



- (51) **International Patent Classification:**
G06F 9/30 (2006.01) *G06F 9/305* (2006.01)
- (21) **International Application Number:**
PCT/US201 1/067035
- (22) **International Filing Date:**
22 December 2011 (22.12.2011)
- (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, Santa Clara, California 95052 (US).
- (72) **Inventors; and**
- (75) **Inventors/Applicants (for US only):** **OULDI-AHMED-VALL, Elmoustapha** [MR/US]; 1436 East Chicago Circle, Chandler, Arizona 85225 (US). **GIRKAR, Milind Baburao** [US/US]; 1049 West Olive Avenue #3, Sunnyvale, California 94086 (US). **VALENTINE, Robert C.** [IL/IL]; Rechov Hadganiot 33-5, 36054 Kiryat Tivon (IL). **SAIR, Suleyman** [US/US]; 14640 South 4th Avenue, Phoenix, Arizona 85045 (US). **CORBAL SAN ADRIAN, Jesus** [ES/ES]; Casp 33, 5-2 Barcelona 08010, Barcelona (ES).
- (74) **Agents:** **REPLOGLE, Eric S.** et al; Blakely, Sokoloff, Taylor & Zaffian 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).

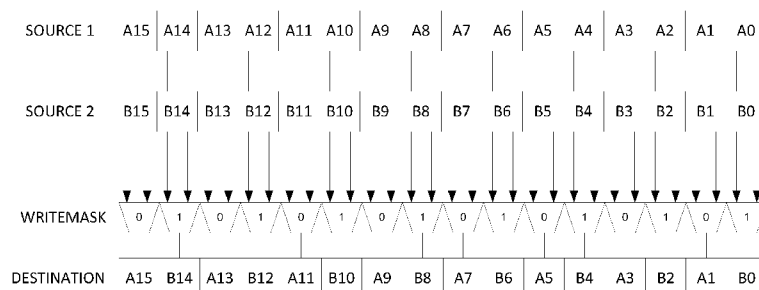
Declarations under Rule 4.17:

— of inventorship (Rule 4.17(ivf))

Published:

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

— with amended claims and statement (Art. 19(1))

(54) **Title:** BROADCAST OPERATION ON MASK REGISTER**FIGURE 1**

(57) **Abstract:** Embodiments of systems, apparatuses, and methods for performing a mask broadcast instruction in a computer processor are described. In some embodiments, the execution of a mask broadcast instruction causes a broadcast of a data element of the source operand to a destination register of the destination operand according to the broadcast size.

BROADCAST OPERATION ON MASK REGISTER

FIELD OF INVENTION

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

5 BACKGROUND

Merging data from vector sources based on control-flow information is a common issue of vector based architectures. For example, to vectorize the following code one needs: 1) a way to generate a vector of Booleans that indicate whether $a[i] > 0$ is true and 2) a way to, based on that vector of Booleans, select either value from two sources ($A[i]$ or $B[i]$) and write the contents
10 into a different destinations ($C[i]$).

```
For (i=0; i<N; i++)  
{  
    C[i] = (a[i]>0? A[i] : B[i];  
}
```

15 To use the mask data $a[i]$, one or more mask registers is filled with the mask data that is part of the array $a[]$. If the mask data is used to select data from different arrays, such as $A[]$ and $B[]$, the mask data is also known as a writemask.

BRIEF DESCRIPTION OF THE DRAWINGS

20 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:

Figure 1 illustrates an example of using a writemask.

Figures 2AB illustrate examples of a mask broadcast instruction's execution.

Figures 3AB illustrate examples of pseudo code of a mask broadcast instruction.

25 Figure 4 illustrates an embodiment of the use of a mask broadcast instruction in a processor.

Figure 5 illustrates an embodiment of a method for processing a mask broadcast instruction.

30 Figure 6 illustrates an embodiment of a method for processing a mask broadcast instruction.

Figures 7A, 7B, and 7C are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention.

Figure 8 is a block diagram of a register architecture according to one embodiment of the invention.

5 Figure 9A is a block diagram illustrating an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention.

10 Figure 9B is a block diagram illustrating 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.

Figures 10A and 10B are block diagrams illustrating an exemplary out-of-order architectures according to embodiments of the invention.

Figure 11 is a block diagram illustrating a processor that may have more than one core according to embodiments of the invention.

15 Figure 12 is a block diagram of a system in accordance with one embodiment of the invention.

Figure 13 is a block diagram of a second system in accordance with an embodiment of the invention.

20 Figure 14 is a block diagram of a third system in accordance with an embodiment of the invention.

Figure 15 is a block diagram of a SoC in accordance with an embodiment of the invention.

25 Figure 16 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

30 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.

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.

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.

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).

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 through the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands.

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).

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 as 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).

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) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme, has been , has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developers Manual, October 201 1; and see Intel® Advanced Vector Extensions Programming Reference, June 201 1).

Mask Broadcast

Below are embodiments of an instruction generically called "mask broadcast," and embodiments of systems, architectures, instruction formats etc. that may be used to execute such an instruction, that is beneficial in several different areas including what was described in the background. The execution of a mask broadcast instruction efficiently deals with the loading of mask register(s) with mask data. In one embodiment, the mask data is also called a writemask when the mask data is used to select source data for a vector register. In other words, the execution of a mask broadcast instruction causes a processor to perform a broadcast of data from either one source or a plurality of sources into a mask register. In some embodiments, at least one of the sources is a register such as a 128-, 256-, 512-bit vector register, etc. In some embodiments, at least one of the source operands is a collection of data elements associated with a starting memory location. Additionally, in some embodiments, data elements of one or both sources go through a data transformation such as swizzle, broadcast, conversion, etc. (examples will be discussed herein) prior to any mask broadcasting. In another embodiment, the destination is a register such as an 8-bit mask register , 16-bit mask register, 32-bit mask register, 64-bit mask register, etc. In one embodiment, the kbroadcast instruction can be a VEX type of an instruction.

An exemplary format of this instruction is "KBROADCAST{BAV/D/Q} k1, k2/memory {k3 }," where the operands k1 is the destination mask register, k2/memory is the first source and k3 is an optional other source that is ANDed with the first source. In one embodiment, KBROADCAST {B/W/D/Q} uses the first source and broadcasts some or all of the contents of the first source to the destination mask register. In one embodiment, KBROADCAST {B/W/D/Q} uses the least significant bit of the source to broadcast to the mask register. In another embodiment, some or all of the contents of the first source is ANDed with contents of the second source. In addition, KBROADCAST{B/W/D/Q} broadcasts the data into a collection of consecutive bits in the destination mask register. The number of bits broadcast is

based on the suffix of the instruction name. For example and in one embodiment, "B" means that sixty-four bits of data is broadcasted, "W" thirty-two bits of data (a word) is broadcasted, "D" sixteen bits of data is broadcasted (double word), "Q" eight bits of data is broadcasted (quad word) for a resulting mask register on 512-bit vector registers. In some embodiments, the destination writemask is also of a different size (8 bits, 32 bits, etc.). KBROADCAST is the instruction's opcode. Typically, each operand is explicitly defined in the instruction. 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 like "W" described later. In most embodiments, W will indicate that each data elements are either 32 or 64 bits. If the data elements are 32 bits in size, and the sources are 512 bits in size, then there are sixteen (16) data elements per source.

An example of how a writemask is used is illustrated in Figure 1. In this example, there are two sources each having 16 data elements. In most cases, one of these sources is a register (for this example, source 1 is treated as being a 512-bit register such as a ZMM register with 16 32-bit data elements, however, other data element and register sizes may be used such as XMM and YMM registers and 16- or 64-bit data elements). The other (optional) source is either a register or a memory location (in this illustration source 2 is the other source). If the second source is a memory location, in most embodiments it is placed into a temporary register prior to any broadcasting of the sources. Additionally, data elements of the memory location may undergo a data transformation prior to that placement into the temporary register. The mask pattern shown is 0x5555.

In this example, for each bit position of the writemask that has a value of "1" it is an indication that the corresponding data element of the second source (source 2) should be written into the corresponding data element position of the destination register. Accordingly, the first, third, fifth, etc. bit positions of source 2 (B0, B2, B4, etc.) are written into the first, third, fifth, etc. data element positions of the destination. Where the writemask has a "0" value, the data element of the first source is written into the corresponding data element position of the destination. Of course, the use of "1" and "0" could be flipped depending upon the implementation. Additionally, while this figure and above description considers the respective first positions to be the least significant positions, in some embodiments the first positions are the most significant positions.

Figure 2A illustrates an example of a mask broadcast instruction's execution using one source. In Figure 2A, content of the source 200 is broadcasted into the writemask 202. In one

embodiment, the least significant bit is broadcasted from the source 200 into each writemask. For example and in one embodiment, the least significant bit of the source 200 is broadcasted into the least significant bit of the writemask 202. As another example and in another embodiment, the least significant bit of the source 200 is broadcasted into the whole of the writemask 202. The number of bits written into the writemask is based on the suffix of the instructions (e.g., 8, 16, 32, 64 bits, etc.). For example and in one embodiment, the least significant bit of source 200 AO is broadcasted into the first eight bits of writemask 202.

Figure 2B illustrates an example of a mask broadcast instruction's execution using two sources. In Figure 2B, content of the source 252 is ANDed with the contents of source 254 and are broadcasted into the writemask 256. In one embodiment, the same content of one source is ANDed with different content of the other source. For example and in one embodiment, the least significant bit of source 252 is ANDed with different content of source 254. In this embodiment, the result of this AND operation is stored into a corresponding location of the writemask 256. For example and in one embodiment, the least significant bit of the source 252 AO is ANDed with each of the first eight bits of the source 254 (e.g., B7, B6, B5, B4, B3, B2, B1, and B0). The results of these AND operations are written into the corresponding bits of the writemask 256.

An example of the kbroadcast instruction used in a code sequence is as follows:

```

for(i)
{
20   bool useAlpha = bitTable[i];
   for(j)
   {
     ...
     if( useAlpha )
25     {
       C[i][j] = Alpha[i][j] - Beta[i][j];
     }
     else
     {
30     C[i][j] = Beta[i][j];
     }
   }
}

```

}

In the above code, the scalar Boolean useAlpha determines if array Alpha is used for all the elements in row i. Using the kbroadcast instruction, a compiler can broadcast useAlpha into a mask register (say kl). The if statement boils down to a subtraction into C with sources Alpha and Beta under the writemask kl and a move into C from Beta under the inverse of kl. If there was another if conditional in either the "if" or the "else" part (say if B[i][j]>0), the compiler can use the two source kbroadcast to merge the useAlpha and B[i][j]>0 masks.

Figures 3A and 3B illustrate examples of pseudo code for different embodiments of a mask broadcast instruction. In Figure 3A, the pseudo code 302 illustrates a mask broadcast from one source. In Figure 3B, the pseudo code 352 illustrates a mask broadcast from two sources that are ANDed together.

Figure 4 illustrates an embodiment of the use of a mask broadcast instruction in a processor. A mask broadcast instruction with a destination operand, a two source operands, an offset (if any), and a writemask is fetched at 401. In some embodiments, the destination operand is a 16-bit register (such as a "k" mask register detailed later). At least one of the source operands may be a memory source operand. In other embodiments, one source may be a mask register and the other source may be memory, or both sources may be mask registers.

The mask broadcast instruction is decoded at 403. 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.

The source operand value(s) are retrieved/read at 405. If both sources are registers then those 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.

