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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
3 January 2014 (03.01.2014)

WIPO | PCT

(10) International Publication Number
WO 2014/004397 A1

(51) International Patent Classification:
G06F 17/16 (2006.01) *G06F 12/00* (2006.01)

(21) International Application Number:
PCT/US2013/047393

(22) International Filing Date:
24 June 2013 (24.06.2013)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
13/538,523 29 June 2012 (29.06.2012) US

(71) Applicant: **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, MS: RNB-4-150, Santa Clara, California 95054 (US).

(72) Inventors: **GUERON, Shay**; 18A Adam HaCohen Street, 32714 Haifa (IL). **KRASNOV, Vlad**; Avraham Perez 2, 36680 Nesher (IL). **VALENTINE, Robert**; Rechov Hadganiot 33-5, 36054 Kiryat Tivon (IL). **SPERBER, Zeev**; 32nd Igal Alon Street, 30900 Zichron Yackov (IL). **GRADSTEIN, Amit**; 16th Hada Street, 30500 Binyamina (IL). **RUBANOVICH, Simon**; Egoz Street 10/4, 34792 Haifa (IL).

(74) Agents: **VINCENT, Lester J.** et al.; Blakely Sokoloff Taylor & Zafman, 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, BN, 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, 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, PA, 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.

[Continued on next page]

(54) Title: VECTOR MULTIPLICATION WITH ACCUMULATION IN LARGE REGISTER SPACE

(57) Abstract: An apparatus is described having an instruction execution pipeline that has a vector functional unit to support a vector multiply add instruction. The vector multiply add instruction to multiply respective K bit elements of two vectors and accumulate a portion of each of their respective products with another respective input operand in an X bit accumulator, where X is greater than K.

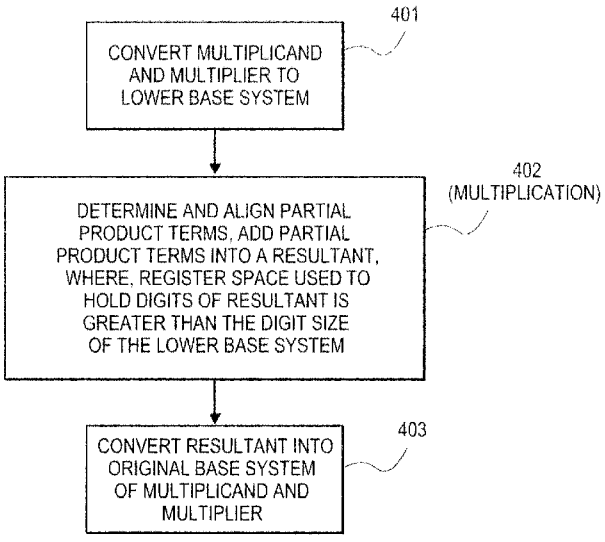


FIG. 4A

WO 2014/004397 A1

(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, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

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

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

VECTOR MULTIPLICATION WITH ACCUMULATION IN LARGE REGISTER SPACE

Background

Field of Invention

The present invention pertains to the computing sciences generally, and, more specifically to an apparatus and method for vector multiplication with accumulation in large register space.

Background

Fig. 1 shows a high level diagram of a processing core 100 implemented with logic circuitry on a semiconductor chip. The processing core includes a pipeline 101. The pipeline consists of multiple stages each designed to perform a specific step in the multi-step process needed to fully execute a program code instruction. These typically include at least: 1) instruction fetch and decode; 2) data fetch; 3) execution; 4) write-back. The execution stage performs a specific operation identified by an instruction that was fetched and decoded in prior stage(s) (e.g., in step 1) above) upon data identified by the same instruction and fetched in another prior stage (e.g., step 2) above). The data that is operated upon is typically fetched from (general purpose) register storage space 102. New data that is created at the completion of the operation is also typically "written back" to register storage space (e.g., at stage 4) above).

The logic circuitry associated with the execution stage is typically composed of multiple "execution units" or "functional units" 103_1 to 103_N that are each designed to perform its own unique subset of operations (e.g., a first functional unit performs integer math operations, a second functional unit performs floating point instructions, a third functional unit performs load/store operations from/to cache/memory, etc.). The collection of all operations performed by all the functional units corresponds to the "instruction set" supported by the processing core 100.

Two types of processor architectures are widely recognized in the field of computer science: "scalar" and "vector". A scalar processor is designed to execute instructions that perform operations on a single set of data, whereas, a vector processor is designed to execute instructions that perform operations on multiple sets of data. Figs. 2A and 2B present a comparative example that demonstrates the basic difference between a scalar processor and a vector processor.

Fig. 2A shows an example of a scalar AND instruction in which a single operand set, A and B, are ANDed together to produce a singular (or "scalar") result C (i.e., AB=C). By contrast, Fig. 2B shows an example of a vector AND instruction in which two operand sets, A/B and D/E, are respectively ANDed together in parallel to simultaneously produce a vector result

C, F (i.e., A.AND.B=C and D.AND.E=F). As a matter of terminology, a "vector" is a data element having multiple "elements". For example, a vector V = Q, R, S, T, U has five different elements: Q, R, S, T and U. The "size" of the exemplary vector V is five (because it has five elements).

Fig. 1 also shows the presence of vector register space 104 that is different that general purpose register space 102. Specifically, general purpose register space 102 is nominally used to store scalar values. As such, when, the any of execution units perform scalar operations they nominally use operands called from (and write results back to) general purpose register storage space 102. By contrast, when any of the execution units perform vector operations they nominally use operands called from (and write results back to) vector register space 107. Different regions of memory may likewise be allocated for the storage of scalar values and vector values.

Note also the presence of masking logic 104_1 to 104_N and 105_1 to 105_N at the respective inputs to and outputs from the functional units 103_1 to 103_N. In various implementations, only one of these layers is actually implemented – although that is not a strict requirement. For any instruction that employs masking, input masking logic 104_1 to 104_N and/or output masking logic 105_1 to 105_N may be used to control which elements are effectively operated on for the vector instruction. Here, a mask vector is read from a mask register space 106 (e.g., along with input data vectors read from vector register storage space 107) and is presented to at least one of the masking logic 104, 105 layers.

Over the course of executing vector program code each vector instruction need not require a full data word. For example, the input vectors for some instructions may only be 8 elements, the input vectors for other instructions may be 16 elements, the input vectors for other instructions may be 32 elements, etc. Masking layers 104/105 are therefore used to identify a set of elements of a full vector data word that apply for a particular instruction so as to effect different vector sizes across instructions. Typically, for each vector instruction, a specific mask pattern kept in mask register space 106 is called out by the instruction, fetched from mask register space and provided to either or both of the mask layers 104/105 to "enable" the correct set of elements for the particular vector operation.

Fig. 3 shows a standard "schoolbook" multiplication process within a base 10 system. As observed in Fig. 3, each digit in a multiplicand 301 is multiplied by each digit in a multiplier 302 to create an array of partial products 303. Each partial product is aligned with the location of its respective multiplier digit. The aligned partial product terms are added together to produce multiplication result 304.

Note the presence of the carry terms 305. Carry terms 305_1 through 305_5 can be created not only when the partial product terms are added to produce the final result, but also, as part of the determination of each partial product term itself. For example, carry term 305_1 is created during the summation of the partial products, while, each of carry terms 305_2 through 305_4 is generated in determining a particular partial product.

In order to perform multiplication operations a processing core embedded on a semiconductor chip essentially performs mathematical operations that are similar to the multiplication processes discussed above. Specifically, partial product terms are generated, and, the partial product terms are added to produce a final result. In the case of vector instructions, however, carry terms can present problems.

For example, any "special logic circuitry" needed to recognize and account for any generated carry terms can become substantial in size as such logic circuitry would be needed for every element of the maximum vector size supported by the processor. Non-vector "integer" execution logic of a processor may be designed to use special "flags" and corresponding flag circuitry to handle carry terms. However, as integer operations are essentially scalar operations, only one instance of such circuitry needs to be implemented.

As such, a common processor design point for a processor that supports both integer and vector instructions is to design special flag circuitry for the integer instructions but not the vector instructions (or at least a limited version of flag circuitry for the vector instructions). Without the flag circuitry and its corresponding support for carry terms, the designers of a processor's vector instruction execution logic face the challenge of accounting for carry terms in their vector multiplication instruction execution logic by some other technique.

