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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
27 June 2013 (27.06.2013)

WIPO | PCT

(10) International Publication Number
WO 2013/095668 Al

(51) International Patent Classification:
G06F 9/06 (2006.01) *G06F 9/305* (2006.01)
G06F 9/30 (2006.01)

(21) International Application Number:
PCT/US201 1/067275

(22) International Filing Date:
23 December 201 1 (23. 12.201 1)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant *(for all designated States except US):* **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, MS: RNB-4-150, Sunnyvale, California 95052 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only):* **OULD-AHMED-VALL, Elmoustapha** [MR/US]; 5000 West Chandler Boulevard, MS: CH7-401, Chandler, Arizona 85226 (US). **WILLHALM, Thomas** [DE/DE]; Hardstr, 51/2, 69207 Sandhausen, BW (DE).

(74) Agents: **NICHOLSON, David F.** et al; Blakely, Sokoloff, Taylor & Zafman LLP, 1279 Oakmead Parkway, Sunnyvale, California 94085 (US).

(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, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, 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, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, 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, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(ivf)*

Published:

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

WO 2013/095668 A1

(54) Title: SYSTEMS, APPARATUSES, AND METHODS FOR PERFORMING VECTOR PACKED COMPRESSION AND REPEAT

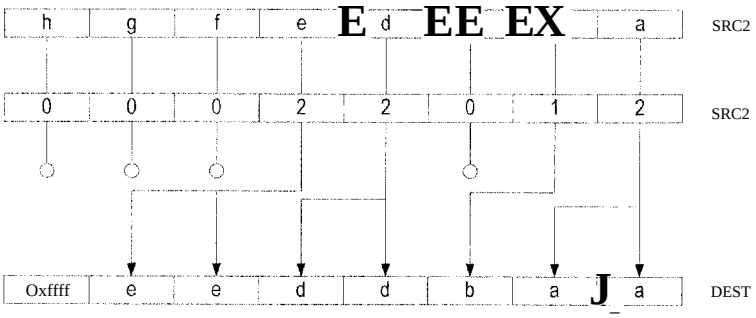


FIGURE 1

(57) Abstract: Embodiments of systems, apparatuses, and methods for performing in a computer processor vector packed compression and repeat in response to a single vector packed compression and repeat instruction that includes a first and second source vector register operand, a destination vector register operand, and an opcode are described.

SYSTEMS, APPARATUSES, AND METHODS FOR PERFORMING VECTOR PACKED COMPRESSION AND REPEAT

FIELD OF INVENTION

[0001] The field of invention relates generally to computer processor architecture, and, more specifically, to instructions which when executed cause a particular result.

BACKGROUND

[0002] An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, and may include the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). The term instruction generally refers herein to macro-instructions - that is instructions that are provided to the processor (or instruction converter that translates (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morphs, emulates, or otherwise converts an instruction to one or more other instructions to be processed by the processor) for execution - as opposed to micro-instructions or micro-operations (micro-ops) - that is the result of a processor's decoder decoding macro-instructions.

[0003] The ISA is distinguished from the microarchitecture, which is the internal design of the processor implementing the instruction set. Processors with different microarchitectures can share a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale CA implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. For example, the same register architecture of the ISA may be implemented in different ways in different microarchitectures using well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB), and a retirement register file; the use of multiple maps and a pool of registers), etc. Unless otherwise specified, the phrases register architecture, register file, and register are used herein to refer to that which is visible to the

software/programmer and the manner in which instructions specify registers. Where a specificity is desired, the adjective logical, architectural, or software visible will be used to indicate registers/files in the register architecture, while different adjectives will be used to designation registers in a given microarchitecture (e.g., physical register, reorder buffer, retirement register, register pool).

[0004] An instruction set includes one or more instruction formats. A given instruction format defines various fields (number of bits, location of bits) to specify, among other things, the operation to be performed (opcode) and the operand(s) on which that operation is to be performed. Some instruction formats are further broken down though the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (sourcel/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.

[0005] Scientific, financial, auto-vectorized general purpose, RMS (recognition, mining, and synthesis), and visual and multimedia applications (e.g., 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation) often require the same operation to be performed on a large number of data items (referred to as "data parallelism"). Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data items. SIMD technology is especially suited to processors that can logically divide the bits in a register into a number of fixed-sized data elements, each of which represents a separate value. For example, the bits in a 256-bit register may be specified as a source operand to be operated on as four separate 64-bit packed data elements (quad-word (Q) size data elements), eight separate 32-bit packed data

elements (double word (D) size data elements), sixteen separate 16-bit packed data elements (word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). This type of data is referred to as packed data type or vector data type, and operands of this data type are referred to as packed data operands or vector operands. In other words, a packed data item or vector refers to a sequence of packed data elements, and a packed data operand or a vector operand is a source or destination operand of a SIMD instruction (also known as a packed data instruction or a vector instruction).

[0006] By way of example, one type of SIMD instruction specifies a single vector operation to be performed on two source vector operands in a vertical fashion to generate a destination vector operand (also referred to as a result vector operand) of the same size, with the same number of data elements, and in the same data element order. The data elements in the source vector operands are referred to as source data elements, while the data elements in the destination vector operand are referred to a destination or result data elements. These source vector operands are of the same size and contain data elements of the same width, and thus they contain the same number of data elements. The source data elements in the same bit positions in the two source vector operands form pairs of data elements (also referred to as corresponding data elements; that is, the data element in data element position 0 of each source operand correspond, the data element in data element position 1 of each source operand correspond, and so on). The operation specified by that SIMD instruction is performed separately on each of these pairs of source data elements to generate a matching number of result data elements, and thus each pair of source data elements has a corresponding result data element. Since the operation is vertical and since the result vector operand is the same size, has the same number of data elements, and the result data elements are stored in the same data element order as the source vector operands, the result data elements are in the same bit positions of the result vector operand as their corresponding pair of source data elements in the source vector operands. In addition to this exemplary type of SIMD instruction, there are a variety of other types of SIMD instructions (e.g., that has only one or has more than two source vector operands, that operate in a horizontal fashion, that generates a result vector operand that is of a different size, that has a different size data elements, and/or that has a different data element order). It should be understood that the term destination vector

operand (or destination operand) is defined as the direct result of performing the operation specified by an instruction, including the storage of that destination operand at a location (be it a register or at a memory address specified by that instruction) so that it may be accessed as a source operand by another instruction (by specification of that same location by the another instruction).

[0007] The SIMD technology, such as that employed by the Intel ® Core™ processors having an instruction set including x86, MMX™, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, and SSE4.2 instructions, has enabled a significant improvement in application performance. An additional set of SIMD extensions, referred to the Advanced Vector Extensions (AVX) (AVXl and AVX2) and using the Vector Extensions (VEX) coding scheme, has been , has been released and/or published (e.g., see Intef 64 and IA-32 Architectures Software Developers Manual, October 2011; and see Intef Advanced Vector Extensions Programming Reference, June 2011).

Brief Description of the Drawings

[0008] The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements and in which:

[0009] **Figure 1** illustrates an exemplary illustration of an operation of exemplary instructions for VPCOMPRESSN.

[0010] **Figure 2** illustrates an embodiment of the use of a VPCOMPRESSN instruction in a processor.

[0011] **Figure 3** illustrates an embodiment of a method for processing a VPCOMPRESSN instruction.

[0012] **Figure 4** illustrates exemplary pseudo-code for processing this instruction.

[0013] **Figure 5** illustrates a correlation between the number of one active bit vector writemask elements and the vector size and the data element size according to one embodiment of the invention.

[0014] **Figure 6A** illustrates an exemplary AVX instruction format.

[0015] **Figure 6B** illustrates which fields from Figure 6A make up a full opcode field and a base operation field.

[0016] **Figure 6C** illustrates which fields from Figure 6A make up a register index field.

[0017] **Figures 7A-7B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention.

[0018] **Figures 8A-D** are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention.

[0019] **Figure 9** is a block diagram of a register architecture 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 12** is a block diagram of a processor that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention.

[0024] **Figure 13** is a block diagram of an exemplary system in accordance with an embodiment of the present invention.

[0025] **Figure 14** is a block diagram of a first more specific exemplary system in accordance with an embodiment of the present invention.

[0026] **Figure 15** is a block diagram of a second more specific exemplary system in accordance with an embodiment of the present invention.

[0027] **Figure 16** is a block diagram of a SoC in accordance with an embodiment of the present invention.

[0028] **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

[0029] In the following description, numerous specific details are set forth. However, it is understood that embodiments of the invention may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description.

[0030] 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 may 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.

[0031] Overview

[0032] In the description below, there are some items that may need explanation prior to describing the operations of this particular instruction in the instruction set architecture. One such item is called a "writemask register" which is generally used to predicate an operand to conditionally control per-element computational operation (below, the term mask register may also be used and it refers to a writemask register such as the ***"k"*** registers discussed below). As used below, a writemask register stores a plurality of bits (16, 32, 64, etc.) wherein each active bit of the writemask register governs the operation/update of a packed data element of a vector register during SIMD processing. Typically, there is more than one writemask register available for use by a processor core.

[0033] The instruction set architecture includes at least some SIMD instructions that specify vector operations and that have fields to select source registers and/or destination registers from these vector registers (an exemplary SIMD instruction may specify a vector operation to be performed on the contents of one or more of the vector registers, and the result of that vector operation to be stored in one of the vector registers). Different embodiments of the invention may have different sized vector registers and support more/less/different sized data elements.

[0034] The size of the multi-bit data elements specified by a SIMD instruction (e.g., byte, word, double word, quad word) determines the bit locations of the "data element positions" within a vector register, and the size of the vector operand determines the number of data elements. A packed data element refers to the data stored in a particular position. In other words, depending on the size of the data elements in the destination operand and the size of the destination operand (the total number of bits in the destination operand) (or put another way, depending on the size of the destination operand and the number of data elements within the destination operand), the bit locations of the multi-bit data element positions within the resulting vector operand change (e.g., if the destination for the resulting vector operand is a vector register, then the bit locations of the multi-bit data element positions within the destination vector register change). For example, the bit locations of the multi-bit data elements are different between a vector operation that operates on 32-bit data elements (data element position 0 occupies bit locations 31:0, data element position 1 occupies bit locations 63:32, and so on) and a vector operation that operates on 64-bit data elements (data element position 0 occupies bit locations 63:0, data element position 1 occupies bit locations 127:64, and so on).

[0035] Additionally, there is a correlation between the number of one active bit vector writemask elements and the vector size and the data element size according to one embodiment of the invention as shown in Figure 5. Vector sizes of 128-bits, 256-bits, and 512-bits are shown, although other widths are also possible. Data element sizes of 8-bit bytes (B), 16-bit words (W), 32-bit doublewords (D) or single precision floating point, and 64-bit quadwords (Q) or double precision floating point are considered, although other widths are also possible. As shown, when the vector size is 128-bits, 16-bits may be used for masking when the vector's data element size is 8-bits, 8-bits may be used for masking when the vector's data element size is 16-bits, 4-bits may be used for masking when the vector's data element size is 32-bits, and 2-bits may be used for masking when the vector's data element size is 64-bits. When the vector size is 256-bits, 32-bits may be used for masking when the packed data element width is 8-bits, 16-bits may be used for masking when the vector's data element size is 16-bits, 8-bits may be used for masking when the vector's data element size is 32-bits, and 4-bits may be used for masking when the vector's data element size is 64-bits. When the vector

size is 512-bits, 64-bits may be used for masking when the vector's data element size is 8-bits, 32-bits may be used for masking when the vector's data element size is 16-bits, 16-bits may be used for masking when the vector's data element size is 32-bits, and 8-bits may be used for masking when the vector's data element size is 64-bits.

[0036] Depending upon the combination of the vector size and the data element size, either all 64-bits, or only a subset of the 64-bits, may be used as a write mask. Generally, when a single, per-element masking control bit is used, the number of bits in the vector writemask register used for masking (active bits) is equal to the vector size in bits divided by the vector's data element size in bits.

[0037] Below are embodiments of an instruction generically called a vector packed compression and repeat ("VPCOMPRESSN") instruction and embodiments of systems, architectures, instruction formats, etc. that may be used to execute such an instruction that is beneficial in several different areas. This instruction implements a restricted permutation where the order of the elements is maintained but elements can be repeated or skipped. The execution of a VPCOMPRESSN instruction causes the selective storage of packed data operands of a first source vector register into one or more packed data element positions of the destination vector register based on the contents of a second source vector register. More particularly, for each packed data element position of the first source vector register, a corresponding packed data element position of the second vector register determines the number of times (from 0 to n) that the packed data of the packed data element position of the first source register is to be written into previously unused least significant packed data element positions of the destination vector register.

[0038] Figure 1 illustrates an exemplary illustration of an operation of exemplary instructions for VPCOMPRESSN. In the illustrated example all of the registers have 8 packed data element positions. Of course this is merely an example of sizes.