If there is any data element transformation to be performed (such as an upconversion, broadcast, swizzle, etc. which are detailed later) it may be performed at 407. For example, a 16-bit data element from memory may be upconverted into a 32-bit data element or data elements may be swizzled from one pattern to another (e.g., XYZW XYZW XYZW ... XYZW to XXXXXXXX YYYYYYYY ZZZZZZZZZZ WWWWWWWW).

The mask broadcast instruction (or operations comprising such an instruction such as microoperations) is executed by execution resources at 409. This execution causes a broadcast of data from one or more sources into the destination mask register(s). For example, the least

significant bit of a data element of the source operand is broadcasted over a consecutive collection of bits of a mask register. As another example, the least significant bit of one source is ANDed with data from another source, where the result of the ANDed operation is stored into a corresponding location in the mask register. Examples of such a mask broadcasting are

5 illustrated in Figures 2AB.

The resulting data elements of the mask broadcast are stored into the destination register at 411. Again, examples of this are shown in Figures 2AB. While 409 and 411 have been illustrated separately, in some embodiments they are performed together as a part of the execution of the instruction.

10 While the above has been illustrated in one type of execution environment it is easily modified to fit in other environments such as the in-order and out-of-order environments detailed below.

Figure 5 illustrates an embodiment of a method for processing a mask broadcast instruction. In this embodiment it is assumed that some, if not all, of the operations 401-407
15 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 (source(s) and destination) retrieval shown.

At 501, the first source data, optional second source data, and destination data size are received. For example, a first source data element of the first source data is received from a first
20 source operand. In one embodiment, the first source data element is the least significant bit of the first source data element stored in the first source operand. As another example, the optional second source data is received from a second source operand. In some embodiments, the destination size is received from the corresponding instruction operands. In another embodiment, the destination size is fixed based on the instruction name. In this embodiment, a suffix of the
25 instruction name determines the destination size. For example and in one embodiment, "B" means that sixty-four bits of data is broadcasted, "W" thirty-two bits of data (a word) is broadcasted, "D" sixteen bits of data is broadcasted (double word), "Q" eight bits of data is broadcasted (quad word) for a resulting mask register on 512-bit vector registers.

At 503-511, a loop is performed to broadcast data to mask registers. At 505, the
30 broadcast data is set as the first source data. For example, the least significant bit of a data element of the first source data is the broadcast data. While in one embodiment, the first source data is the same throughout the loop, in an alternate embodiment, the first source data may vary

during the loop execution. At 507, if the second source data is used, the corresponding second source data is ANDed with the broadcast data. For example, content of the source 252 is ANDed with the contents of source 254 and are broadcasted into the mask registers 256 as described in Figure 2B above. If no second source is used, no operation is performed at 507. At 509, the
5 broadcast data is copied to the corresponding destination location. For example, the content of source 202 is copied to the appropriate destination location 204 as described in Figure 2A above. At 511, the loop ends.

Figure 6 illustrates an embodiment of a method for processing a mask broadcast instruction. In this embodiment it is assumed that some, if not all, of the operations 401-407
10 have been performed prior to 601. At 601, whether the value for each of the destination bit positions requires a combination of two sources is determined.

If the mask broadcast values are from one source, at 603, for each destination bit position of the writemask, the corresponding value(s) are stored in that destination bit position. For example, the least significant bit of the source is stored into the corresponding bit position of the
15 writemask as described above in Figure 2A. If the mask broadcast values is a combination of sources, at 605, for each destination bit position of the writemask, the corresponding source values are ANDed together and the resulting value is stored in the that destination bit position. For example, the least significant bit of the source 252 AO is ANDed with the first eight bits of the source 254, where the resulting value is written into the corresponding bit position of the
20 writemask 256 as described above in Figure 2B. In some embodiments, 603 and 605 are performed in parallel.

While Figures 5 and 6 have discussed mask broadcasting based on a single bit from a first source, other embodiments can be envisioned (mask broadcasting using more than a single, broadcasting using a pattern of bits, etc.). Additionally, it should be clearly understood that other
25 types of mask broadcasting can be used. An advantage of doing the mask broadcasting as a single instruction is that a program would have a smaller binary that has instruction cache implications. For example and in one embodiment, during execution, there could be less stress on fetch, decode, execute resources in the pipeline. As a result, this program would potentially execute faster.

30 ***Exemplary Instruction Formats***

Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below.

Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

VEX Instruction Format

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

Figure 7A illustrates an exemplary AVX instruction format including a VEX prefix 702, real opcode field 730, Mod R/M byte 740, SIB byte 750, displacement field 762, and IMM8 772. **Figure 7B** illustrates which fields from Figure 7A make up a full opcode field 774 and a base operation field 742. **Figure 7C** illustrates which fields from Figure 7A make up a register index field 744.

VEX Prefix (Bytes 0-2) 702 is encoded in a three-byte form. The first byte is the Format Field 740 (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 705 (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 715 (VEX byte 1, bits [4:0] - mmmmm) includes content to encode an implied leading opcode byte. W Field 764 (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 720 (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 Is 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 768 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 725 (VEX byte 2, bits [1:0]-pp) provides additional bits for the base operation field.

Real Opcode Field 730 (Byte 3) is also known as the opcode byte. Part of the opcode is specified in this field.

MOD R/M Field 740 (Byte 4) includes MOD field 742 (bits [7-6]), Reg field 744 (bits [5-3]), and R/M field 746 (bits [2-0]). The role of Reg field 744 may include the following:
 5 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 746 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) - The content of Scale field 750 (Byte 5) includes SS752 (bits
 10 [7-6]), which is used for memory address generation. The contents of SIB.xxx 754 (bits [5-3]) and SIB.bbb 756 (bits [2-0]) have been previously referred to with regard to the register indexes Xxxx and Bbbb.

The Displacement Field 762 and the immediate field (IMM8) 772 contain address data.

Exemplary Encoding into VEX

15 An exemplary encoding into VEX for an instruction is illustrated in Appendix A below.

Exemplary Encoding into the Specific Vector Friendly Instruction Format

Exemplary Register Architecture

Figure 8 is a block diagram of a register architecture 800 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 810 that are 512 bits
 20 wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15.

Write mask registers 815 - in the embodiment illustrated, there are 8 write mask registers
 25 (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers 815 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.

30 General-purpose registers 825 - in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address

memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

Scalar floating point stack register file (x87 stack) 845, on which is aliased the MMX packed integer flat register file 850 - in the embodiment illustrated, the x87 stack is an eight-
5 element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

Alternative embodiments of the invention may use wider or narrower registers.

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

Exemplary Core Architectures, Processors, and Computer Architectures

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general
15 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
20 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
25 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
30 processors and computer architectures.

Exemplary Core Architectures

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

Figure 9A 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 9B** 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 9A-B illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

In Figure 9A, a processor pipeline 900 includes a fetch stage 902, a length decode stage 904, a decode stage 906, an allocation stage 908, a renaming stage 910, a scheduling (also known as a dispatch or issue) stage 912, a register read/memory read stage 914, an execute stage 916, a write back/memory write stage 918, an exception handling stage 922, and a commit stage 924.

Figure 9B shows processor core 990 including a front end unit 930 coupled to an execution engine unit 950, and both are coupled to a memory unit 970. The core 990 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 990 may be a special-purpose core, such as, for example, a network or communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit 930 includes a branch prediction unit 932 coupled to an instruction cache unit 934, which is coupled to an instruction translation lookaside buffer (TLB) 936, which is coupled to an instruction fetch unit 938, which is coupled to a decode unit 940. The decode unit 940 (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 940 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 990 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 940 or otherwise within the front end unit 930). The decode unit 940 is coupled to a rename/allocator unit 952 in the execution engine unit 950.

5 The execution engine unit 950 includes the rename/allocator unit 952 coupled to a retirement unit 954 and a set of one or more scheduler unit(s) 956. The scheduler unit(s) 956 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 956 is coupled to the physical register file(s) unit(s) 958. Each of the physical register file(s) units 958 represents one or more physical register files,
10 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 958 comprises a vector registers unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector
15 registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) 958 is overlapped by the retirement unit 954 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 954 and the
20 physical register file(s) unit(s) 958 are coupled to the execution cluster(s) 960. The execution cluster(s) 960 includes a set of one or more execution units 962 and a set of one or more memory access units 964. The execution units 962 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
25 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) 956, physical register file(s) unit(s) 958, and execution cluster(s) 960 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
30 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) 964). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

The set of memory access units 964 is coupled to the memory unit 970, which includes a data TLB unit 972 coupled to a data cache unit 974 coupled to a level 2 (L2) cache unit 976. In one exemplary embodiment, the memory access units 964 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 972 in the memory unit 970. The instruction cache unit 934 is further coupled to a level 2 (L2) cache unit 976 in the memory unit 970. The L2 cache unit 976 is coupled to one or more other levels of cache and eventually to a main memory.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 900 as follows: 1) the instruction fetch 938 performs the fetch and length decoding stages 902 and 904; 2) the decode unit 940 performs the decode stage 906; 3) the rename/allocator unit 952 performs the allocation stage 908 and renaming stage 910; 4) the scheduler unit(s) 956 performs the schedule stage 912; 5) the physical register file(s) unit(s) 958 and the memory unit 970 perform the register read/memory read stage 914; the execution cluster 960 perform the execute stage 916; 6) the memory unit 970 and the physical register file(s) unit(s) 958 perform the write back/memory write stage 918; 7) various units may be involved in the exception handling stage 922; and 8) the retirement unit 954 and the physical register file(s) unit(s) 958 perform the commit stage 924.

The core 990 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 990 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2, etc.), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a

combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology).

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated
5 embodiment of the processor also includes separate instruction and data cache units 934/974 and a shared L2 cache unit 976, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the
10 cache may be external to the core and/or the processor.

Specific Exemplary In-Order Core Architecture

Figures 10A-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
15 interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

Figure 10A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1002 and with its local subset of the Level 2 (L2) cache 1004, according to embodiments of the invention. In one embodiment, an instruction decoder 1000
20 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 1006 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 1008 and a vector unit 1010 use separate register sets (respectively, scalar registers 1012 and vector registers 1014) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 1006,
25 alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

The local subset of the L2 cache 1004 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
30 own local subset of the L2 cache 1004. Data read by a processor core is stored in its L2 cache subset 1004 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 1004 and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1012-bits wide per direction.

Figure 10B is an expanded view of part of the processor core in Figure 10A according to embodiments of the invention. Figure 10B includes an LI data cache 1006A part of the LI cache 1004, as well as more detail regarding the vector unit 1010 and the vector registers 1014. Specifically, the vector unit 1010 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 1028), 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 1020, numeric conversion with numeric convert units 1022A-B, and replication with replication unit 1024 on the memory input. Write mask registers 1026 allow predicating resulting vector writes.

Processor with integrated memory controller and graphics

Figure 11 is a block diagram of a processor 1100 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 11 illustrate a processor 1100 with a single core 1102A, a system agent 1110, a set of one or more bus controller units 1116, while the optional addition of the dashed lined boxes illustrates an alternative processor 1100 with multiple cores 1102A-N, a set of one or more integrated memory controller unit(s) 1114 in the system agent unit 1110, and special purpose logic 1108.

Thus, different implementations of the processor 1100 may include: 1) a CPU with the special purpose logic 1108 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 1102A-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 1102A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 1102A-N being a large number of general purpose in-order cores. Thus, the processor 1100 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 1100 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.

