(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
07 September 2018 (07.09.2018)

WIPO | PCT

(10) International Publication Number
WO 2018/158603 A1

(51) International Patent Classification:
G06F 9/30 (2018.01) *G06F 9/38* (2018.01)

(21) International Application Number:
PCT/IB2017/000333

(22) International Filing Date:
28 February 2017 (28.02.2017)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95054 (US).

(72) Inventors: **PLOTNIKOV, Mikhail**; Turgeneva Str. 30, Nizhny Novgorod 603024 (RU). **ERMOLAEV, Igor**; Fruktovaja 5/3, 67, Nizhny Novgorod 603093 (RU).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:
— *of inventorship (Rule 4.17(iv))*

Published:
— *with international search report (Art. 21(3))*

WO 2018/158603 A1

(54) Title: STRIDESHIFT INSTRUCTION FOR TRANSPOSING BITS INSIDE VECTOR REGISTER

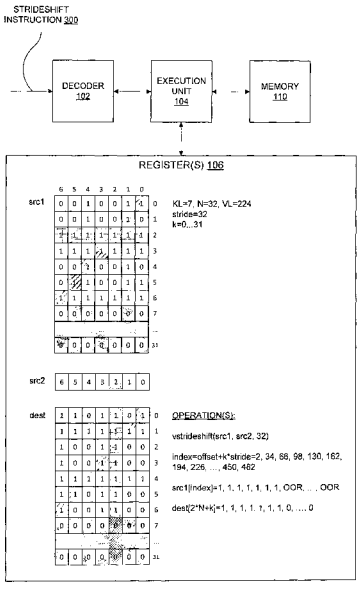


Fig. 3

(57) Abstract: A processor includes a decode circuit to decode an instruction into a decoded instruction and an execution circuit to execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

STRIDESHIFT INSTRUCTION FOR TRANSPOSING BITS INSIDE VECTOR REGISTER

TECHNICAL FIELD

[0001] Embodiments of the invention relate to the field of computer instruction set architecture; and more specifically, to a strideshift instruction for transposing bits inside a vector register.

BACKGROUND

[0002] A processor or set of processors, executes instructions from an instruction set, e.g., the instruction set architecture (ISA). The instruction set is the part of the computer architecture related to programming, and generally includes the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). It should be noted that the term instruction as used herein generally refers to a macro-instruction (e.g., an instruction that is provided to the processor for execution), as opposed to a micro-instruction (e.g., an instruction that results from a processor's decoder decoding macro-instructions).

[0003] Modern processors often include instructions to provide operations that are computationally intensive, but offer a high level of data parallelism that can be exploited through an efficient implementation using various data storage devices, such as for example, single-instruction multiple-data (SIMD) vector registers. In SIMD execution, a single instruction operates on multiple data elements concurrently or simultaneously. This is typically implemented by extending the width of various resources such as registers and arithmetic logic units (ALUs), allowing them to hold and operate on multiple data elements, respectively.

[0004] Determining the maximum and minimum values contained in a vector may be useful for various purposes. One conventional technique for determining the maximum value contained in a vector extracts the upper half of the vector to another register, determines the maximum value using a pairwise maxps instruction between two vectors, extracts the upper half of the half-vector, and repeats this process until the maxps instruction is performed on just two elements. The minimum value contained in a vector can be determined in similar manner. With this technique, determining the maximum and minimum values contained in a vector having 16

elements requires executing a sequence of 16 instructions. Also, a number of temporal registers are used to store intermediate results.

[0005] Another conventional technique for determining the maximum value contained in a vector uses an instruction that performs a square all-to-all comparisons of the elements of the vector using a given comparison operation (e.g., greater than or equal to) and stores the result as bit values, where a bit value of binary '1' indicates comparison is true and binary '0' indicates otherwise. This instruction allows the results of all comparisons of elements to each other to be obtained by a single instruction. Determining the minimum value contained in the vector may require executing another instruction that performs a square all-to-all comparison of the elements of the vector using a different comparison operation (e.g., less than or equal to) than the comparison operation used to determine the maximum value. In some microarchitectures, the square all-to-all comparison instruction can turn into a long sequence of micro-instructions and have high latency. The conventional techniques for determining the maximum and minimum values contained in a vector may thus be inefficient.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] **Figure 1** is a diagram illustrating a sequence of instructions being executed to determine the maximum value contained in a vector, according to conventional techniques;

[0007] **Figure 2** is a block diagram illustrating a hardware processor and a memory for executing instructions, according to some embodiments;

[0008] **Figure 3** is a diagram illustrating a hardware processor that decodes and executes a strideshift instruction, according to some embodiments;

[0009] **Figure 4** is a diagram illustrating a sequence of instructions being executed to determine the minimum values contained in a vector, according to some embodiments;

[0010] **Figure 5** is a diagram illustrating the result of executing a strideshift instruction that uses a cyclic addressing mode, according to some embodiments;

[0011] **Figure 6** is a flow diagram of a process for processing a strideshift instruction, according to some embodiments;

[0012] **Figures 7A-7B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

[0013] **Figure 7A** is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention;

[0014] **Figure 7B** is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention;

[0015] **Figure 8A** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention;

[0016] **Figure 8B** is a block diagram illustrating the fields of the specific vector friendly instruction format 800 that make up the full opcode field 774 according to one embodiment of the invention;

[0017] **Figure 8C** is a block diagram illustrating the fields of the specific vector friendly instruction format 800 that make up the register index field 744 according to one embodiment of the invention;

[0018] **Figure 8D** is a block diagram illustrating the fields of the specific vector friendly instruction format 800 that make up the augmentation operation field 750 according to one embodiment of the invention;

[0019] **Figure 9** is a block diagram of a register architecture 900 according to one embodiment of the invention;

[0020] **Figure 10A** 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;

[0021] **Figure 10B** 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;

[0022] **Figures 11A-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;

[0023] **Figure 11A** is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1102 and with its local subset of the Level 2 (L2) cache 1104, according to embodiments of the invention;

[0024] **Figure 11B** is an expanded view of part of the processor core in **Figure 11A** according to embodiments of the invention;

[0025] **Figure 12** is a block diagram of a processor 1200 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention;

[0026] **Figures 13-16** are block diagrams of exemplary computer architectures;

[0027] **Figure 13** shown a block diagram of a system in accordance with one embodiment of the present invention;

[0028] **Figure 14** is a block diagram of a first more specific exemplary system in accordance with an embodiment of the present invention;

[0029] **Figure 15** is a block diagram of a second more specific exemplary system in accordance with an embodiment of the present invention;

[0030] **Figure 16** is a block diagram of a SoC in accordance with an embodiment of the present invention; and

[0031] **Figure 17** 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.

DETAILED DESCRIPTION

[0032] In the following description, numerous specific details are set forth. However, it is understood that embodiments of the disclosure may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail to not obscure the understanding of this description.

[0033] References in the specification to "one embodiment," "an embodiment," "an example embodiment," etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment need not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to affect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

[0034] Determining the maximum and minimum values contained in a vector may be useful for various purposes. One conventional technique for determining the maximum value contained

in a vector extracts the upper half of the vector to another register, determines the maximum value using a pairwise maxps instruction between two vectors, extracts the upper half of the half-vector, and repeats this process until the maxps instruction is performed on just two elements. A sequence of instructions for determining the maximum value contained in a vector is as follows for a vector zmm_index having 16 elements (KL=16).

```
vshuff32x4 zmm1, zmm_index, zmm_index, 238
vmaxps zmm2, zmm1, zmm_index
vshuff32x4 zmm0, zmm2, zmm2, 85
vmaxps zmm3, zmm0, zmm2
vpshufd zmm4, zmm3, 78
vmaxps zmm5, zmm3, zmm4
vpshufd zmm6, amm5, 177
vmaxps zmm_max, zmm5, zmm6
```

[0035] A similar sequence of instructions can be executed to determine the minimum value contained in a vector. A sequence of instructions for determining both the maximum and minimum values contained in a vector is as follows for a vector zmm_index having 16 elements (KL=16).

```
vshuff32x4 zmm12, zmm_index, zmm_index, 238
vshuff32x4 zmm3, zmm_index, zmm_index, 238
vmaxps zmm15, zmm12, zmm_index
vminps zmm6, zmm3, zmm_index
vshuff32x4 zmm14, zmm15, zmm15, 85
vshuff32x4 zmm5, zmm6, zmm6, 85
vmaxps zmm16, zmm14, zmm15
vminps zmm7, zmm5, zmm6
vpshufd zmm17, zmm16, 78
vpshufd zmm8, zmm7, 78
vmaxps zmm18, zmm16, zmm17
vminps zmm9, zmm7, zmm8
```

vpshufd zmm19, zmm18, 177

vpshufd zmm10, zmm9, 177

vmaxps zmm_max, zmm18, zmm19

vminps zmm_min, zmm9, zmm10

[0036] With this technique, determining the maximum and minimum values contained in a vector having 16 elements requires executing 8 shuffle instructions, 4 vminps instructions, and 4 vmaxps instructions, for a total sequence of 16 instructions. Also, a number of temporal registers are used to store intermediate results.

[0037] Another conventional technique for determining the maximum value contained in a vector employs an instruction that performs a square all-to-all comparison of the elements of the vector using a given comparison operation (e.g., greater than or equal to) and stores the result as bit values, where a bit value of binary '1' indicates comparison is true and binary '0' indicates otherwise. This instruction allows the results of all comparisons of elements to each other to be obtained by a single instruction. Such an instruction may be referred to herein as a square all-to-all comparison instruction.

[0038] **Figure 1** is a diagram illustrating a sequence of instructions being executed to determine the maximum value contained in a vector, according to conventional techniques. In the diagram, each vertical "column" represents the same lane of a vector register. The "offset" refers to the position of an element. In this example, there are seven (KL = 7) elements in a vector. The values and sizes of the vectors are provided by way of example for purposes of illustration. It should be understood that other values and sizes may be utilized. Zmm_index is the vector containing the values from which the maximum value is to be determined. In this example, zmm_index contains the values 5, 7, 1, 1, 7, 3, and 1 (in order from least significant bit (LSB) to most significant bit (MSB)). Executing the vconf_sqr_ge(zmm_index) instruction produces zmm_sqr_conf. The vconf_sqr_ge instruction performs a square all-to-all comparison of the elements of a vector using a greater than or equal to comparison operation. It can be seen from zmm_sqr_conf that the "column(s)" corresponding to the maximum value (there can be multiple of them, as in this example) contain all binary '1's in the 7x7 matrix. Executing the vpcmpeq(zmm_sqr_conf, zmm_cmp) instruction compares zmm_sqr_conf to a pre-defined 7x7 matrix containing all binary '1's (zmm_cmp) to produce k_mask_max, which contains binary '1's in the bit position(s) corresponding to the maximum value. Executing the

vcompress(k_mask_max, zmm_index) instruction compresses zmm_index with k_mask_max to produce zmm_max, which contains the maximum value in its LSB lane (the maximum value is 7 in this example).

[0039] The problem with determining the minimum value from zmm_sqr_conf (the result of the vconf_sqr_ge instruction) is the existence of duplicated minimum values. If there were no duplicated minimum values, then the "column" of zmm_sqr_conf corresponding to the minimum value would contain all binary '0's and it would be possible to extract the minimum value by comparing zmm_sqr_conf with a 7x7 matrix containing all binary '0's.

[0040] When there are duplicated minimum values, determining the minimum value contained in the vector may require performing another square all-to-all comparison of the elements of the vector, but using a different comparison operation than the comparison operation used to determine the maximum value. The result may be compared to a pre-defined matrix to extract the minimum value. For example, a first technique may perform a square all-to-all comparison of the elements of the vector using a less than or equal to comparison operation (e.g., using a vconf_sqr_le instruction) and compare the result to a pre-defined matrix containing all binary '1's. A second technique may perform a square all-to-all comparison of the elements of the vector using a greater than comparison operation and compare the result to a pre-defined matrix containing all binary '0's. A sequence of instructions for determining the maximum and minimum values contained in a vector is as follows for a vector zmm_index (using the first technique).

(1) zmm_sqr_vconf_ge = vconf_sqr_ge(zmm_index)
(2) k1 = vpcmpeq(zmm_sqr_vconf_ge, zmm_cmp)
(3) zmm_max = vcompress(k1, zmm_index)
(4) zmm_sqr_vconf_le = vconf_sqr_le(zmm_index)
(5) k2 = vpcmpeq(zmm_sqr_vconf_le, zmm_cmp)
(6) zmm_min = vcompress(k2, zmm_index)

[0041] In some microarchitectures, a square all-to-all comparison instruction (e.g., vconf_sqr_ge and vconf_sqr_le) can turn into a long sequence of micro-instructions and have high latency. Depending on the number of comparators present in the microarchitecture, this technique for determining the maximum and minimum values contained in a vector can, in some

situations, perform worse than the technique described above that use the shuffle and vmaxps/vminps instructions. The conventional techniques for determining the maximum and minimum values contained in a vector may thus be inefficient.

[0042] Certain embodiments disclosed herein overcome the disadvantages of the conventional techniques by providing a strideshift instruction that allows the maximum and minimum values contained in a vector to be determined based on the result of a single square all-to-all comparison instruction. The strideshift instruction can be used to transpose the result of the square all-to-all comparison instruction with respect to a main diagonal (e.g., (0, 0) -> (KL-1, KL-1). The transposed result can be compared with a pre-defined matrix to extract the minimum value. With the strideshift instruction (e.g., vstrideshift), both the maximum and minimum values contained in a vector (zmm_index) can be determined with a total of 6 instructions as follows:

(1) zmm_sqr_conf = vconf_sqr_ge(zmm_index)

(2) k1 = vpcmpeq(zmm_sqr_conf, zmm_cmp)

(3) zmm_max = vcompress(k1, zmm_index)

(4) zmm_transposed = vstrideshift(zmm_sqr_conf)

(5) k2 = vpcmpeq(zmm_transposed, zmm_cmp)

(6) zmm_min = vcompress(k2, zmm_index)

[0043] Compared to the sequence of instructions that determines the maximum and minimum values contained in a vector based on executing shuffle and vmaxps/vminps instructions, the above sequence of instructions that includes the strideshift instruction is shorter (6 instructions vs. 16 instructions) and uses less number of temporal registers. Compared to the sequence of instructions that determines the maximum and minimum values contained in a vector based on executing two square all-to-all comparison instructions (e.g., vconf_sqr_ge and vconf_sqr_le), the above sequence of instructions that includes the strideshift instruction may have better performance and lower latency depending on whether the square all-to-all comparison instruction or the strideshift instruction has better performance characteristics on the underlying microarchitecture.

[0044] **Figure 2** is a block diagram illustrating a hardware processor and a memory for executing instructions, according to some embodiments. Depicted hardware processor 100 includes a hardware decoder 102 (e.g., decode unit or decode circuit) and a hardware execution unit 104 (e.g., execution circuit). Depicted hardware processor 100 includes register(s) 106.

Registers 106 may include one or more registers to perform operations in, e.g., additionally or alternatively to access of (e.g., load or store) data in memory 110. It should be noted that the figures herein may not depict all data communication connections. One of ordinary skill in the art will appreciate that this is to not obscure certain details in the figures. It should be noted that a double headed arrow in the figures may not require two-way communication. For example, it may indicate one-way communication (e.g., to or from that component or device). Any or all combinations of communications paths may be utilized in certain embodiments herein.

[0045] Hardware decoder 102 may receive an instruction (e.g., macro-instruction) and decode the instruction (e.g., into micro-instructions and/or micro-operations). Execution unit 104 may execute the decoded instruction to perform one or more operations. The decoder 102 and the execution unit 104 may decode and execute any of the instructions disclosed herein (e.g., strideshift instruction). Certain embodiments disclosed herein provide a strideshift instruction that can be decoded and executed by the decoder 102 and execution unit 104, respectively. As will be described in additional detail below, this instruction may be used to more efficiently determine the maximum and minimum values contained in a vector compared to conventional techniques.

[0046] In one embodiment, a strideshift instruction has the following definition:

```
VSTRIDE[C]SHIFT{B, W} dest{k1}, src1, src2, imm8
(KL, VL) = (64, 512), (32, 512) // where KL is the number of elements in the source/destination
vector and VL is the vector length
N = VL/KL // granularity of the destination in bits: 8 or 16 (B, W)
stride = imm8 // stride in bits 0...255

for (i = 0; i < KL; i++) {        // loop over i-elements of the destination
        if (k1[i]) {
                offset = src2[i] // starting position 0...255 for 8 bit granularity, 0...511 for 16 bit
                                 //granularity
                for (k = 0; k < N; k++) {      //k is a bit position in each i-element of the
                                               //destination
```

```
                index = offset + k * stride
                if (cyclic_addressing) //"C" letter in opcode of instruction
                        // store bit value from src1 to k-th bit in i-th element of destination
                        // (cyclic addressing mode)
                        dest[i * N + k] = src1[index & (VL – 1)]
                else
                        //// if index is out of range, then zero destination bit
                        dest[i * N + k] = (index < VL) ? src1[index] : 0
            }
        }
        else {  // zeroing mode
            dest[i] = 0      // zero destination bit if k1[i] == 0
        }
}
```

In this instruction, {B, W} indicates the size of supported elements (e.g., byte (B) and word (W)) and {k1} indicates the write mask. In this instruction, KL is the number of elements in the destination vector (dest), VL is the number of bits in the destination vector, N is the number of bits in packed element of the destination vector, stride is taken as an immediate operand (or can be src3), src1 is the vector to be transposed, and src2 is the vector containing starting positions. In one embodiment, operation of this instruction may be described as follows: take bits from src1 starting from the bit position indicated in src2[i] and going with the stride over bits in src1 (e.g., src1[index], where index = offset + k * stride). The N bits taken from src1 are stored as consecutive bits in the i-th element of the destination vector (dest). Here, all vectors are addressed as one-dimensional arrays with VL bits. There are two possible addressing modes for src1. Cyclic addressing mode and range addressing mode. In cyclic addressing mode, when index reaches the MSB of src1, wrap-around takes effect by taking index & (VL – 1). In range addressing mode, when index is out of the VL – 1 range, the destination bit (in the destination vector) is zeroed. The destination vector is masked based on the write mask.