Brief Description of the Drawings:

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:

Fig. 1 (Prior Art) shows an instruction processing pipeline;

Figs. (Prior Art) 2a and 2b pertain to vector processing;

Fig. 3 (Prior Art) shows an example of school book multiplication;

Fig. 4a shows a process for accounting for carry terms by accumulating summed values in register space that is larger than multiplier and multiplicand digit size;

Fig. 4b shows an example of the process of Fig. 4a;

Fig. 4c shows an example of school book multiplication for comparison against Fig. 4b;

Fig. 4d shows a sequence of instructions that can perform the exemplary method of Fig. 4b;

Fig. 4e shows an exemplary process for converting the base system of the resultant multiplication into the original base system of the multiplicand and multiplier;

Fig. 5a shows an embodiment of execution unit logic circuitry for a VMULTADDLO instruction;

Fig. 5b shows an embodiment of execution unit logic circuitry for a VMULTADDHI instruction;

Fig. 5c shows reuse of the design of an integer multiplier for the VMULTADDLO and VMULTADDHI instructions;

Figure 6A illustrates an exemplary AVX instruction format;

Figure 6B illustrates which fields from Figure 6A make up a full opcode field and a base operation field;

Figure 6C illustrates which fields from Figure 6A make up a register index field;

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

Figure 8A-8D are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention;

Figure 9 is a block diagram of a register architecture according to one embodiment of the invention;

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;

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;

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;

Figure 13 is a block diagram of a exemplary system in accordance with an embodiment of the present invention;

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

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

Figure 16 is a block diagram of a SoC in accordance with an embodiment of the present invention;

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

The present description discloses a technique for performing vector multiplication by accumulating summed partial product terms in registers having a width larger than the digits expressed in the partial product terms. Because summations are written to larger registers, any summation result that exceeds the digit size, which in traditional implementations produce a "carry term" that would need to be handled with special carry logic, naturally expands into the additional register space. As such, special carry logic such as flag logic and flag handling logic normally used for integer operations need not be implemented for vector multiplication operations. The original multiplicand and multiplier operands of the vector multiplication technique may be expressed in a base system having a digit size that can fully consume the register size. As such, the multiplication operation may be preceded by a conversion process that converts the input operands from their original higher base system to a lower base system characterized by smaller digits. The result of the multiplication may subsequently be converted back to the original base system.

Figs. 4a depicts a process for performing multiplication that accounts for carry terms by forcing summations of partial product terms to be accumulated in a register size that is larger than the maximum number of digits that can result in the summation sequence. Because the register size is larger than the size of the summation result there is "room" to store any carry over from the addition within the register.

Referring to Fig. 4a, a conversion process may be executed 401 prior to the multiplication to effectively convert the multiplicand and multiplier into a lower base system so as to create digits that are smaller in size than the registers that will store the summations determined from them.

Partial product terms are then determined, their respective digits aligned and added into a resultant, where, the register size holding each digit in the resultant is larger than the maximum digit size that is possible given the respective sizes of the multiplicand and multiplier 402. As the size of the digits in the resultant may expand beyond the size of the digits that emanated from

the initial conversion 401, the resultant at the conclusion of the summation of partial products may be effectively expressed in a different base system than the specific base system that the multiplier and multiplier were converted into by the conversion process 401.

More specifically, the resultant may be expressed with a base value that is between the respective base values of the original and converted forms of the multiplier and multiplicand. For example, if the multiplier and multiplicand are initially expressed as radix 64 (i.e., 264, or, 64 bit digits) and the conversion process 401 converts the multiplier and multiplicand into radix 52 (i.e., 252, or, 52 bit digits), the resultant of multiplication 402 may be expressed radix m (i.e., 2m, or, m bit digits) where $64 > m > 52$.

As such, another conversion process 403 is performed to convert the multiplication resultant into the base system that the multiplicand and multiplier were originally expressed in prior to the initial conversion process 401.

Fig. 4b shows an example of the process discussed just above with respect to Fig. 4a. The specific example of Fig. 4b is further supported by Figs. 4c and 4d which will be discussed in more detail below. The specific example of Figs. 4b through 4d are directed to a system where the multiplicand and multiplier are initially expressed in radix 64 form and are converted into radix 52 form in the initial conversion 401. It should be readily apparent to those of ordinary skill that the teachings herein extend to any base system.

Referring to Fig. 4b, the multiplicand 404_1 and multiplier 405_1 may each be initially represented with a respective vector where each element of the vector corresponds to a different 64 bit digit of the multiplier or multiplicand.

A conversion process 406 may then convert, again as an example, the multiplier and multiplicand such that they are each represented with 52 bit digits 404_2 and 405_2. In so doing, the number of digits (i.e., the vector size) of either or both of the multiplier and multiplicand may increase as part of the conversion process (even through the numerical value is unchanged on either side of the conversion process). For example, as observed in Fig. 4b, the initial multiplicand 404_1 is expressed as a three element vector and the initial multiplier 405_1 is expressed as a two element vector. The conversion process 406 is observed to convert the multiplicand 404_1 into a four element vector 404_2 and the multiplier 405_1 into a three element vector 405_2.

Here, note that each digit of the converted operands 404_2, 405_2 depicts a left-wise field of 0s (e.g., left-wise field 407). This representational feature is meant to depict the conversion from 64 bit digits in the original operands 404_1, 405_1 to 52 bit digits in the converted operands 404_2, 405_2. Notably, the physical hardware used to contain the converted operands

404_2, 405_2 is still "wide enough" to hold 64 bit digits. As such, each converted 52 bit digit is left-wise appended with a field of 12 zeros. Said another way, each element of the converted vectors 404_2, 405_2 is a 64 bit element which contains a 52 bit digit appended by 12 packed zeros on its left side.

The multiplicand and multiplier 404_2, 405_2 as represented in their new "52 bit digit format" are then multiplied 408. In the example of Fig. 4b, because the multiplication of two 52 bit digits can produce a 104 bit result, and, only 64 bit vector element sizes are supported by the underlying hardware, two different vector instruction types (VPMULADDLO and VPMULADDHI) are used to separately provide the "lower ordered" 52 bits of the partial product terms and the "higher ordered" 52 bits of the partial product terms.

Here, referring to Fig. 4c, Fig. 4c shows standard schoolbook form of the partial products of the converted multiplicand 404_2 and multiplier 405_2. Note that the partial product terms respect the fact that the multiplication of two 52 bit digits can result in a 104 bit digit. As such, for example, the partial product of the a'0 and b'0 digits 420 consumes two vector element locations 420_1, 420_2, (a "HI" and a "LO") each location supporting 52 bits with 12 left-wise packed zeros.

Accordingly, a first type of instruction (VPMULADDLO) is used to determine the lower of the vector elements 420_1 of the partial product term 420, and, a second type of instruction (VPMULADDHI) is used to determine the higher of the vector elements 420_2 of the partial products. Essentially, VPMULADDLO returns the lower 52 bits of the 104 bit resultant of a'0 X b'0, and, VPMULADDHI returns the higher 52 bits of the 104 bit resultant of a'0 x b'0. Other embodiments may be designed where the upper and lower portions of the multiplication that are calculated or accumulated by the instructions are something other than an upper half and a lower half.

Returning to Fig. 4b, note that the individual digits of the partial product terms have been rearranged to take advantage of vector operation of the VPMULADD instructions and "pack" the operands so as to consume less total instructions. Nevertheless, the summation through a specific digit (vector element) location is correct when compared against the schoolbook format of Fig. 4c. For example, summation 421_1 of Fig. 4b sums the same partial product digits as summation 421_2 of Fig. 4c.

In order to take advantage of the vector operation of the VPMULADD instructions, note that a broadcast instruction may be executed prior to execution of the VPMULADD instructions in order to create one of its operands. Fig. 4d shows an instruction level representation of the exemplary multiplication of Fig. 4b. Here, the conversion 406 of the 64 bit digit multiplicand

and multiplier operands 404_1, 405_1 into 52 bit digit operands 404_2, 405_2 is performed with instruction(s) 430. As the mathematical execution for converting digits from one base system to another is readily achievable to those of ordinary skill, an example of the specific instruction(s) need not be provided in the present discussion.

After conversion, the newly formatted 52 bit digit operands 404_2, 405_2 are stored in vector register R1 (which stores the multiplicand 501_2) and vector register R2 (which stores the multiplier 502_2). A vector whose size is at least equal to the maximum size of the multiplication result and whose elements are each zero is also created and stored in R3. Here, an iteration counter i is set to i = 0 as an initial condition. A first broadcast instruction (VBROADCAST) is then executed 431 which extracts the lowest ordered element of the multiplier in R2 (i.e., b'0) and replicates it across the vector size of the multiplicand 404_2. In this case, multiplicand 404_2 has four elements. As such, the first VBROADCAST instruction 431 returns in R4 a vector having four copies of b'0 as its four lowest ordered vector element locations.

A VPMULADDLO instruction 432 and a VPMULADDHI instruction 434 are each executed subsequently where each instruction accepts the contents of R1, R3 and R4 as its input operands. In the particular embodiment of Fig. 4d, the VPMULADDLO and VPMULADDHI instructions are "multiply accumulate" instructions. As such, the instructions not only perform vector multiplication, but also, vector addition. Fig. 5a shows an embodiment of the logic design of a VPMULADDLO execution unit and Fig. 5b shows an embodiment of the logic design of a VPMULADDHI execution unit.

Each execution unit includes an array of multipliers and adders, where, each individual multiplier and adder within the array is capable of operating from same positioned elements of two input vectors but each multiplier and adder operates from a different vector element position. For simplicity Figs. 5a and 5b only show a multiplier 501 and corresponding adder 502 of one vector element position.

