to by a fixed change in the given dimension indicated by the instruction.

14. The method of claim 13, wherein storing the result packed data operand comprises storing the result packed data operand in which the fixed change is a unit decrement in the given dimension.

15. The method of claim 13, further comprising generating each Morton coordinate of the result packed data operand by:

setting all lowest order zeroed bits corresponding to the given dimension of each corresponding Morton coordinate of the source packed data operand, which are lower order than each lowest order set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand; and

clearing the lowest order set bit corresponding to the given dimension of each Morton coordinate of the source packed data operand.

16. The method of claim 13, wherein receiving comprises receiving the instruction that is able to indicate the dimensionality of the multi-dimensional space as being any one of at least two dimensional and three dimensional.

17. The method of claim 13, wherein receiving comprises receiving the instruction having an immediate that indicates the dimensionality of the multi-dimensional space.

18. The method of claim 13, wherein receiving comprises receiving the instruction having an opcode that indicates the dimensionality of the multi-dimensional space.

19. The method of claim 13, wherein receiving comprises receiving the instruction having at least two bits that are able to indicate the given dimension as being any one of at least three different dimensions.

20. The method of claim 13, further comprising generating the Morton coordinates of the result packed data operand without de-interleaving bits of the Morton coordinates of the source packed data operand that correspond to different dimensions.

21. An article of manufacture comprising a non-transitory machine-readable storage medium, the non-transitory machine-readable storage medium storing an instruction,

the instruction to indicate a source packed data operand that is to include a plurality of Morton coordinates, a given dimension of a multi-dimensional space having points that the Morton coordinates are to be mapped to, and a destination storage location, and the instruction if executed by a machine is to cause the machine to perform operations comprising:

- 5 storing a result packed data operand in the destination storage location, wherein the result packed data operand is to include a plurality of Morton coordinates that are each to correspond to a different one of the Morton coordinates of the source packed data operand, wherein the Morton coordinates of the result packed data operand are to be mapped to points in the multi-dimensional space that differ from the points that the corresponding Morton coordinates of the
- 10 source packed data operand are to be mapped to by a decrement of one in the given dimension indicated by the instruction.

22. The article of manufacture of claim 21, wherein an opcode of the instruction allows a plurality of different dimensionalities of the multi-dimensional space, and wherein the instruction has one or more bits to indicate a dimensionality of the multi-dimensional space.

- 15 23. An apparatus comprising means for performing the method of any one of claims 13 to 20.

24. An article of manufacture comprising a machine-readable medium, which stores an instruction, which if executed by a machine, is operative to cause the machine to perform the method of any one of claims 13 to 20.

- 20 25. An electronic device comprising an interconnect, the processor of any one of claims 1 to 12 coupled with the interconnect, and a dynamic random access memory (DRAM) coupled with the interconnect.