[0047] **Figure 3** is a diagram illustrating a hardware processor that decodes and executes a strideshift instruction, according to some embodiments. Strideshift instruction 300 may be

decoded by the decoder 102 and executed by the execution unit 104. Data may be accessed in register(s) 106 and/or memory 110. The strideshift instruction 300 may take a first input vector and a second input vector as operands. The first input vector is the vector to be transposed. The second input vector is the vector containing starting positions. The strideshift instruction 300 may also take the stride as an operand. In one embodiment, the execution unit 104 executes the strideshift instruction 300 (e.g., vstrideshift) to cause the first input vector to be transposed with respect to the main diagonal ((0, 0) -> (KL-1, KL-1)) and to cause the result to be stored as a destination vector. The transposition effectively takes bits corresponding to a "row" in the first input vector and stores them as consecutive bits in the destination vector to form a "column" in the destination vector. In the example shown in the diagram, src1 is the first input vector, src2 is the second input vector, and dest is the destination vector. For purposes of illustration, the specifics related to the transposition of "row" 2 of src1 is shown in the diagram. It should be understood that the other "rows" can be transposed in a similar manner. In this example KL=7, VL=224, N=32, and stride=32. Since the value of the third element (element 2) of src2 is 2, the offset is set to 2. The index is calculated as offset+k*stride, where k =0...31. The resulting index values (2, 34, 66, 98, ... 482) correspond to the bit positions in src1 that form "row" 2. The bits located at these bit positions are stored as consecutive bits in "column" 2 of dest (since this is the "column" that corresponds to element 2 of src2). In this example, the strideshift instruction 300 uses range addressing mode. As such, when the index goes out of range (OOR) (e.g., exceeds VL-1, which in this example is 223), the destination bit is zeroed.

[0048] In this way, the execution unit 104 may execute strideshift instruction 300 to access a given bit of the first input vector located at a bit position indicated by an element of the second input vector (e.g., the bit located at bit position 2) and stride over bits of the first input vector using a stride (e.g., skips over every 32 bits) to access bits of the first input vector that are located at a strided bit position with respect to the given bit (e.g., bits located at bit positions 34, 66, 98...). The execution unit 104 may then store the given bit and the bits that are located at a strided position with respect to the given bit as consecutive bits in a destination vector.

[0049] As previously mentioned, with the strideshift instruction, the following sequence of instruction may be executed to determine the maximum and minimum values contained in an input vector zmm_index:

(1) zmm_sqr_conf = vconf_sqr_ge(zmm_index)

(2) k1 = vpcmpeq(zmm_sqr_conf, zmm_cmp)

(3) zmm_max = vcompress(k1, zmm_index)

(4) zmm_transposed = vstrideshift(zmm_sqr_conf)

(5) k2 = vpcmpeq(zmm_transposed, zmm_cmp)

(6) zmm_min = vcompress(k2, zmm_index)

In the above sequence of instructions, zmm_index is the vector containing the values from which the maximum and minimum values are to be determined. In the above sequence of instructions, zmm_cmp is a predefined (constant) vector containing a square matrix containing all binary '1's. In the above sequence of instructions, (1) is an instruction to perform square all-to-all comparisons of the elements of the vector using a greater than or equal to comparison operation (e.g., and stores the result as bit values, where a bit value of binary '1' indicates comparison is true and binary '0' indicates otherwise). In the above sequence of instructions, (2) is an instruction to perform vector comparison. In the above sequence of instructions, (3) is an instruction to perform vector compression. In the above sequence of instructions, (4) is a strideshift instruction (only the first operand is shown for purposes of simplicity). In the above sequence of instructions, (5) is an instruction to perform vector comparison. In the above sequence of instructions, (6) is an instruction to perform vector compression.

[0050] **Figure 4** is a diagram illustrating a sequence of instructions being executed to determine the minimum values contained in a vector, according to some embodiments. In the diagram, each vertical "column" is the same lane of a vector register. The "offset" refers to the position of an element. In this example, there are seven (KL = 7) elements in a vector. The values and sizes of the vectors are provided by way of example for purposes of illustration. It should be understood that other values and sizes may be utilized. In this example, zmm_index is the vector containing the values from which the minimum value is to be determined. In this example, zmm_index contains the values 5, 7, 1, 1, 7, 3, and 1 (in order from LSB to MSB). Executing the vconf_sqr_ge(zmm_index) instruction produces zmm_sqr_conf. This may have already been computed as part of determining the maximum value contained in zmm_index. In this example, zmm_start is the vector containing starting positions (to be used by the vstrideshift instruction). In this example, zmm_start contains the values 0, 1, 2, 3, 4, 5, and 6 (in order from LSB to MSB). Executing the vstrideshift(zmm_sqr_conf_ge, zmm_start, 32) instruction

produces zmm_transposed. In this example, zmm_cmp is a pre-defined 7x7 matrix containing all binary '1's. Executing the vpcmpeq(zmm_transposed, zmm_cmp) instruction produces k_mask_min, which contains binary '1's in the bit position(s) corresponding to the minimum value. Executing the vcompress(k_mask_min, zmm_index) instruction produces zmm_min, which contains the minimum value in its LSB lane (the minimum value is 1 in this example). In this way, the minimum value contained in zmm_index is determined without having to execute a second square all-to-all comparison instruction (in addition to the square all-to-all comparison instruction that was executed as part of determining the maximum value).

[0051] **Figure 5** is a diagram illustrating the result of executing a strideshift instruction that uses a cyclic addressing mode, according to some embodiments. The diagram shows the vstridecshiftw(src1, src2, 32) instruction being executed. This instructions uses cyclic addressing mode (as indicated by the "c" in vstridecshiftw). In this example, src1 contains data that is packed as double words and src2 contains starting positions. Since data in src1 is packed as double words, the stride is set to 32. The write mask (e.g., k1) has word-level granularity to make the data in destination to also be packed as double words (with 16 meaningful lower bits corresponding to 16 rows in src1 and with 16 upper bits zeroed). The result of executing the vstridecshiftw(src1, src2, 32) instruction is stored in dest.

[0052] Since src2[12] contains a value of 136, the vstridecshiftw instruction starts with accessing the bit of src1 located at bit position 136, which is in "row" 8 and "column" 4 of src1. The vstridecshiftw instruction then strides over the bits of src1 according to the specified stride to access the remaining bits in "row" 8. It should be noted that since the vstridecshiftw instruction uses cyclic addressing mode, wrap-around takes effect after reaching bit position 488 to access bit positions 8, 40, 72, and 104. The starting bit and the bits located at a strided bit position with respect to the starting bit position are stored as consecutive bits in "column" 12 of dest (since this "column" corresponds to the element of src2 that contained the starting position).

[0053] **Figure 6** is a flow diagram of a process for processing a strideshift instruction, according to some embodiments. The operations in the flow diagrams will be described with reference to the exemplary embodiments of the other figures. However, it should be understood that the operations of the flow diagrams can be performed by embodiments other than those discussed with reference to the other figures, and the embodiments discussed with reference to

these other figures can perform operations different than those discussed with reference to the flow diagrams.

[0054] In one embodiment, the process is initiated when a decoder 102 (e.g., decode circuit) decodes an instruction (e.g., a strideshift instruction) into a decoded instruction (block 610). An execution unit 104 (e.g., execution circuit) executes the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector (block 620). In one embodiment, the execution circuit 104 is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector (e.g., index = (offset + k * stride)). In one embodiment, the execution circuit 104 is to stride over bits of the first input vector using a cyclic addressing mode (wrap-around). In such an embodiment, the execution circuit 104 may determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one (e.g., index & (VL-1)). In one embodiment, the execution circuit 104 is to stride over bits of the first input vector using a range addressing mode. In such an embodiment, the execution circuit 104 may store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range (e.g., out of VL-1 range). In one embodiment, the consecutive bits in the destination register correspond to an element of the destination vector and this element of the destination vector corresponds to the element of the second input vector that indicates the bit position of the first bit of the first input vector). In one embodiment, the instruction specifies the stride. In one embodiment, the execution circuit 104 is to mask bits of the destination vector using a write mask.

Instruction Sets

[0055] An instruction set may include one or more instruction formats. A given instruction format may define various fields (e.g., number of bits, location of bits) to specify, among other

things, the operation to be performed (e.g., opcode) and the operand(s) on which that operation is to be performed and/or other data field(s) (e.g., mask). 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 as the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developer's Manual, September 2014; and see Intel® Advanced Vector Extensions Programming Reference, October 2014).

Exemplary Instruction Formats

[0056] 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.

Generic Vector Friendly Instruction Format

[0057] 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.

[0058] **Figures 7A-7B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. **Figure 7A**

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 7B** 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 700 for which are defined class A and class B instruction templates, both of which include no memory access 705 instruction templates and memory access 720 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.

[0059] 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 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 (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).

[0060] The class A instruction templates in **Figure 7A** include: 1) within the no memory access 705 instruction templates there is shown a no memory access, full round control type operation 710 instruction template and a no memory access, data transform type operation 715 instruction template; and 2) within the memory access 720 instruction templates there is shown a memory access, temporal 725 instruction template and a memory access, non-temporal 730 instruction template. The class B instruction templates in **Figure 7B** include: 1) within the no memory access 705 instruction templates there is shown a no memory access, write mask control, partial round control type operation 712 instruction template and a no memory access, write mask control, vsize type operation 717 instruction template; and 2) within the memory access 720 instruction templates there is shown a memory access, write mask control 727 instruction template.

[0061] The generic vector friendly instruction format 700 includes the following fields listed below in the order illustrated in **Figures 7A-7B**.

[0062] Format field 740 – 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 sense that it is not needed for an instruction set that has only the generic vector friendly instruction format.

[0063] Base operation field 742 – its content distinguishes different base operations.

[0064] Register index field 744 – 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).

[0065] Modifier field 746 – 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 705 instruction templates and memory access 720 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.

[0066] Augmentation operation field 750 – 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 768, an alpha field 752, and a beta field 754. The augmentation operation field 750 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

[0067] Scale field 760 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses 2^{scale} * index + base).

[0068] Displacement Field 762A– its content is used as part of memory address generation (e.g., for address generation that uses 2^{scale} * index + base + displacement).

[0069] Displacement Factor Field 762B (note that the juxtaposition of displacement field 762A directly over displacement factor field 762B 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^{scale} * index + base + 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 774 (described later herein) and the data manipulation field 754C. The displacement field 762A and the displacement factor field 762B are optional in the sense that they are not used for the no memory access 705 instruction templates and/or different embodiments may implement only one or none of the two.

[0070] Data element width field 764 – 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.

[0071] Write mask field 770 – 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 770 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 770 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 770 content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's 770 content to directly specify the masking to be performed.

[0072] Immediate field 772 – 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.

[0073] Class field 768 – its content distinguishes between different classes of instructions. With reference to **Figures 7A-B**, the contents of this field select between class A and class B instructions. In **Figures 7A-B**, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 768A and class B 768B for the class field 768 respectively in **Figures 7A-B**).

Instruction Templates of Class A

[0074] In the case of the non-memory access 705 instruction templates of class A, the alpha field 752 is interpreted as an RS field 752A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 752A.1 and data transform 752A.2 are respectively specified for the no memory access, round type operation 710 and the no memory access, data transform type operation 715 instruction templates), while the beta field 754 distinguishes which of the operations of the specified type is to be performed. In the no memory access 705 instruction templates, the scale field 760, the displacement field 762A, and the displacement scale filed 762B are not present.

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

[0075] In the no memory access full round control type operation 710 instruction template, the beta field 754 is interpreted as a round control field 754A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field 754A includes a suppress all floating point exceptions (SAE) field 756 and a round operation control field 758, 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 758).

[0076] SAE field 756 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 756 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.

[0077] Round operation control field 758 – 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 758 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 750 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

[0078] In the no memory access data transform type operation 715 instruction template, the beta field 754 is interpreted as a data transform field 754B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

[0079] In the case of a memory access 720 instruction template of class A, the alpha field 752 is interpreted as an eviction hint field 752B, whose content distinguishes which one of the eviction hints is to be used (in **Figure 7A**, temporal 752B.1 and non-temporal 752B.2 are respectively specified for the memory access, temporal 725 instruction template and the memory access, non-temporal 730 instruction template), while the beta field 754 is interpreted as a data manipulation field 754C, 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 720 instruction templates include the scale field 760, and optionally the displacement field 762A or the displacement scale field 762B.

[0080] 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

[0081] 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

[0082] 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

[0083] In the case of the instruction templates of class B, the alpha field 752 is interpreted as a write mask control (Z) field 752C, whose content distinguishes whether the write masking controlled by the write mask field 770 should be a merging or a zeroing.

[0084] In the case of the non-memory access 705 instruction templates of class B, part of the beta field 754 is interpreted as an RL field 757A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 757A.1 and vector length (VSIZE) 757A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 712 instruction template and the no memory access, write mask control, VSIZE type operation 717 instruction template), while the rest of the beta field 754 distinguishes which of the operations of the specified type is to be performed. In the no memory access 705 instruction templates, the scale field 760, the displacement field 762A, and the displacement scale filed 762B are not present.

[0085] In the no memory access, write mask control, partial round control type operation 710 instruction template, the rest of the beta field 754 is interpreted as a round operation field 759A

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).

[0086] Round operation control field 759A – just as round operation control field 758, 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 759A 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 750 content overrides that register value.

[0087] In the no memory access, write mask control, VSIZE type operation 717 instruction template, the rest of the beta field 754 is interpreted as a vector length field 759B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

[0088] In the case of a memory access 720 instruction template of class B, part of the beta field 754 is interpreted as a broadcast field 757B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 754 is interpreted the vector length field 759B. The memory access 720 instruction templates include the scale field 760, and optionally the displacement field 762A or the displacement scale field 762B.

[0089] With regard to the generic vector friendly instruction format 700, a full opcode field 774 is shown including the format field 740, the base operation field 742, and the data element width field 764. While one embodiment is shown where the full opcode field 774 includes all of these fields, the full opcode field 774 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 774 provides the operation code (opcode).

[0090] The augmentation operation field 750, the data element width field 764, and the write mask field 770 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

[0091] The combination of write mask field and data element width field create typed instructions in that they allow the mask to be applied based on different data element widths.

[0092] 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 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-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 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.

Exemplary Specific Vector Friendly Instruction Format

[0093] **Figure 8A** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. **Figure 8A** shows a specific vector friendly instruction format 800 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 vector friendly instruction format 800 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 7** into which the fields from **Figure 8A** map are illustrated.

[0094] It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 800 in the context of the generic vector friendly instruction format 700 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 800 except where claimed. For example, the generic vector friendly instruction format 700 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 800 is shown as having fields of specific sizes. By way of specific example, while the data element width field 764 is illustrated as a one bit field in the specific vector friendly instruction format 800, the invention is not so limited (that is, the generic vector friendly instruction format 700 contemplates other sizes of the data element width field 764).

[0095] The generic vector friendly instruction format 700 includes the following fields listed below in the order illustrated in **Figure 8A**.

[0096] EVEX Prefix (Bytes 0-3) 802 - is encoded in a four-byte form.

[0097] Format Field 740 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 740 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

[0098] The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

[0099] REX field 805 (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 757BEX 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.

[00100] REX' field 710 – this is the first part of the REX' field 710 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.

[00101] Opcode map field 815 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

[00102] Data element width field 764 (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).

[00103] EVEX.vvvv 820 (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 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 820 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.

[00104] EVEX.U 768 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.

[00105] Prefix encoding field 825 (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 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 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.

[00106] Alpha field 752 (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.

[00107] Beta field 754 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s_{2-0}, EVEX.r_{2-0}, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with βββ) – as previously described, this field is context specific.

[00108] REX' field 710 – 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.

[00109] Write mask field 770 (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).

[00110] Real Opcode Field 830 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

[00111] MOD R/M Field 840 (Byte 5) includes MOD field 842, Reg field 844, and R/M field 846. As previously described, the MOD field's 842 content distinguishes between memory access and non-memory access operations. The role of Reg field 844 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 846 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.

[00112] Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 750 content is used for memory address generation. SIB.xxx 854 and SIB.bbb 856 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[00113] Displacement field 762A (Bytes 7-10) – when MOD field 842 contains 10, bytes 7-10 are the displacement field 762A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

[00114] Displacement factor field 762B (Byte 7) – when MOD field 842 contains 01, byte 7 is the displacement factor field 762B. 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 762B is a reinterpretation of disp8; when using displacement factor field 762B, 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 762B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 762B 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 772 operates as previously described.

Full Opcode Field

[00115] **Figure 8B** is a block diagram illustrating the fields of the specific vector friendly instruction format 800 that make up the full opcode field 774 according to one embodiment of the invention. Specifically, the full opcode field 774 includes the format field 740, the base operation field 742, and the data element width (W) field 764. The base operation field 742 includes the prefix encoding field 825, the opcode map field 815, and the real opcode field 830.

Register Index Field

[00116] **Figure 8C** is a block diagram illustrating the fields of the specific vector friendly instruction format 800 that make up the register index field 744 according to one embodiment of the invention. Specifically, the register index field 744 includes the REX field 805, the REX' field 810, the MODR/M.reg field 844, the MODR/M.r/m field 846, the VVVV field 820, xxx field 854, and the bbb field 856.