[0039] As detailed above, for each data element position of the first source vector register, beginning at the least significant data element position, a corresponding data element position of the second source vector register determines the number of times that the data of that data element position will be copied (stored) into the destination vector register beginning with the least significant data element position.

[0040] Exemplary Format

[0041] An exemplary format of this instruction is "VPCOMPRESSN DEST, SRC1, SRC2" where DEST, SRC1, and SRC2 are all vector registers (such as 128-, 256-, 512-bit register, etc.) and VPCOMPRESSN is the instruction's opcode. SRC1 and SRC2 are the first and second source operands respectively and DEST is the destination operand. In some embodiments, the VPCOMPRESSN instruction operates on memory locations instead of registers and/or uses immediate values for the source. The size of the data elements may be defined in the "prefix" of the instruction such as through the use of an indication of data granularity bit. In most embodiments, this bit will indicate that each data elements are either 32 or 64 bits, however, other variations may be used. In some embodiments, the source operand is not a register, but rather a memory location.

[0042] Exemplary Methods of Execution

[0043] Figure 2 illustrates an embodiment of the use of a VPCOMPRESSN instruction in a processor. A VPCOMPRESSN instruction with a two source vector register operands and a destination vector register operand is fetched at 201.

[0044] The VPCOMPRESSN instruction is decoded by decoding logic at 203. Depending on the instruction's format, a variety of data may be interpreted at this stage such as if there is to be a data transformation, which registers to write to and retrieve, what memory address to access, etc.

[0045] The source operand values are retrieved/read at 205. For example, the source registers are read. If one or both of the source operands is a memory operand, then the data elements associated with that operand are retrieved. In some embodiments, data elements from memory are stored into a temporary register.

[0046] The VPCOMPRESSN instruction (or operations comprising such an instruction such as microoperations) is executed by execution resources such as one or more functional units at 207 to determine, for each packed data element position of the first source vector register, how many times that packed data element position's packed data element is to be stored in the destination vector register based on the value of a corresponding packed data element position of the second source vector register. This evaluation begins at the least significant packed data element position of the source vector registers.

[0047] The determined packed data elements of the first source vector register are stored value times in the destination register at 209. While 207 and 209 have been illustrated separately, in some embodiments they are performed together as a part of the execution of the instruction. This storage begins at the least significant packed data element position of the destination vector register as illustrated in Figure 1.

[0048] Figure 3 illustrates an embodiment of a method for processing a VPCOMPRESSN instruction. In this embodiment it is assumed that some, if not all, of the operations 201-205 have been performed earlier, however, they are not shown in order to not obscure the details presented below. For example, the fetching and decoding are not shown, nor is the operand retrieval shown. In some embodiments, all of the packed data elements of the destination vector register are set to "0" prior any re-writing of its packed data elements.

[0049] At 301, a determination of a value of a least significant packed data element position of the second source vector register is made. For example, what is the value of SRC2[0]? After that determination is made, another determination is made as to whether or not the value is greater than zero at 303.

[0050] If the value is zero, then the next least significant packed data element position of the second source vector register is evaluated for its value at 305. Since the previously evaluated packed data element position was zero then the corresponding data element of the first source register will not be stored into the destination register.

[0051] If the value is not zero, then a write of the corresponding packed data element position of the first source vector register is into "value" number of the least significant unused packed data element positions of the destination register is made at 311. For example, SRC[0]'s value of "a" is written in the two least significant unused packed data element positions of the destination register (i.e., DST[1:0]). If the packed data element positions of the destination vector register have previously been set to all zero, then a simple check of which ones are not still zero will determine the least significant unused packed data element positions of the destination register.

[0052] After the data has been written, in some embodiments a check is made to determine if all of the packed data element positions of the second source vector register have been evaluated at 309. If so, then the instruction is complete. Depending on the embodiment, if all

of the bit values have been evaluated, the method may be complete if all of the packed data element positions of the destination register have been written to. However, in some embodiments, if there are some unused (unwritten) packed data element positions in the destination vector register a preset value is written to all of these unused packed data element positions. For example, all "Is" or all "0s" may be written to each of these unused packed data element positions.

[0053] If not, then in some embodiments a check is made to determine if all of there are any unused packed data element positions left in the destination register at 313. If there are not, then a programmer visible exception may be thrown to give an indication of where the instruction left off.

[0054] If there are unused packed data element positions left in the destination register, then the method continues at step 305.

[0055] Additionally, while the above description indicates a serial write, in some embodiments no data is written until all of the second source's packed data element positions have been evaluated.

[0056] Figure 4 illustrates exemplary pseudo-code for processing this instruction.

[0057] Exemplary Instruction Formats

[0058] Embodiments of the instruction(s) described herein may be embodied in different formats. For example, the instruction(s) described herein may be embodied as a VEX, generic vector friendly, or other format. Details of VEX and a generic vector friendly format are discussed below. 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.

[0059] VEX Instruction Format

[0060] VEX encoding allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 128 bits. The use of a VEX prefix provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as A = A + B, which overwrites a source operand. The use of a VEX prefix enables operands to perform nondestructive operations such as A = B + C.

[0061] **Figure 6A** illustrates an exemplary AVX instruction format including a VEX prefix 602, real opcode field 630, Mod R/M byte 640, SIB byte 650, displacement field 662, and IMM8 672. **Figure 6B** illustrates which fields from Figure 6A make up a full opcode field 674 and a base operation field 642. **Figure 6C** illustrates which fields from Figure 6A make up a register index field 644.

[0062] VEX Prefix (Bytes 0-2) 602 is encoded in a three-byte form. The first byte is the Format Field 640 (VEX Byte 0, bits [7:0]), which contains an explicit C4 byte value (the unique value used for distinguishing the C4 instruction format). The second-third bytes (VEX Bytes 1-2) include a number of bit fields providing specific capability. Specifically, REX field 605 (VEX Byte 1, bits [7-5]) consists of a VEX.R bit field (VEX Byte 1, bit [7] - R), VEX.X bit field (VEX byte 1, bit [6] - X), and VEX.B bit field (VEX byte 1, bit[5] - B). Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding VEX.R, VEX.X, and VEX.B. Opcode map field 615 (VEX byte 1, bits [4:0] - mmmmm) includes content to encode an implied leading opcode byte. W Field 664 (VEX byte 2, bit [7] - W) - is represented by the notation VEX.W, and provides different functions depending on the instruction. The role of VEX.vvvv 620 (VEX Byte 2, bits [6:3]-vvvv) may include the following: 1) VEX.vvvv encodes the first source register operand, specified in inverted (Is complement) form and is valid for instructions with 2 or more source operands; 2) VEX.vvvv encodes the destination register operand, specified in I s complement form for certain vector shifts; or 3) VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. If VEX.L 668 Size field (VEX byte 2, bit [2]-L) = 0, it indicates 128 bit vector; if VEX.L = 1, it indicates 256 bit vector. Prefix encoding field 625 (VEX byte 2, bits [1:0]-pp) provides additional bits for the base operation field.

[0063] Real Opcode Field 630 (Byte 3) is also known as the opcode byte. Part of the opcode is specified in this field.

[0064] MOD R/M Field 640 (Byte 4) includes MOD field 642 (bits [7-6]), Reg field 644 (bits [5-3]), and R/M field 646 (bits [2-0]). The role of Reg field 644 may include the following: encoding either the destination register operand or a source register operand (the rrr of Rrrr), or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 646 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.

[0065] Scale, Index, Base (SIB) - The content of Scale field 650 (Byte 5) includes SS652 (bits [7-6]), which is used for memory address generation. The contents of SIB.xxx 654 (bits [5-3]) and SIB.bbb 656 (bits [2-0]) have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[0066] The Displacement Field 662 and the immediate field (IMM8) 672 contain address data.

[0067] Generic Vector Friendly Instruction Format

[0068] 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.

[0069] **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.

[0070] 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).

[0071] 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.

[0072] The generic vector friendly instruction format 700 includes the following fields listed below in the order illustrated in Figures 7A-7B.

[0073] 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.

[0074] Base operation field 742 - its content distinguishes different base operations.

[0075] 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).

[0076] 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.

[0077] 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.

[0078] 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).

[0079] 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).

[0080] 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.

[0081] 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.

[0082] 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.

[0083] 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.

[0084] 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).

[0085] Instruction Templates of Class A

[0086] 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.

[0087] No-Memory Access Instruction Templates - Full Round Control Type Operation

[0088] 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).

[0089] 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.

[0090] 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.

[0091] No Memory Access Instruction Templates - Data Transform Type Operation

[0092] 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).

[0093] 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.

[0094] 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.

[0095] Memory Access Instruction Templates -Temporal

[0096] 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.

[0097] Memory Access Instruction Templates - Non-Temporal

[0098] Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the lst-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.

[0099] Instruction Templates of Class B

[00100] 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.

[00101] 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.

[00102] 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).

[00103] 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.

[00104] 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).

[00105] 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.

[00106] 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).

[00107] 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.

[00108] 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.

[00109] 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.

[00110] Exemplary Specific Vector Friendly Instruction Format

[00111] **Figure 8** is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. Figure 8 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 8 map are illustrated.

[00112] 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).

[00113] The generic vector friendly instruction format 700 includes the following fields listed below in the order illustrated in Figure 8A.

[00114] EVEX Prefix (Bytes 0-3) 802 - is encoded in a four-byte form.

[00115] 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).

[00116] The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

[00117] 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 I s 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.

[00118] 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.

[00119] Opcode map field 815 (EVEX byte 1, bits [3:0] - mmmm) - its content encodes an implied leading opcode byte (OF, OF 38, or OF 3).

[00120] 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).

[00121] 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.

[00122] 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.

[00123] 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.

[00124] 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.

[00125] Beta field 754 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s_{2-0}, EVEX.r_{2-0}, EVEX.rrl, EVEX.LL0, EVEX.LLB; also illustrated with βββ) - as previously described, this field is context specific.

[00126] 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.

[00127] 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).

[00128] Real Opcode Field 830 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field .

[00129] 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.

[00130] 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.

[00131] 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.

[00132] 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 Mod RM/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).

[00133] Immediate field 772 operates as previously described.

[00134] Full Opcode Field

[00135] 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.

[00136] Register Index Field

[00137] 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 WW field 820, xxx field 854, and the bbb field 856.

[00138] Augmentation Operation Field

[00139] 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.

[00140] When U=l, the alpha field 752 (EVEX byte 3, bit [7] - EH) is interpreted as the write mask control (Z) field 752C. When U=l 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-i)** 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-}i)** is interpreted as the vector length field 759B (EVEX byte 3, bit [6-5]- **L_{1-0}).** When U=l 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_{i_o}) and the broadcast field 757B (EVEX byte 3, bit [4]- B).

[00141] ***Exemplary Register Architecture***

[00142] **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 zmmO through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymmO-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

xmmO-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

[00143] 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.

[00144] Write mask registers 915 - in the embodiment illustrated, there are 8 write mask registers (kO 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.

[00145] 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.

[00146] 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.

[00147] 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.

[00148] Exemplary Core Architectures, Processors, and Computer Architectures

[00149] 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.

[00150] Exemplary Core Architectures

[00151] In-order and out-of-order core block diagram

[00152] **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.

[00153] 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.

[00154] 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.

[00155] 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.

[00156] 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.

[00157] 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.

[00158] 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.

[00159] 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, and/or some form of the generic vector friendly instruction format (U=0 and/or U=l) previously described), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[00160] 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).

[00161] 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 (LI) 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.

[00162] Specific Exemplary In-Order Core Architecture

[00163] **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.

[00164] 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 LI 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 (LI) 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).

[00165] 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.

[00166] 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 LI data cache 1106A part of the LI 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.

[00167] Processor with integrated memory controller and graphics

[00168] **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.

[00169] 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.

[00170] 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, 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.

[00171] 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.

[00172] 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.

[00173] Exemplary Computer Architectures

[00174] **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.

[00175] 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 is 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.

[00176] 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.

[00177] 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.

[00178] 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.

[00179] 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.

[00180] 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.

[00181] 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.

[00182] 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.

[00183] 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 1439. 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.

[00184] 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.

[00185] 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.

[00186] 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.

[00187] 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.

[00188] 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.

[00189] 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 202A-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.

[00190] 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.

[00191] 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.

[00192] 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.

[00193] 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.

[00194] 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 rewritables (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.

[00195] 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.

[00196] Emulation (including binary translation, code morphing, etc.)

[00197] 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.

[00198] **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.

Claims

What is claimed is:

1. A method of performing in a computer processor vector packed compression and repeat in response to a single vector packed compression and repeat instruction that includes a first and second source vector register operand, a destination vector register operand, and an opcode, the method comprising steps of:

executing the vector packed compression and repeat instruction to determine, for each packed data element position of the first source vector register, how many times that packed data element position's packed data element is to be stored in the destination vector register based on the value of a corresponding packed data element position of the second source vector register;

storing each packed data element of a packed data element position of the first source vector register into the destination vector register value number of times based on the determination of the corresponding data element of the second source vector register's value.