5 The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 1106, and external memory (not shown) coupled to the set of integrated memory controller units 1114. The set of shared cache units 1106 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 1112 interconnects the integrated graphics logic 1108, the set of shared cache
10 units 1106, and the system agent unit 1110/integrated memory controller unit(s) 1114, 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 1106 and cores 1102A-N.

In some embodiments, one or more of the cores 1102A-N are capable of multi-threading.
15 The system agent 1110 includes those components coordinating and operating cores 1102A-N. The system agent unit 1110 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 1102A-N and the integrated graphics logic 1108. The display unit is for driving one or more externally connected displays.

20 The cores 1102A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 1102A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Computer Architectures

25 **Figures 12-15** 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
30 various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

Referring now to **Figure 12**, shown is a block diagram of a system 1200 in accordance with one embodiment of the present invention. The system 1200 may include one or more processors 1210, 1215, which are coupled to a controller hub 1220. In one embodiment the controller hub 1220 includes a graphics memory controller hub (GMCH) 1290 and an
5 Input/Output Hub (IOH) 1250 (which may be on separate chips); the GMCH 1290 includes memory and graphics controllers to which are coupled memory 1240 and a coprocessor 1245; the IOH 1250 is coupled input/output (I/O) devices 1260 to the GMCH 1290. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 1240 and the coprocessor 1245 are coupled directly to the processor 1210,
10 and the controller hub 1220 in a single chip with the IOH 1250.

The optional nature of additional processors 1215 is denoted in Figure 12 with broken lines. Each processor 1210, 1215 may include one or more of the processing cores described herein and may be some version of the processor 1100.

The memory 1240 may be, for example, dynamic random access memory (DRAM),
15 phase change memory (PCM), or a combination of the two. For at least one embodiment, the controller hub 1220 communicates with the processor(s) 1210, 1215 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 1295.

In one embodiment, the coprocessor 1245 is a special-purpose processor, such as, for
20 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 1220 may include an integrated graphics accelerator.

There can be a variety of differences between the physical resources 1210, 1215 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power
25 consumption characteristics, and the like.

In one embodiment, the processor 1210 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 1210 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 1245. Accordingly, the processor 1210 issues these
30 coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 1245. Coprocessor(s) 1245 accept and execute the received coprocessor instructions.

Referring now to **Figure 13**, shown is a block diagram of a first more specific exemplary system 1300 in accordance with an embodiment of the present invention. As shown in Figure 13, multiprocessor system 1300 is a point-to-point interconnect system, and includes a first processor 1370 and a second processor 1380 coupled via a point-to-point interconnect 1350.

5 Each of processors 1370 and 1380 may be some version of the processor 1100. In one embodiment of the invention, processors 1370 and 1380 are respectively processors 1210 and 1215, while coprocessor 1338 is coprocessor 1245. In another embodiment, processors 1370 and 1380 are respectively processor 1210 coprocessor 1245.

Processors 1370 and 1380 are shown including integrated memory controller (IMC) units
10 1372 and 1382, respectively. Processor 1370 also includes as part of its bus controller units point-to-point (P-P) interfaces 1376 and 1378; similarly, second processor 1380 includes P-P interfaces 1386 and 1388. Processors 1370, 1380 may exchange information via a point-to-point (P-P) interface 1350 using P-P interface circuits 1378, 1388. As shown in Figure 13, IMCs 1372 and 1382 couple the processors to respective memories, namely a memory 1332 and a memory
15 1334, which may be portions of main memory locally attached to the respective processors.

Processors 1370, 1380 may each exchange information with a chipset 1390 via individual P-P interfaces 1352, 1354 using point to point interface circuits 1376, 1394, 1386, 1398. Chipset 1390 may optionally exchange information with the coprocessor 1338 via a high-performance interface 1339. In one embodiment, the coprocessor 1338 is a special-purpose processor, such
20 as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

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

Chipset 1390 may be coupled to a first bus 1316 via an interface 1396. In one embodiment, first bus 1316 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.

30 As shown in Figure 13, various I/O devices 1314 may be coupled to first bus 1316, along with a bus bridge 1318 which couples first bus 1316 to a second bus 1320. In one embodiment, one or more additional processor(s) 1315, 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 1316. In one embodiment, second bus 1320 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 1320 including, for example, a keyboard and/or mouse 1322, communication devices 1327 and a storage unit 1328 such as a disk drive or other mass storage device which may include instructions/code and data 1330, in one embodiment. Further, an audio I/O 1324 may be coupled to the second bus 1320. Note that other architectures are possible. For example, instead of the point-to-point architecture of Figure 13, a system may implement a multi-drop bus or other such architecture.

Referring now to **Figure 14**, shown is a block diagram of a second more specific exemplary system 1400 in accordance with an embodiment of the present invention. Like elements in Figures 13 and 14 bear like reference numerals, and certain aspects of Figure 13 have been omitted from Figure 14 in order to avoid obscuring other aspects of Figure 14.

Figure 14 illustrates that the processors 1370, 1380 may include integrated memory and I/O control logic ("CL") 1372 and 1382, respectively. Thus, the CL 1372, 1382 include integrated memory controller units and include I/O control logic. Figure 14 illustrates that not only are the memories 1332, 1334 coupled to the CL 1372, 1382, but also that I/O devices 1414 are also coupled to the control logic 1372, 1382. Legacy I/O devices 1415 are coupled to the chipset 1390.

Referring now to **Figure 15**, shown is a block diagram of a SoC 1500 in accordance with an embodiment of the present invention. Similar elements in Figure 11 bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In Figure 15, an interconnect unit(s) 1502 is coupled to: an application processor 1510 which includes a set of one or more cores 202A-N and shared cache unit(s) 1106; a system agent unit 1110; a bus controller unit(s) 1116; an integrated memory controller unit(s) 1114; a set or one or more coprocessors 1520 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 1530; a direct memory access (DMA) unit 1532; and a display unit 1540 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 1520 include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

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

Program code, such as code 1330 illustrated in Figure 13, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

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

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

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

memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

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

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

In some cases, an instruction converter may be used to convert an instruction from a
10 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
15 processor, off processor, or part on and part off processor.

Figure 16 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
20 be implemented in software, firmware, hardware, or various combinations thereof. Figure 16 shows a program in a high level language 1602 may be compiled using an x86 compiler 1604 to generate x86 binary code 1606 that may be natively executed by a processor with at least one x86 instruction set core 1616. The processor with at least one x86 instruction set core 1616 represents any processor that can perform substantially the same functions as an Intel processor
25 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 1604 represents a compiler that is
30 operable to generate x86 binary code 1606 (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 1616. Similarly, Figure 16 shows the program in the high level language 1602 may be compiled using

an alternative instruction set compiler 1608 to generate alternative instruction set binary code 1610 that may be natively executed by a processor without at least one x86 instruction set core 1614 (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 1612 is used to convert the x86 binary code 1606 into code that may be natively executed by the processor without an x86 instruction set core 1614. This converted code is not likely to be the same as the alternative instruction set binary code 1610 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 1612 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 1606.

Certain operations of the instruction(s) in the vector friendly instruction format disclosed herein may be performed by hardware components and may be embodied in machine-executable instructions that are used to cause, or at least result in, a circuit or other hardware component programmed with the instructions performing the operations. The circuit may include a general-purpose or special-purpose processor, or logic circuit, to name just a few examples. The operations may also optionally be performed by a combination of hardware and software.

Execution logic and/or a processor may include specific or particular circuitry or other logic responsive to a machine instruction or one or more control signals derived from the machine instruction to store an instruction specified result operand. For example, embodiments of the instruction(s) disclosed herein may be executed in one or more the systems of Figures 12-15 and embodiments of the instruction(s) in the vector friendly instruction format may be stored in program code to be executed in the systems. Additionally, the processing elements of these figures may utilize one of the detailed pipelines and/or architectures (e.g., the in-order and out-of-order architectures) detailed herein. For example, the decode unit of the in-order architecture may decode the instruction(s), pass the decoded instruction to a vector or scalar unit, etc.

The above description is intended to illustrate preferred embodiments of the present invention. From the discussion above it should also be apparent that especially in such an area of technology, where growth is fast and further advancements are not easily foreseen, the invention can may be modified in arrangement and detail by those skilled in the art without departing from

the principles of the present invention within the scope of the accompanying claims and their equivalents. For example, one or more operations of a method may be combined or further broken apart.

Alternative Embodiments

5 While embodiments have been described which would natively execute the vector friendly instruction format, alternative embodiments of the invention may execute the vector friendly instruction format through an emulation layer running on a processor that executes a different instruction set (e.g., a processor that executes the MIPS instruction set of MIPS Technologies of Sunnyvale, CA, a processor that executes the ARM instruction set of ARM
10 Holdings of Sunnyvale, CA). Also, while the flow diagrams in the figures show a particular order of operations performed by certain embodiments of the invention, it should be understood that such order is exemplary (e.g., alternative embodiments may perform the operations in a different order, combine certain operations, overlap certain operations, etc.).

In the description above, for the purposes of explanation, numerous specific details
15 have been set forth in order to provide a thorough understanding of the embodiments of the invention. It will be apparent however, to one skilled in the art, that one or more other embodiments may be practiced without some of these specific details. The particular embodiments described are not provided to limit the invention but to illustrate embodiments of the invention. The scope of the invention is not to be determined by the
20 specific examples provided above but only by the claims below.

CLAIMS**What is claimed is:**

1. A method of performing mask broadcast instruction in a computer processor,
5 comprising:
 fetching the mask broadcast instruction, wherein the mask broadcast instruction
includes a destination operand, a source operand, and broadcast size;
 decoding the fetched mask broadcast instruction; and
 executing the decoded mask broadcast instruction to perform a broadcast of a data
10 element of the source operand to a destination register of the destination operand according to
the broadcast size.
2. The method of claim 1, wherein the destination register is a mask register.
- 15 3. The method of claim 1, wherein the data element is a least significant bit of data in
the source operand.
4. The method of claim 1, wherein the broadcast size is derived from the name of the
mask register instruction.
20
5. The method of claim 4, wherein the broadcast size is selected from the group
consisting of 8, 16, 32, and 64 bits.
6. The method of claim 1, wherein the source is a 512-bit register.
25
7. The method of claim 1, wherein the broadcasting is done in parallel.
8. The method of claim 1, wherein the to perform a broadcast further comprises to
combine the data element of the source with another data element of another source into a result,
30 and to broadcast the result to the destination register.

9. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to protect data stored in a storage system of a device from malware alternation, the method, comprising:

in response to a mask broadcast instruction that includes a destination operand, a first
5 source operand, and broadcast size,

retrieving a data element of the first source operand as a broadcast data,

for each destination position of the destination operand according to the
broadcast size, storing that broadcast data into the destination position.

10 10. The non-transitory machine-readable medium of claim 9, further comprising for the
each destination position:

combining the broadcast data with another data element of a second source operand.

11. The non-transitory machine-readable medium of claim 10, wherein the combining
15 in an AND operation.

12. The non-transitory machine-readable medium of claim 10, wherein the second
operand is a 512-bit register.

20 13. The non-transitory machine-readable medium of claim 10, wherein the combining
is done in parallel.

14. The non-transitory machine-readable medium of claim 9, wherein the destination
operand is a 16-bit mask register.

25 15. The non-transitory machine-readable medium of claim 9, wherein the data element
is a least significant bit of data in the source operand.

16. The non-transitory machine-readable medium of claim 9, wherein the broadcast
30 size is derived from the name of the mask register instruction.