Augmentation Operation Field

[00117] **Figure 8D** is a block diagram illustrating the fields of the specific vector friendly instruction format 800 that make up the augmentation operation field 750 according to one embodiment of the invention. When the class (U) field 768 contains 0, it signifies EVEX.U0 (class A 768A); when it contains 1, it signifies EVEX.U1 (class B 768B). When U=0 and the MOD field 842 contains 11 (signifying a no memory access operation), the alpha field 752 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 752A. When the rs field 752A contains a 1 (round 752A.1), the beta field 754 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the round control field 754A. The round control field 754A includes a one bit SAE field 756 and a two bit round operation field 758. When the rs field 752A contains a 0 (data transform 752A.2), the beta field 754 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data transform field 754B. When U=0 and the MOD field 842 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 752 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 752B and the beta field 754 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 754C.

[00118] When U=1, the alpha field 752 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 752C. When U=1 and the MOD field 842 contains 11 (signifying a no memory access operation), part of the beta field 754 (EVEX byte 3, bit [4]- S_0) is interpreted as the RL field 757A; when it contains a 1 (round 757A.1) the rest of the beta field 754 (EVEX byte 3, bit [6-5]- S_{2-1}) is interpreted as the round operation field 759A, while when the RL field 757A contains a 0 (VSIZE 757.A2) the rest of the beta field 754 (EVEX byte 3, bit [6-5]- S_{2-1}) is interpreted as the vector length field 759B (EVEX byte 3, bit [6-5]- L_{1-0}). When U=1 and the MOD field 842 contains 00, 01, or 10 (signifying a memory access operation), the beta field 754 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 759B (EVEX byte 3, bit [6-5]- L_{1-0}) and the broadcast field 757B (EVEX byte 3, bit [4]- B).

Exemplary Register Architecture

[00119] **Figure 9** is a block diagram of a register architecture 900 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 910 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 800 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 759B	A (Figure 7A; U=0)	710, 715, 725, 730	zmm registers (the vector length is 64 byte)
	B (Figure 7B; U=1)	712	zmm registers (the vector length is 64 byte)
Instruction templates that do include the vector length field 759B	B (Figure 7B; U=1)	717, 727	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 759B

[00120] In other words, the vector length field 759B 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 759B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 800 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.

[00121] Write mask registers 915 - 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 915 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.

[00122] General-purpose registers 925 - 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.

[00123] Scalar floating point stack register file (x87 stack) 945, on which is aliased the MMX packed integer flat register file 950 - 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.

[00124] 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

[00125] 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

[00126] **Figure 10A** 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 10B** 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 10A-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.

[00127] In **Figure 10A**, a processor pipeline 1000 includes a fetch stage 1002, a length decode stage 1004, a decode stage 1006, an allocation stage 1008, a renaming stage 1010, a scheduling (also known as a dispatch or issue) stage 1012, a register read/memory read stage 1014, an execute stage 1016, a write back/memory write stage 1018, an exception handling stage 1022, and a commit stage 1024.

[00128] **Figure 10B** shows processor core 1090 including a front end unit 1030 coupled to an execution engine unit 1050, and both are coupled to a memory unit 1070. The core 1090 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 1090 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.

[00129] The front end unit 1030 includes a branch prediction unit 1032 coupled to an instruction cache unit 1034, which is coupled to an instruction translation lookaside buffer (TLB) 1036, which is coupled to an instruction fetch unit 1038, which is coupled to a decode unit 1040. The decode unit 1040 (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 signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit 1040 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 1090 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 1040 or otherwise within the front end unit 1030). The decode unit 1040 is coupled to a rename/allocator unit 1052 in the execution engine unit 1050.

[00130] The execution engine unit 1050 includes the rename/allocator unit 1052 coupled to a retirement unit 1054 and a set of one or more scheduler unit(s) 1056. The scheduler unit(s) 1056 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 1056 is coupled to the physical register file(s) unit(s) 1058. Each of the physical register file(s) units 1058 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 (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 1058 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) 1058 is overlapped by the retirement unit 1054 to illustrate various ways in which 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 1054 and the physical register file(s) unit(s) 1058 are coupled to the execution cluster(s) 1060. The execution cluster(s) 1060 includes a set of one or more execution units 1062 and a set of one or more memory access units 1064. The execution units 1062 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) 1056, physical register file(s)

unit(s) 1058, and execution cluster(s) 1060 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) 1064). 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.

[00131] The set of memory access units 1064 is coupled to the memory unit 1070, which includes a data TLB unit 1072 coupled to a data cache unit 1074 coupled to a level 2 (L2) cache unit 1076. In one exemplary embodiment, the memory access units 1064 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 1072 in the memory unit 1070. The instruction cache unit 1034 is further coupled to a level 2 (L2) cache unit 1076 in the memory unit 1070. The L2 cache unit 1076 is coupled to one or more other levels of cache and eventually to a main memory.

[00132] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 1000 as follows: 1) the instruction fetch 1038 performs the fetch and length decoding stages 1002 and 1004; 2) the decode unit 1040 performs the decode stage 1006; 3) the rename/allocator unit 1052 performs the allocation stage 1008 and renaming stage 1010; 4) the scheduler unit(s) 1056 performs the schedule stage 1012; 5) the physical register file(s) unit(s) 1058 and the memory unit 1070 perform the register read/memory read stage 1014; the execution cluster 1060 perform the execute stage 1016; 6) the memory unit 1070 and the physical register file(s) unit(s) 1058 perform the write back/memory write stage 1018; 7) various units may be involved in the exception handling stage 1022; and 8) the retirement unit 1054 and the physical register file(s) unit(s) 1058 perform the commit stage 1024.

[00133] The core 1090 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 1090 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.

[00134] 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).

[00135] 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 1034/1074 and a shared L2 cache unit 1076, 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

[00136] **Figures 11A-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.

[00137] **Figure 11A** is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1102 and with its local subset of the Level 2 (L2) cache 1104, according to embodiments of the invention. In one embodiment, an instruction decoder 1100 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 1106 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 1108 and a vector unit 1110 use separate register sets (respectively, scalar registers 1112 and vector registers 1114) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 1106, 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).

[00138] The local subset of the L2 cache 1104 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 1104. Data read by a processor core is stored in its L2 cache subset 1104 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 1104 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.

[00139] **Figure 11B** is an expanded view of part of the processor core in **Figure 11A** according to embodiments of the invention. **Figure 11B** includes an L1 data cache 1106A part of the L1 cache 1104, as well as more detail regarding the vector unit 1110 and the vector registers 1114. Specifically, the vector unit 1110 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 1128), 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 1120, numeric conversion with numeric convert units 1122A-B, and replication with replication unit 1124 on the memory input. Write mask registers 1126 allow predicating resulting vector writes.

[00140] **Figure 12** is a block diagram of a processor 1200 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 12** illustrate a processor 1200 with a single core 1202A, a system agent 1210, a set of one or more bus controller units 1216, while the optional addition of the dashed lined boxes illustrates an alternative processor 1200 with multiple cores 1202A-N, a set of one or more integrated memory controller unit(s) 1214 in the system agent unit 1210, and special purpose logic 1208.

[00141] Thus, different implementations of the processor 1200 may include: 1) a CPU with the special purpose logic 1208 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 1202A-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 1202A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 1202A-N being a large number of general purpose in-order cores. Thus, the processor 1200 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 1200 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.

[00142] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 1206, and external memory (not shown) coupled to the set of integrated memory controller units 1214. The set of shared cache units 1206 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 1212 interconnects the integrated graphics logic 1208 (integrated graphics logic 1208 is an example of and is also referred to herein as special purpose logic), the set of shared cache units 1206, and the system agent unit 1210/integrated memory controller unit(s) 1214, 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 1206 and cores 1202-A-N.

[00143] In some embodiments, one or more of the cores 1202A-N are capable of multi-threading. The system agent 1210 includes those components coordinating and operating cores 1202A-N. The system agent unit 1210 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 1202A-N and the integrated graphics logic 1208. The display unit is for driving one or more externally connected displays.

[00144] The cores 1202A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 1202A-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

[00145] **Figures 13-16** 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.

[00146] Referring now to **Figure 13**, shown is a block diagram of a system 1300 in accordance with one embodiment of the present invention. The system 1300 may include one or more processors 1310, 1315, which are coupled to a controller hub 1320. In one embodiment the controller hub 1320 includes a graphics memory controller hub (GMCH) 1390 and an Input/Output Hub (IOH) 1350 (which may be on separate chips); the GMCH 1390 includes memory and graphics controllers to which are coupled memory 1340 and a coprocessor 1345; the IOH 1350 couples input/output (I/O) devices 1360 to the GMCH 1390. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 1340 and the coprocessor 1345 are coupled directly to the processor 1310, and the controller hub 1320 in a single chip with the IOH 1350.

[00147] The optional nature of additional processors 1315 is denoted in **Figure 13** with broken lines. Each processor 1310, 1315 may include one or more of the processing cores described herein and may be some version of the processor 1200.

[00148] The memory 1340 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 1320 communicates with the processor(s) 1310, 1315 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 1395.

[00149] In one embodiment, the coprocessor 1345 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 1320 may include an integrated graphics accelerator.

[00150] There can be a variety of differences between the physical resources 1310, 1315 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

[00151] In one embodiment, the processor 1310 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 1310 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 1345. Accordingly, the processor 1310 issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 1345. Coprocessor(s) 1345 accept and execute the received coprocessor instructions.

[00152] Referring now to **Figure 14**, shown is a block diagram of a first more specific exemplary system 1400 in accordance with an embodiment of the present invention. As shown in **Figure 14**, multiprocessor system 1400 is a point-to-point interconnect system, and includes a first processor 1470 and a second processor 1480 coupled via a point-to-point interconnect 1450. Each of processors 1470 and 1480 may be some version of the processor 1200. In one embodiment of the invention, processors 1470 and 1480 are respectively processors 1310 and 1315, while coprocessor 1438 is coprocessor 1345. In another embodiment, processors 1470 and 1480 are respectively processor 1310 coprocessor 1345.

[00153] Processors 1470 and 1480 are shown including integrated memory controller (IMC) units 1472 and 1482, respectively. Processor 1470 also includes as part of its bus controller units point-to-point (P-P) interfaces 1476 and 1478; similarly, second processor 1480 includes P-P interfaces 1486 and 1488. Processors 1470, 1480 may exchange information via a point-to-point (P-P) interface 1450 using P-P interface circuits 1478, 1488. As shown in **Figure 14**, IMCs 1472 and 1482 couple the processors to respective memories, namely a memory 1432 and a memory 1434, which may be portions of main memory locally attached to the respective processors.

[00154] Processors 1470, 1480 may each exchange information with a chipset 1490 via individual P-P interfaces 1452, 1454 using point to point interface circuits 1476, 1494, 1486, 1498. Chipset 1490 may optionally exchange information with the coprocessor 1438 via a high-performance interface 1492. In one embodiment, the coprocessor 1438 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.

[00155] 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 processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

[00156] Chipset 1490 may be coupled to a first bus 1416 via an interface 1496. In one embodiment, first bus 1416 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 the present invention is not so limited.

[00157] As shown in **Figure 14**, various I/O devices 1414 may be coupled to first bus 1416, along with a bus bridge 1418 which couples first bus 1416 to a second bus 1420. In one embodiment, one or more additional processor(s) 1415, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus 1416. In one embodiment, second bus 1420 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 1420 including, for example, a keyboard and/or mouse 1422, communication devices 1427 and a storage unit 1428 such as a disk drive or other mass storage device which may include instructions/code and data 1430, in one embodiment. Further, an audio I/O 1424 may be coupled to the second bus 1420. Note that other architectures are possible. For example, instead of the point-to-point architecture of **Figure 14**, a system may implement a multi-drop bus or other such architecture.

[00158] Referring now to **Figure 15**, shown is a block diagram of a second more specific exemplary system 1500 in accordance with an embodiment of the present invention. Like elements in **Figures 14 and 15** bear like reference numerals, and certain aspects of **Figure 14** have been omitted from **Figure 15** in order to avoid obscuring other aspects of **Figure 15**.

[00159] **Figure 15** illustrates that the processors 1470, 1480 may include integrated memory and I/O control logic ("CL") 1472 and 1482, respectively. Thus, the CL 1472, 1482 include integrated memory controller units and include I/O control logic. **Figure 15** illustrates that not only are the memories 1432, 1434 coupled to the CL 1472, 1482, but also that I/O devices 1514

are also coupled to the control logic 1472, 1482. Legacy I/O devices 1515 are coupled to the chipset 1490.

[00160] Referring now to **Figure 16**, shown is a block diagram of a SoC 1600 in accordance with an embodiment of the present invention. Similar elements in **Figure 12** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 16**, an interconnect unit(s) 1602 is coupled to: an application processor 1610 which includes a set of one or more cores 1202A-N, which include cache units 1204A-N, and shared cache unit(s) 1206; a system agent unit 1210; a bus controller unit(s) 1216; an integrated memory controller unit(s) 1214; a set or one or more coprocessors 1620 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 1630; a direct memory access (DMA) unit 1632; and a display unit 1640 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 1620 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.

[00161] 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.

[00162] Program code, such as code 1430 illustrated in **Figure 14**, 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.

[00163] 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.

[00164] 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.

[00165] 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.

[00166] 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.)

[00167] 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.

[00168] **Figure 17** 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 17** shows a program in a high level language 1702 may be compiled using an x86 compiler 1704 to generate x86 binary code 1706 that may be natively executed by a processor with at least one x86 instruction set core 1716. The processor with at least one x86 instruction set core 1716 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 1704 represents a compiler that is operable to generate x86 binary code 1706 (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 1716. Similarly, **Figure 17** shows the program in the high level language 1702 may be compiled using an alternative instruction set compiler 1708 to generate alternative instruction set binary code 1710 that may be natively executed by a processor without at least one x86 instruction set core 1714 (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 1712 is used to convert the x86 binary code 1706 into code that may be natively executed by the processor without an x86 instruction set core 1714. This converted code is not likely to be the same as the alternative instruction set binary code 1710 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 1712 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 1706.

EXAMPLES

[00169] Example 1 is a processor. The processor includes a decode circuit to decode an instruction into a decoded instruction and an execution circuit to execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

[00170] Example 2 includes the substance of example 1. In this example, the execution circuit is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

[00171] Example 3 includes the substance of example 1. In this example, the execution circuit is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

[00172] Example 4 includes the substance of example 3. In this example, the execution circuit is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

[00173] Example 5 includes the substance of example 1. In this example, the execution circuit is to stride over bits of the first input vector using a range addressing mode.

[00174] Example 6 includes the substance of example 5. In this example, the execution circuit is to store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range.

[00175] Example 7 includes the substance of example 1. In this example, the consecutive bits in the destination vector correspond to an element of the destination vector, and the element of the destination vector corresponds to the element of the second input vector.

[00176] Example 8 includes the substance of example 1. In this example, the instruction specifies the stride.

[00177] Example 9 includes the substance of example 1. In this example, the execution circuit is to mask bits of the destination vector using a write mask.

[00178] Example 10 is a method performed by a processor. The method includes decoding an instruction into a decoded instruction and executing the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

[00179] Example 11 includes the substance of example 10. In this example, the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

[00180] Example 12 includes the substance of example 10. In this example, the execution is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

[00181] Example 13 includes the substance of example 12. In this example, the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

[00182] Example 14 includes the substance of example 10. In this example, execution is to stride over bits of the first input vector using a range addressing mode.

[00183] Example 15 includes the substance of example 14. In this example, the execution is to store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range.

[00184] Example 16 includes the substance of example 10. In this example, the consecutive bits in the destination vector correspond to an element of the destination vector, and where the element of the destination vector corresponds to the element of the second input vector.

[00185] Example 17 includes the substance of example 10. In this example, the instruction specifies the stride.

[00186] Example 18 includes the substance of example 10. In this example, the execution is to mask bits of the destination vector using a write mask.

[00187] Example 19 is a non-transitory machine readable medium. The non-transitory machine readable medium has instruction stored therein, which when executed by a processor, causes the processor to decode an instruction into a decoded instruction and execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

[00188] Example 20 includes the substance of example 19. In this example, the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

[00189] Example 21 includes the substance of example 19. In this example, the execution is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

[00190] Example 22 includes the substance of example 21. In this example, the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

[00191] Example 23 includes the substance of example 19. In this example, the execution is to stride over bits of the first input vector using a range addressing mode.

[00192] Example 24 includes the substance of example 23. In this example, the execution is to store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range.

[00193] Example 25 includes the substance of example 19. In this example, the consecutive bits in the destination vector correspond to an element of the destination vector, and where the element of the destination vector corresponds to the element of the second input vector.

[00194] Example 26 includes the substance of example 19. In this example, the instruction specifies the stride.

[00195] Example 27 includes the substance of example 19. In this example, the execution is to mask bits of the destination vector using a write mask.

[00196] Example 28 is a hardware processor. The hardware processor includes a decoding means to decode an instruction into a decoded instruction and an executing means to execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

[00197] Example 29 includes the substance of example 28. In this example, the executing means is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

[00198] Example 30 includes the substance of example 28. In this example, the executing means is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

[00199] Example 31 includes the substance of example 30. In this example, the executing means is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

[00200] Example 32 includes the substance of example 28. In this example, the executing means is to stride over bits of the first input vector using a range addressing mode.

[00201] Example 33 includes the substance of example 32. In this example, the executing means is to store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range.

[00202] Example 34 includes the substance of example 28. In this example, the consecutive bits in the destination vector correspond to an element of the destination vector, and where the element of the destination vector corresponds to the element of the second input vector.

[00203] Example 35 includes the substance of example 28. In this example, the instruction specifies the stride.

[00204] Example 36 includes the substance of example 28. In this example, the executing means is to mask bits of the destination vector using a write mask.

[00205] Example 37 is a system for executing instructions. The system includes a memory and a processor coupled to the memory. The processor includes a decode circuit to decode an instruction into a decoded instruction and an execution circuit to execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

[00206] Example 38 includes the substance of example 37. In this example, the execution circuit is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

[00207] While the invention has been described in terms of several embodiments, those skilled in the art will recognize that the invention is not limited to the embodiments described, can be practiced with modification and alteration within the spirit and scope of the appended claims. The description is thus to be regarded as illustrative instead of limiting.