2. The method of claim 1, wherein the storing begins at the least significant packed data element position of the destination vector register and the packed data elements are stored in consecutive packed data element positions of the destination vector register.

3. The method of claim 1, wherein the executing and storing steps further comprise:

determining a value of a least significant packed data element position of the second source vector register;

determining if that value is greater than 0;

if the value is greater than 0, storing a corresponding packed data element position of the first source vector register's packed data element value number of times, wherein these packed data elements are stored in consecutively beginning at a least significant packed data element position of the destination vector register; and

if the value is 0, determining a value of a next least significant packed data element position of the second source vector register;

if the value of the next least significant data element position is greater than 0, storing a corresponding packed data element position of the first source vector register's packed data element value number of times, wherein these packed data elements are stored in consecutively beginning at a least significant packed data element position of the destination vector register that has not been previously written to.

4. The method of claim 3, further comprising:

repeating the determining and storing steps until all of the packed data element position of the second source vector register's values have been determined and appropriate .

5. The method of claim 4, further comprising:

writing a preset value into all unused packed data element positions of the destination register after all of the packed data element positions of the first source vector register have been written into the destination vector register.

6. The method of claim 5, wherein the preset value is a value of all Is.

7. The method of claim 1, providing a programmer visible exception when all of packed data element positions of the destination vector register have been written to, but there are still packed data elements from the first source vector register that are to be written to the destination vector register.

8. The method of claim 1, wherein the vector registers are all a same size of 128-bit, 256-bit, or 512-bit.

9. An article of manufacture comprising:

a tangible machine-readable storage medium having stored thereon an occurrence of an instruction, wherein the instruction's format specifies as its source operands a first and second vector register and specifies as its destination a single vector register, and wherein the instruction format includes an opcode which instructs a machine, responsive to the single occurrence of the single instruction, to cause a determination, for each packed data element position of the first

source vector register, how many times that packed data element position's packed data element is to be stored in the destination vector register based on the value of a corresponding packed data element position of the second source vector register, storage of each packed data element of a packed data element position of the first source vector register into the destination vector register value number of times based on the determination of the corresponding data element of the second source vector register's value.

10. The article of manufacture of claim 9, wherein the storing begins at the least significant packed data element position of the destination vector register and the packed data elements are stored in consecutive packed data element positions of the destination vector register.

11. The article of manufacture of claim 9, further to cause:

a determination of a value of a least significant packed data element position of the second source vector register;

a determination of if that value is greater than 0;

if the value is greater than 0, storage of a corresponding packed data element position of the first source vector register's packed data element value number of times, wherein these packed data elements are stored in consecutively beginning at a least significant packed data element position of the destination vector register; and

if the value is 0, a determination of a value of a next least significant packed data element position of the second source vector register;

if the value of the next least significant data element position is greater than 0, storage of a corresponding packed data element position of the first source vector register's packed data element value number of times, wherein these packed data elements are stored in consecutively beginning at a least significant packed data element position of the destination vector register that has not been previously written to.

12. The article of manufacture of claim 9, further to:

repeat until all of the packed data element position of the second source vector register's values have been determined and appropriate .

13. The article of manufacture of claim 9, further to:

write a preset value into all unused packed data element positions of the destination register after all of the packed data element positions of the first source vector register have been written into the destination vector register.

14. The article of manufacture of claim 9, wherein the preset value is a value of all Is.

15. The article of manufacture of claim 9, wherein the vector registers are all a same size of 128-bit, 256-bit, or 512-bit.

16. An apparatus comprising;

a hardware decoder to decode a single vector packed compression and repeat instruction that includes a first and second source vector register operand, a destination vector register operand, and an opcode;

execution logic to determine, for each packed data element position of the first source vector register, how many times that packed data element position's packed data element is to be stored in the destination vector register based on the value of a corresponding packed data element position of the second source vector register and store each packed data element of a packed data element position of the first source vector register into the destination vector register value number of times based on the determination of the corresponding data element of the second source vector register's value.

17. The apparatus of claim 16, wherein the storgae begins at the least significant packed data element position of the destination vector register and the packed data elements are stored in consecutive packed data element positions of the destination vector register.

18. The apparatus of claim 17, wherein the execution logic further to:

repeat the determination and storage until all of the packed data element position of the second source vector register's values have been determined and appropriate .

19. The apparatus of claim 18, wherein the execution logic further to:

writie a preset value into all unused packed data element positions of the destination register after all of the packed data element positions of the first source vector register have been written into the destination vector register.

8. The apparatus of claim 16, wherein the vector registers are all a same size of 128-bit, 256-bit, or 512-bit.

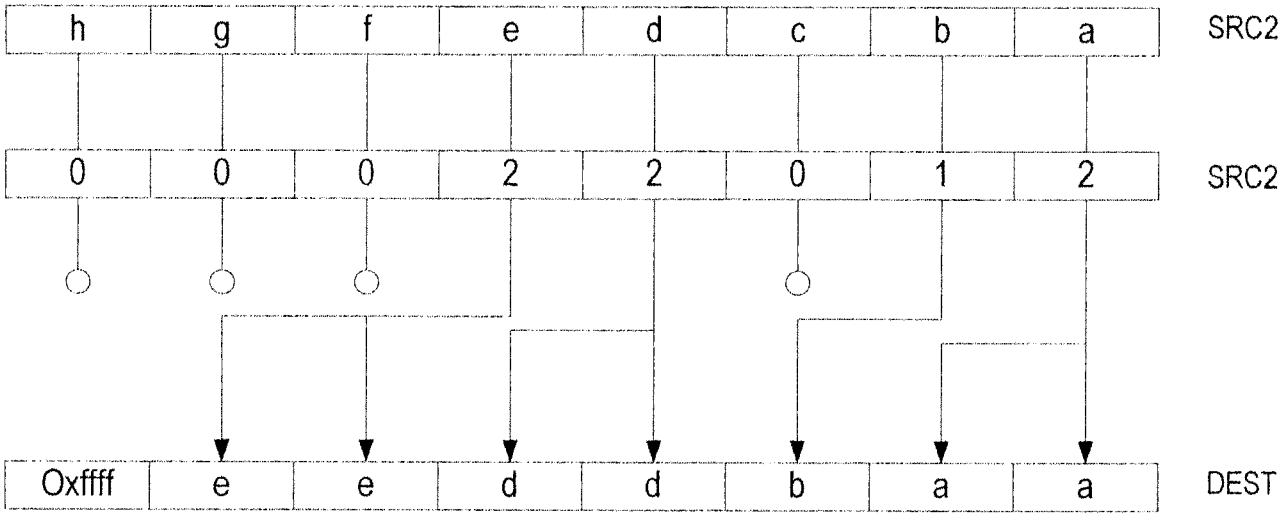


FIGURE 1

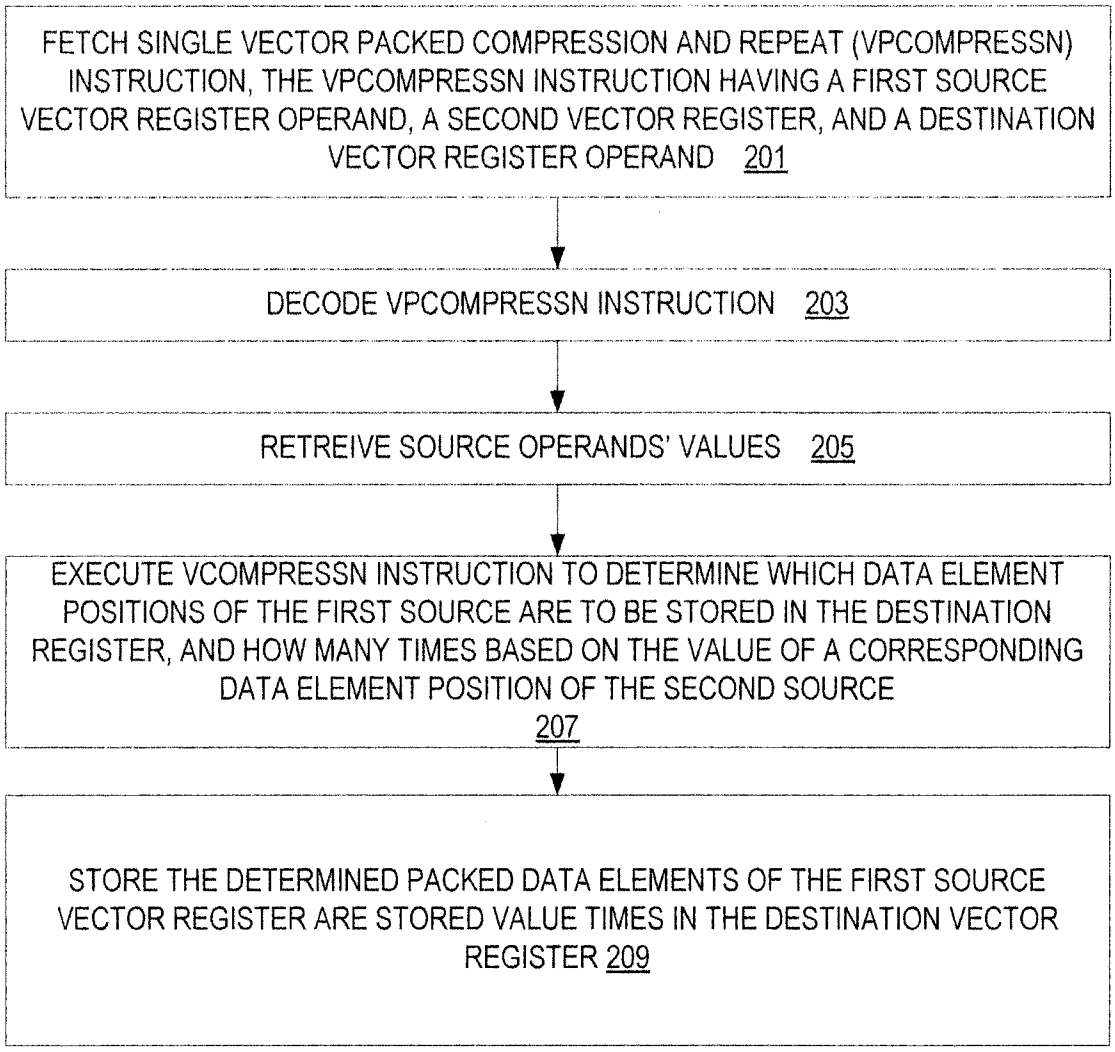

FETCH SINGLE VECTOR PACKED COMPRESSION AND REPEAT (VPCOMPRESSN) INSTRUCTION, THE VPCOMPRESSN INSTRUCTION HAVING A FIRST SOURCE VECTOR REGISTER OPERAND, A SECOND VECTOR REGISTER, AND A DESTINATION VECTOR REGISTER OPERAND 201
DECODE VPCOMPRESSN INSTRUCTION 203
RETREIVE SOURCE OPERANDS' VALUES 205
EXECUTE VCOMPRESSN INSTRUCTION TO DETERMINE WHICH DATA ELEMENT POSITIONS OF THE FIRST SOURCE ARE TO BE STORED IN THE DESTINATION REGISTER, AND HOW MANY TIMES BASED ON THE VALUE OF A CORRESPONDING DATA ELEMENT POSITION OF THE SECOND SOURCE 207
STORE THE DETERMINED PACKED DATA ELEMENTS OF THE FIRST SOURCE VECTOR REGISTER ARE STORED VALUE TIMES IN THE DESTINATION VECTOR REGISTER 209