As observed in Fig. 5a, the size of the register space used to hold each input operand presented to the multiplier 501 is X bits (e.g., 64 bits). However, a maximum of K (e.g., 52 bits) bits of these are used by the multiplier in performing the multiplication where K < X. Notably the register space used to hold each input operand can be used to hold elements of other vectors of vector operations other than the vector operations presently being described, where, the maximum width of the register space X (e.g., 64 bits) can be utilized for input operand data. These other vector operations may be performed with execution logic circuitry other than then execution logic circuitry of Fig. 5a and Fig. 5b. The execution logic circuitry used to perform

the other vector operations may be implemented within execution units of the pipeline other than the execution unit(s) that the execution logic circuitry of Fig. 5a and Fig. 5b is implemented within. As such, the other execution logic/execution unit support vector operations having input operands expressed in a base system that is higher than the base system of the input operands utilized by the logic circuitry of Figs. 5a and 5b.

The maximum size of the substantive resultant of the multiplication is L = 2Kbits. The execution logic of the VPMULADDLO instruction, as observed in Fig. 5a, for a particular vector element location, extracts a lower Q (e.g., 52) bits of the multiplication result and feeds these bits into one input of an adder. A third input operand is respectively provided to a second input of the adder. Here, the third input vector operand corresponds to a respective element of a third input vector operand.

In the particular embodiment of Figs. 5a and 5b, the resultant of the addition operation performed by the adder 502 is stored "back" in the same register that provided the third (additive) input operand. As such, in this particular embodiment, the VPMULADDLO and VPMULADDHI instructions have an instruction format that supports definition of both an input operand "source" register and resultant "destination" register as the same register.

The execution of the VPMULADDHI instruction is similar to that of the VPMULADDLO instruction except that the higher Q bits of the multiplication resultant are fed to the multiplier's respective adder.

Referring back to Fig. 4d, the initial execution of the VPMULADDLO instruction 432 provides, in R3, the lower 52 bits of the multiplication of b'0 with a'3 through a'0 as kept in R4.

As such, referring to Figs. 4b, the set of partial product terms 440 can be viewed as any of: i) the output of the respective multipliers of the VPMULADDLO instruction 432; or, ii) the contents of R3 after the execution of the initial VPMULADDLO instruction 432 is complete. The contents of R3 are formally represented in data structure 441 of Fig. 4b. As will be more clear in the following discussion, R3 behaves as an accumulator that collects partial sums of the partial product terms over the course of the multiplication sequence.

Referring to Fig, 4d, the contents of R4 are then shifted 433 to the left by one vector element location to setup correct alignment of the input operands for the VPMULADDHI instruction. The VPMULADDHI instruction is then executed 434. Data structure 442 of Fig. 4b shows the results of the output of the multipliers during the course of the execution of the initial VPMULADDHI instruction 434.

The addition operation of the VPMULADDHI instruction 434 adds the contents of R3 (i.e., the resultant of the prior VPMULADDLO instruction 432) and stores the result of the addition

"back into" R3. As such, referring to data structure 441 of Fig. 4b, R3 now holds: i) partial product term 443 in the lowest ordered element 444; ii) the summation of partial product terms 445 and 446 in the second to lowest ordered element 447; iii) the summation of partial product terms 448 and 449 in element 450; iv) the summation of partial product terms 451 and 452 in element 453.

The instruction pattern of instructions 431 through 434 of Fig. 4d is then repeated for each NEXT i until the multiplication complete (after the i = 3 iteration is complete). With each completion of each iteration (i.e., the completion of each VPMULADDHI instruction) summed aligned partial products are accumulated in R3.

Notably, over the course of the summations and corresponding accumulation within R3, the size of the digits in any of the elements of R3 may have exceeded 52 bits. As R3 is implemented in this example with hardware that supports vectors having element sizes of 64 bits, there is sufficient room in each of the elements of R3 to accommodate the expansion of the digit size.

Lastly, as the digit size of the accumulated values in R3 may have expanded at the completion of the multiplication to a value greater than 52, the base system represented in R3 after multiplication is complete may no longer be radix 52. As such, a conversion 435 from the resulting base system of R3 to the original radix 64 system is performed.

Fig. 4e shows an example of a process flow for converting the resultant of the multiplication into the original base system of the multiplicand and multiplier. According to the process of Fig. 4e, assume that the original base system of the multiplicand and multiplier was a radix M system (i.e., 2M, or, M digits). In the example discussed above with respect to Figs. 4b and 4d, M = 64. According to the process of Fig. 4e, the maximum digit size of the digits in the multiplication result is identified 460. In an embodiment, this is performed by identifying the bit location of the most significant "1" of all the digits in the multiplication result. As observed in Fig. 4, the maximum digit size is kept as a variable K. Thus, for example, if the most significant bit amongst the digits in the multiplication result of the example of Figs. 4b and 4d was located in the 55th bit position, K = 55.

A variable TEMP is set to a value 0 as an initial condition 461. The value of TEMP is added 462 to the value of the next lowest ordered digit in the multiplication result (which, for the initial iteration corresponds to the lowest ordered digit in the multiplication result). The value of TEMP is then divided by 2M and the remainder is kept in a variable V 463. The value of V is retained/recognized as the next least ordered digit in the original (2M) base system 464 (which, again, for the initial iteration corresponds to the lowest ordered digit in the multiplication result).

The value of TEMP is then recalculated as TEMP/(2M) 465 and the process iteratively repeats for each next digit in the multiplication result until each digit in the multiplication result has been processed.

Referring to Figs. 5a and 5b, it is pertinent to point out that any of the depicted SourceDest, Source1 or Source 2 registers may be: i) registers in the vector register space of the processing core; ii) registers within an operand fetch stage of the instruction pipeline that preset operands to the execution unit; or, iii) registers at the "front end" of the execution unit (e.g., that receive input operands from an instruction fetch stage of the instruction execution pipeline).

Fig. 5c shows that the design of multiplier 501 of the execution units of Figs. 5a and 5b may be substantially the same if not identical to the design of a multiplier within an integer(as opposed to vector) execution unit of an instruction execution pipeline 550 of a processing core. Here, as is known in the art, integer processing can be performed in a floating point format. According to one common approach, the mantissa of the floating point format is 53 bits. As such, in order to support integer floating point multiplication operations, there exists in the instruction execution pipeline 550 an integer execution unit 551 that is coupled to or otherwise receives operands from integer register space 552 and contains a 53 bit by 53 bit multiplier 553.

In an embodiment, the same (or substantially the same) design for the integer multiplier 553 is "ported over" and "reused" within the execution unit(s) 554, 555 that support the improved vector multiplication discussed at length above. As such, multiple instances of the same/substantially the same multiplier design is not only effectively coupled to the integer register space 552 of the instruction execution pipeline 550 but also the vector register space of the 556 of the instruction execution pipeline 550. In particular, note that the size of the digits supported by the integer multiplier may be greater than or equal to the number of bits that corresponds to the lower base system that digits of the vector multiplication's multiplicand and multiplier are converted down to from their original base system expression.

The enclosed techniques and methods are expected to be particularly useful when embedded in cryptographic processes including public key cryptographic processes such as RSA, DSA, GF(p), GF(p*q), GF(n)), ECC over GF(p) or DH key exchange processes.

The instruction format of the VPMULADDHI and VPMULADDLO instructions can be implemented in various ways. Essentially, each of the VPMULADDHI and VPMULADDLO instructions can be viewed as vector instructions that multiply K bit elements but accumulate the resultant products of the K bit elements in X bit elements, where X > K. In various embodiments X and K may be specified in the instruction format. For example, X and K (and whether the HI or LO product portions is to be accumulated) may be effectively specified in any

opcode information of the instruction format and/or, any immediate operand information of the instruction format. The following discussion pertains to some specific vector instruction format embodiments. Here, X and K (and whether a HI or LO portion is accumulated) may be effectively encoded into any suitable field of information discussed below including but not limited to any opcode and/or immediate operand information.

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.

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 (1s complement) form and is valid for instructions with 2 or more source operands; 2) VEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. If VEX.L 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.

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

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.

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.

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

Generic Vector Friendly Instruction Format

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

Figures 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.

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).

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.

The generic vector friendly instruction format 700 includes the following fields listed below in the order illustrated in Figures 7A-7B. In conjunction with the discussions above of Figs. 4a,b,c,d and 5,a,b,c in an embodiment, referring to the format details provided below in Figures 7A-B and 8, either a non memory access instruction type 705 or a memory access instruction type 720 may be utilized. Addresses for the read mask(s), input vector operand(s) and destination may be identified in register address field 744 described below. In a further embodiment, the write mask is specified in write mask field 770.

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.

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

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).

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.

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.

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 2scale * index + base).

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

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 2scale * 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.

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.

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.

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.

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

Instruction Templates of Class A

In the case of the non-memory access 705 instruction templates of class A, the alpha field 752 is interpreted as an RS field 752A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 752A.1 and data transform 752A.2 are respectively specified for the no memory access, round type operation 710 and the no

memory access, data transform type operation 715 instruction templates), while the beta field 754 distinguishes which of the operations of the specified type is to be performed. In the no memory access 705 instruction templates, the scale field 760, the displacement field 762A, and the displacement scale filed 762B are not present.

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

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).

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.