CLAIMS

What is claimed is:

1. A processor comprising:

   a decode circuit to decode an instruction into a decoded instruction; and

   an execution circuit to execute the decoded instruction to:

   access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

2. The processor of claim 1, wherein the execution circuit is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

3. The processor of claim 1, wherein the execution circuit is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

4. The processor of claim 3, wherein the execution circuit is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

5. The processor of claim 1, wherein the execution circuit is to stride over bits of the first input vector using a range addressing mode.

6. The processor of claim 5, wherein the execution circuit is to store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range.

7. The processor of any one of claims 1-6, wherein the consecutive bits in the destination vector correspond to an element of the destination vector, and wherein the element of the destination vector corresponds to the element of the second input vector.

8. The processor of any one of claims 1-6, wherein the instruction specifies the stride.

9. The processor of any one of claims 1-6, wherein the execution circuit is to mask bits of the destination vector using a write mask.

10. A method performed by a processor comprising:

decoding an instruction into a decoded instruction; and

executing the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

11. The method of claim 10, wherein the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

12. The method of claim 10, wherein the execution is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

13. The method of claim 12, wherein the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

14. The method of claim 10, wherein the execution is to stride over bits of the first input vector using a range addressing mode.

15. The method of claim 14, wherein the execution is to store a binary '0' in a bit of the destination vector in response to a determination that a strided bit position with respect to the first bit of the first input vector is out of range.

16. A non-transitory machine readable medium having stored therein instructions, which when executed by a processor, causes the processor to:

decode an instruction into a decoded instruction; and

execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

17. The non-transitory machine readable medium of claim 16, wherein the execution is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

18. The non-transitory machine readable medium of claim 16, wherein the execution is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

19. The non-transitory machine readable medium of claim 16, wherein the execution is to stride over bits of the first input vector using a range addressing mode.

20. The non-transitory machine readable medium of any one of claims 16-19, wherein the consecutive bits in the destination vector correspond to an element of the destination vector,

and wherein the element of the destination vector corresponds to the element of the second input vector.

21. The non-transitory machine readable medium of any one of claims 16-19, wherein the instruction specifies the stride.

22. A hardware processor comprising:

a decoding means to decode an instruction into a decoded instruction; and

an executing means to execute the decoded instruction to access a first bit of a first input vector located at a bit position indicated by an element of a second input vector, stride over bits of the first input vector using a stride to access bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector, and store the first bit of the first input vector and the bits of the first input vector that are located at a strided bit position with respect to the first bit of the first input vector as consecutive bits in a destination vector.

23. The hardware processor of claim 22, wherein the executing means is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector.

24. The hardware processor of claim 22, wherein the executing means is to stride over bits of the first input vector using the stride using a cyclic addressing mode.

25. The hardware processor of claim 24, wherein the executing means is to determine a strided bit position with respect to the first bit of the first input vector based on adding a multiple of the stride to the bit position of the first bit of the first input vector to obtain an index and performing a modulo operation on the index and a length of the destination vector in bits minus one.

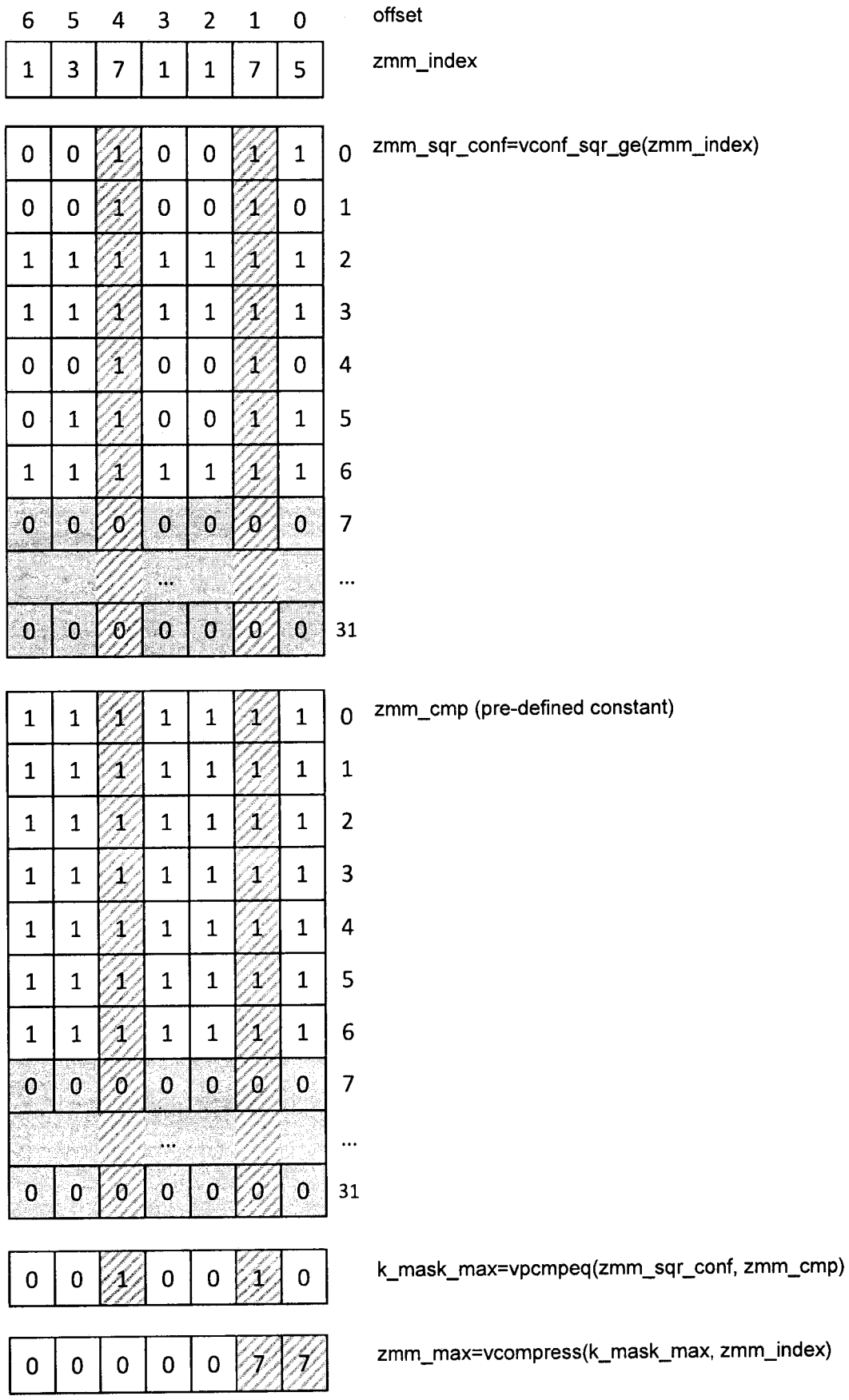

6 5 4 3 2 1 0
offset
1 3 7 1 1 7 5
zmm_index
zmm_sqr_conf=vconf_sqr_ge(zmm_index)
zmm_cmp (pre-defined constant)
k_mask_max=vpcmpeq(zmm_sqr_conf, zmm_cmp)
zmm_max=vcompress(k_mask_max, zmm_index)

Fig. 1

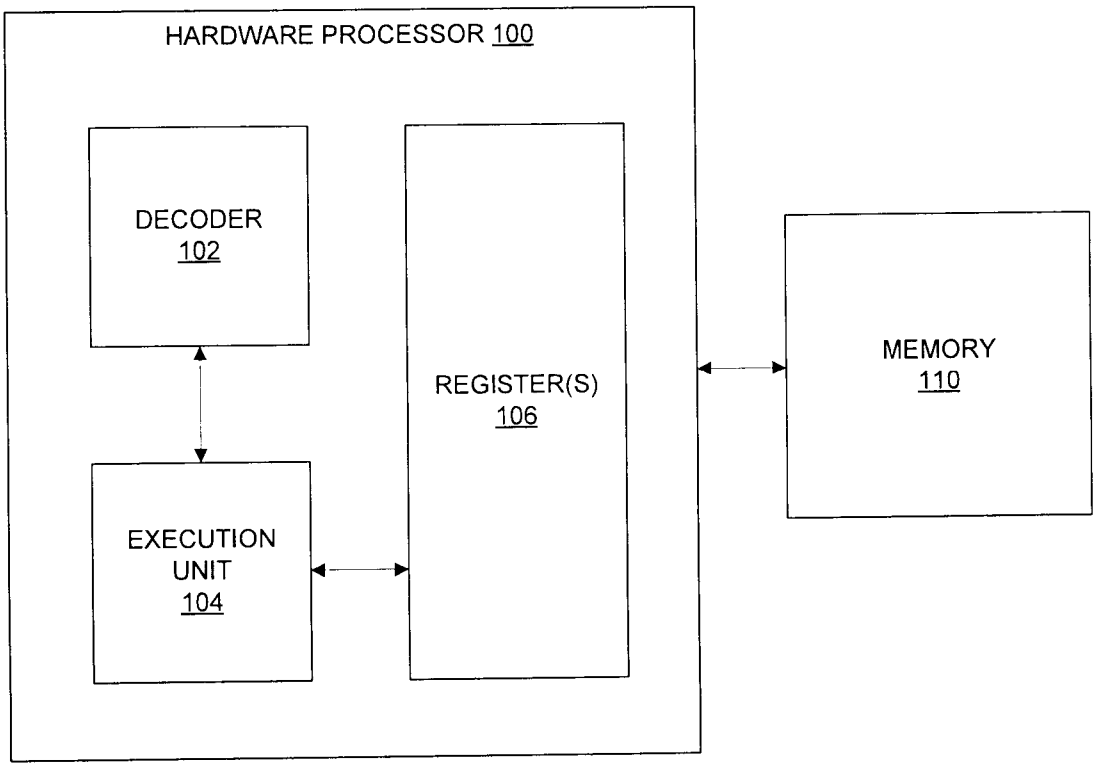

HARDWARE PROCESSOR 100
DECODER
102
EXECUTION
UNIT
104
REGISTER(S)
106
MEMORY
110

Fig. 2

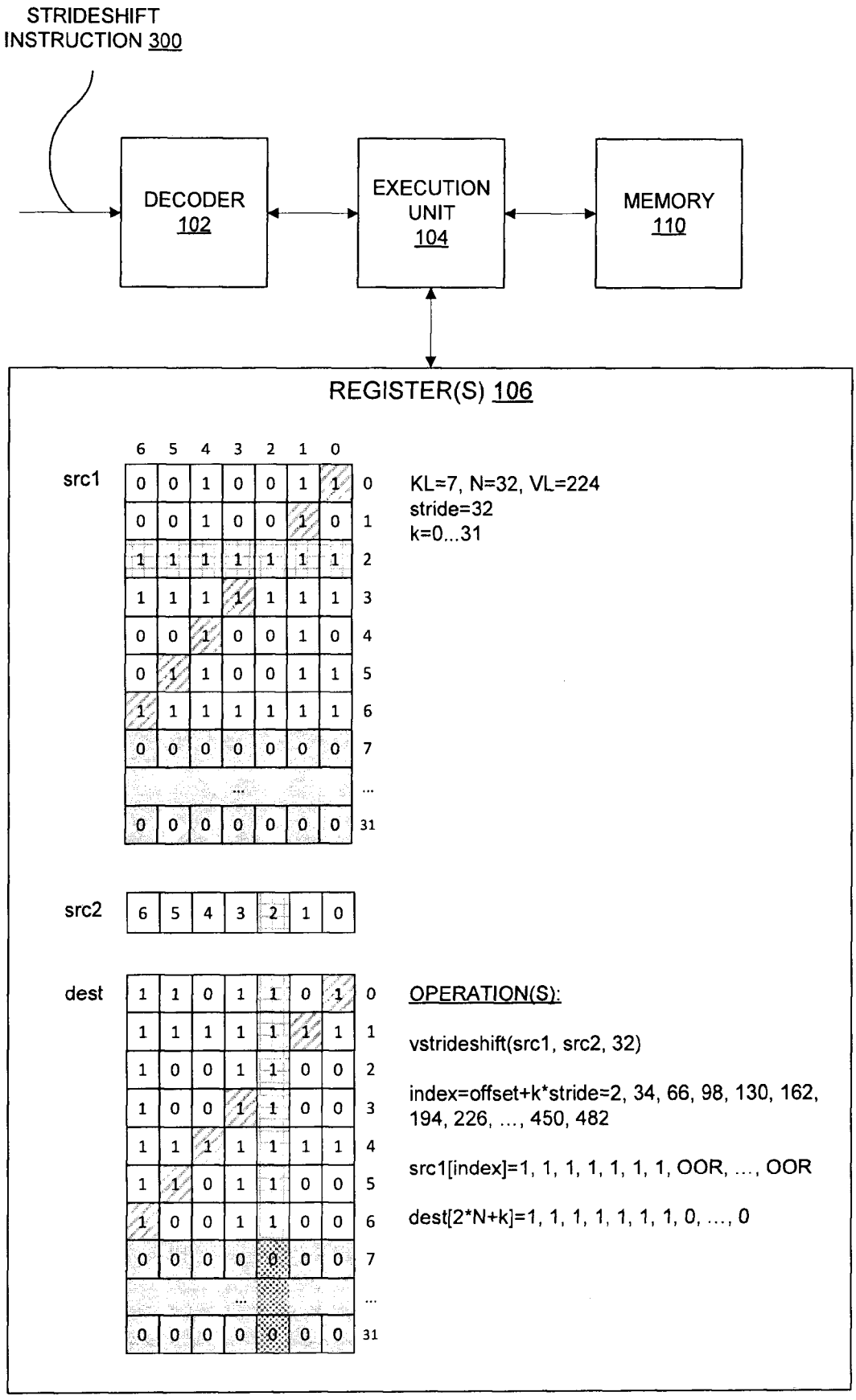

STRIDESHIFT
INSTRUCTION 300
DECODER
102
EXECUTION
UNIT
104
MEMORY
110
REGISTER(S) 106
src1
6 5 4 3 2 1 0
0 0 1 0 0 1 1 0
0 0 1 0 0 1 0 1
1 1 1 1 1 1 1 2
1 1 1 1 1 1 1 3
0 0 1 0 0 1 0 4
0 1 1 0 0 1 1 5
1 1 1 1 1 1 1 6
0 0 0 0 0 0 0 7
... ...
0 0 0 0 0 0 0 31
KL=7, N=32, VL=224
stride=32
k=0...31
src2
6 5 4 3 2 1 0
dest
1 1 0 1 1 0 1 0
1 1 1 1 1 1 1 1
1 0 0 1 1 0 0 2
1 0 0 1 1 0 0 3
1 1 1 1 1 1 1 4
1 1 0 1 1 0 0 5
1 0 0 1 1 0 0 6
0 0 0 0 0 0 0 7
... ...
0 0 0 0 0 0 0 31
OPERATION(S):
vstrideshift(src1, src2, 32)
index=offset+k*stride=2, 34, 66, 98, 130, 162, 194, 226, ..., 450, 482
src1[index]=1, 1, 1, 1, 1, 1, 1, OOR, ..., OOR
dest[2*N+k]=1, 1, 1, 1, 1, 1, 1, 0, ..., 0

Fig. 3

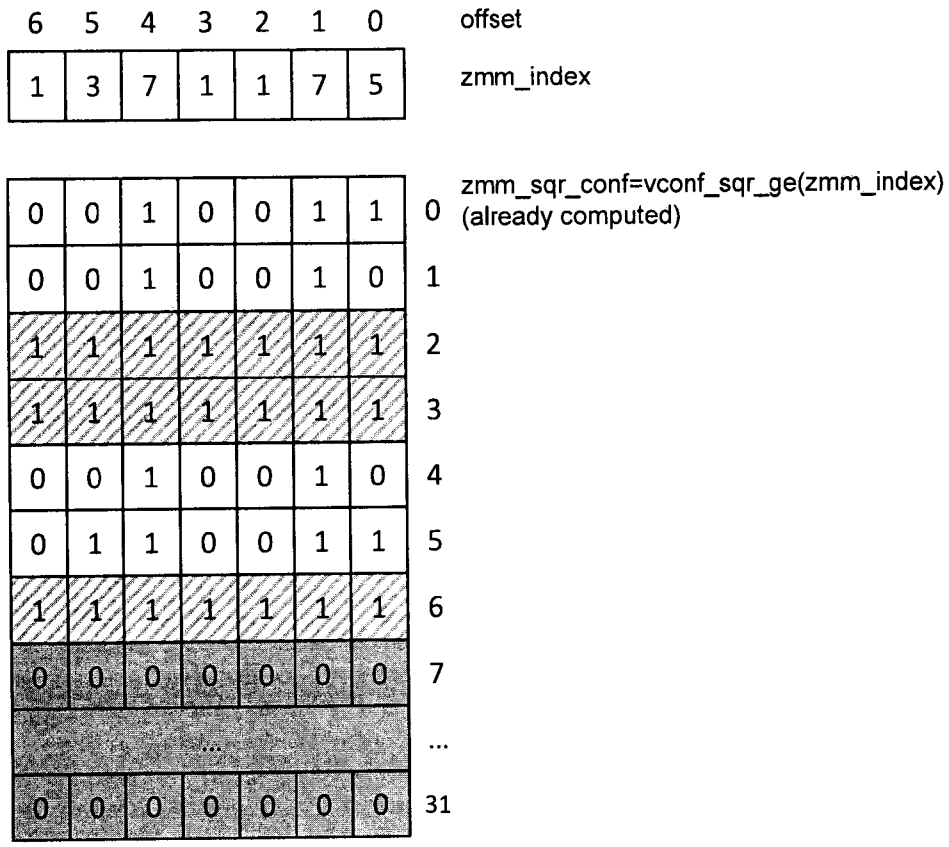

6 5 4 3 2 1 0
offset
1 3 7 1 1 7 5
zmm_index
zmm_sqr_conf=vconf_sqr_ge(zmm_index)
(already computed)
0 0 1 0 0 1 1 0
0 0 1 0 0 1 0 1
1 1 1 1 1 1 1 2
1 1 1 1 1 1 1 3
0 0 1 0 0 1 0 4
0 1 1 0 0 1 1 5
1 1 1 1 1 1 1 6
0 0 0 0 0 0 0 7
... ...
0 0 0 0 0 0 0 31