FIGURE 2

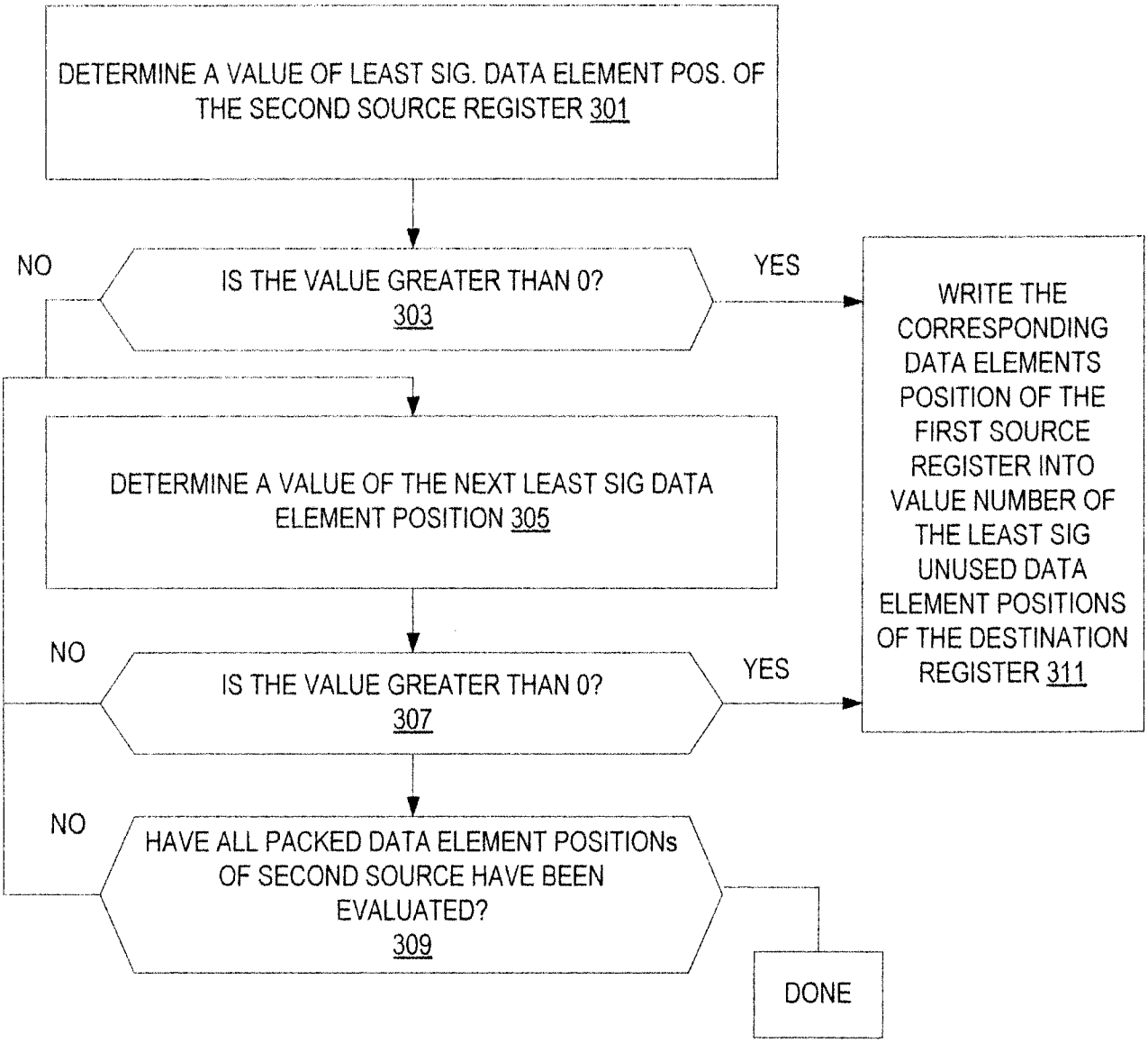

DETERMINE A VALUE OF LEAST SIG. DATA ELEMENT POS. OF THE SECOND SOURCE REGISTER 301
IS THE VALUE GREATER THAN 0? 303
NO
YES
WRITE THE CORRESPONDING DATA ELEMENTS POSITION OF THE FIRST SOURCE REGISTER INTO VALUE NUMBER OF THE LEAST SIG UNUSED DATA ELEMENT POSITIONS OF THE DESTINATION REGISTER 311
DETERMINE A VALUE OF THE NEXT LEAST SIG DATA ELEMENT POSITION 305
IS THE VALUE GREATER THAN 0? 307
NO
YES
HAVE ALL PACKED DATA ELEMENT POSITIONs OF SECOND SOURCE HAVE BEEN EVALUATED? 309
NO
DONE

FIGURE 3

```
j ← 0
for i=0 to n-1 {
        for j=0 to SRC1[1] {
                if (j<n)
                        DEST ← SRC2[i]
                        J ← j+1

end if
        }
}
For i=j_1 to n-1 {

        DEST[i] ←-1
```

FIGURE 4

NUMBER OF ONE BIT VECTOR WRITE MASK ELEMENTS

DATA ELEMENT SIZE FOR VECTOR	VECTOR SIZE		
	128 BITS	256 BITS	512 BITS
8-BIT BYTES	16	32	64
16-BIT WORDS	8	16	32
32-BIT DWORDS / SP	4	8	16
64-BIT QWORDS / DP	2	4	8

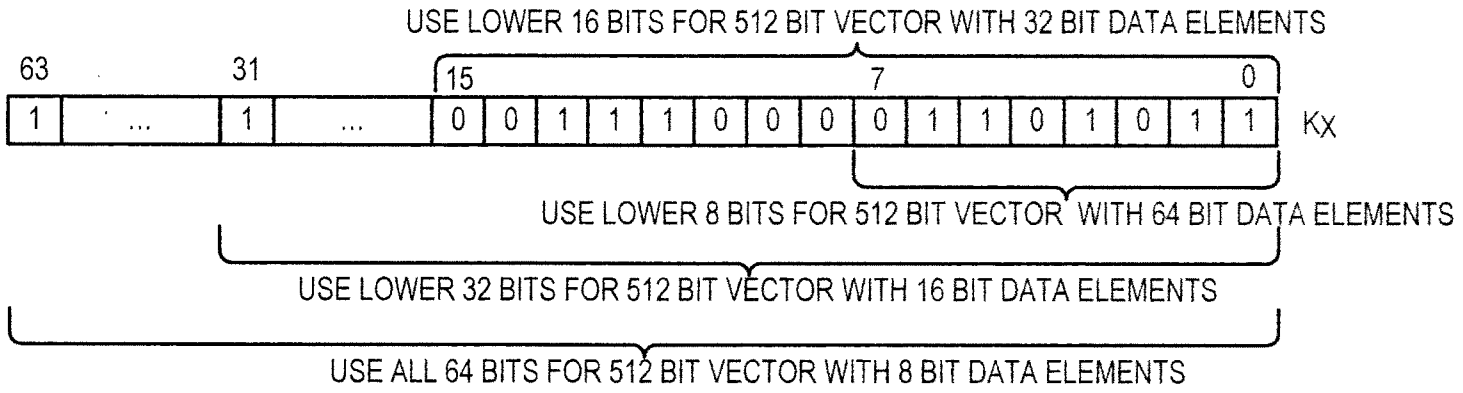
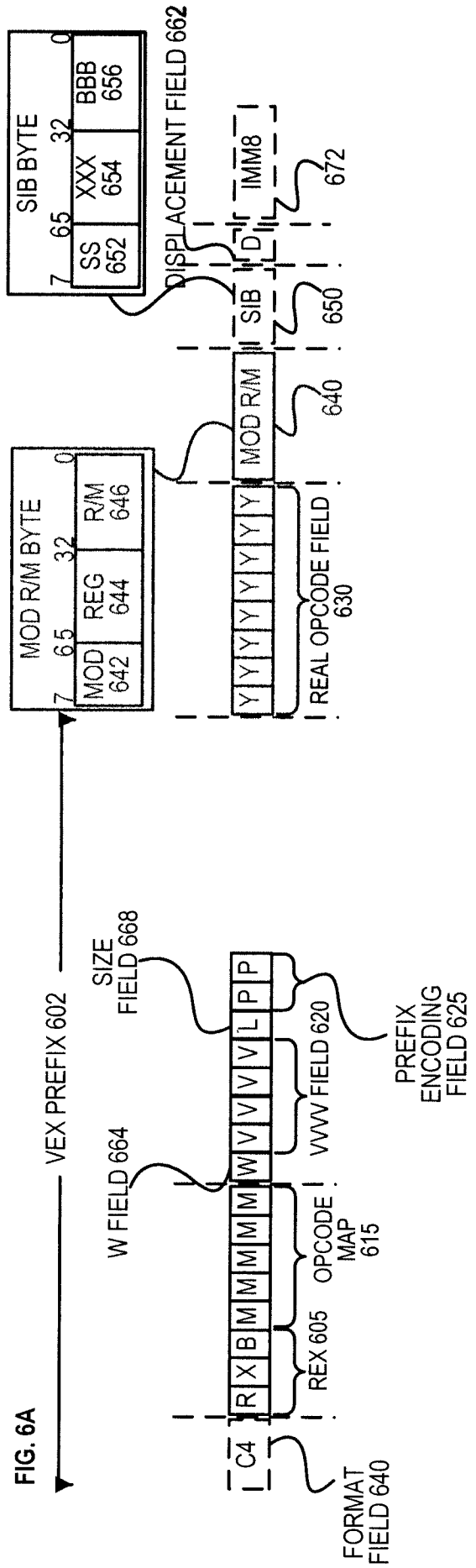
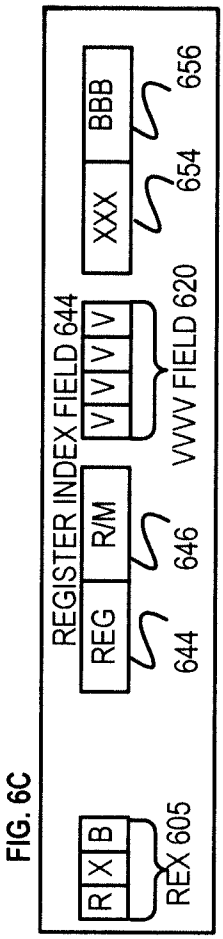
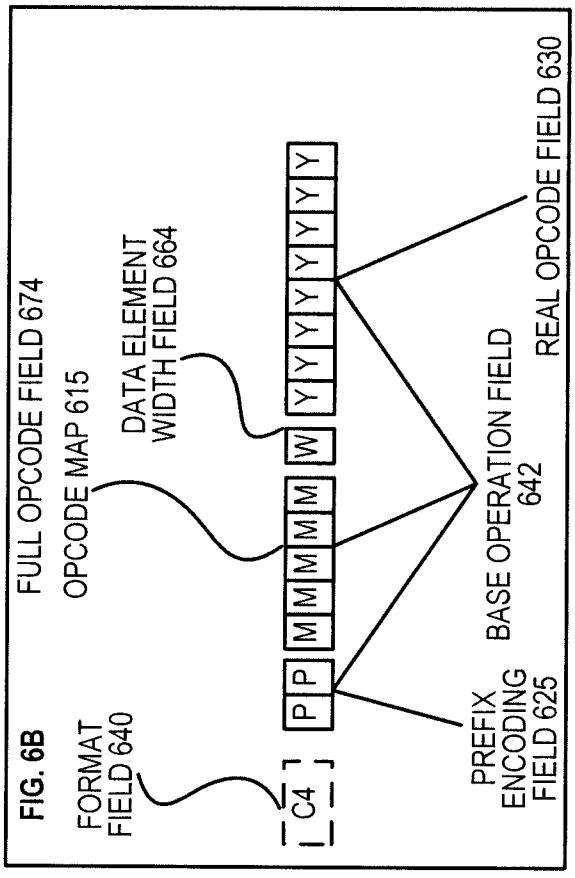


Figure 5


FIG. 6A
VEX PREFIX 602
C4
R X B M M M M M
W V V V V L P P
REX 605
OPCODE MAP 615
W FIELD 664
SIZE FIELD 668
VVVV FIELD 620
PREFIX ENCODING FIELD 625
FORMAT FIELD 640
MOD R/M BYTE
MOD 642
REG 644
R/M 646
SIB BYTE
SS 652
XXX 654
BBB 656
DISPLACEMENT FIELD 662
SIB
D
IMM8
672
MOD R/M
650
640
Y Y Y Y Y Y Y Y
REAL OPCODE FIELD 630


FIG. 6C
REGISTER INDEX FIELD 644
REG
R/M
644
646
V V V V
VVVV FIELD 620
XXX
BBB
654
656
R X B
REX 605


FIG. 6B
FULL OPCODE FIELD 674
OPCODE MAP 615
DATA ELEMENT WIDTH FIELD 664
FORMAT FIELD 640
C4
P P M M M M W Y Y Y Y Y Y Y Y
PREFIX ENCODING FIELD 625
BASE OPERATION FIELD 642
REAL OPCODE FIELD 630