Round operation control field 758 – its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 758 allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 750 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

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).

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.

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

Memory Access Instruction Templates – Temporal

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

Memory Access Instruction Templates – Non-Temporal

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

Instruction Templates of Class B

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.

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.

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).

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.

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).

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.

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).

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.

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.

The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may

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

Exemplary Specific Vector Friendly Instruction Format

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.

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).

The generic vector friendly instruction format 700 includes the following fields listed

below in the order illustrated in Figure 8A.

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

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).

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

REX field 805 (EVEX Byte 1, bits [7-5]) – consists of a EVEX.R bit field (EVEX Byte 1, bit [7] – R), EVEX.X bit field (EVEX byte 1, bit [6] – X), and 757BEX byte 1, bit[5] – B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 1111B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

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.

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

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).

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.

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.

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.

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.

Beta field 754 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s2-0, EVEX.r2-0, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with βββ) – as previously described, this field is context specific.

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.

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).

Real Opcode Field 830 (Byte 4) is also known as the opcode byte. Part of the opcode is

specified in this field.

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.

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.

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.

Displacement factor field 762B (Byte 7) – when MOD field 842 contains 01, byte 7 is the displacement factor field 762B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since disp8 is sign extended, it can only address between -128 and 127 bytes offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values -128, -64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field 762B is a reinterpretation of disp8; when using displacement factor field 762B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is based on the assumption that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field 762B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 762B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

Immediate field 772 operates as previously described.

Full Opcode Field

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

Register Index Field

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

Augmentation Operation Field

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.

When U=1, the alpha field 752 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 752C. When U=1 and the MOD field 842 contains 11 (signifying a no memory access operation), part of the beta field 754 (EVEX byte 3, bit [4]- S0) 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]- S2-1) is interpreted as the round operation field 759A, while when the RL field 757A contains a 0 (VSIZE 757.A2) the rest of the beta field 754 (EVEX byte 3, bit [6-5]- S2-1)

is interpreted as the vector length field 759B (EVEX byte 3, bit [6-5]- L1-0). When U=1 and the MOD field 842 contains 00, 01, or 10 (signifying a memory access operation), the beta field 754 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 759B (EVEX byte 3, bit [6-5]- L1-0) and the broadcast field 757B (EVEX byte 3, bit [4]- B).

Exemplary Register Architecture

Figure 9 is a block diagram of a register architecture 900 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 910 that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format 800 operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 759B	A (Figure 7A; U=0)	710, 715, 725, 730	zmm registers (the vector length is 64 byte)
	B (Figure 7B; U=1)	712	zmm registers (the vector length is 64 byte)
Instruction Templates that do include the vector length field 759B	B (Figure 7B; U=1)	717, 727	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 759B

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.

Write mask registers 915 - in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers 915 are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

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.

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.

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

Exemplary Core Architectures, Processors, and Computer Architectures

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

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

Exemplary Core Architectures

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

Figure 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.

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.

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.

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.

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.

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.

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.

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=1) previously described), thereby allowing the operations used by many multimedia applications to be performed using packed data.

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

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units 1034/1074 and a shared L2 cache unit 1076, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

Specific Exemplary In-Order Core Architecture

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.

Figure 11A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 1102 and with its local subset of the Level 2 (L2) cache 1104, according to embodiments of the invention. In one embodiment, an instruction decoder 1100 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 1106 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 1108 and a vector unit 1110 use separate register sets (respectively, scalar registers 1112 and vector registers 1114) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 1106, alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

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.

Figure 11B is an expanded view of part of the processor core in Figure 11A according to embodiments of the invention. Figure 11B includes an L1 data cache 1106A part of the L1 cache 1104, as well as more detail regarding the vector unit 1110 and the vector registers 1114. Specifically, the vector unit 1110 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 1128), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit 1120, numeric conversion with numeric convert units 1122A-B, and replication with replication unit 1124 on the memory input. Write mask registers 1126 allow predicating resulting vector writes.

Processor with integrated memory controller and graphics

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.

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.

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.

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.

The cores 1202A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 1202A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

Exemplary Computer Architectures

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

[0001] 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.

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.

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 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.

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

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

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

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

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

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

processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

Figure 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

1. An apparatus, comprising:

an instruction execution pipeline having a vector functional unit to support a vector multiply add instruction, said vector multiply add instruction to multiply respective K bit elements of two vectors and accumulate a portion of each of their respective products with another respective input operand in an X bit accumulator, where X is greater than K.

2. The apparatus of claim 1 where X and K are specified in the instruction format of the vector multiply instruction.

3. The apparatus of claim 1 wherein said vector functional unit includes respective multiplier instances to said multiply said respective K bit elements, each of said multiplier instances being substantially the same as an integer floating point multiplier within another execution unit of said pipeline.

4. The apparatus of claim 1 wherein said respective input operand is provided by said X bit accumulator.

5. The apparatus of claim 1 where K = 52 and X = 64.

6. The apparatus of claim 1 wherein X is a nominal bit width of vector elements processed by said pipeline.

7. The apparatus of claim 6 wherein said execution pipeline is coupled to vector register space that provides for vectors composed of X bit elements.

8. The apparatus of claim 7 wherein said accumulator is implemented with one of said registers.

9. The apparatus of claim 1 wherein said portion is a half.

10. The apparatus of claim 1 wherein said instruction execution pipeline supports a second vector multiply add instruction, said second vector multiply add instruction to multiply respective K bit elements of two vectors and accumulate a different portion than said portion of

each of their respective products with another respective input operand in an X bit accumulator, where X is greater than K.

11. A method, comprising:

accepting a multiplicand operand and a multiplier operand from first and second registers, said multiplier operand and said multiplicand operand being elements of a first vector and a second vector respectively and submitted for a vector processing instruction, said registers being wider than a maximum permissible size of said multiplicand operand and said multiplier operand;

multiplying said multiplicand operand and said multiplier operand to produce a multiplication result;

using said first and second registers to provide operands for another vector processing instruction where at least one of said operands is wider than said maximum permissible size.

12. The method of claim 11 wherein said another operand is a result of a previous multiply and accumulate operation.

13. The method of claim 11 wherein said method further comprises accumulating a portion of said multiplication result with another operand.

14. The method of claim 13 wherein said accumulating produces a digit size larger than said maximum permissible size.

15. The method of claim 13 wherein said portion is an upper half of said multiplication result.

16. The method of claim 13 wherein said portion is a lower half of said multiplication result.

17. A computing system, comprising:

a processing core comprising an instruction execution pipeline, said instruction execution pipeline including:

i) a first vector execution unit, said first vector execution unit to perform vector multiplication, said first vector execution unit coupled to receive vector operands, said vector execution unit having a multiplier that provides a maximum number of significant digits in

performing said vector multiplication for a particular vector element location, wherein one half of said maximum number of significant digits is less than an amount of register space provided for elements of vectors processed by said instruction execution pipeline;

ii) a first integer execution unit, said first integer execution unit having a second multiplier having at least a substantially same design as said multiplier.

18. The computing system of claim 17 wherein said first vector execution unit further comprises logic to select a lower half of said maximum number of significant digits.

19. The computing system of claim 17 wherein said first vector execution unit further comprises logic to select a higher half of said maximum number of significant digits.

20. The computing system of claim 17 wherein said first vector execution unit further comprises an adder to add an output of said multiplier with an operand provided to said first vector execution unit other than said multiplicand and multiplier of said vector multiplication.