Fig. 4

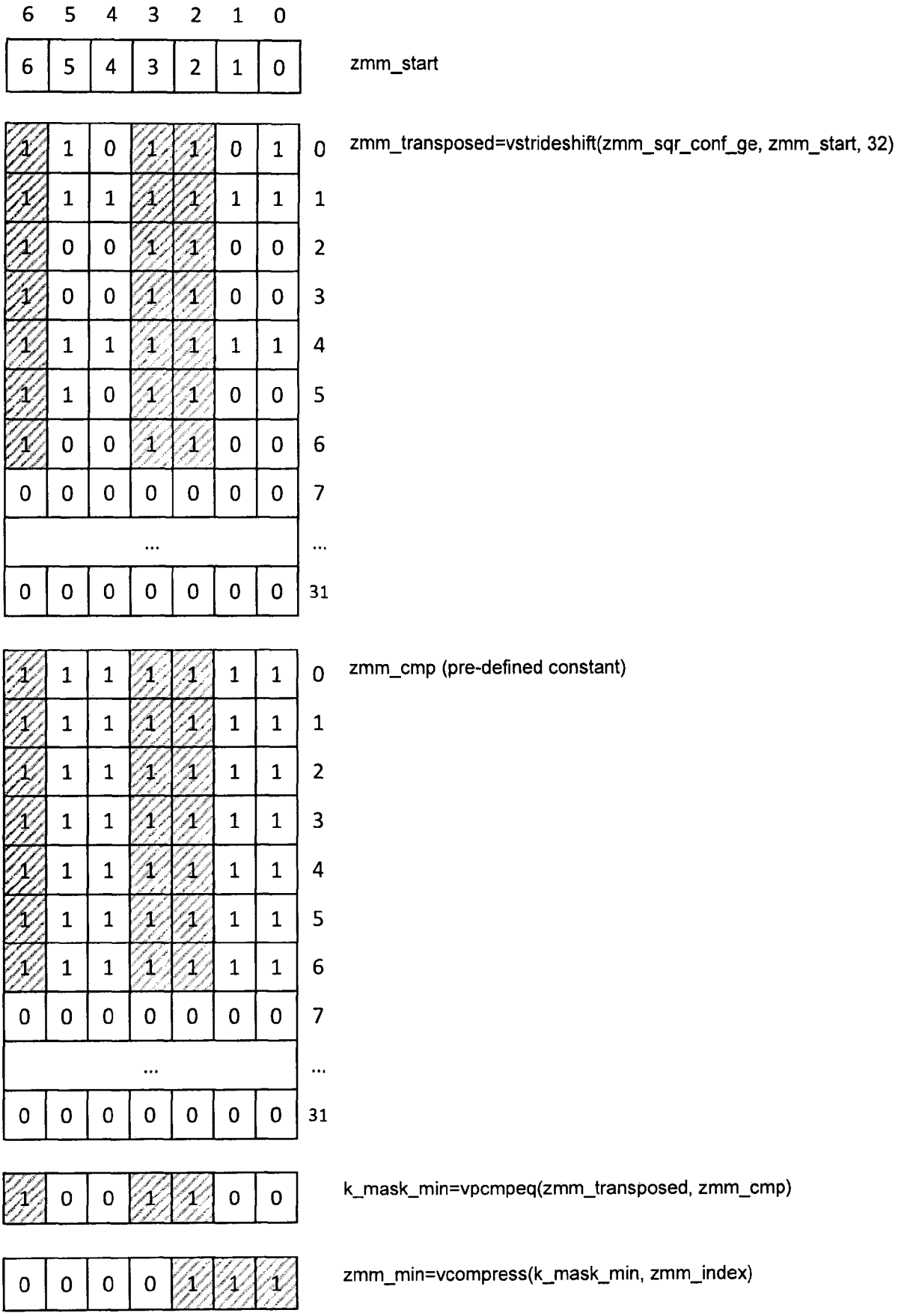

6 5 4 3 2 1 0
6 5 4 3 2 1 0
zmm_start
1 1 0 1 1 0 1 0
1 1 1 1 1 1 1 1
1 0 0 1 1 0 0 2
1 0 0 1 1 0 0 3
1 1 1 1 1 1 1 4
1 1 0 1 1 0 0 5
1 0 0 1 1 0 0 6
0 0 0 0 0 0 0 7
... ...
0 0 0 0 0 0 0 31
zmm_transposed=vstrideshift(zmm_sqr_conf_ge, zmm_start, 32)
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 2
1 1 1 1 1 1 1 3
1 1 1 1 1 1 1 4
1 1 1 1 1 1 1 5
1 1 1 1 1 1 1 6
0 0 0 0 0 0 0 7
... ...
0 0 0 0 0 0 0 31
zmm_cmp (pre-defined constant)
1 0 0 1 1 0 0
k_mask_min=vpcmpeq(zmm_transposed, zmm_cmp)
0 0 0 0 1 1 1
zmm_min=vcompress(k_mask_min, zmm_index)

Fig. 4 (CONT')

Fig. 5

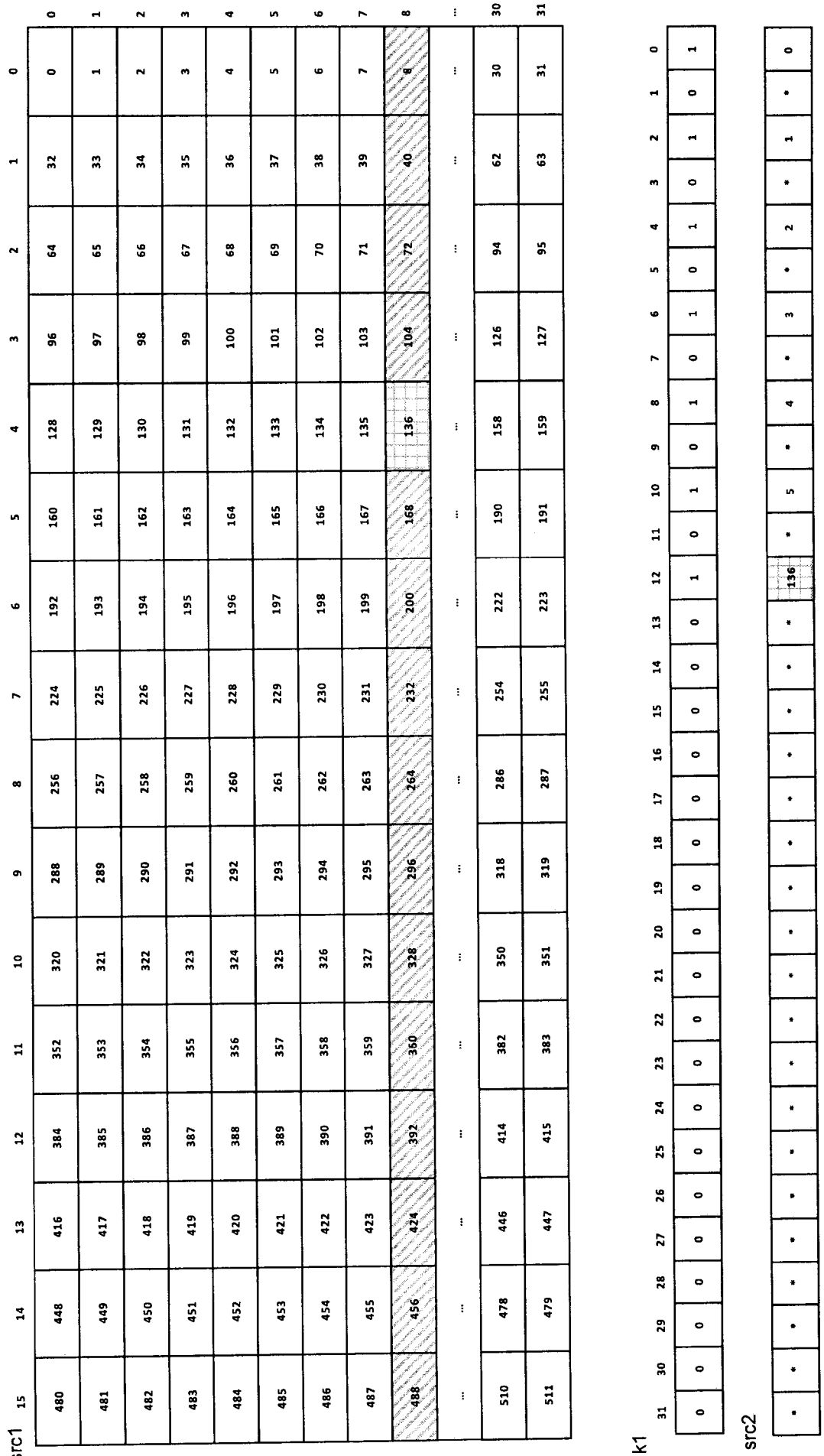
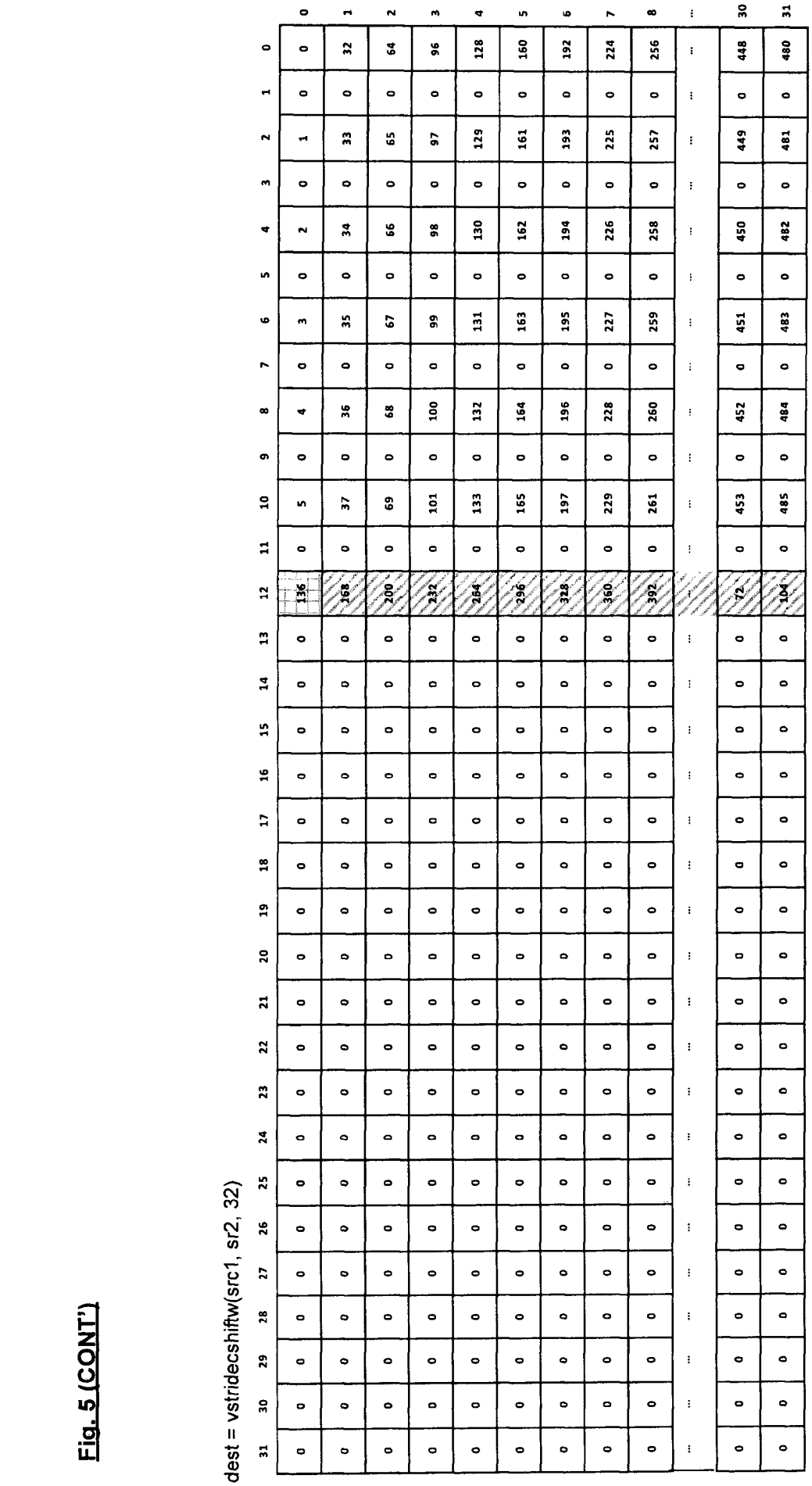


Fig. 5 (CONT')



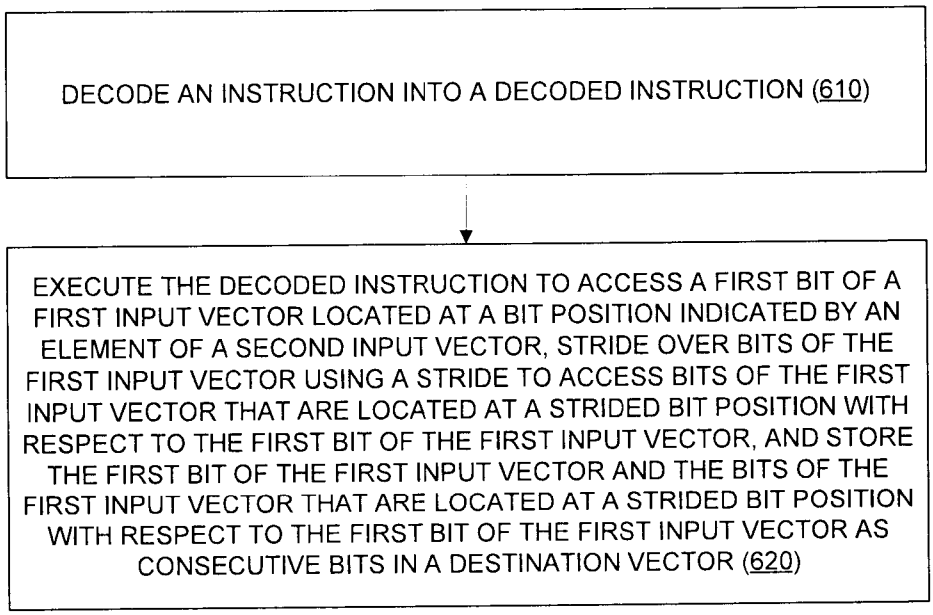

DECODE AN INSTRUCTION INTO A DECODED INSTRUCTION (610)
EXECUTE THE DECODED INSTRUCTION TO ACCESS A FIRST BIT OF A FIRST INPUT VECTOR LOCATED AT A BIT POSITION INDICATED BY AN ELEMENT OF A SECOND INPUT VECTOR, STRIDE OVER BITS OF THE FIRST INPUT VECTOR USING A STRIDE TO ACCESS BITS OF THE FIRST INPUT VECTOR THAT ARE LOCATED AT A STRIDED BIT POSITION WITH RESPECT TO THE FIRST BIT OF THE FIRST INPUT VECTOR, AND STORE THE FIRST BIT OF THE FIRST INPUT VECTOR AND THE BITS OF THE FIRST INPUT VECTOR THAT ARE LOCATED AT A STRIDED BIT POSITION WITH RESPECT TO THE FIRST BIT OF THE FIRST INPUT VECTOR AS CONSECUTIVE BITS IN A DESTINATION VECTOR (620)