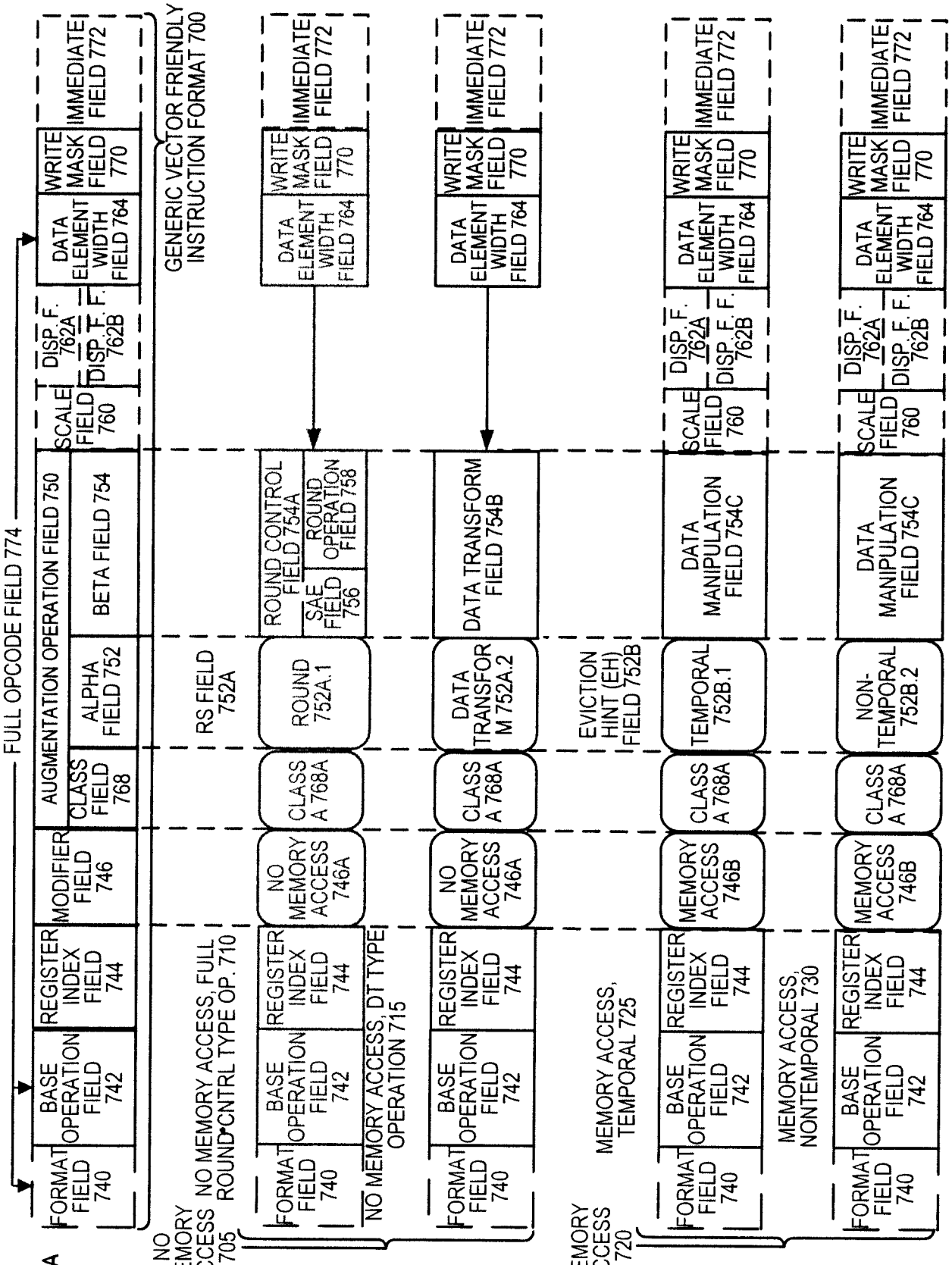

FULL OPCODE FIELD 774
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
SCALE FIELD 760
DISP. F. 762A
DISP. F. F. 762B
DATA ELEMENT WIDTH FIELD 764
WRITE MASK FIELD 770
IMMEDIATE FIELD 772
NO MEMORY ACCESS 705
NO MEMORY ACCESS, FULL ROUND CNTRL TYPE OP. 710
NO MEMORY ACCESS
NO MEMORY ACCESS, DT TYPE OPERATION 715
RS FIELD 752A
ROUND 752A.1
ROUND CONTROL FIELD 754A
SAE FIELD 756
ROUND OPERATION FIELD 758
CLASS A 768A
NO MEMORY ACCESS 746A
DATA TRANSFORM FIELD 754B
DATA TRANSFORM 752A.2
MEMORY ACCESS 720
MEMORY ACCESS, TEMPORAL 725
EVICTION HINT (EH) FIELD 752B
TEMPORAL 752B.1
DATA MANIPULATION FIELD 754C
MEMORY ACCESS 746B
MEMORY ACCESS, NONTEMPORAL 730
NON-TEMPORAL 752B.2

FIG.7A

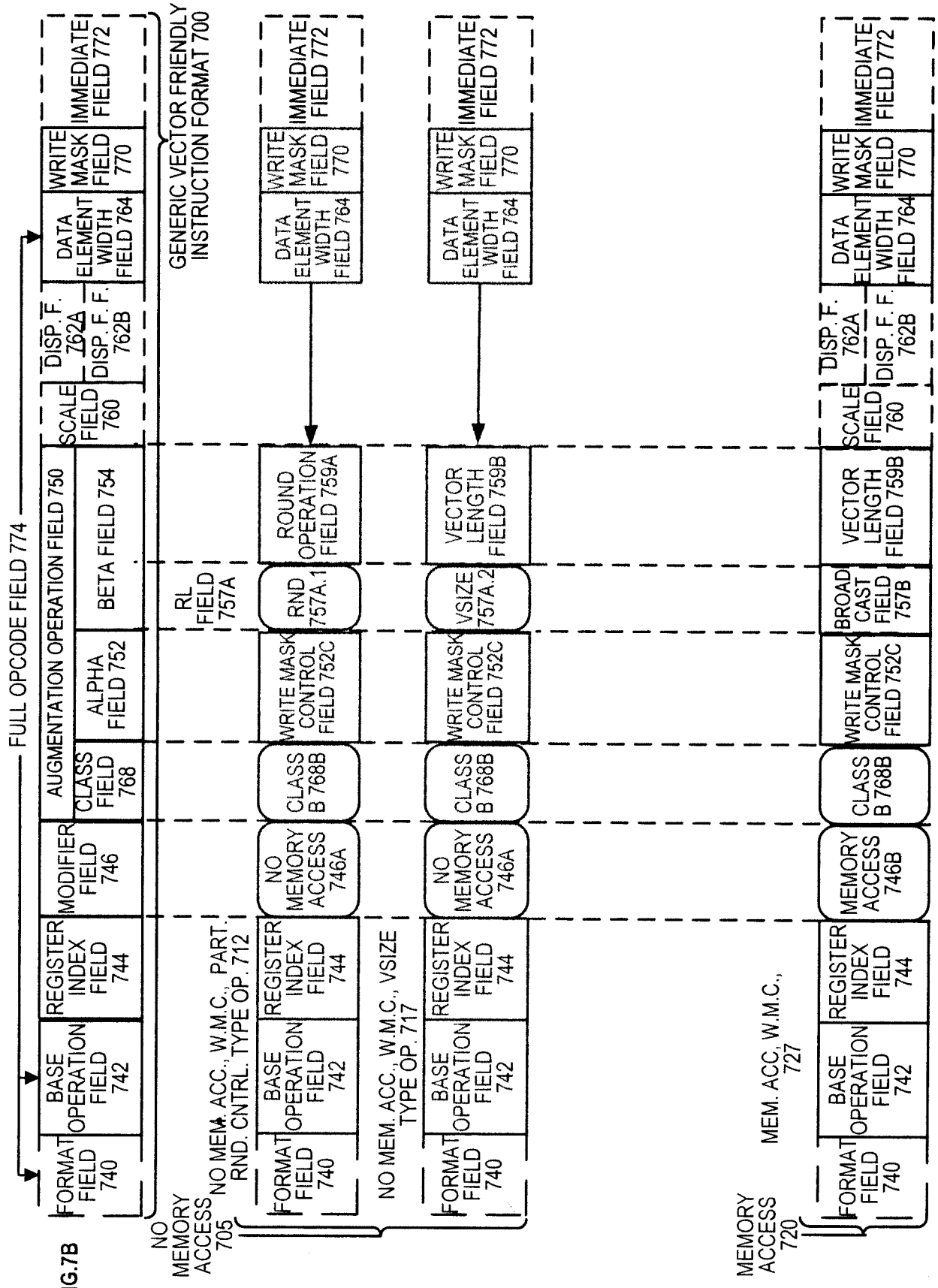

FULL OPCODE FIELD 774
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
ALPHA FIELD 752
BETA FIELD 754
SCALE FIELD 760
DISP. F. 762A
DISP. F. F. 762B
DATA ELEMENT WIDTH FIELD 764
WRITE MASK FIELD 770
IMMEDIATE FIELD 772
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
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
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

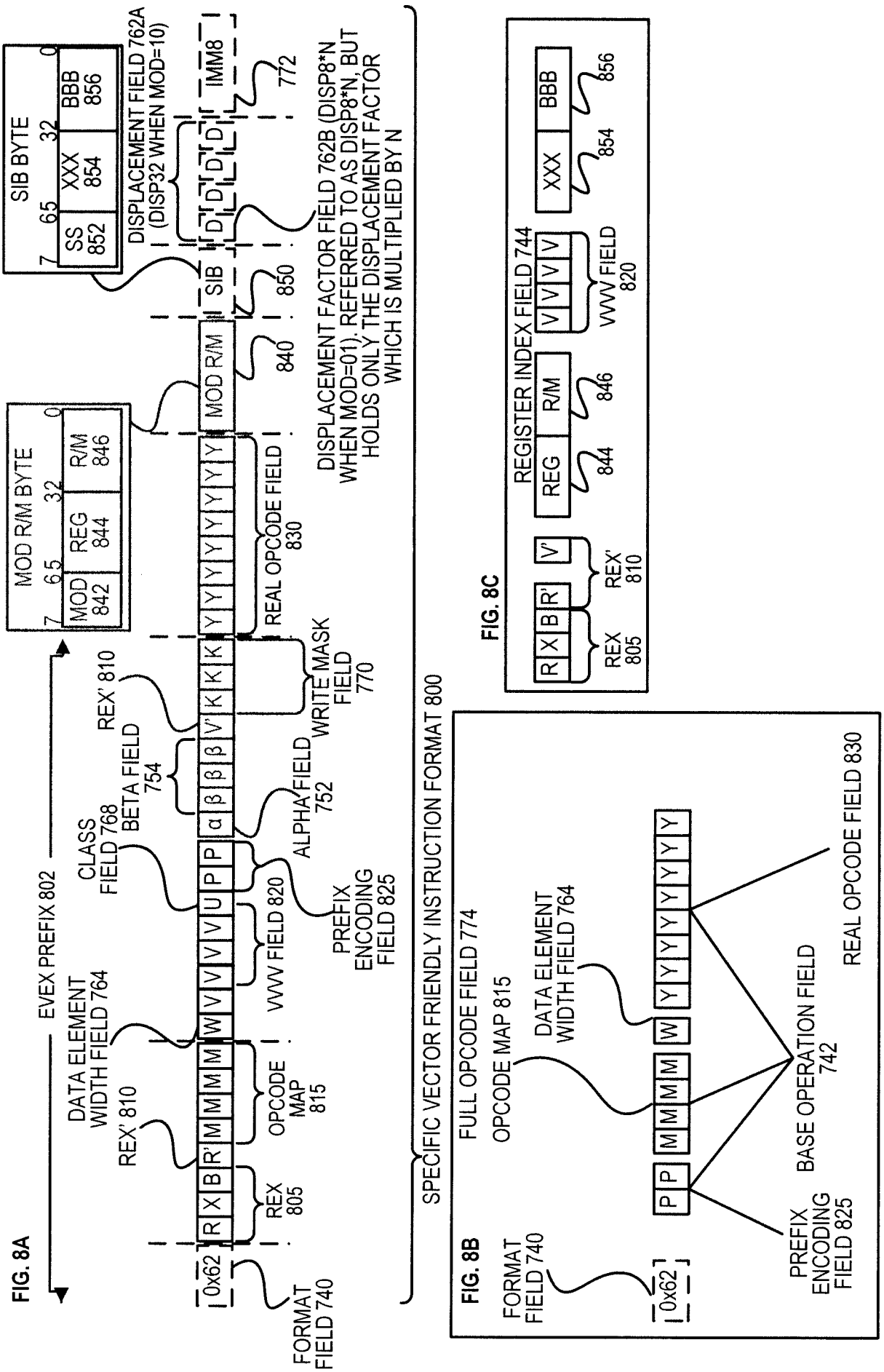
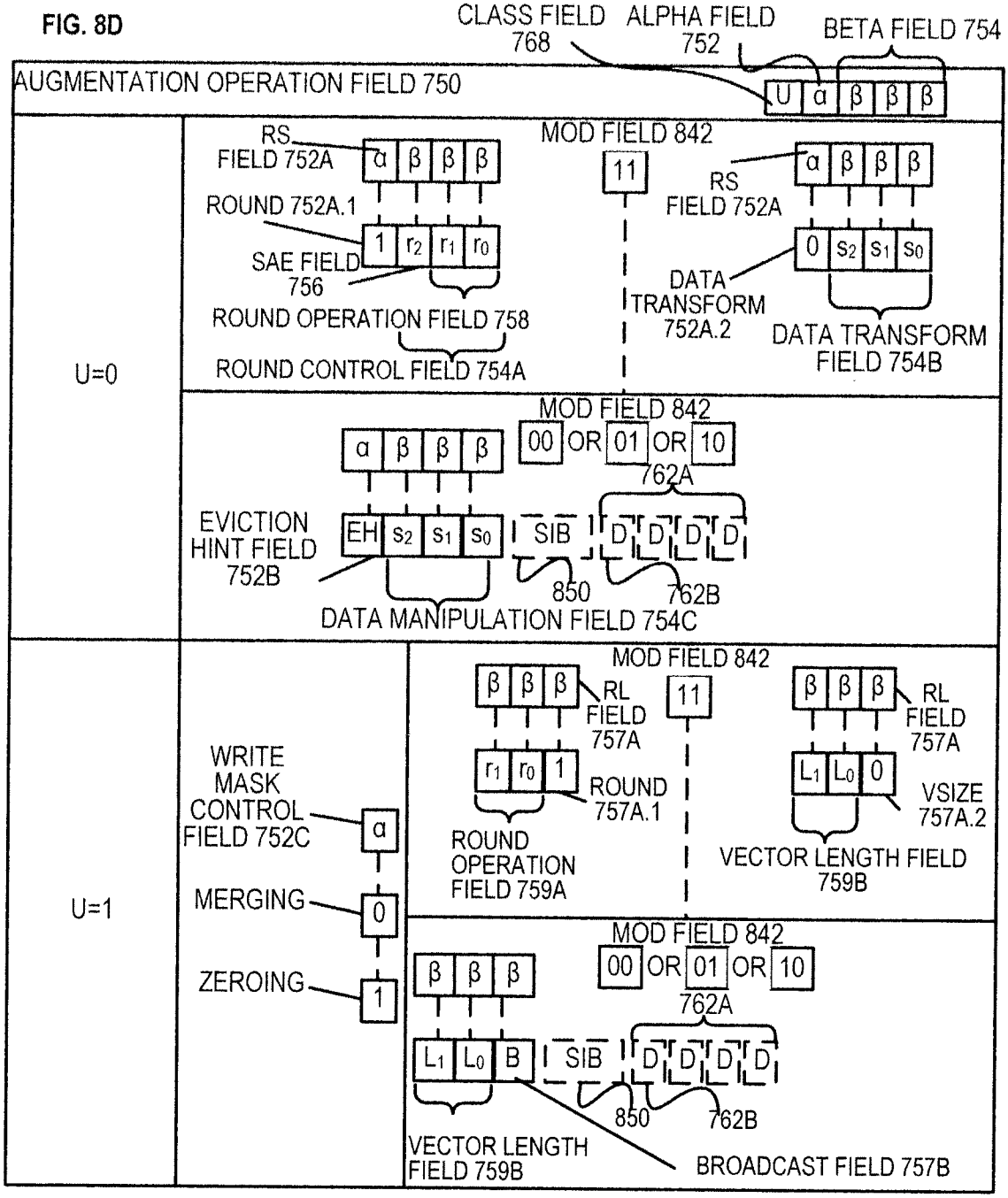