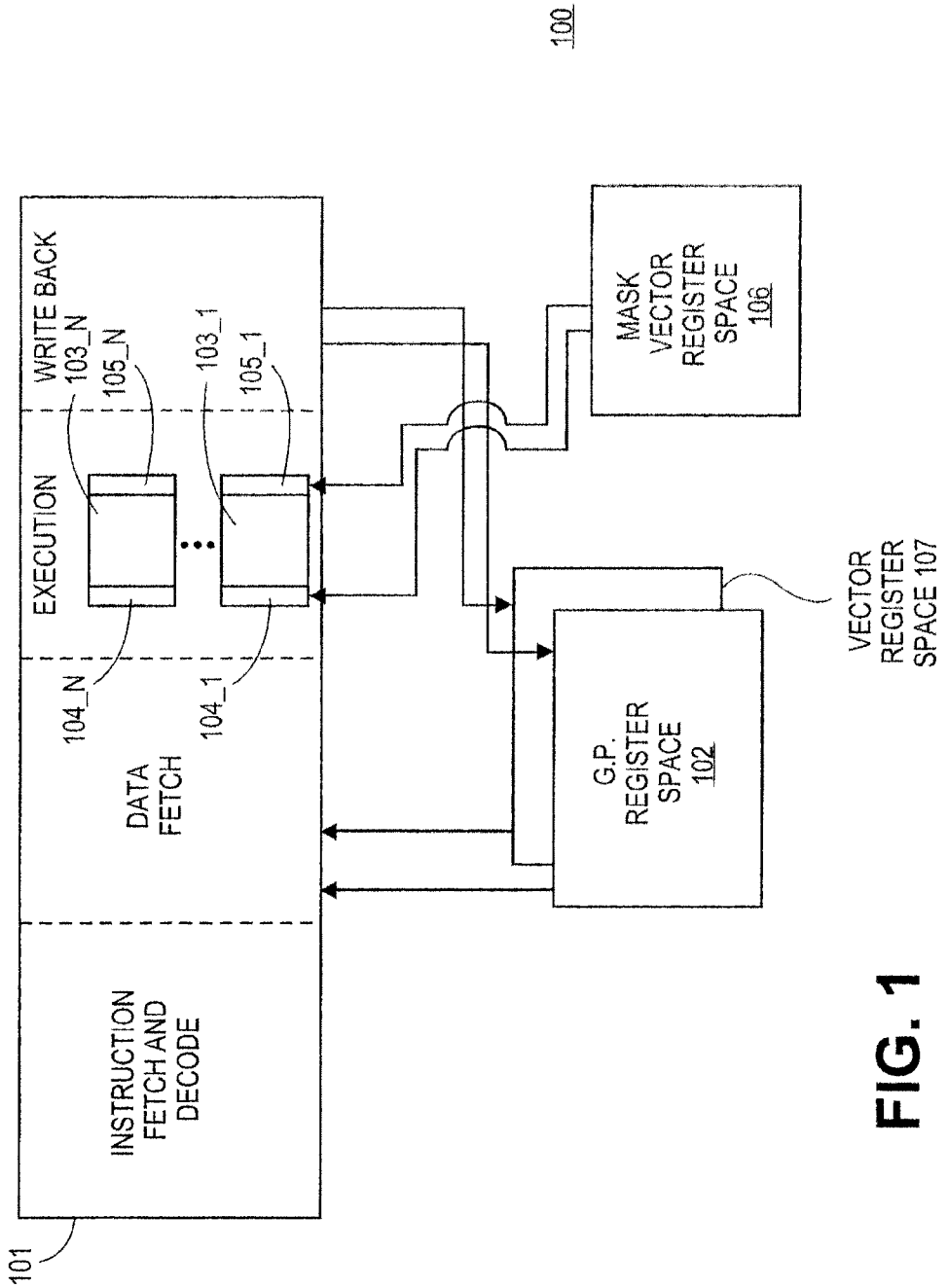

100
WRITE BACK
EXECUTION
DATA FETCH
INSTRUCTION FETCH AND DECODE
101
103_N
105_N
103_1
105_1
104_N
104_1
MASK VECTOR REGISTER SPACE 106
G.P. REGISTER SPACE 102
VECTOR REGISTER SPACE 107

FIG. 1

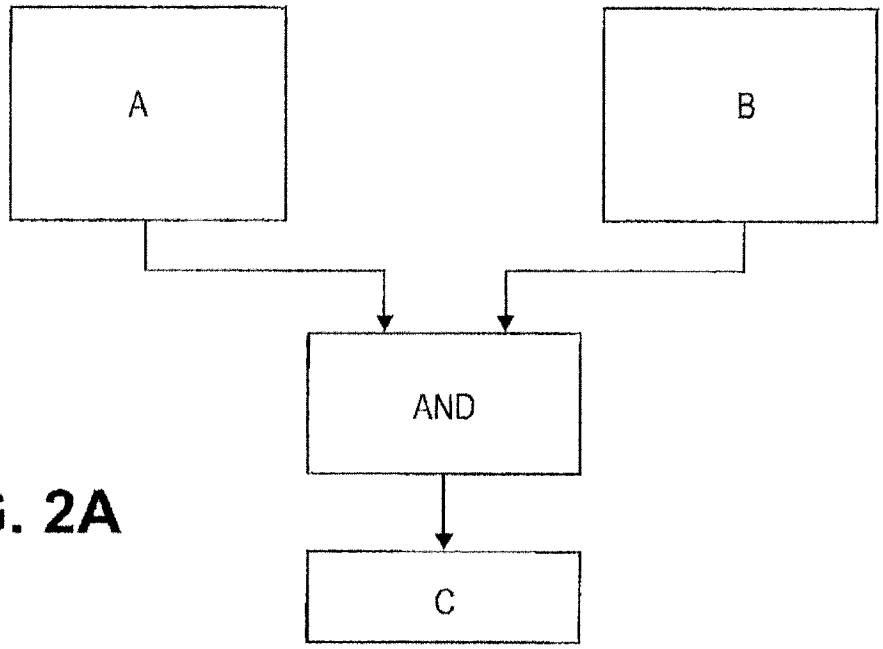


FIG. 2A

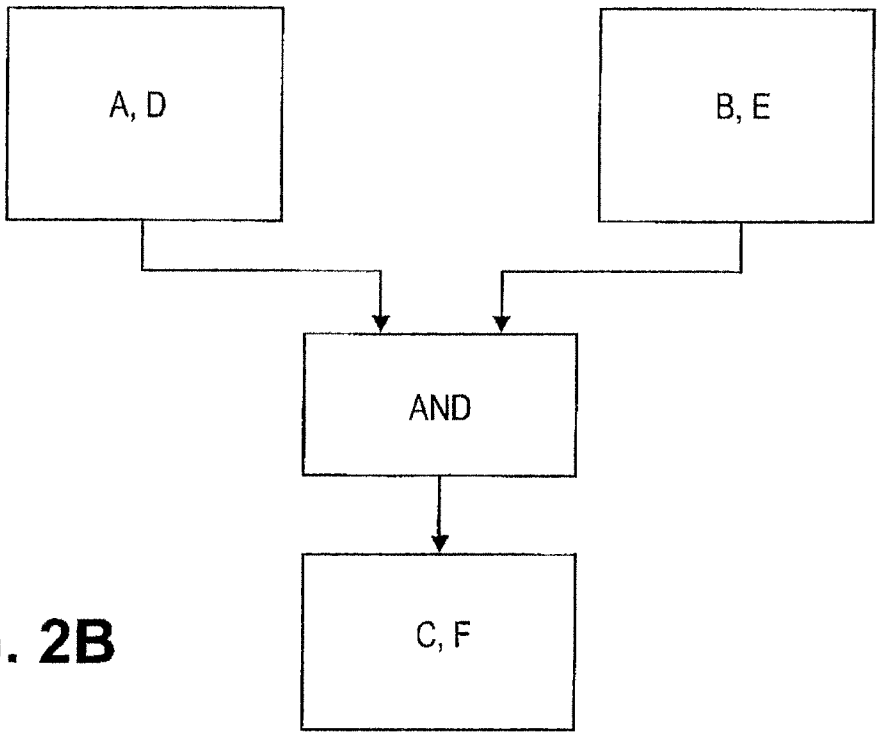


FIG. 2B

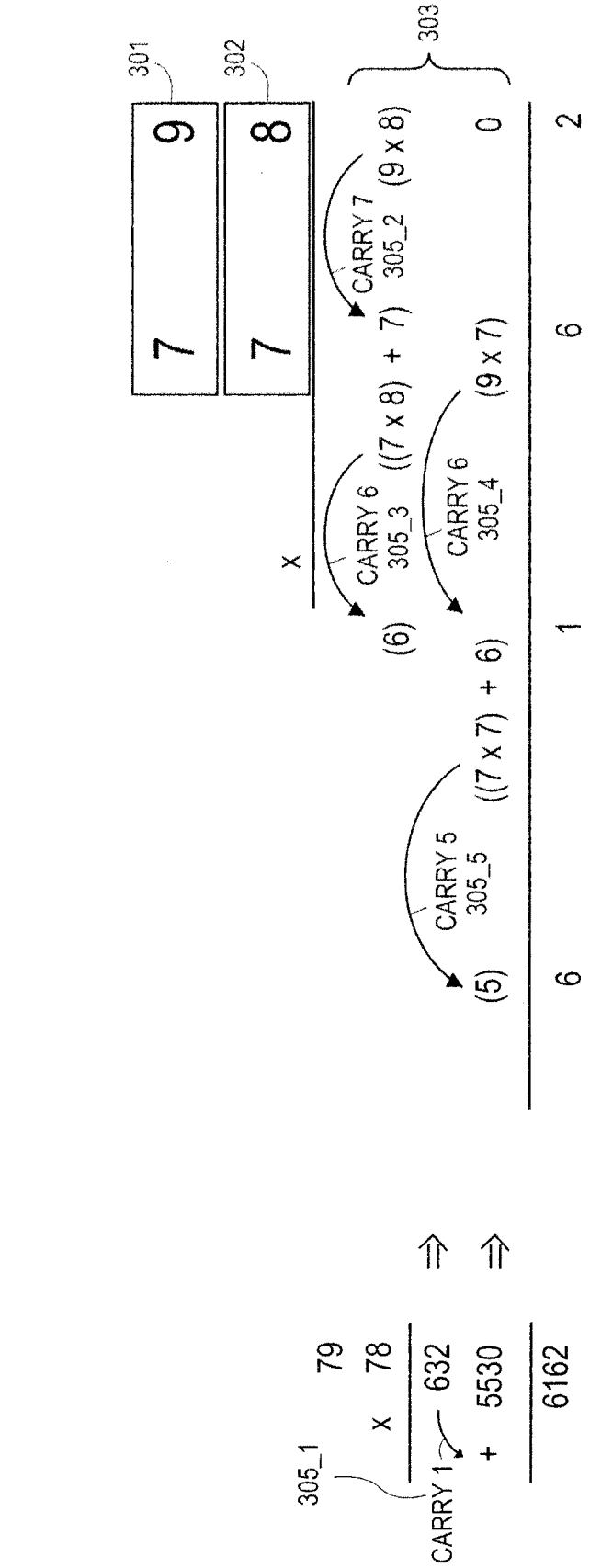

301
302
303
9
8
7
7
x
CARRY 7
305_2
(9 x 8)
0
CARRY 6
305_3
((7 x 8) + 7)
CARRY 6
305_4
(9 x 7)
(6)
(6)
1
6
2
CARRY 5
305_5
((7 x 7) + 6)
(5)
6
305_1
CARRY 1
79
x 78
632
+ 5530
6162