Fig. 6

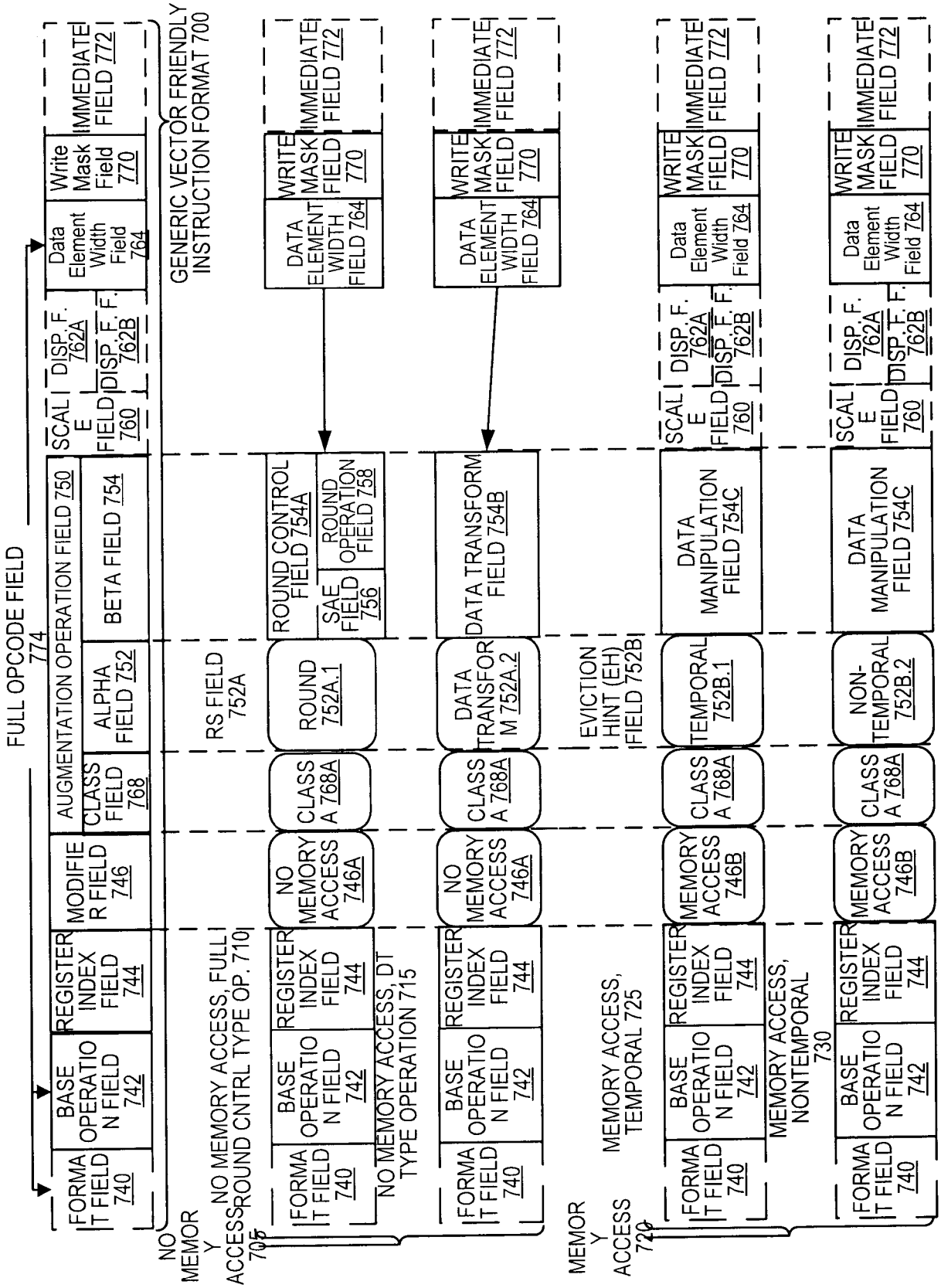

GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
FULL OPCODE FIELD 774
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
SCALE FIELD 760
DISP. F. 762A
DISP. F. 762B
Data Element Width Field 764
Write Mask Field 770
IMMEDIATE FIELD 772
NO MEMORY ACCESS
NO MEMORY ACCESS, FULL ROUND CNTRL TYPE OP. 710
NO MEMORY ACCESS, DT TYPE OPERATION 715
MEMORY ACCESS
MEMORY ACCESS, TEMPORAL 725
MEMORY ACCESS, NONTEMPORAL 730
RS FIELD 752A
ROUND 752A.1
DATA TRANSFORM 752A.2
EVICTION HINT (EH) FIELD 752B
TEMPORAL 752B.1
NON-TEMPORAL 752B.2
ROUND CONTROL FIELD 754A
SAE FIELD 756
ROUND OPERATION FIELD 758
DATA TRANSFORM FIELD 754B
DATA MANIPULATION FIELD 754C
NO MEMORY ACCESS 746A
MEMORY ACCESS 746B
CLASS A 768A
DATA ELEMENT WIDTH FIELD 764
WRITE MASK FIELD 770

FIG.7A

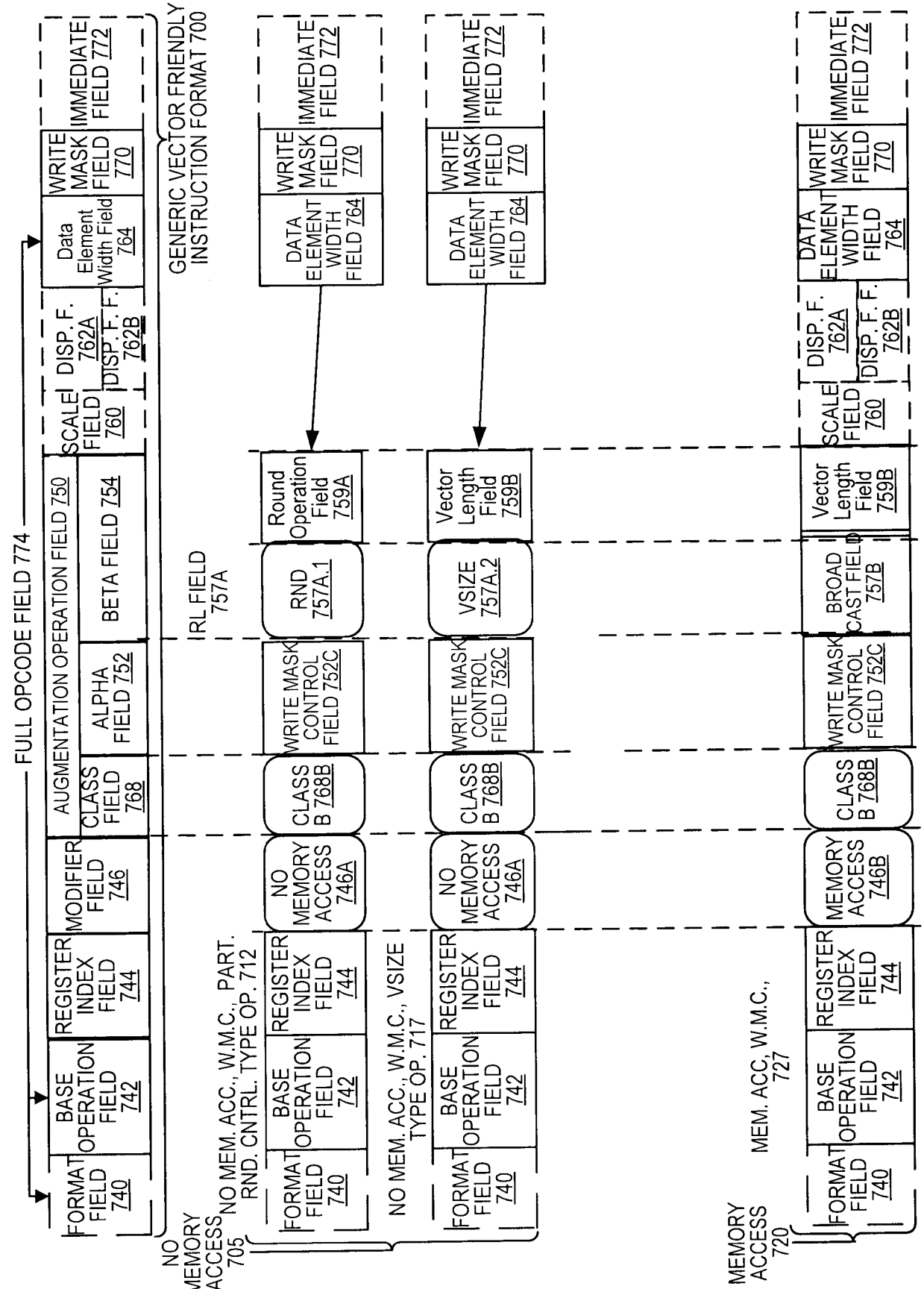

GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
FULL OPCODE FIELD 774
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
SCALE FIELD 760
DISP. F. 762A
DISP. F. F. 762B
Data Element Width Field 764
WRITE MASK FIELD 770
IMMEDIATE FIELD 772
NO MEMORY ACCESS 705
NO MEM. ACC., W.M.C., PART. RND. CNTRL. TYPE OP. 712
NO MEMORY ACCESS 746A
CLASS B 768B
WRITE MASK CONTROL FIELD 752C
RL FIELD 757A
RND 757A.1
Round Operation Field 759A
DATA ELEMENT WIDTH FIELD 764
NO MEM. ACC., W.M.C., VSIZE TYPE OP. 717
VSIZE 757A.2
Vector Length Field 759B
MEMORY ACCESS 720
MEM. ACC, W.M.C., 727
MEMORY ACCESS 746B
BROAD CAST FIELD 757B

FIG. 7B

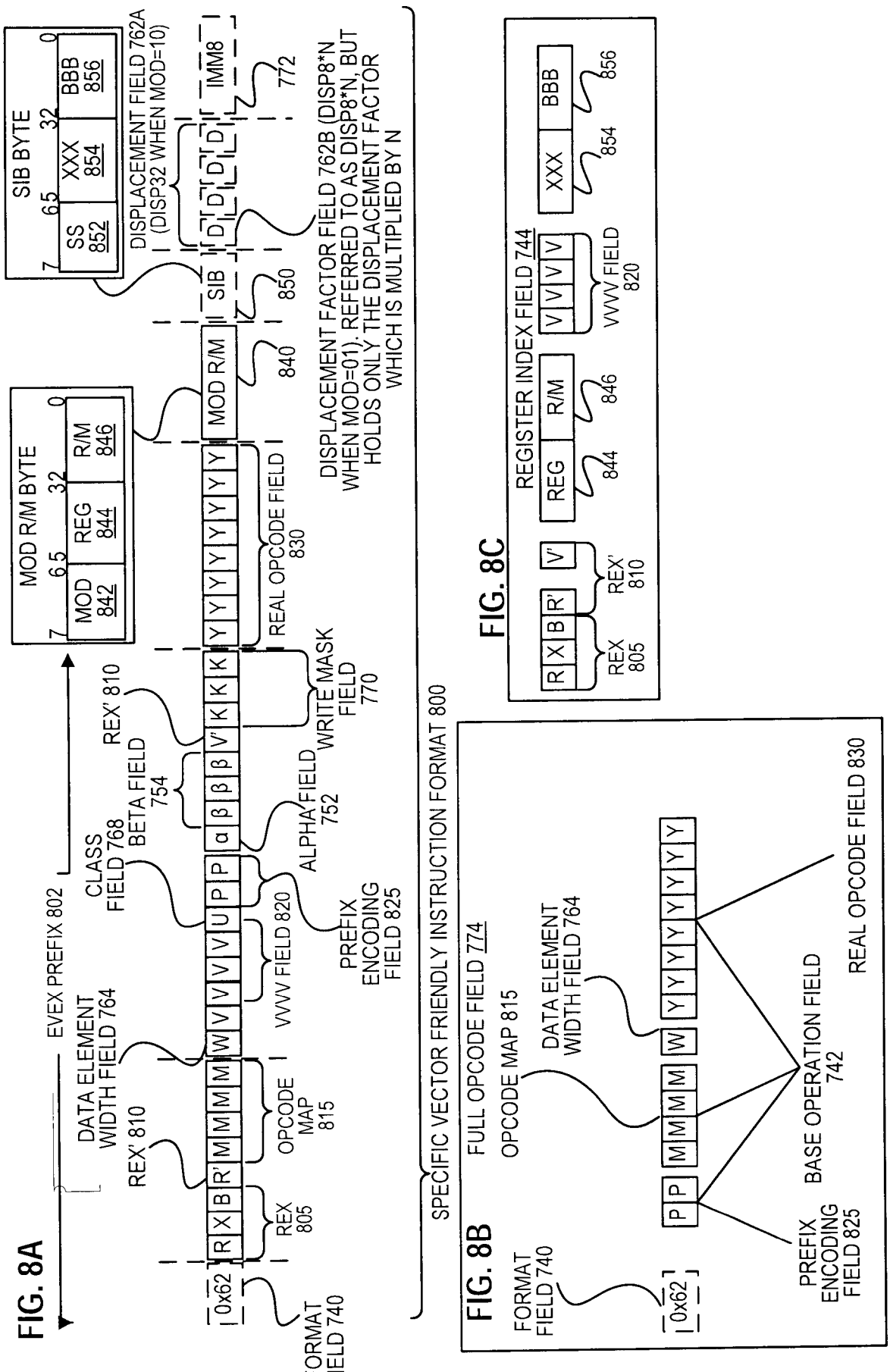
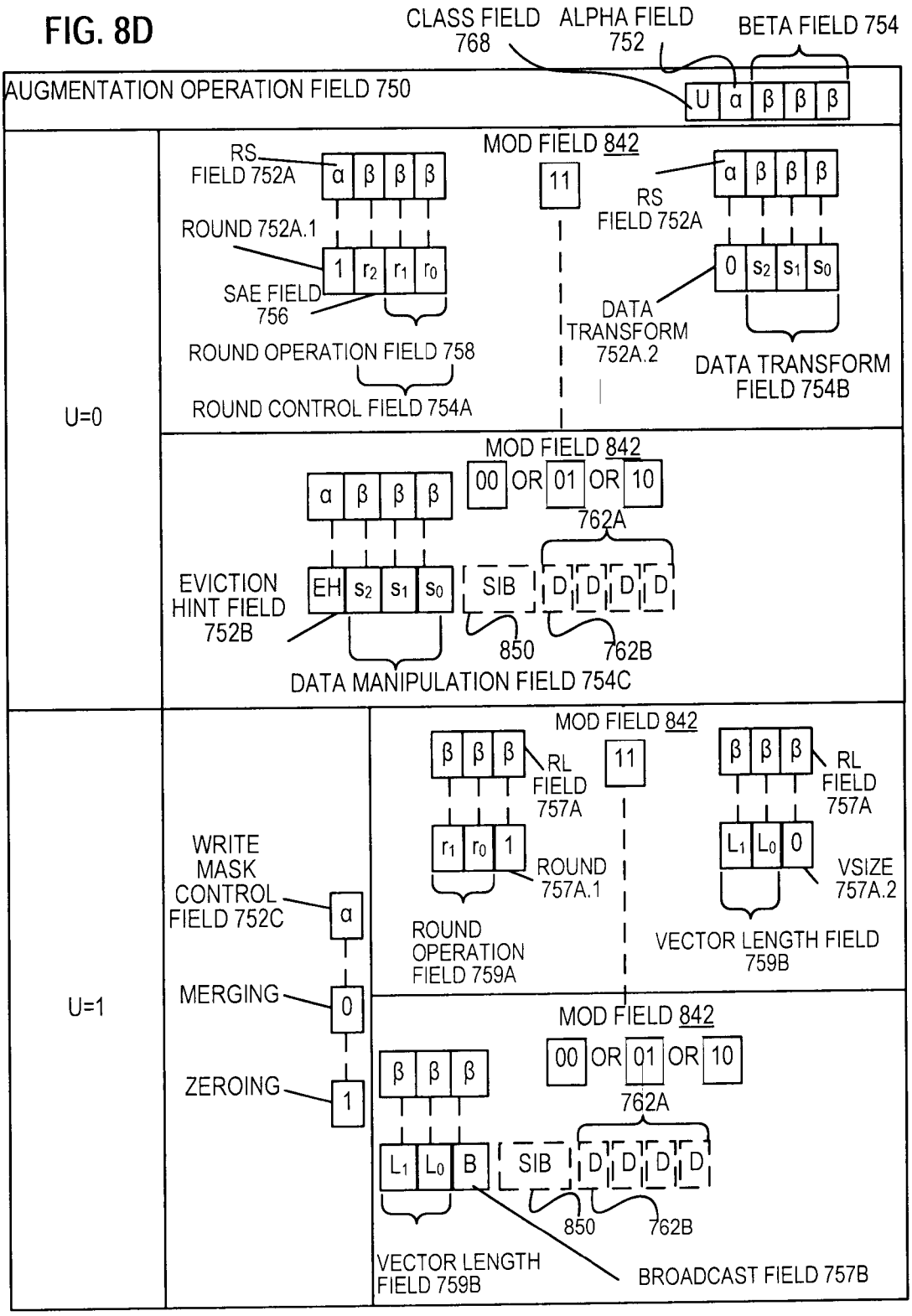


FIG. 8A
EVEX PREFIX 802
FORMAT FIELD 740
0x62
REX 805
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
REX' 810
VVV FIELD 820
CLASS FIELD 768
BETA FIELD 754
ALPHA FIELD 752
REX' 810
PREFIX ENCODING FIELD 825
WRITE MASK FIELD 770
REAL OPCODE FIELD 830
MOD R/M BYTE
MOD 842
REG 844
R/M 846
SIB BYTE
SS 852
XXX 854
BBB 856
DISPLACEMENT FIELD 762A (DISP32 WHEN MOD=10)
DISPLACEMENT FACTOR FIELD 762B (DISP8*N WHEN MOD=01). REFERRED TO AS DISP8*N, BUT HOLDS ONLY THE DISPLACEMENT FACTOR WHICH IS MULTIPLIED BY N
SPECIFIC VECTOR FRIENDLY INSTRUCTION FORMAT 800
FIG. 8B
FORMAT FIELD 740
FULL OPCODE FIELD 774
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
BASE OPERATION FIELD 742
PREFIX ENCODING FIELD 825
REAL OPCODE FIELD 830
FIG. 8C
REGISTER INDEX FIELD 744
REX 805
REX' 810
REG 844
R/M 846
VVV FIELD 820
XXX 854
BBB 856

FIG. 8D


CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
AUGMENTATION OPERATION FIELD 750
U α β β β
U=0
RS FIELD 752A
α β β β
ROUND 752A.1
1 r2 r1 r0
SAE FIELD 756
ROUND OPERATION FIELD 758
ROUND CONTROL FIELD 754A
MOD FIELD 842
11
RS FIELD 752A
α β β β
0 s2 s1 s0
DATA TRANSFORM 752A.2
DATA TRANSFORM FIELD 754B
MOD FIELD 842
α β β β
00 OR 01 OR 10
762A
EVICTION HINT FIELD 752B
EH s2 s1 s0
SIB
D D D D
850
762B
DATA MANIPULATION FIELD 754C
U=1
WRITE MASK CONTROL FIELD 752C
α
MERGING
0
ZEROING
1
MOD FIELD 842
β β β
RL FIELD 757A
11
r1 r0 1
ROUND 757A.1
ROUND OPERATION FIELD 759A
β β β
RL FIELD 757A
L1 L0 0
VSIZE 757A.2
VECTOR LENGTH FIELD 759B
MOD FIELD 842
β β β
00 OR 01 OR 10
762A
L1 L0 B
SIB
D D D D
850
762B
VECTOR LENGTH FIELD 759B
BROADCAST FIELD 757B