17. The non-transitory machine-readable medium of claim 16, wherein the broadcast size is selected from the group consisting of 8, 16, 32, and 64 bits.

18. The non-transitory machine-readable medium of claim 9, wherein the second
5 source operand is a 512-bit register.

19. A processor comprising;
a hardware decoder to decode a mask broadcast instruction, wherein the mask broadcast instruction includes a writemask operand, a destination operand, a first source operand, and a
10 second source operand;
execution logic to perform a broadcast of a data element of the source operand to a destination register of the destination operand according to the broadcast size.

20. The processor of claim 19, further comprising:
15 a source register to store the first data element; and
a destination register to store the broadcasted data element.

AMENDED CLAIMS
received by the International Bureau on 28 September 2012 (28.09.2012)

What is claimed is:

1. A method of performing mask broadcast instruction in a computer processor, comprising:
 fetching the mask broadcast instruction, wherein the mask broadcast instruction includes a destination operand, a source operand, and broadcast size;
 decoding the fetched mask broadcast instruction; and
 executing the decoded mask broadcast instruction to perform a broadcast of a data element of the source operand to a destination register of the destination operand according to the broadcast size, wherein the destination register is a mask register.
2. The method of claim 1, wherein the data element is a least significant bit of data in the source operand.
3. The method of claim 1, wherein the broadcast size is derived from the name of the mask register instruction.
4. The method of claim 3, wherein the broadcast size is selected from the group consisting of 8, 16, 32, and 64 bits.
5. The method of claim 1, wherein the source is a 512-bit register.
6. The method of claim 1, wherein the broadcasting is done in parallel.
7. The method of claim 1, wherein the to perform a broadcast further comprises to combine the data element of the source with another data element of another source into a result, and to broadcast the result to the destination register.
8. A non-transitory machine-readable medium having executable instructions to cause one or more processing units to perform a method to protect data stored in a storage system of a device from malware alternation, the method, comprising:
 in response to a mask broadcast instruction that includes a destination operand, a first source operand, and broadcast size,
 retrieving a data element of the first source operand as a broadcast data,

for each destination position of the destination operand according to the broadcast size, storing that broadcast data into the destination position, wherein the destination register is a mask register.

9. The non-transitory machine-readable medium of claim 8, further comprising for the each destination position:

combining the broadcast data with another data element of a second source operand.

10. The non-transitory machine-readable medium of claim 9, wherein the combining in an AND operation.

11. The non-transitory machine-readable medium of claim 9, wherein the second operand is a 512-bit register.

12. The non-transitory machine-readable medium of claim 9, wherein the combining is done in parallel.

13. The non-transitory machine-readable medium of claim 8, wherein the destination operand is a 16-bit mask register.

14. The non-transitory machine-readable medium of claim 8, wherein the data element is a least significant bit of data in the source operand.

15. The non-transitory machine-readable medium of claim 8, wherein the broadcast size is derived from the name of the mask register instruction.

16. The non-transitory machine-readable medium of claim 15, wherein the broadcast size is selected from the group consisting of 8, 16, 32, and 64 bits.

17. The non-transitory machine-readable medium of claim 8, wherein the second source operand is a 512-bit register.

18. A processor comprising;

a hardware decoder to decode a mask broadcast instruction, wherein the mask broadcast instruction includes a writemask operand, a destination operand, a first source operand, and a second source operand;

execution logic to perform a broadcast of a data element of the source operand to a destination register of the destination operand according to the broadcast size, wherein the destination register is a mask register.

19. The processor of claim 18, further comprising:

a source register to store the first data element; and
the destination register to store the broadcasted data element.

STATEMENT UNDER ARTICLE 19 (1)

Claims 1, 9, 19, and 20 have been amended and claim 2 has been canceled. No new claims have been added to better define the invention. Support for these amendments is found in the specification and claims as originally filed, specifically.

- * Claims 1-7: Claims 2-7 are dependent of claim 1. The support for claims 1-7 can be found at least in Figures 1, 2A, and 2B of the original application and corresponding discussion in the original application.
- Claims 8-17: Claim 8 is the original claim 9, and claims 9-17 are dependent claims of claim 8. The support for claims 8-17 can be found at least in Figures 1, 2A, and 2B of the original application and corresponding discussion in the original application.
- Claim 18-19: Claim 18 is the original claim 19, and claim 19 is dependent claim of claim 18. The support for claims 18-19 can be found at least in Figures 1, 2A, and 2B of the original application and corresponding discussion in the original application.

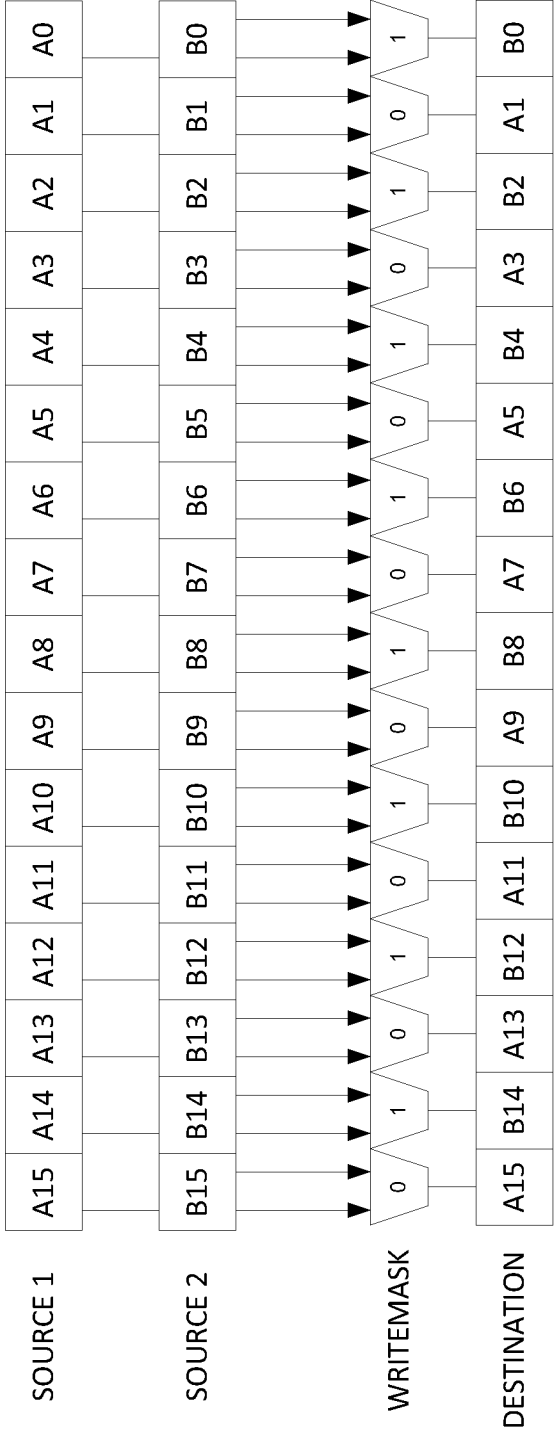


FIGURE 1

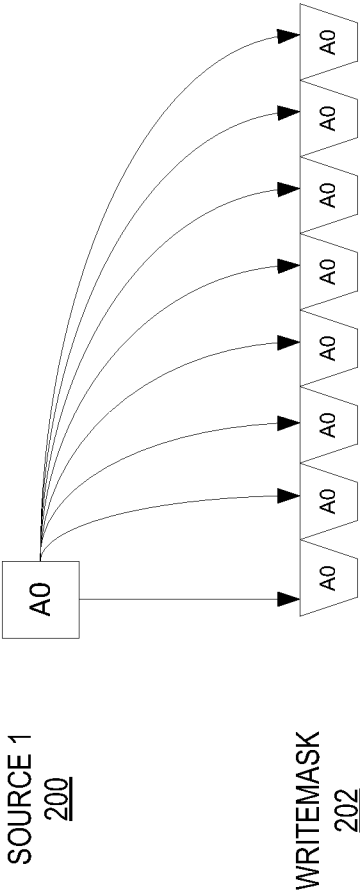


FIGURE 2A

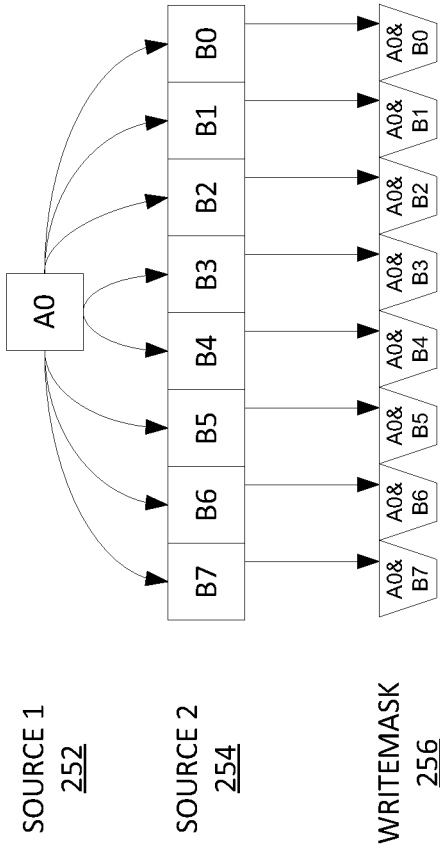


FIGURE 2B

KBROADCAST[B/W/D/Q] K1, K2
OPERATION 302
KL <= 64, 32, 16, 8 ; NUMBER OF MASK BITS TO SET (BASED ON MNEMONIC)
FOR I = <=0 TO KL-1
 DEST[I] = SRC[0];
ENDFOR
DEST[MAX_KL-1:KL] <= 0

FIGURE 3A

KBROADCAST[B/W/D/Q] K1, K2, K3
OPERATION 352
KL <= 64, 32, 16, 8 ; NUMBER OF MASK BITS TO SET (BASED ON MNEMONIC)
FOR I = <=0 TO KL-1
 DEST[I] = SRC2[0] AND SRC1[I];
ENDFOR
DEST[MAX_KL-1:KL] <= 0