FIG. 8A
EVEX PREFIX 802
FORMAT FIELD 740
0x62
REX 805
REX' 810
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
VVV FIELD 820
CLASS FIELD 768
BETA FIELD 754
ALPHA FIELD 752
REX' 810
PREFIX ENCODING FIELD 825
WRITE MASK FIELD 770
REAL OPCODE FIELD 830
MOD R/M BYTE
MOD 842
REG 844
R/M 846
SIB BYTE
SS 852
XXX 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
FULL OPCODE FIELD 774
FORMAT FIELD 740
0x62
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
PREFIX ENCODING FIELD 825
BASE OPERATION FIELD 742
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
MOD FIELD 842
11
RS FIELD 752A
ROUND 752A.1
SAE FIELD 756
ROUND OPERATION FIELD 758
ROUND CONTROL FIELD 754A
DATA TRANSFORM 752A.2
DATA TRANSFORM FIELD 754B
MOD FIELD 842
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
MERGING
ZEROING
MOD FIELD 842
RL FIELD 757A
ROUND 757A.1
ROUND OPERATION FIELD 759A
VSIZE 757A.2
VECTOR LENGTH FIELD 759B
MOD FIELD 842
00 OR 01 OR 10
762A
L1 L0 B
SIB
D D D D
850
762B
VECTOR LENGTH FIELD 759B
BROADCAST FIELD 757B

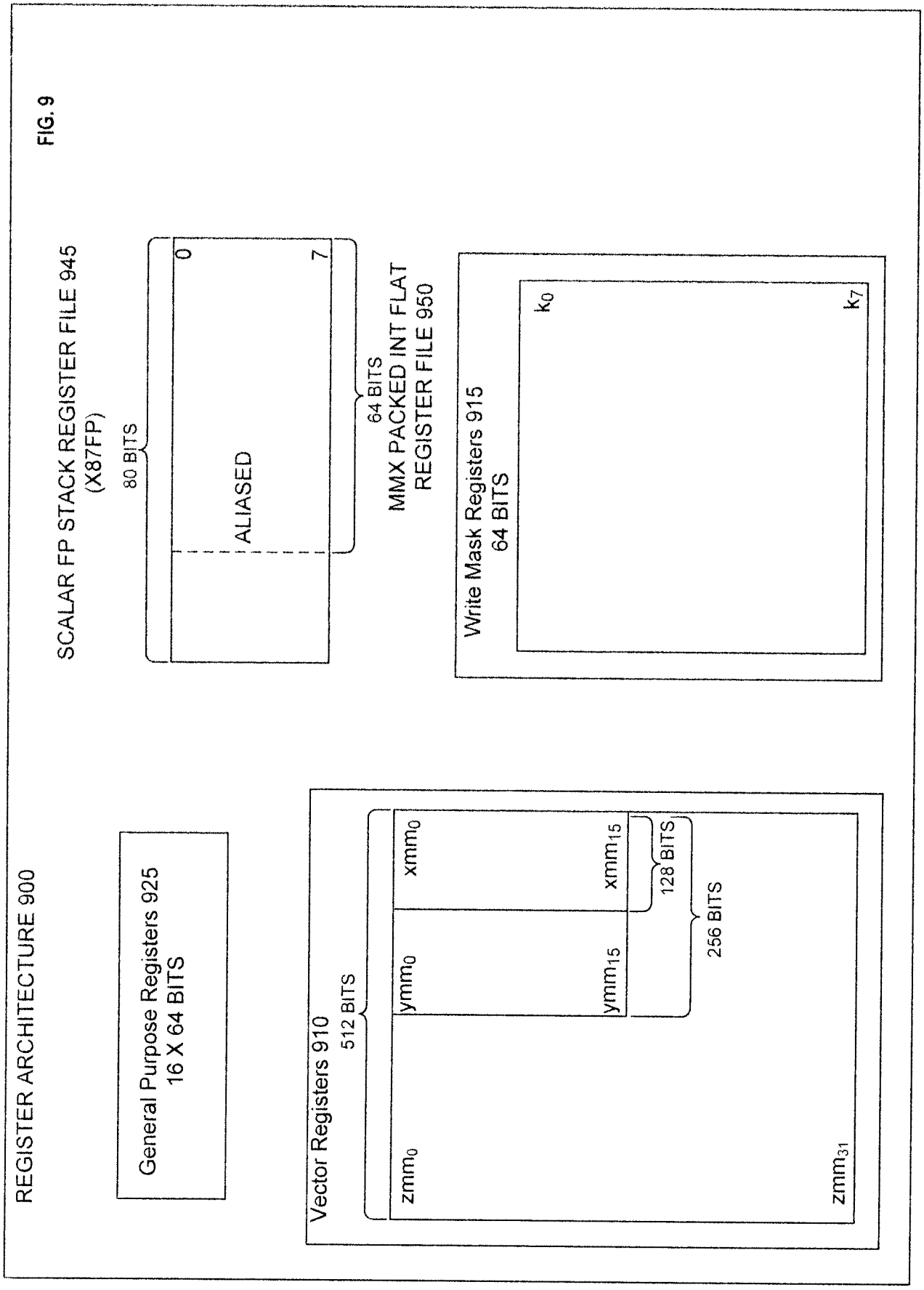

REGISTER ARCHITECTURE 900
General Purpose Registers 925
16 X 64 BITS
SCALAR FP STACK REGISTER FILE 945
(X87FP)
80 BITS
0
ALIASED
7
64 BITS
MMX PACKED INT FLAT
REGISTER FILE 950
Write Mask Registers 915
64 BITS
k_0
k_7
Vector Registers 910
512 BITS
zmm_0
ymm_0
xmm_0
ymm_{15}
xmm_{15}
128 BITS
256 BITS
zmm_{31}