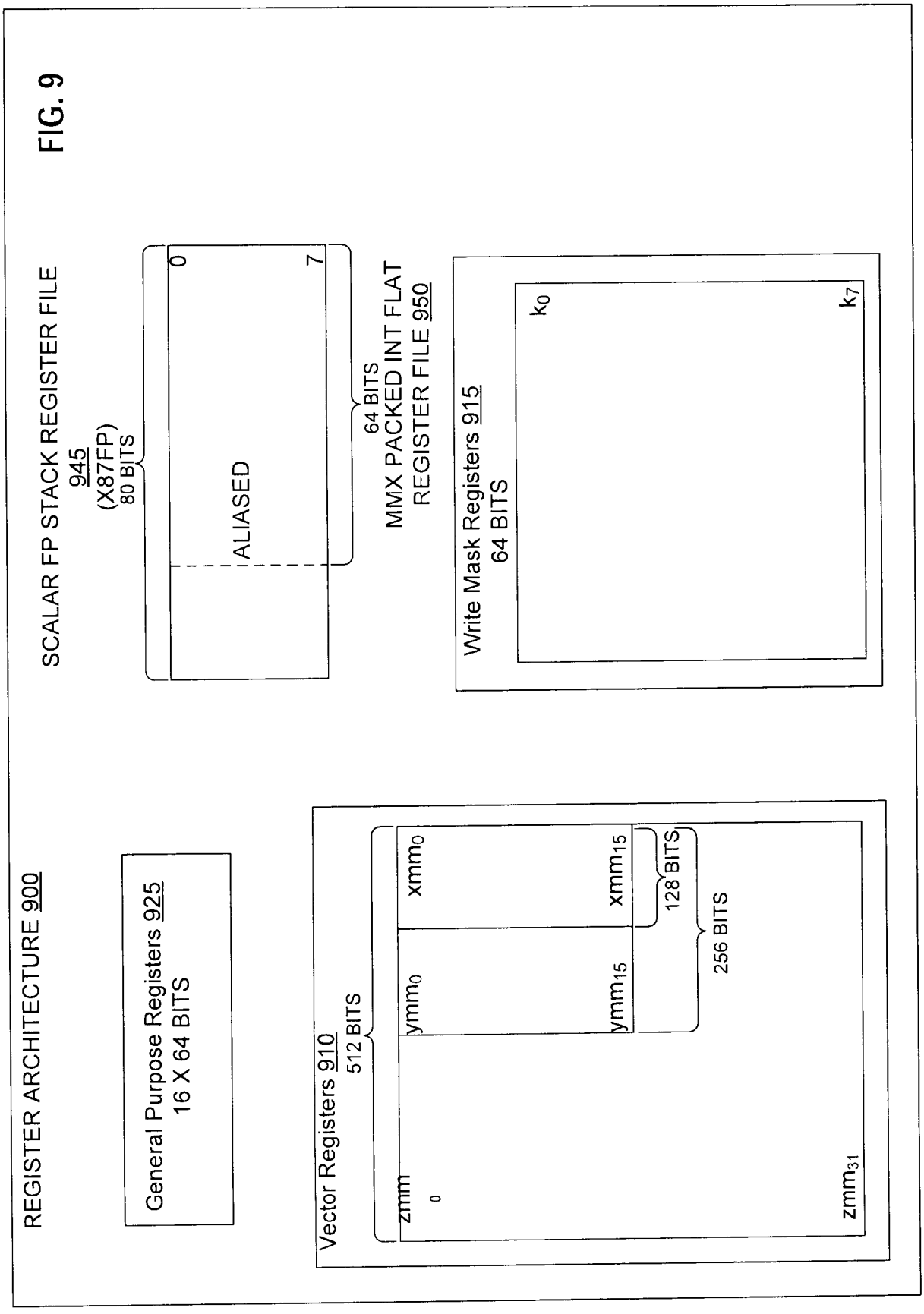

REGISTER ARCHITECTURE 900
General Purpose Registers 925
16 X 64 BITS
Vector Registers 910
512 BITS
zmm
0
ymm0
xmm0
ymm15
xmm15
128 BITS
256 BITS
zmm31
SCALAR FP STACK REGISTER FILE
945
(X87FP)
80 BITS
0
7
ALIASED
64 BITS
MMX PACKED INT FLAT
REGISTER FILE 950
Write Mask Registers 915
64 BITS
k0
k7

FIG. 9

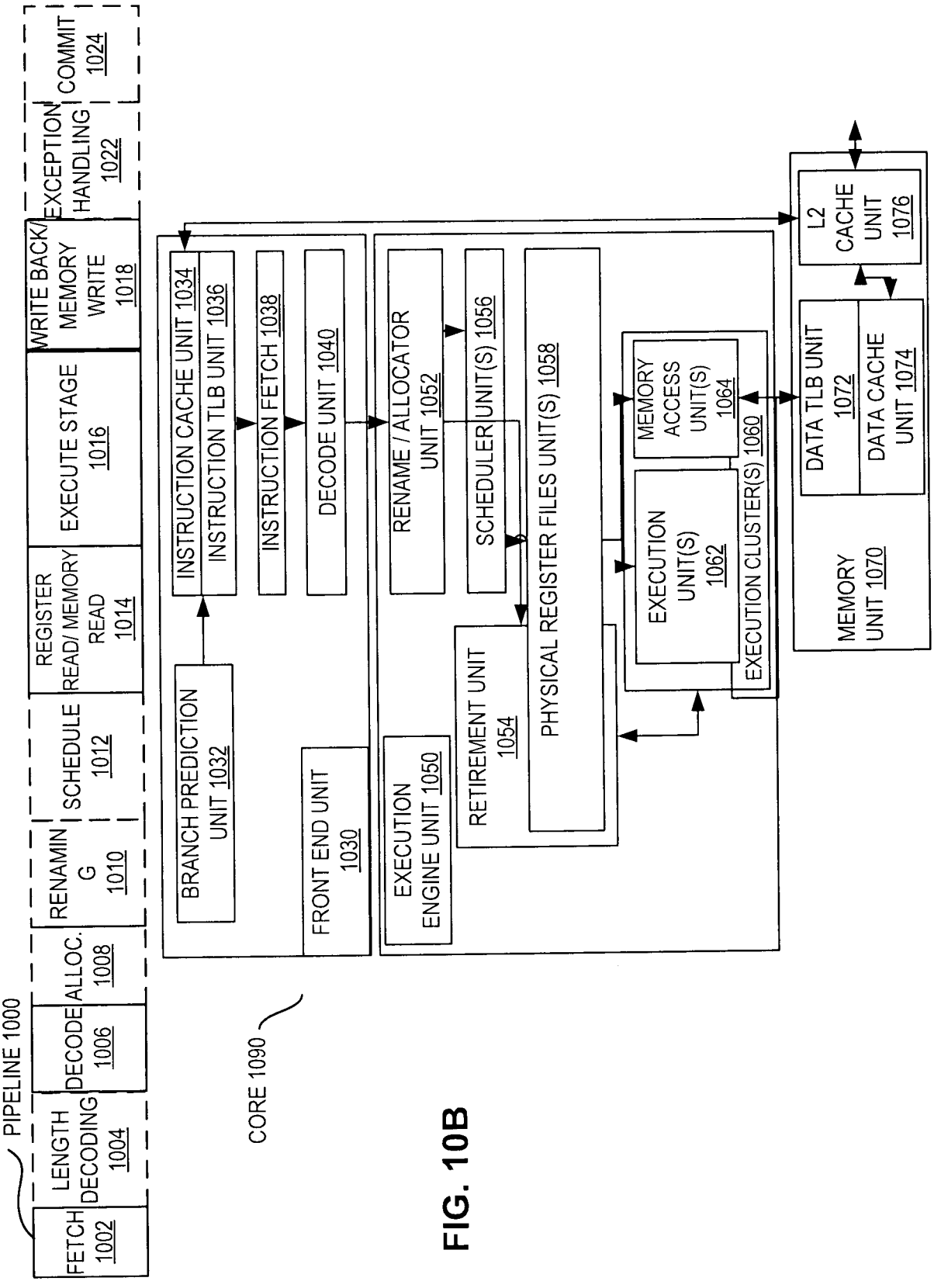


FIG. 10A

FIG. 10B

FIG. 11A

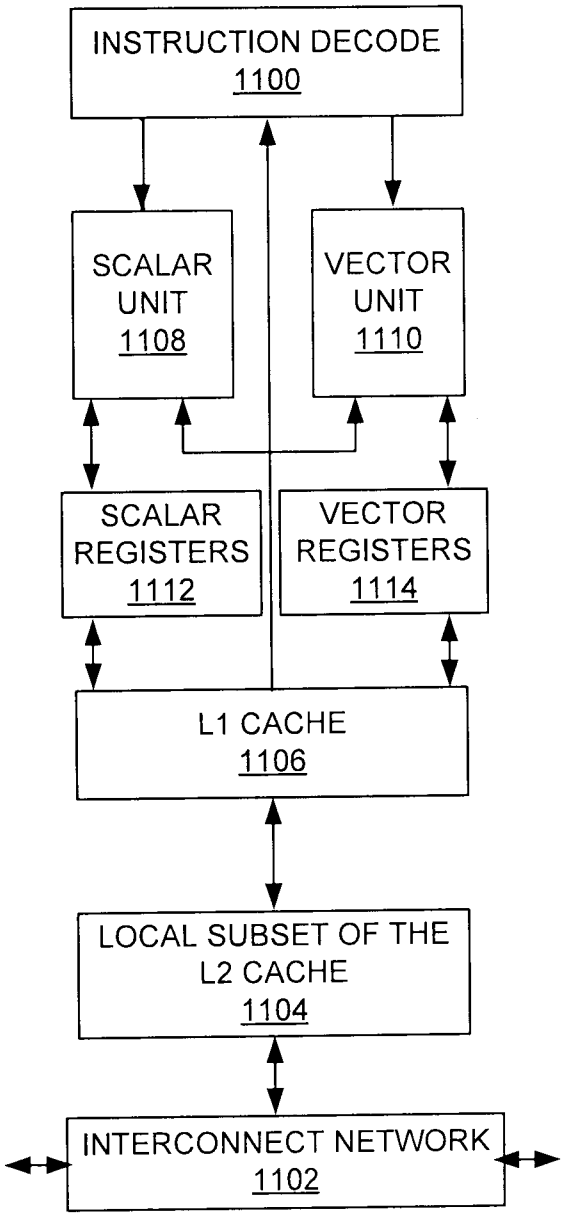
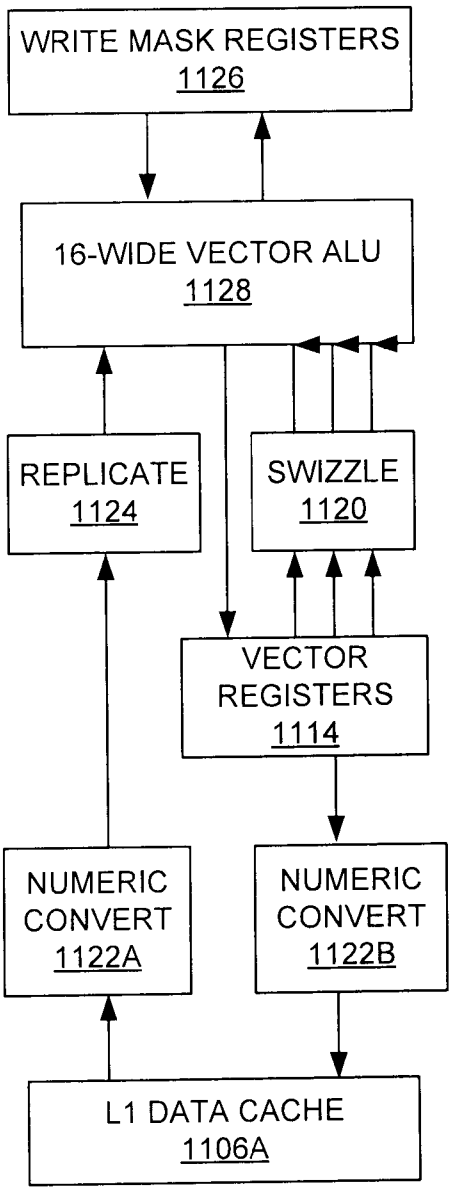


FIG. 11B



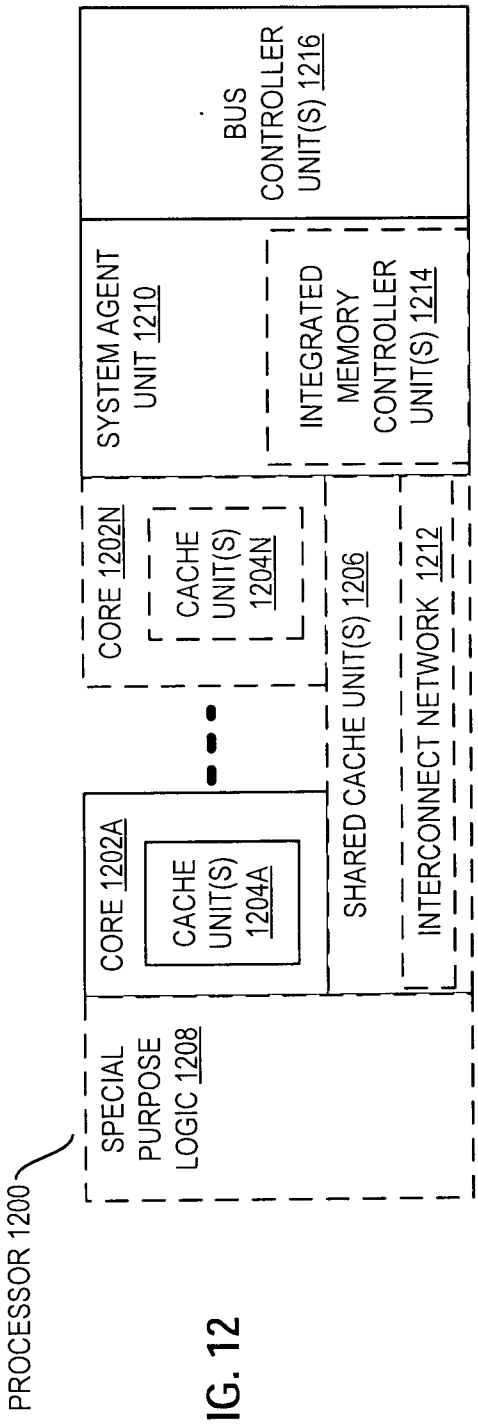

PROCESSOR 1200
SPECIAL PURPOSE LOGIC 1208
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
INTERCONNECT NETWORK 1212
SYSTEM AGENT UNIT 1210
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214
BUS CONTROLLER UNIT(S) 1216