FIGURE 3B

5/17

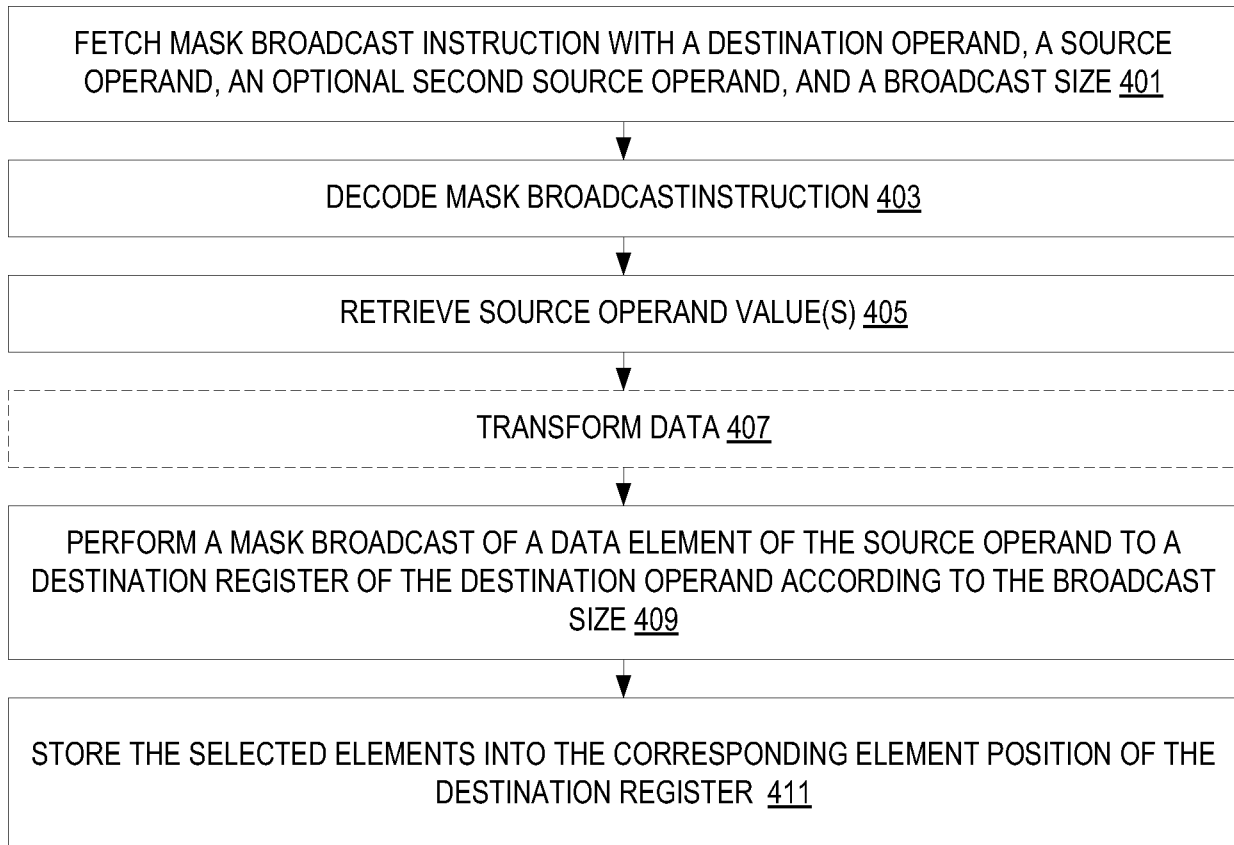


FIGURE 4

6/17

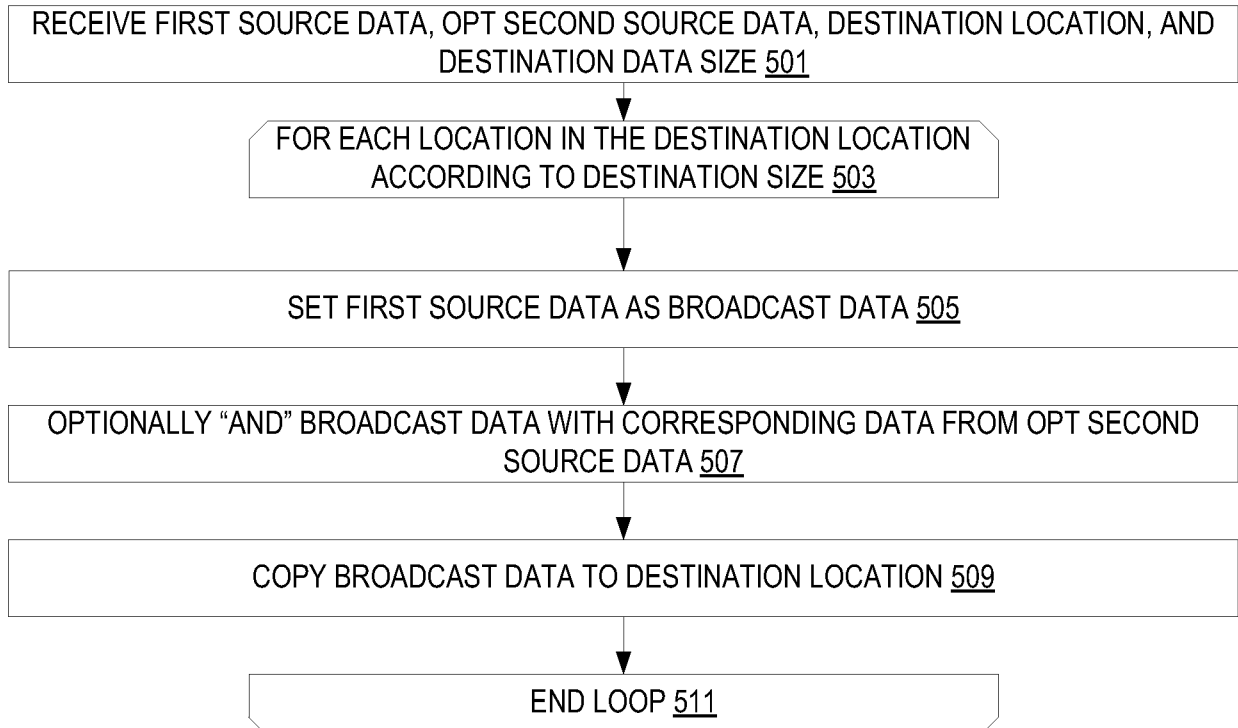


FIGURE 5

7/17

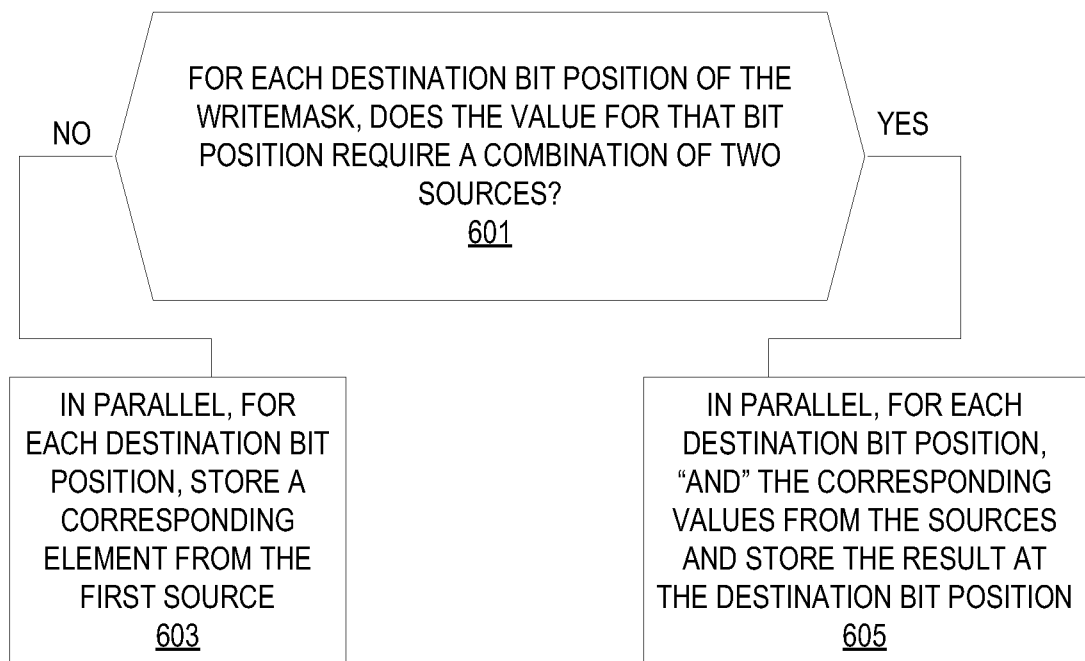
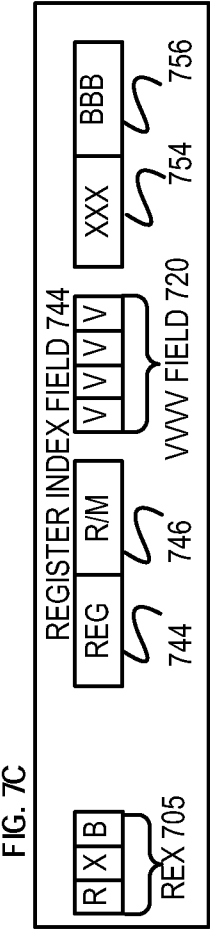
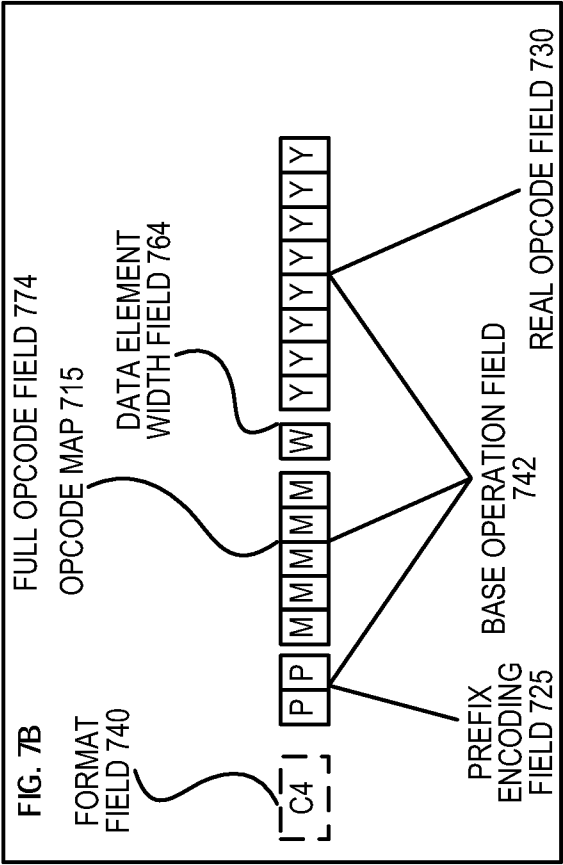
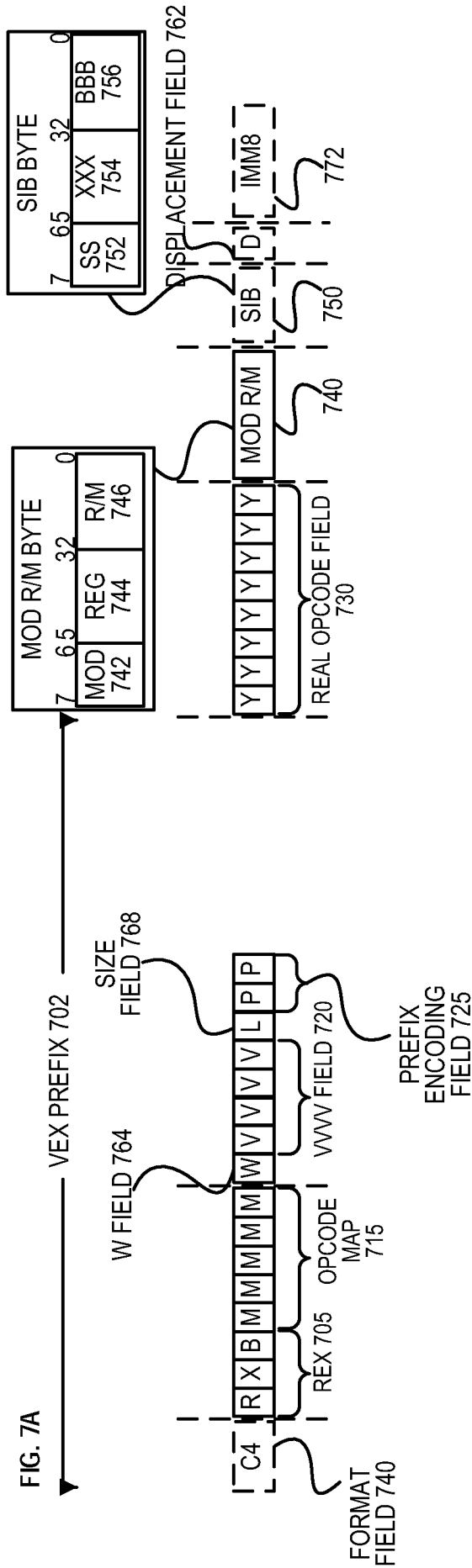
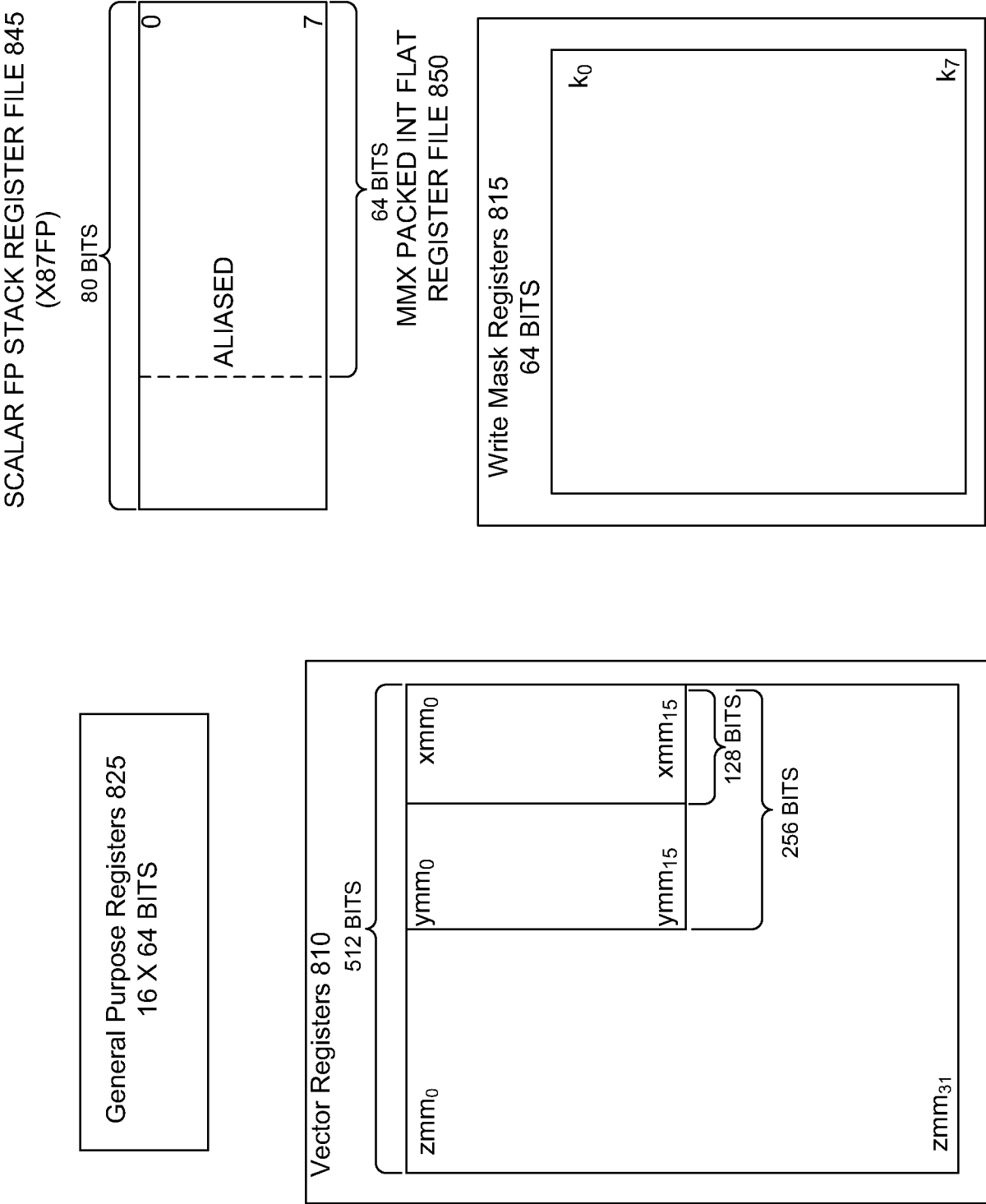