FIG. 3

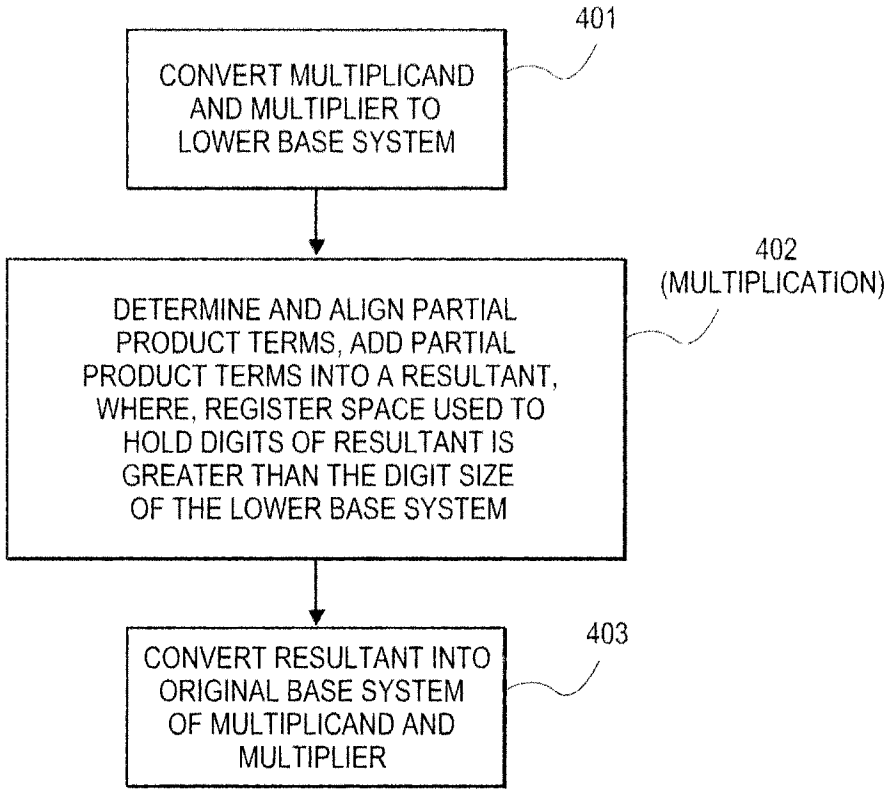

401
CONVERT MULTIPLICAND AND MULTIPLIER TO LOWER BASE SYSTEM
402
(MULTIPLICATION)
DETERMINE AND ALIGN PARTIAL PRODUCT TERMS, ADD PARTIAL PRODUCT TERMS INTO A RESULTANT, WHERE, REGISTER SPACE USED TO HOLD DIGITS OF RESULTANT IS GREATER THAN THE DIGIT SIZE OF THE LOWER BASE SYSTEM
403
CONVERT RESULTANT INTO ORIGINAL BASE SYSTEM OF MULTIPLICAND AND MULTIPLIER

FIG. 4A

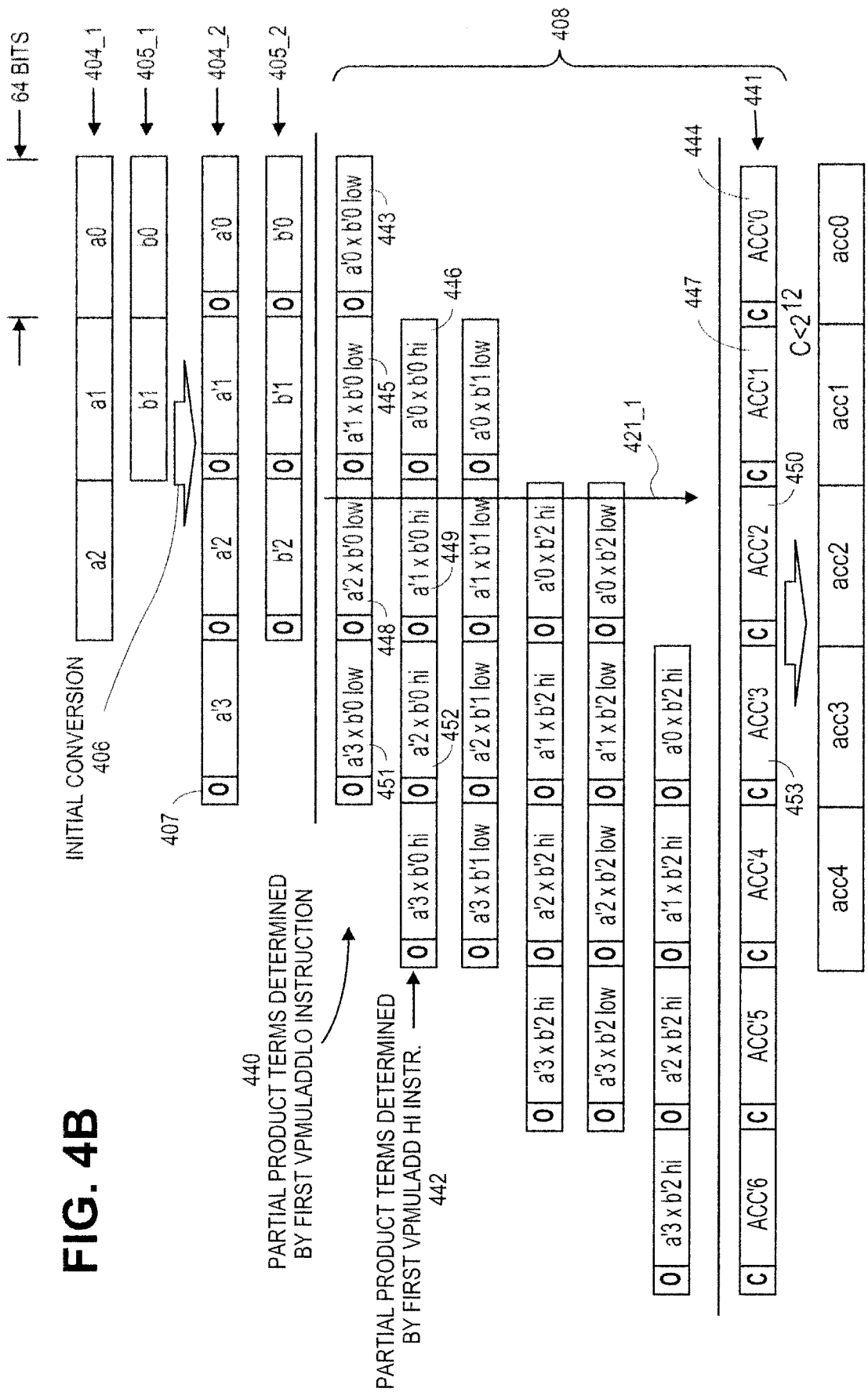

64 BITS
404_1
405_1
404_2
405_2
a2
a1
a0
b1
b0
a'2
a'1
a'0
b'2
b'1
b'0
INITIAL CONVERSION
406
407
408
440
PARTIAL PRODUCT TERMS DETERMINED BY FIRST VPMULADDLO INSTRUCTION
PARTIAL PRODUCT TERMS DETERMINED BY FIRST VPMULADD HI INSTR.
442
421_1
441
444
445
446
447
448
449
450
451
452
453
C<2^12
a'3 x b'0 low
a'2 x b'0 low
a'1 x b'0 low
a'0 x b'0 low
a'3 x b'0 hi
a'2 x b'0 hi
a'1 x b'0 hi
a'0 x b'0 hi
a'3 x b'1 low
a'2 x b'1 low
a'1 x b'1 low
a'0 x b'1 low
a'3 x b'2 hi
a'2 x b'2 hi
a'1 x b'2 hi
a'0 x b'2 hi
a'3 x b'2 low
a'2 x b'2 low
a'1 x b'2 low
a'0 x b'2 low
ACC'6
ACC'5
ACC'4
ACC'3
ACC'2
ACC'1
ACC'0
acc4
acc3
acc2
acc1
acc0