FIG. 9

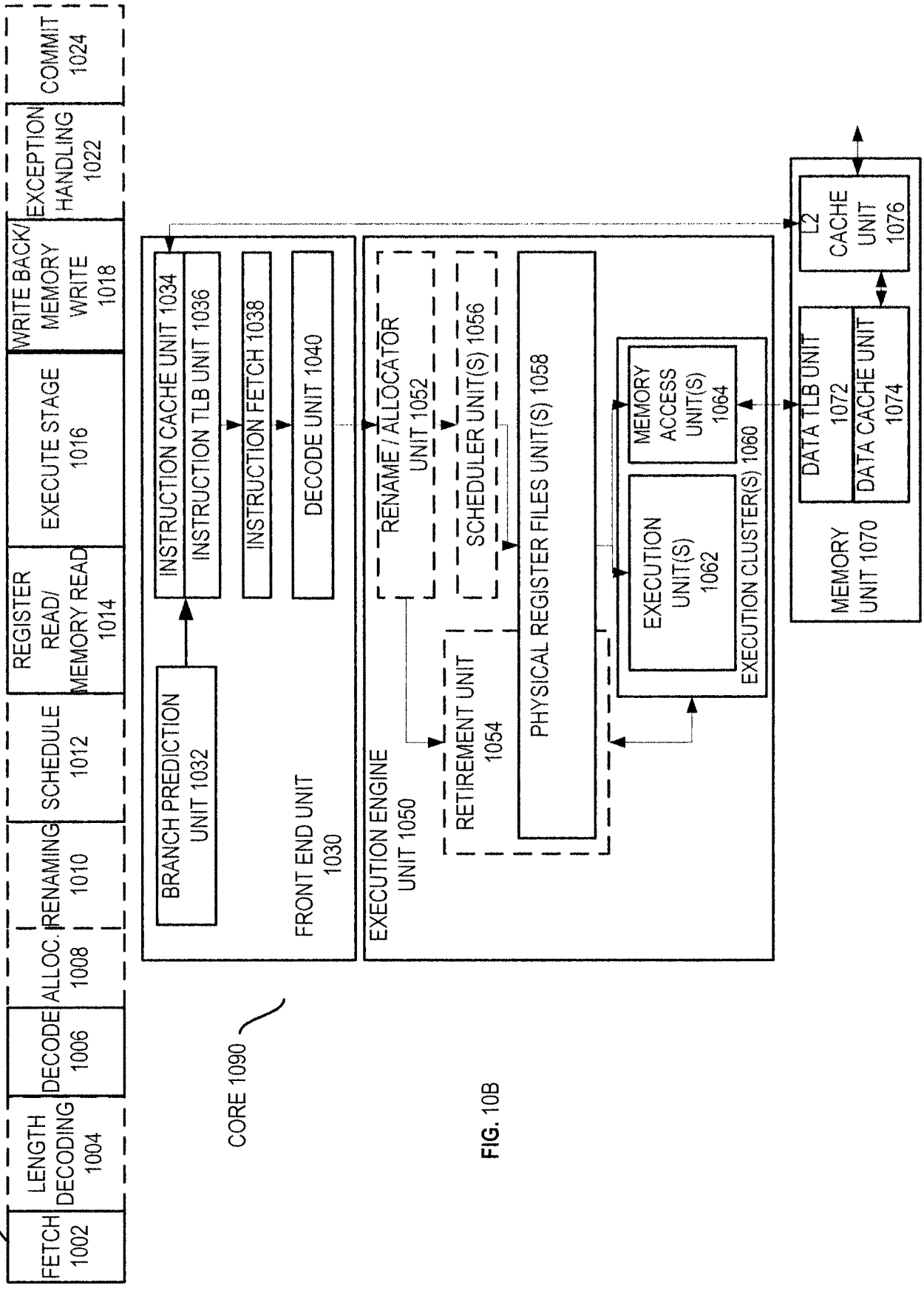


FIG. 10A

FIG. 10B

FIG. 11B

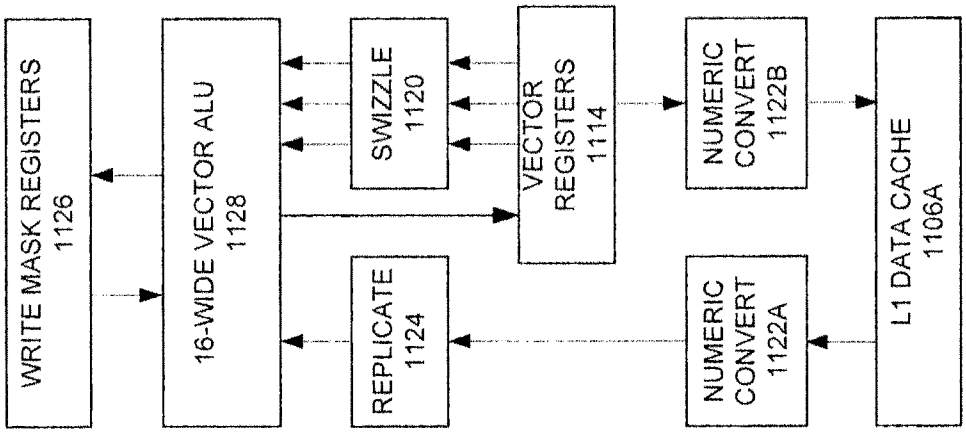
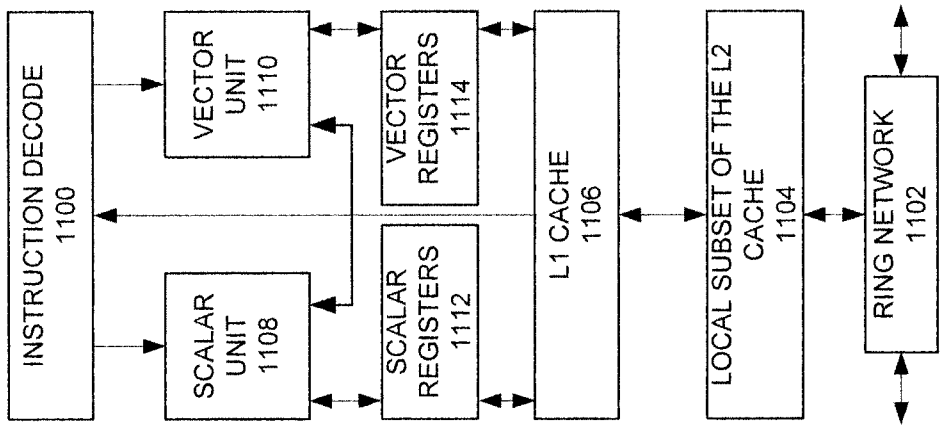


FIG. 11A



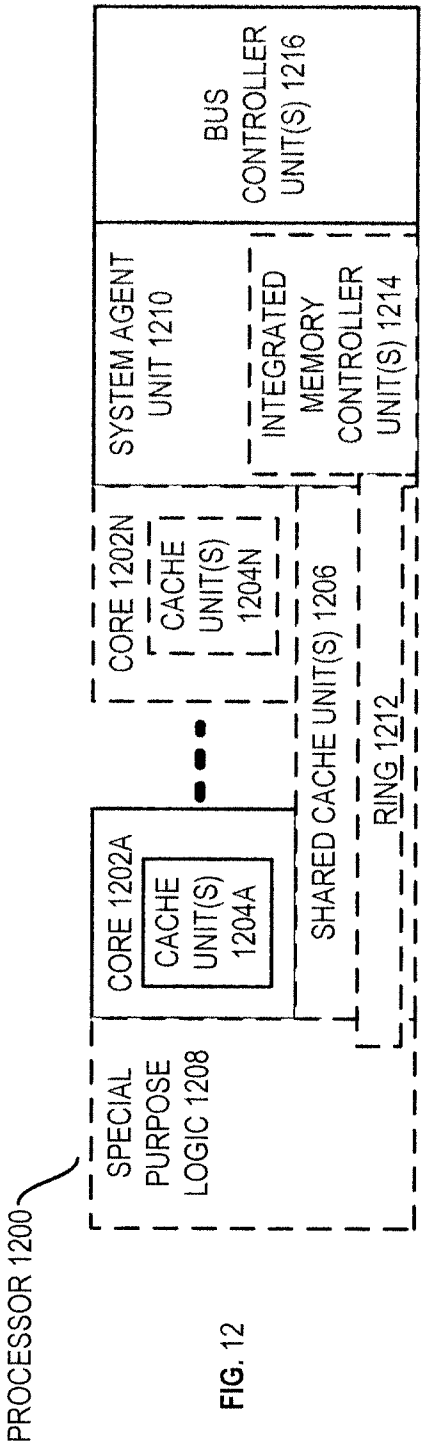

PROCESSOR 1200
SPECIAL PURPOSE LOGIC 1208
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
RING 1212
SYSTEM AGENT UNIT 1210
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214
BUS CONTROLLER UNIT(S) 1216

FIG. 12

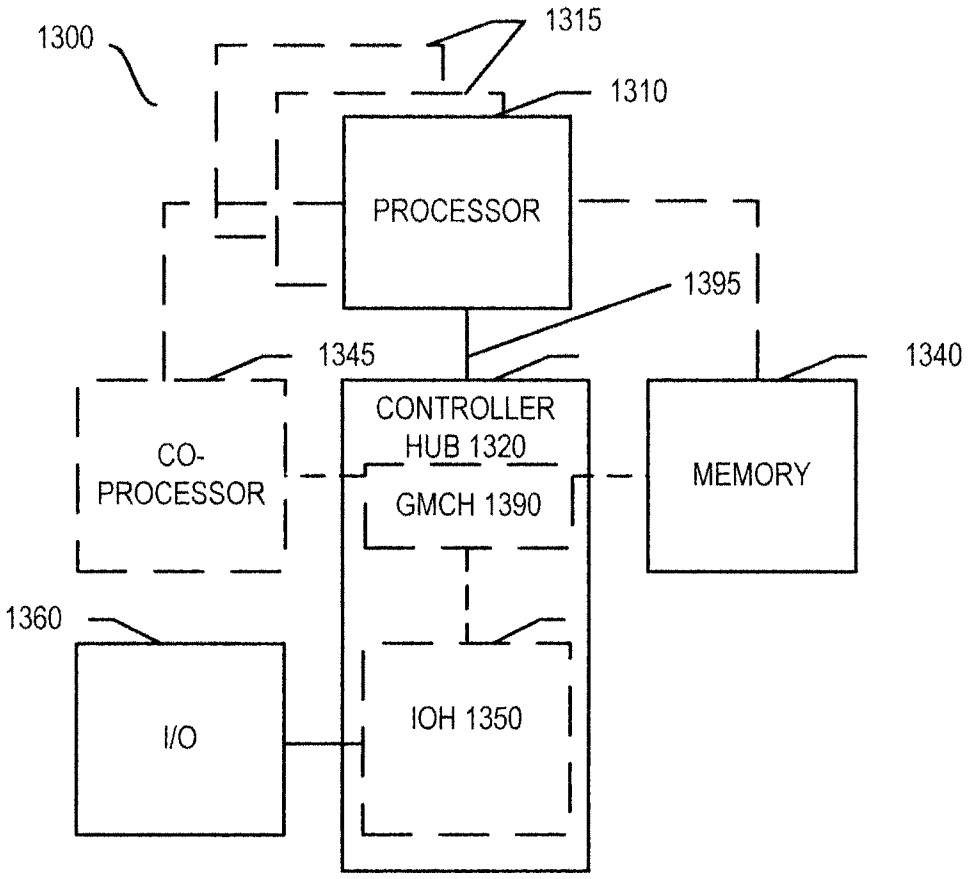

1300
1315
1310
PROCESSOR
1395
1345
1340
CONTROLLER
HUB 1320
CO-
PROCESSOR
GMCH 1390
MEMORY
1360
I/O
IOH 1350

FIG. 13

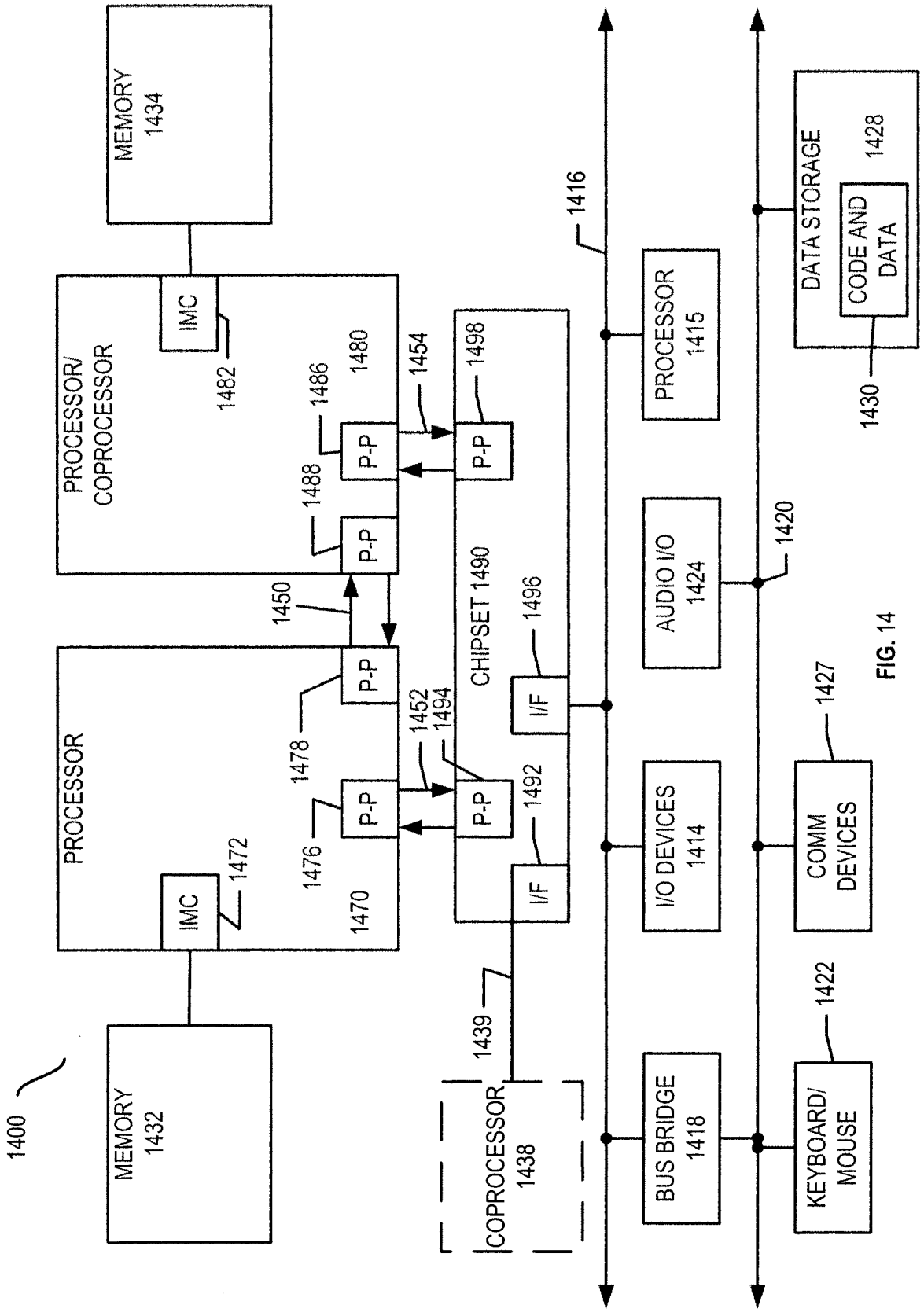

1400
MEMORY 1432
MEMORY 1434
PROCESSOR
IMC
1472
1470
PROCESSOR/ COPROCESSOR
IMC
1482
1480
P-P
1478
1476
1486
1488
1450
1452
1494
1454
1498
CHIPSET 1490
1496
I/F
1492
I/F
1439
COPROCESSOR 1438
1416
PROCESSOR 1415
AUDIO I/O 1424
1420
I/O DEVICES 1414
1427
COMM DEVICES
BUS BRIDGE 1418
1422
KEYBOARD/ MOUSE
DATA STORAGE
CODE AND DATA
1428
1430