FIGURE 6



REGISTER ARCHITECTURE 800

FIG. 8



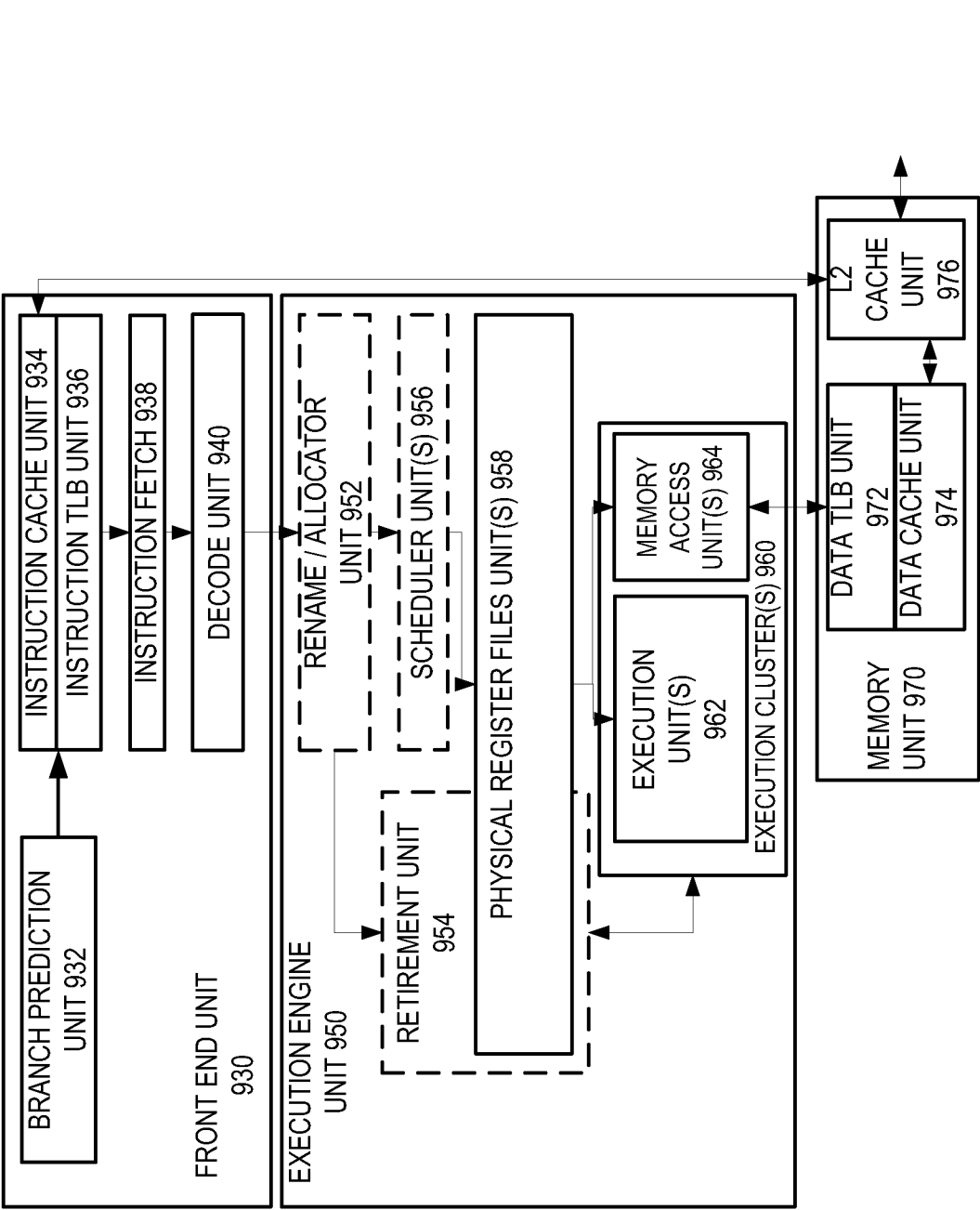
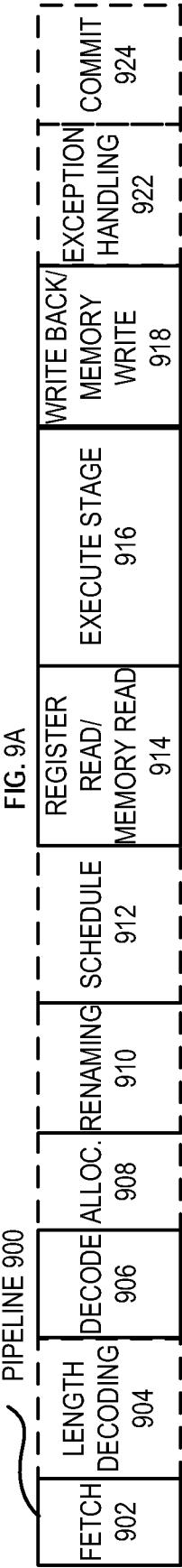