FIG. 4B

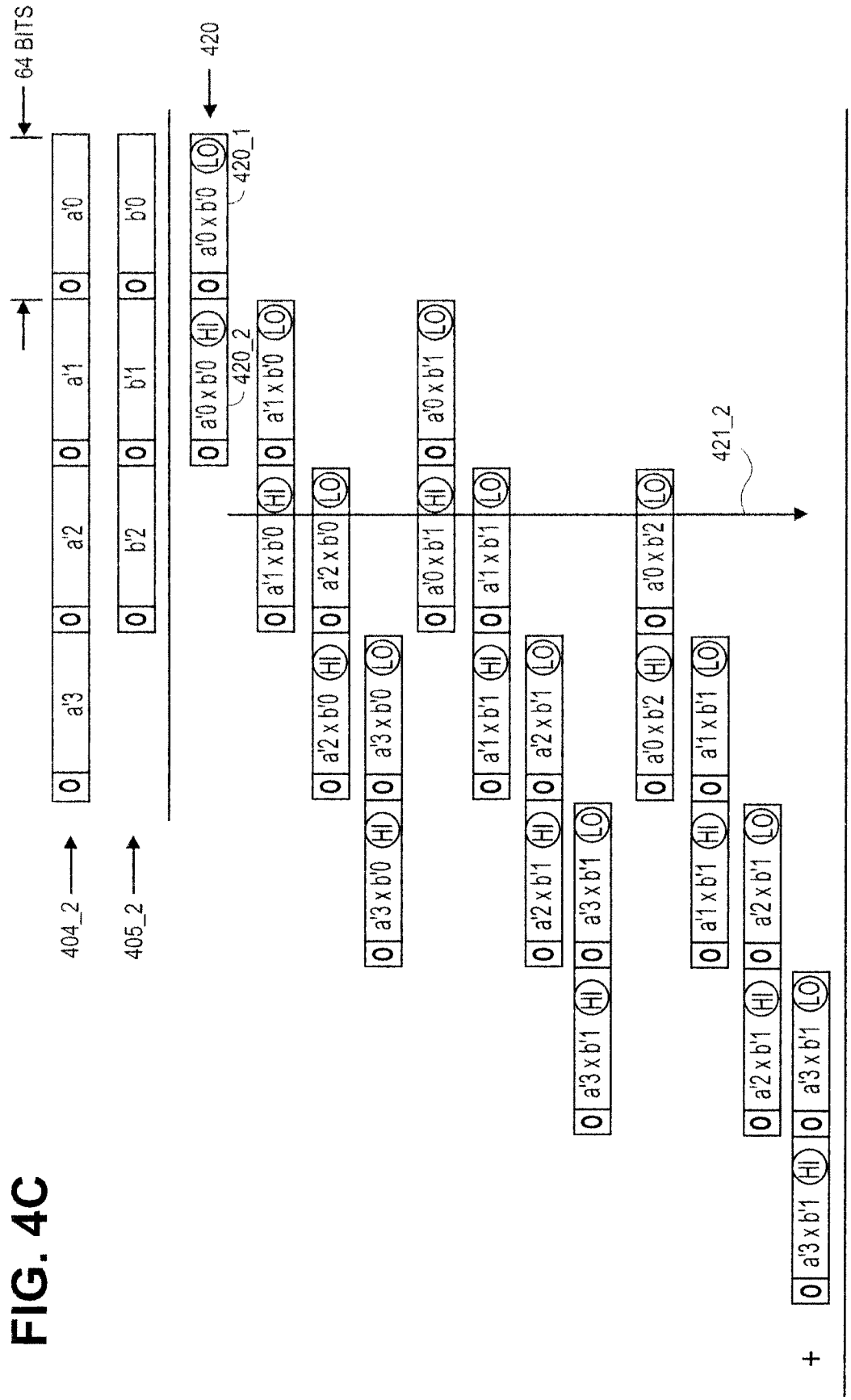

64 BITS
420
420_1
420_2
404_2
405_2
421_2
a'3
a'2
a'1
a'0
b'2
b'1
b'0
a'0 x b'0 (HI)
a'0 x b'0 (LO)
a'1 x b'0 (HI)
a'1 x b'0 (LO)
a'2 x b'0 (HI)
a'2 x b'0 (LO)
a'3 x b'0 (HI)
a'3 x b'0 (LO)
a'0 x b'1 (HI)
a'0 x b'1 (LO)
a'1 x b'1 (HI)
a'1 x b'1 (LO)
a'2 x b'1 (HI)
a'2 x b'1 (LO)
a'3 x b'1 (HI)
a'3 x b'1 (LO)
a'0 x b'2 (HI)
a'0 x b'2 (LO)
a'1 x b'1 (HI)
a'1 x b'1 (LO)
a'2 x b'1 (HI)
a'2 x b'1 (LO)
a'3 x b'1 (HI)
a'3 x b'1 (LO)
+

FIG. 4C

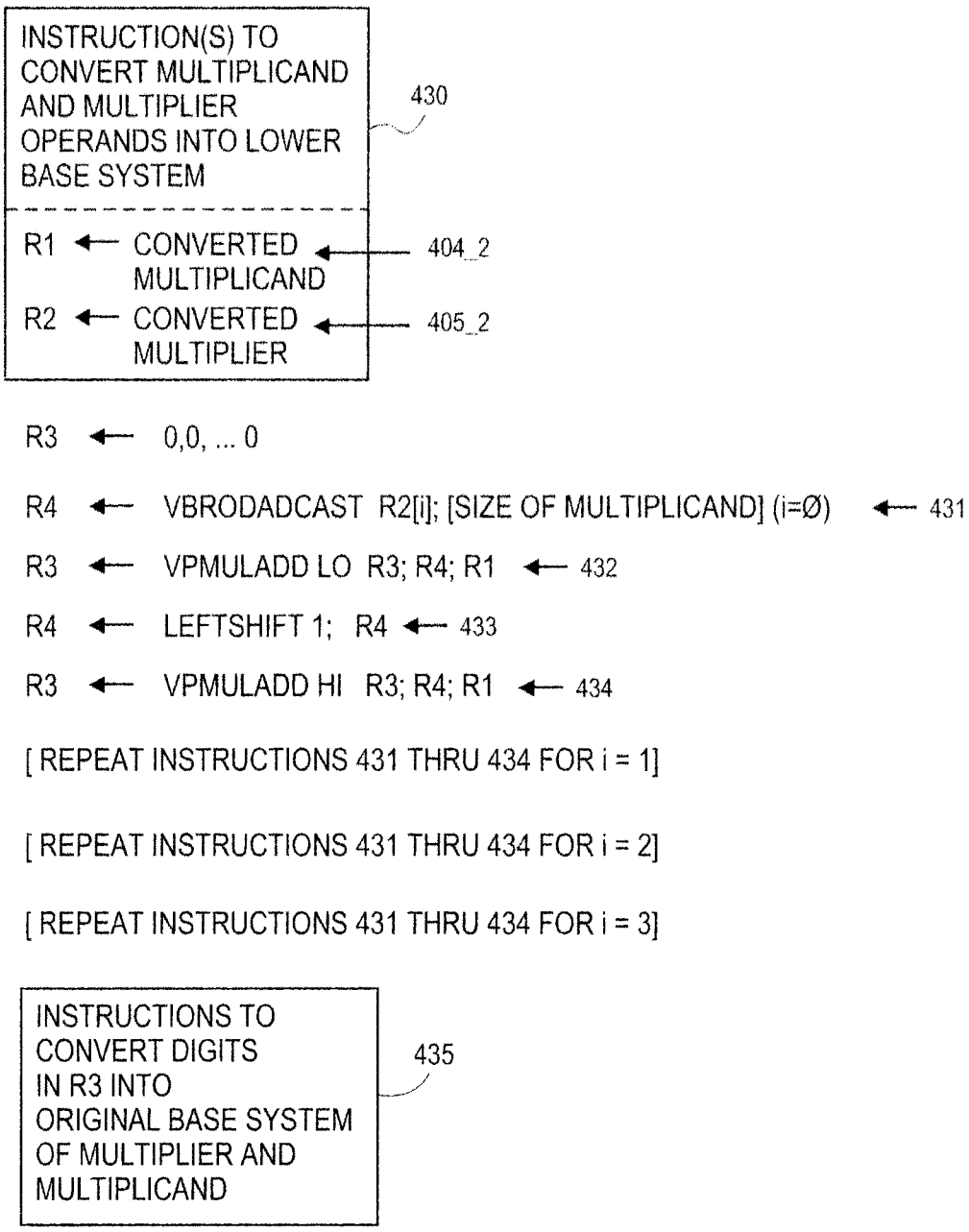

INSTRUCTION(S) TO CONVERT MULTIPLICAND AND MULTIPLIER OPERANDS INTO LOWER BASE SYSTEM
430
R1 ← CONVERTED MULTIPLICAND
404_2
R2 ← CONVERTED MULTIPLIER
405_2
R3 ← 0,0, ... 0
R4 ← VBRODADCAST R2[i]; [SIZE OF MULTIPLICAND] (i=Ø)
431
R3 ← VPMULADD LO R3; R4; R1
432
R4 ← LEFTSHIFT 1; R4
433
R3 ← VPMULADD HI R3; R4; R1
434
[REPEAT INSTRUCTIONS 431 THRU 434 FOR i = 1]
[REPEAT INSTRUCTIONS 431 THRU 434 FOR i = 2]
[REPEAT INSTRUCTIONS 431 THRU 434 FOR i = 3]
INSTRUCTIONS TO CONVERT DIGITS IN R3 INTO ORIGINAL BASE SYSTEM OF MULTIPLIER AND MULTIPLICAND
435