FIG. 14

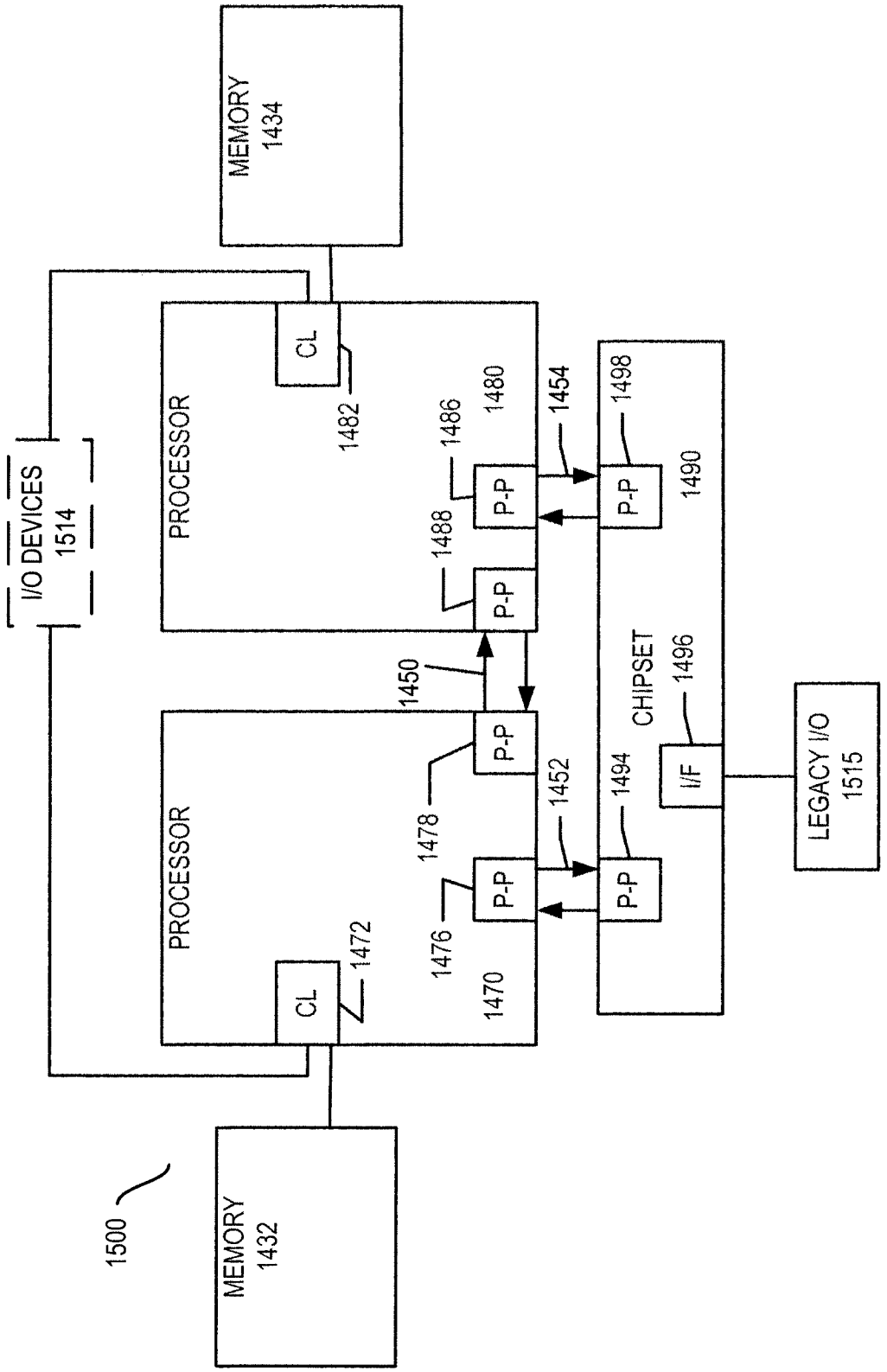

MEMORY
1434
CL
1482
PROCESSOR
1480
1486
P-P
1488
P-P
1454
1498
P-P
1490
P-P
I/O DEVICES
1514
1450
CHIPSET
1496
I/F
LEGACY I/O
1515
1494
1452
1478
P-P
1476
P-P
1470
PROCESSOR
CL
1472
1500
MEMORY
1432

FIG. 15

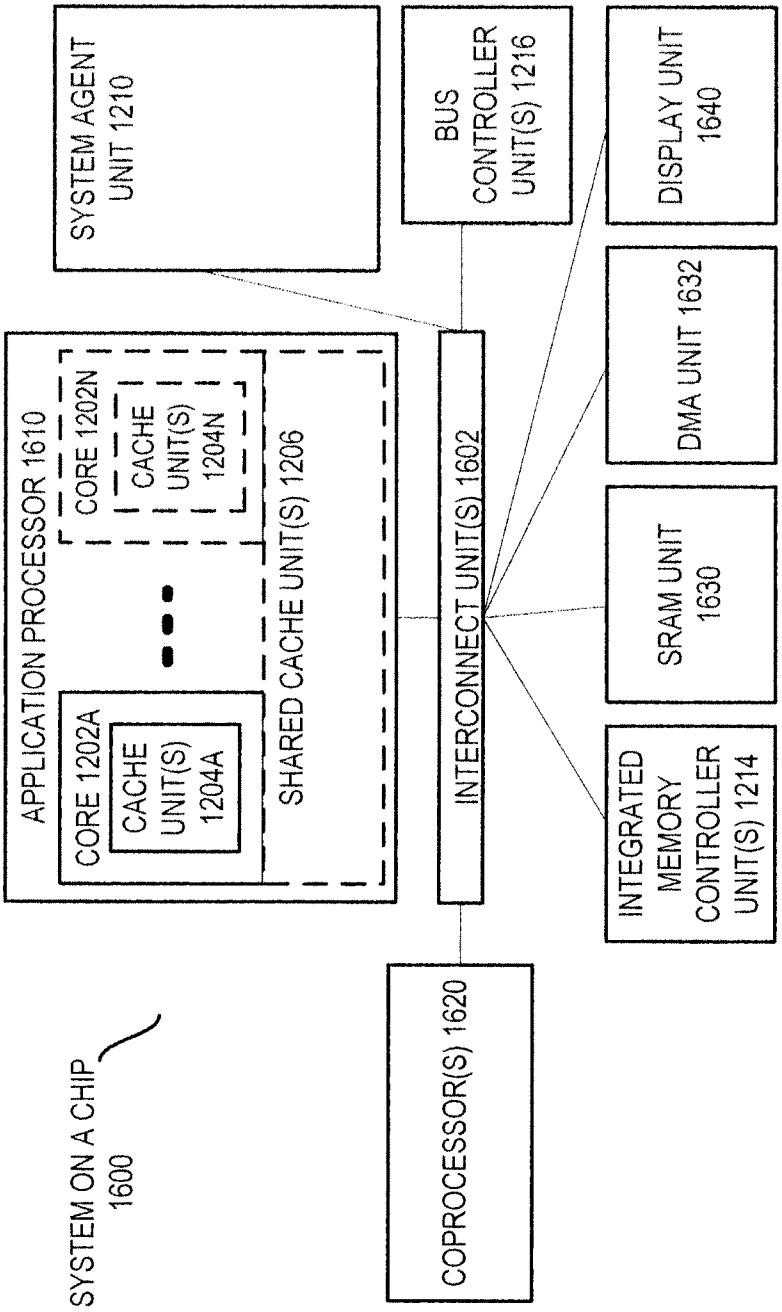

SYSTEM ON A CHIP
1600
APPLICATION PROCESSOR 1610
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
SYSTEM AGENT UNIT 1210
BUS CONTROLLER UNIT(S) 1216
INTERCONNECT UNIT(S) 1602
DISPLAY UNIT 1640
DMA UNIT 1632
SRAM UNIT 1630
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214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

FIG. 16

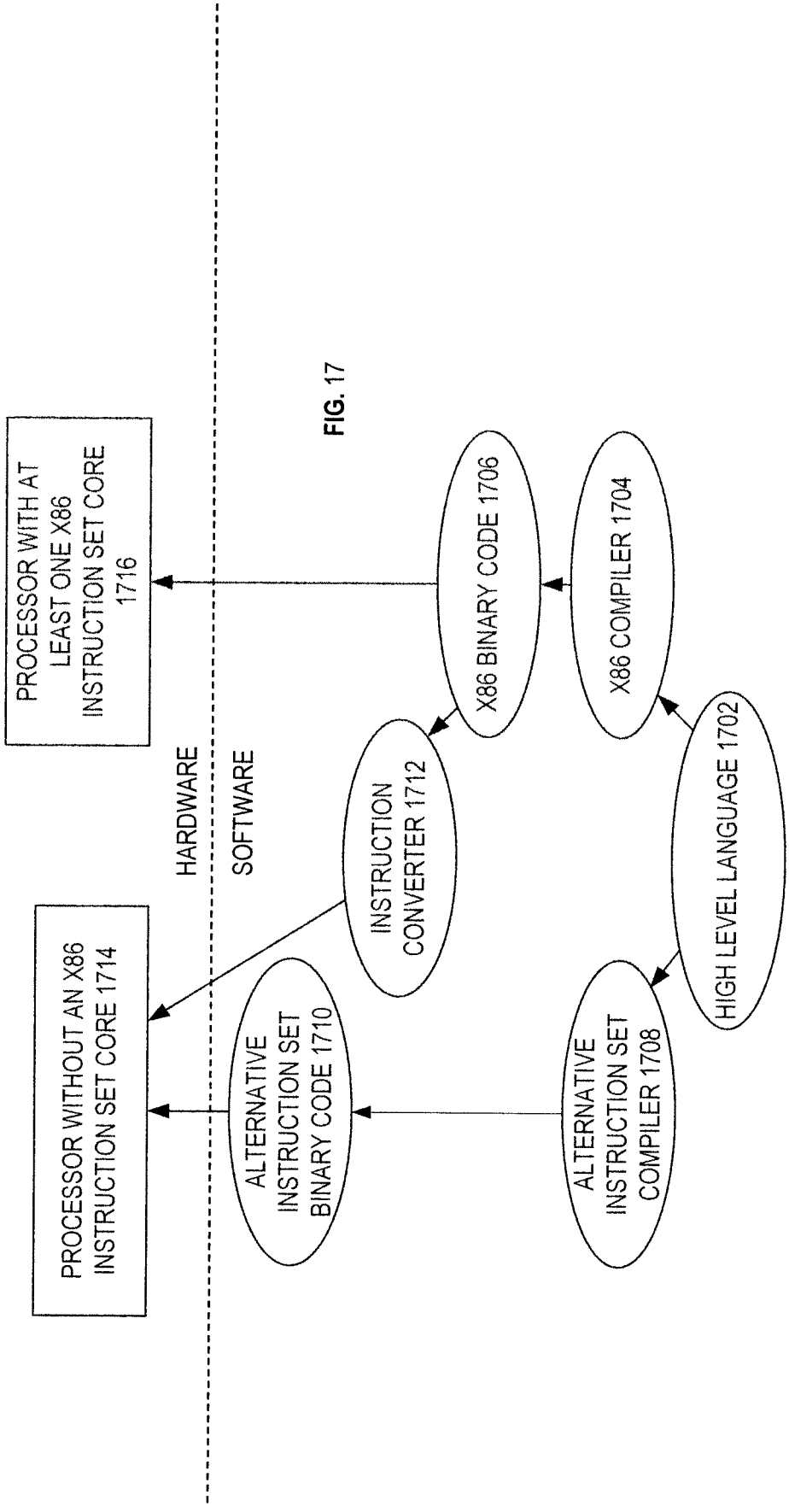

PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE 1716
HARDWARE
SOFTWARE
X86 BINARY CODE 1706
X86 COMPILER 1704
INSTRUCTION CONVERTER 1712
HIGH LEVEL LANGUAGE 1702
PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE 1714
ALTERNATIVE INSTRUCTION SET BINARY CODE 1710
ALTERNATIVE INSTRUCTION SET COMPILER 1708

FIG. 17

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2011/067275

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/06(2006.01)i, G06F 9/30(2006.01)1, G06F 9/305(2006.01)1

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 9/06; G06F 9/30; G06F 15/00; G06F 11/10; G06F 9/315

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

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKOMPASS(KIPO internal) & Keywords: vector register, location, memory, fetch, load, instruction set architecture;

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2008-0244369 A1 (JIANG YIB0 et al.) 02 October 2008 See the abstract, claims 1-20 and figures 1-5.	1-20
A	US 06088782 A (LEE DE-LEI et al.) 11 July 2000 See the abstract, claims 1-33 and figures 1-11.	1-20
A	US 2006-0149938 A1 (HONG JIANG et al.) 06 July 2006 See the abstract, claims 1-22 and figures 1-23.	1-20
A	US 2009-0172349 A1 (SPRANGLE ERIC et al.) 02 July 2009 See the abstract, claims 1-30 and figures 1-8.	1-20

☐ Further documents are listed in the continuation of Box C. ☒ 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 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 AUGUST 2012 (27.08.2012)	**28 AUGUST 2012 (28.08.2012)**

Name and mailing address of the ISA/KR

Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-70 1, Republic of Korea

Facsimile No. 82-42-472-7140

Authorized officer

BOK, Jin Yo

Telephone No. 82-42-481-51 13

Form PCT/ISA/210 (second sheet) **(July** 2009)

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US201 1/067275

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008-0244369 A1	02.10.2008	CN 10 1276637 A	01.10.2008
		CN 10 1276637 B	09.11.2011
		TW 200839780 A	01.10.2008
		US 7774661 B2	10.08.2010
		WO 2008-121147 A1	09.10.2008
US 06088782 A	11.07.2000	None	
US 2006-0149938 A1	06.07.2006	None	
US 2009-0172349 A1	02.07.2009	CN 10 1488083 A	22.07.2009