FIG. 10A

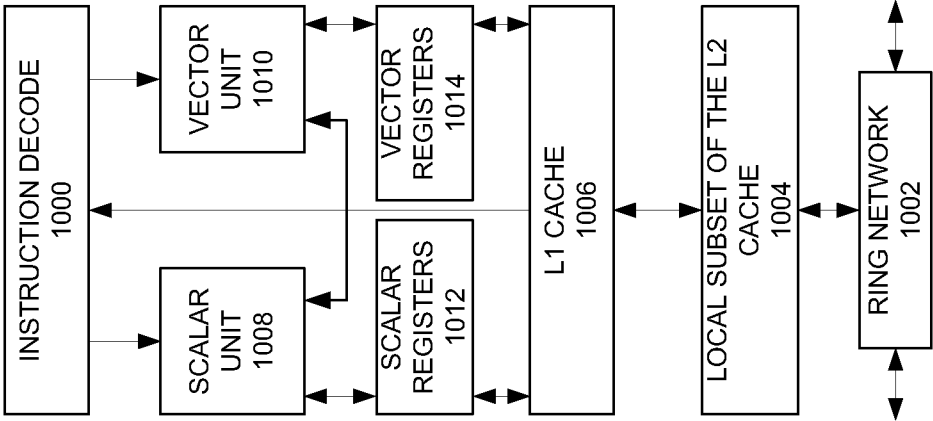
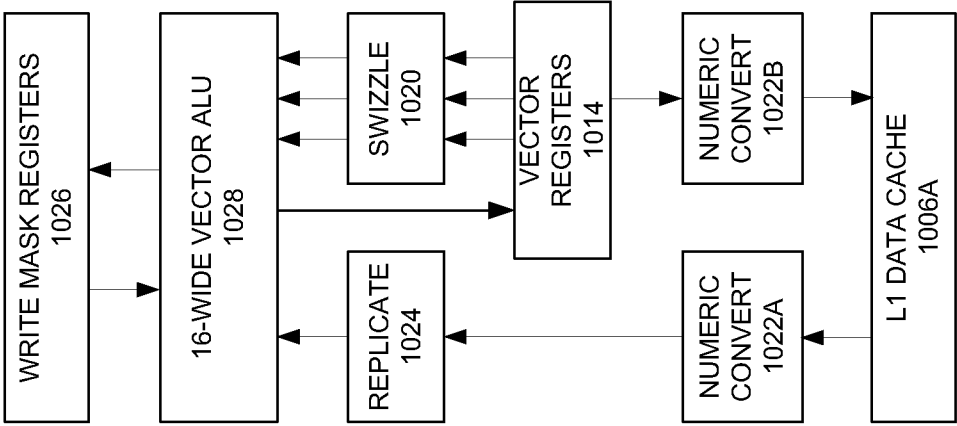
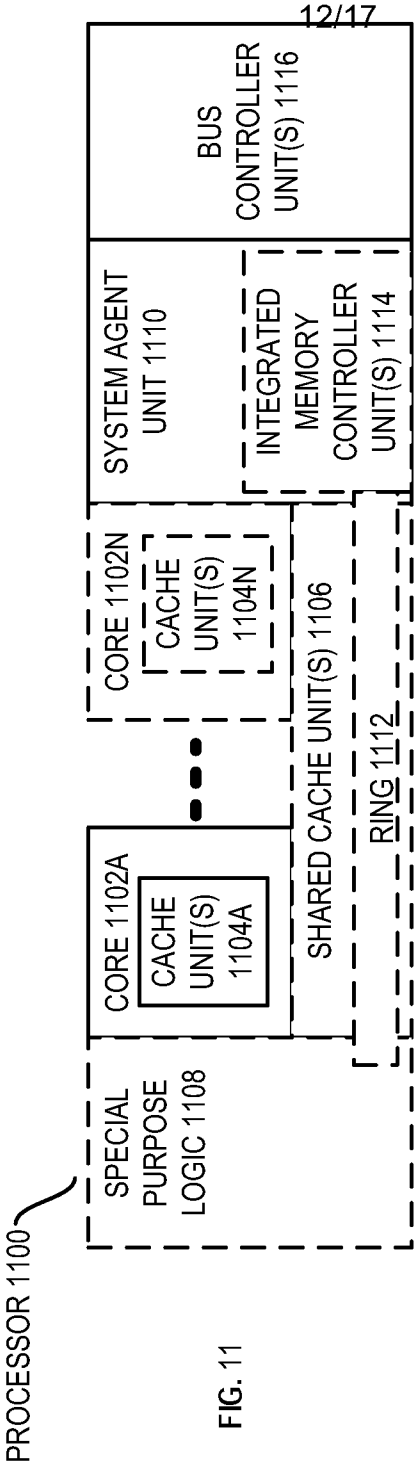


FIG. 10B





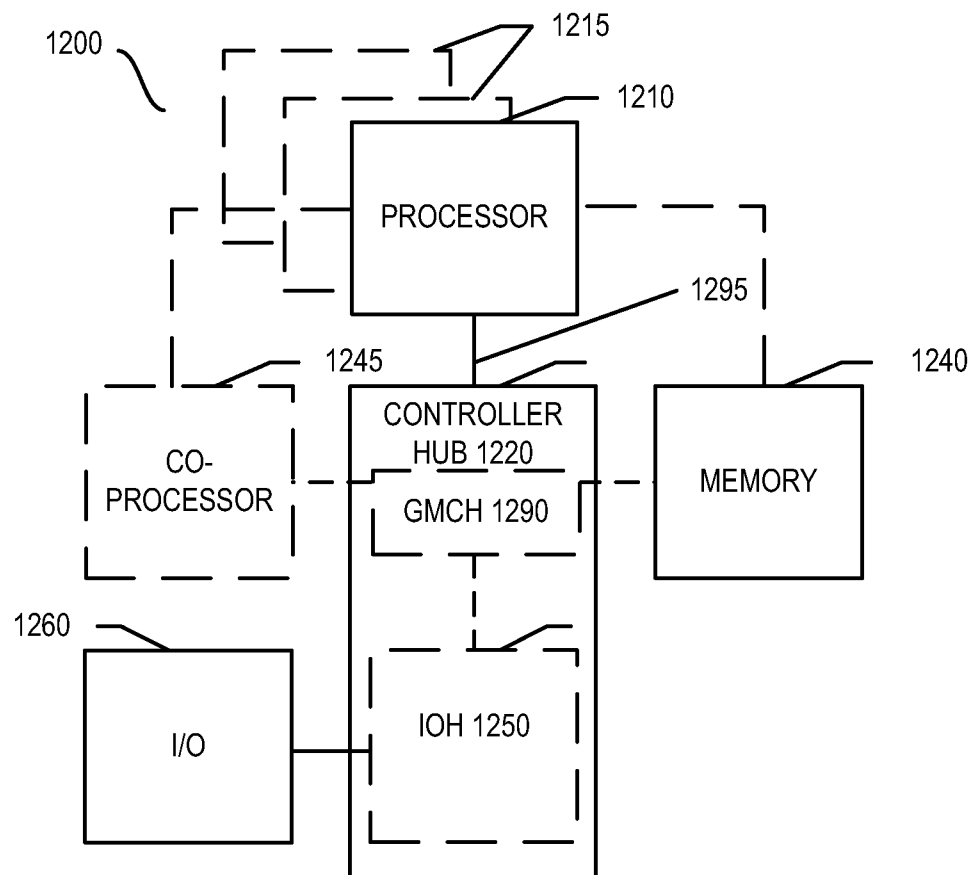
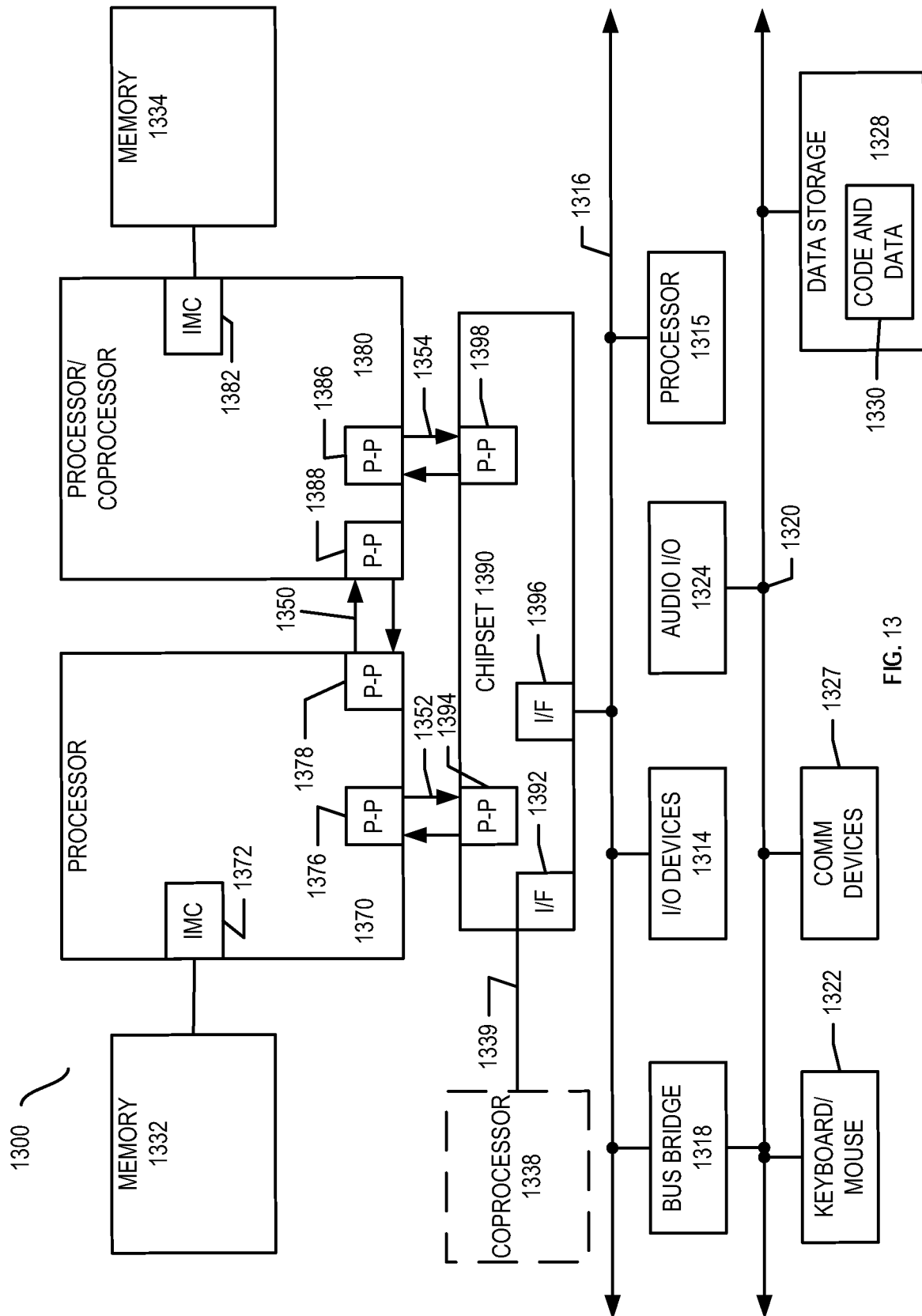