FIG. 12

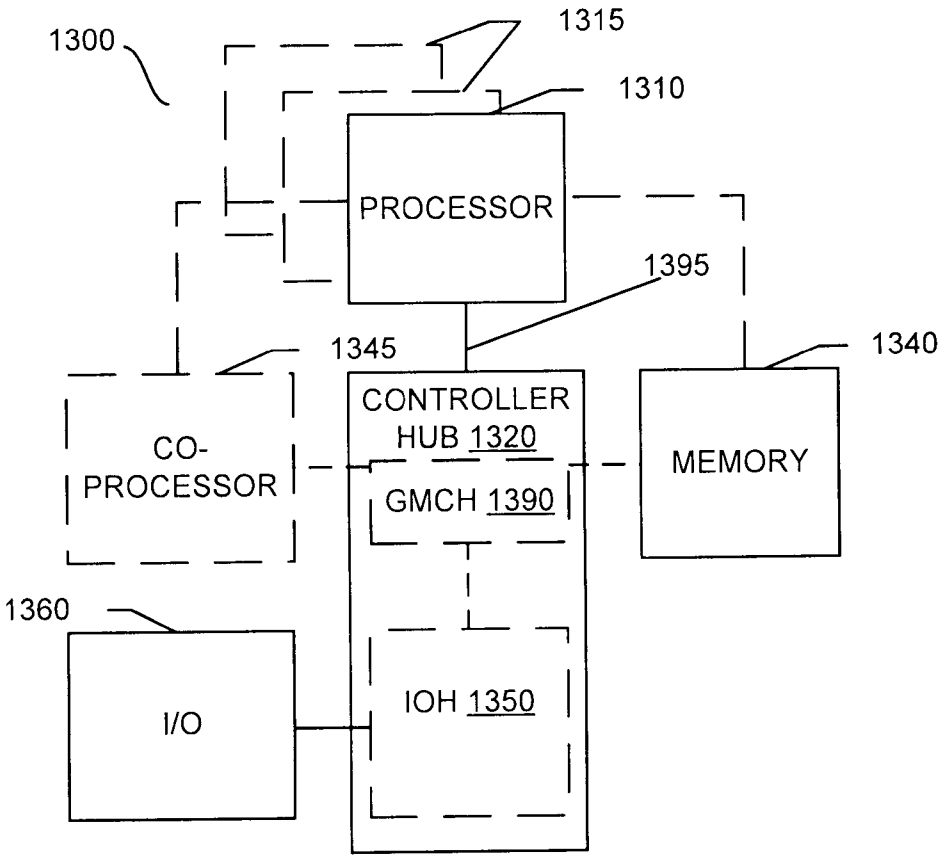

1300
1315
1310
PROCESSOR
1395
1345
1340
CONTROLLER HUB 1320
CO-PROCESSOR
MEMORY
GMCH 1390
1360
I/O
IOH 1350

FIG. 13

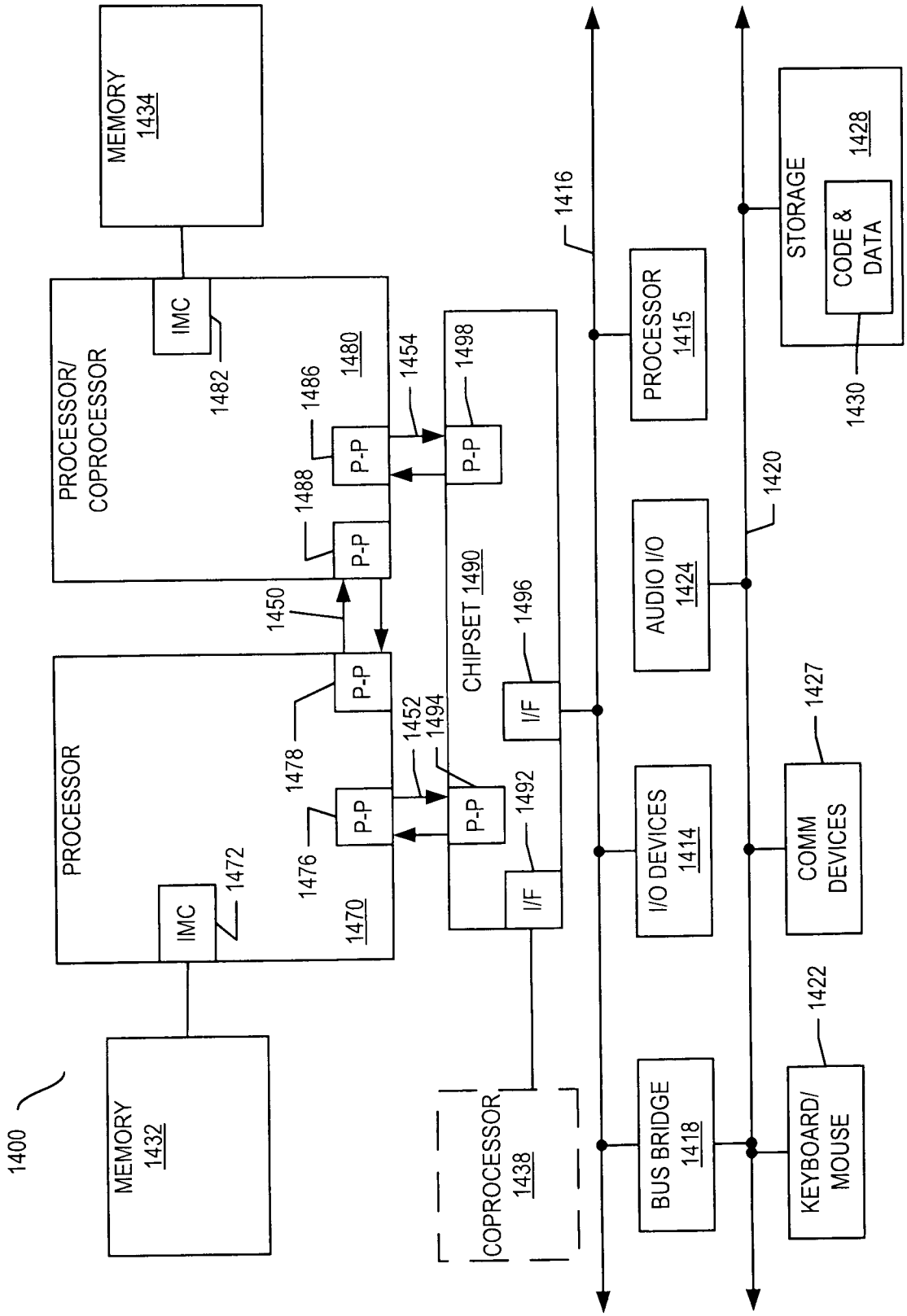

1400
MEMORY 1432
MEMORY 1434
PROCESSOR 1470
IMC 1472
PROCESSOR/ COPROCESSOR 1480
IMC 1482
P-P 1476
P-P 1478
P-P 1486
P-P 1488
1450
1452
1454
CHIPSET 1490
P-P 1492
P-P 1494
P-P 1498
I/F 1496
I/F
COPROCESSOR 1438
1416
PROCESSOR 1415
AUDIO I/O 1424
I/O DEVICES 1414
BUS BRIDGE 1418
1420
STORAGE 1428
CODE & DATA
1430
COMM DEVICES 1427
KEYBOARD/ MOUSE 1422

FIG. 14

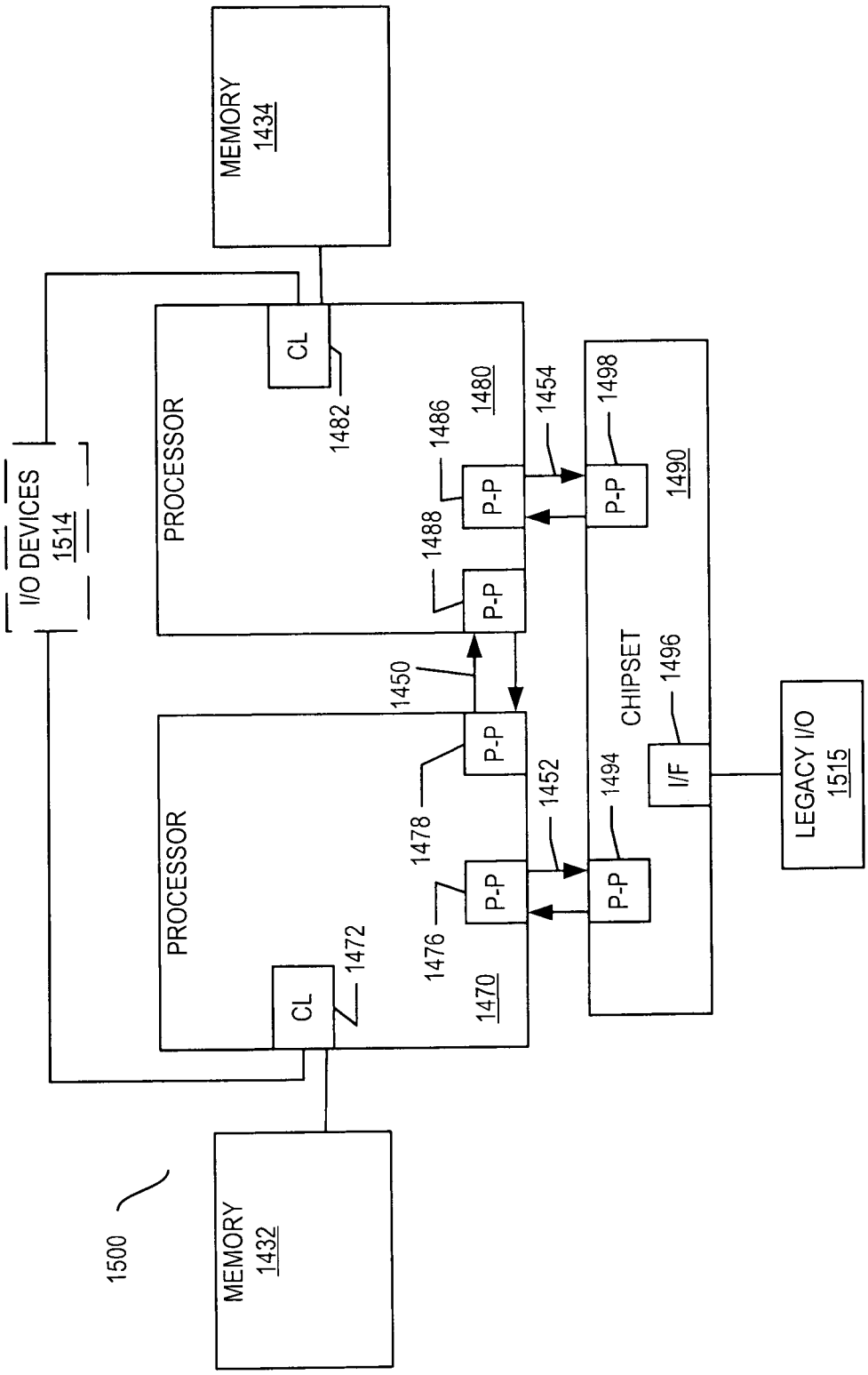

1500
MEMORY
1432
MEMORY
1434
I/O DEVICES
1514
PROCESSOR
CL
1472
1470
1476
1478
1480
1482
1486
1488
1490
1494
1496
1498
CHIPSET
I/F
LEGACY I/O
1515
P-P
1450
1452
1454

FIG. 15

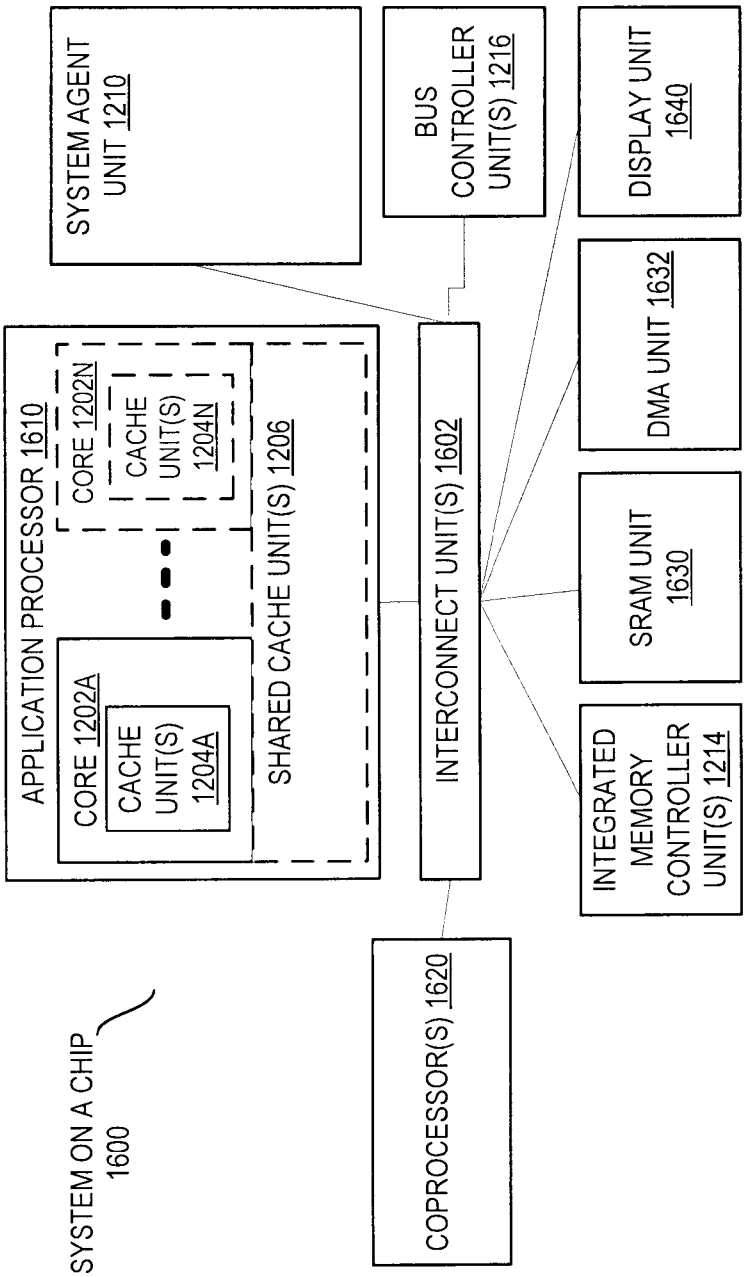

SYSTEM ON A CHIP 1600
APPLICATION PROCESSOR 1610
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
SYSTEM AGENT UNIT 1210
BUS CONTROLLER UNIT(S) 1216
INTERCONNECT UNIT(S) 1602
COPROCESSOR(S) 1620
DISPLAY UNIT 1640
DMA UNIT 1632
SRAM UNIT 1630
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214

FIG. 16

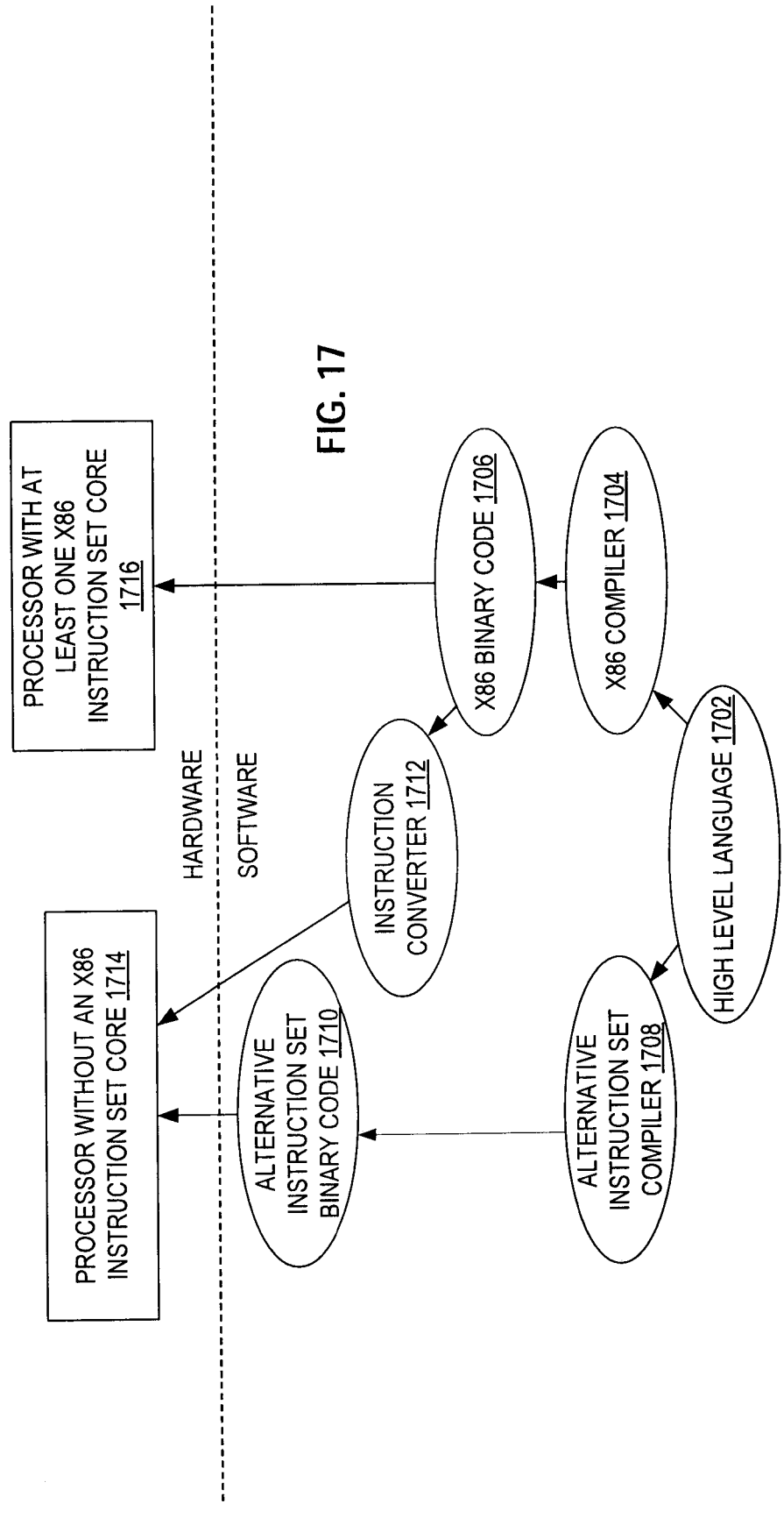

HIGH LEVEL LANGUAGE 1702
X86 COMPILER 1704
X86 BINARY CODE 1706
ALTERNATIVE INSTRUCTION SET COMPILER 1708
INSTRUCTION CONVERTER 1712
ALTERNATIVE INSTRUCTION SET BINARY CODE 1710
PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE 1714
PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE 1716
HARDWARE
SOFTWARE

FIG. 17

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2017/000333

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/30 G06F9/38
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 2012/134555 A1 (INTEL CORP [US]; VALENTINE ROBERT C [IL]; HUGHES CHRISTOPHER J [US]; S) 4 October 2012 (2012-10-04) page 3, line 26 page 36, line 13 page 44, line 14 - line 15 page 36, line 6 page 5, line 3 - line 7 page 5, line 29 page 19, line 5 - line 14 page 5, line 26 page 5, line 24 - line 27 page 5, line 4 figure 1 page 4, line 3 - line 4 ----- -/--	1-25

[X] Further documents are listed in the continuation of Box C.

[X] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
27 October 2017	10/11/2017

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Gratia, Romain

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2017/000333

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 2016/105820 A1 (INTEL CORP [US]) 30 June 2016 (2016-06-30) the whole document -----	1
A	EP 3 026 549 A2 (QUALCOMM INC [US]) 1 June 2016 (2016-06-01) the whole document -----	1

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/IB2017/000333

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 2012134555 A1	04-10-2012	CN 103562856 A	05-02-2014
		DE 112011105121 T5	09-01-2014
		GB 2503169 A	18-12-2013
		JP 5844882 B2	20-01-2016
		JP 2014513340 A	29-05-2014
		JP 2016040737 A	24-03-2016
		KR 20130137702 A	17-12-2013
		TW 201246065 A	16-11-2012
		TW 201525856 A	01-07-2015
		US 2012254591 A1	04-10-2012
		US 2015052333 A1	19-02-2015
		WO 2012134555 A1	04-10-2012
WO 2016105820 A1	30-06-2016	CN 107077333 A	18-08-2017
		EP 3238036 A1	01-11-2017
		KR 20170098806 A	30-08-2017
		SG 11201704324V A	28-07-2017
		TW 201640339 A	16-11-2016
		US 2016188335 A1	30-06-2016
		WO 2016105820 A1	30-06-2016
EP 3026549 A2	01-06-2016	CN 104583938 A	29-04-2015
		EP 2888658 A1	01-07-2015
		EP 3026549 A2	01-06-2016
		EP 3051412 A1	03-08-2016
		KR 20150047547 A	04-05-2015
		US 2014059323 A1	27-02-2014
		WO 2014031129 A1	27-02-2014