FIG. 12



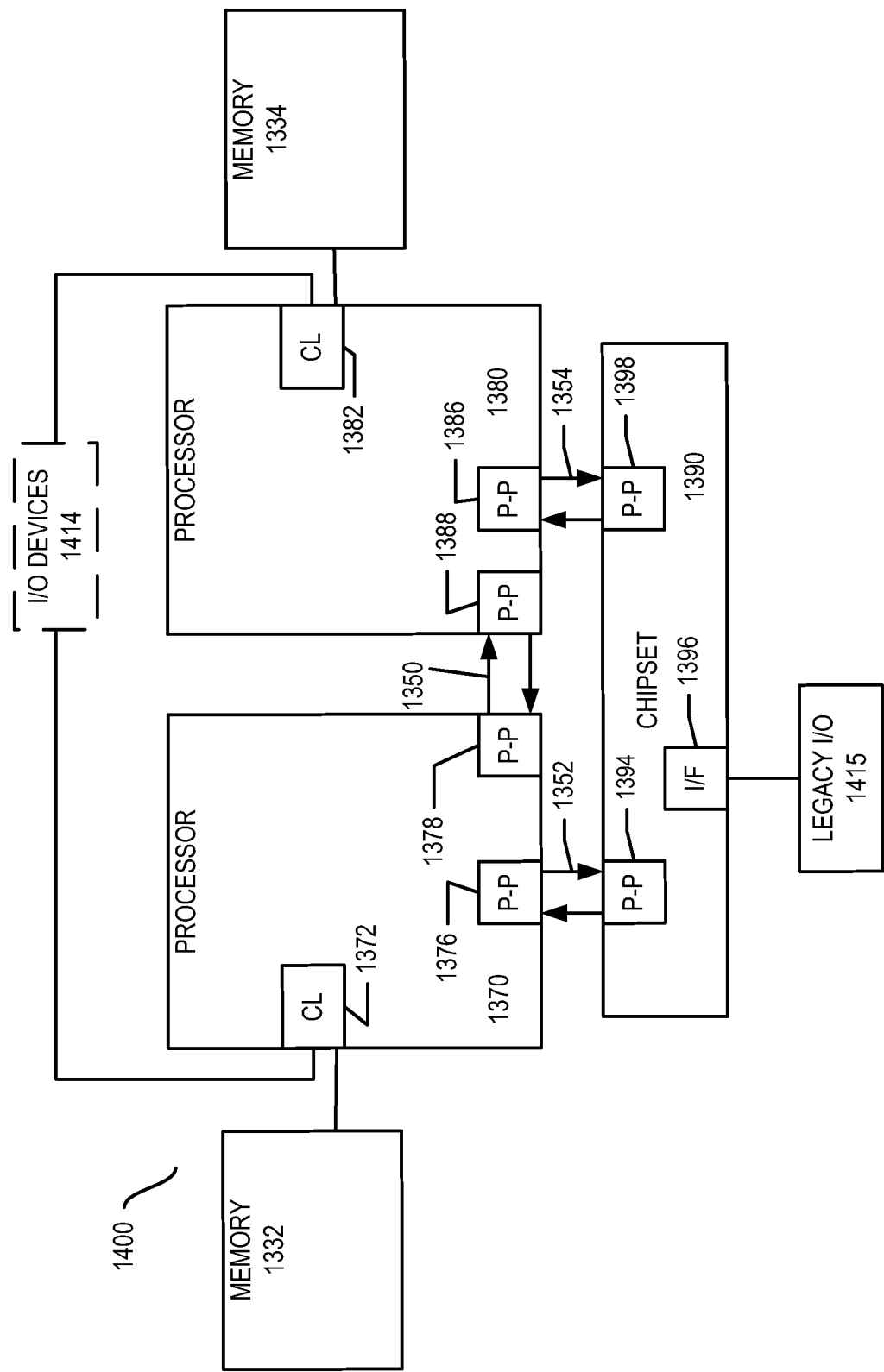


FIG. 14

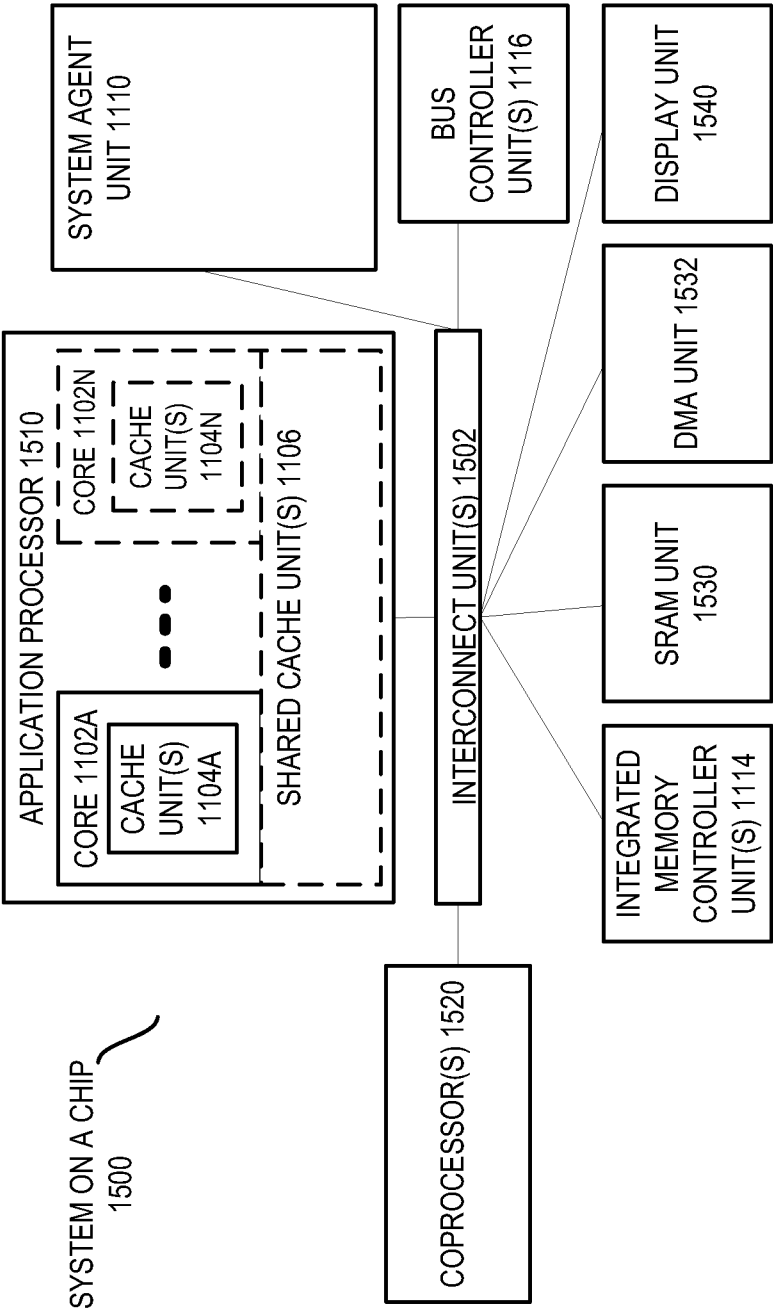


FIG. 15

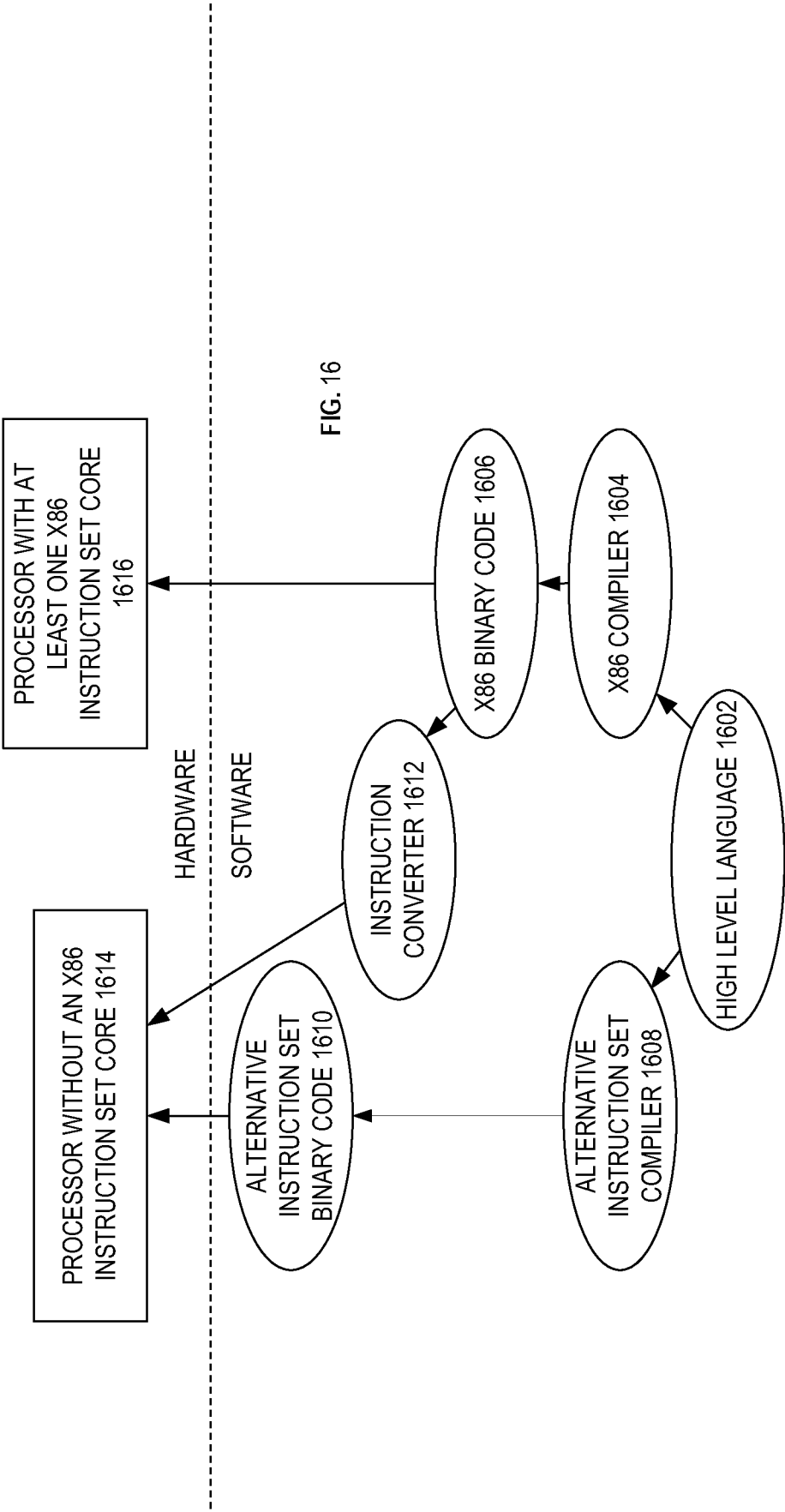


FIG. 16

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US20 11/067035**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i, G06F 9/305(2006.01)i**

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 9/30; G06F 15/00; 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: mask register, broadcast instruction, broadest size, operand

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X A	US 2004-0030863 Al (NIGEL, C. PAVER et al.) 12 February 2004 See paragraphs [0031-0039] ; claims 1-4 .	1, 9, 19 2-8, 10-18 ,20
A	US 2003-0093648 Al (WILLIAM C. MOYER) 15 May 2003 See paragraphs [0033 ,0036] ; claim 1; figures 5-7.	1-20
A	US 2002-0112147 Al (SRINIVAS CHENNUPATY et al.) 15 August 2002 See paragraphs [0022-0027 ,0032-0037] ; claim 6; figures 1,4A-4C.	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

30 AUGUST 2012 (30.08.2012)

Date of mailing of the international search report

31 AUGUST 2012 (31.08.2012)

Name and mailing address of the ISA/KR

Korean Intellectual Property Office
189 Cheongsu-ro, Seo-gu, Daejeon Metropolitan
City, 302-701, Republic of Korea

Facsimile No. 82-42-472-7140

Authorized officer

Jung Pil Seung

Telephone No. 82-42-481-5670



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT /US2011/067035

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2004-0030863 A1	12.02.2004	AU 2003-256870 A1 CN 100394380 CO CN 1688966 A CN 1688966 CO JP 2005-535966 A JP 2009-037599 A JP 201 1-108265 A US 2004-0030862 A1 US 2004-0034760 A1 US 2006-00 15702 A1 US 2006-0149939 A1 US 2007-0204132 A1 US 2008-0209 187 A1 US 2008-0270768 A1 US 2009-0300325 A1 US 6986023 B2 US 7047393 B2 US 7213128 B2 US 7356676 B2 US 7373488 B2 US 7392368 B2 US 7664930 B2 US 8 131981 B2 Wo 2004-015563 A1	25.02.2004 11.06.2008 26.10.2005 11.06.2008 24.11.2005 19.02.2009 02.06.201 1 12.02.2004 19.02.2004 19.01.2006 06.07.2006 30.08.2007 28.08.2008 30.10.2008 03.12.2009 10.01.2006 16.05.2006 01.05.2007 08.04.2008 13.05.2008 24.06.2008 16.02.2010 06.03.2012 19.02.2004
US 2003-0093648 A1	15.05.2003	CN 1610880 A CN 1610880 CO EP 1446717 A1 JP 04-098241 B2 JP 2005-528669 A KR 10-0981033 B1 KR 10-2005-0058240 A TW I313822A TW I313822B US 7228401 B2 Wo 2003-0428 19 A1	27.04.2005 13.12.2006 18.08.2004 11.06.2008 22.09.2005 10.09.2010 16.06.2005 21.08.2009 21.08.2009 05.06.2007 22.05.2003
US 2002-01 12 147 A1	15.08.2002	us 7155601 B2	26.12.2006