FIG. 4D

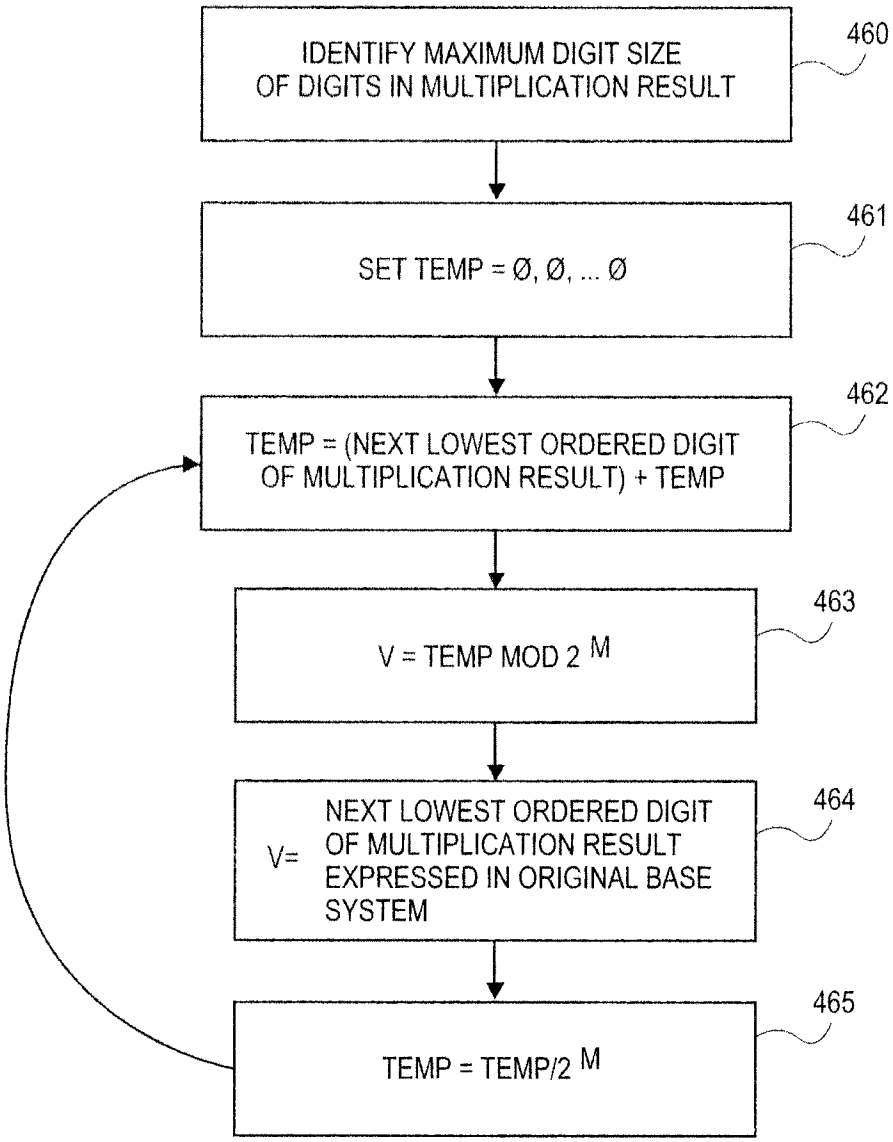

IDENTIFY MAXIMUM DIGIT SIZE OF DIGITS IN MULTIPLICATION RESULT
460
SET TEMP = Ø, Ø, ... Ø
461
TEMP = (NEXT LOWEST ORDERED DIGIT OF MULTIPLICATION RESULT) + TEMP
462
V = TEMP MOD 2 M
463
V= NEXT LOWEST ORDERED DIGIT OF MULTIPLICATION RESULT EXPRESSED IN ORIGINAL BASE SYSTEM
464
TEMP = TEMP/2 M
465

FIG. 4E

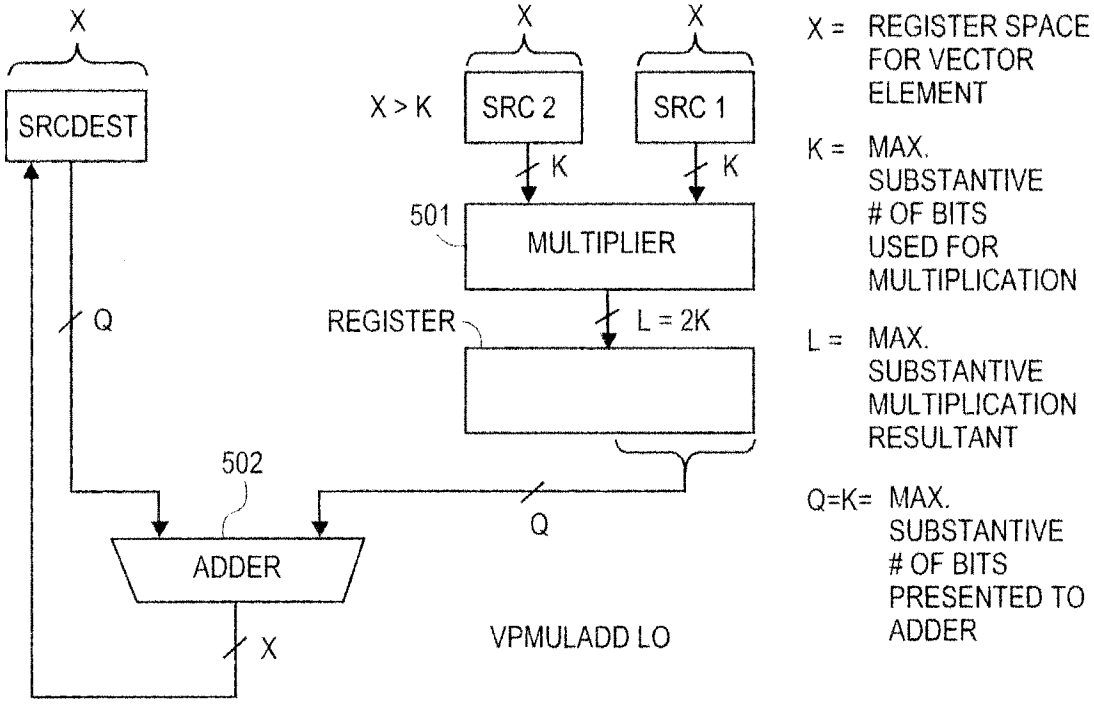

X
SRCDEST
X
X
X > K
SRC 2
SRC 1
K
K
501
MULTIPLIER
Q
REGISTER
L = 2K
502
Q
ADDER
X
VPMULADD LO
X = REGISTER SPACE FOR VECTOR ELEMENT
K = MAX. SUBSTANTIVE # OF BITS USED FOR MULTIPLICATION
L = MAX. SUBSTANTIVE MULTIPLICATION RESULTANT
Q=K= MAX. SUBSTANTIVE # OF BITS PRESENTED TO ADDER

FIG. 5A

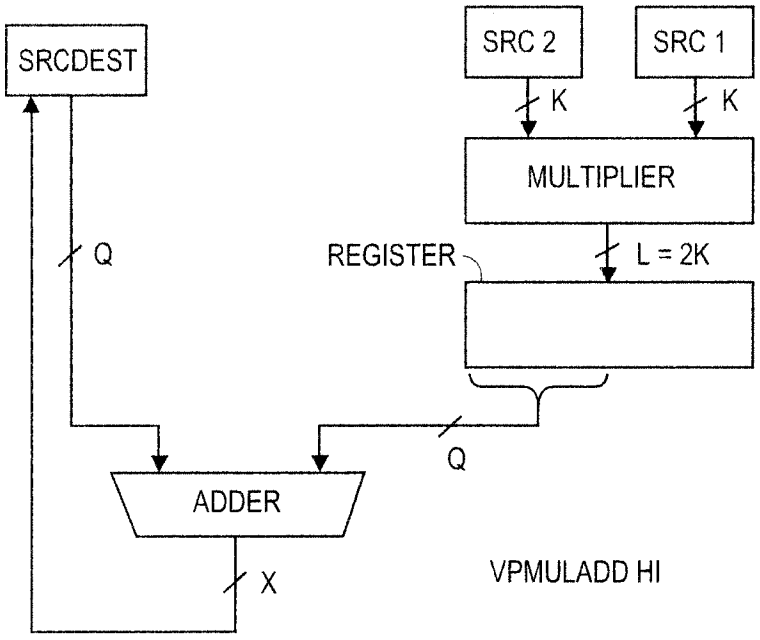

SRCDEST
SRC 2
SRC 1
K
K
MULTIPLIER
Q
REGISTER
L = 2K
Q
ADDER
X
VPMULADD HI

FIG. 5B

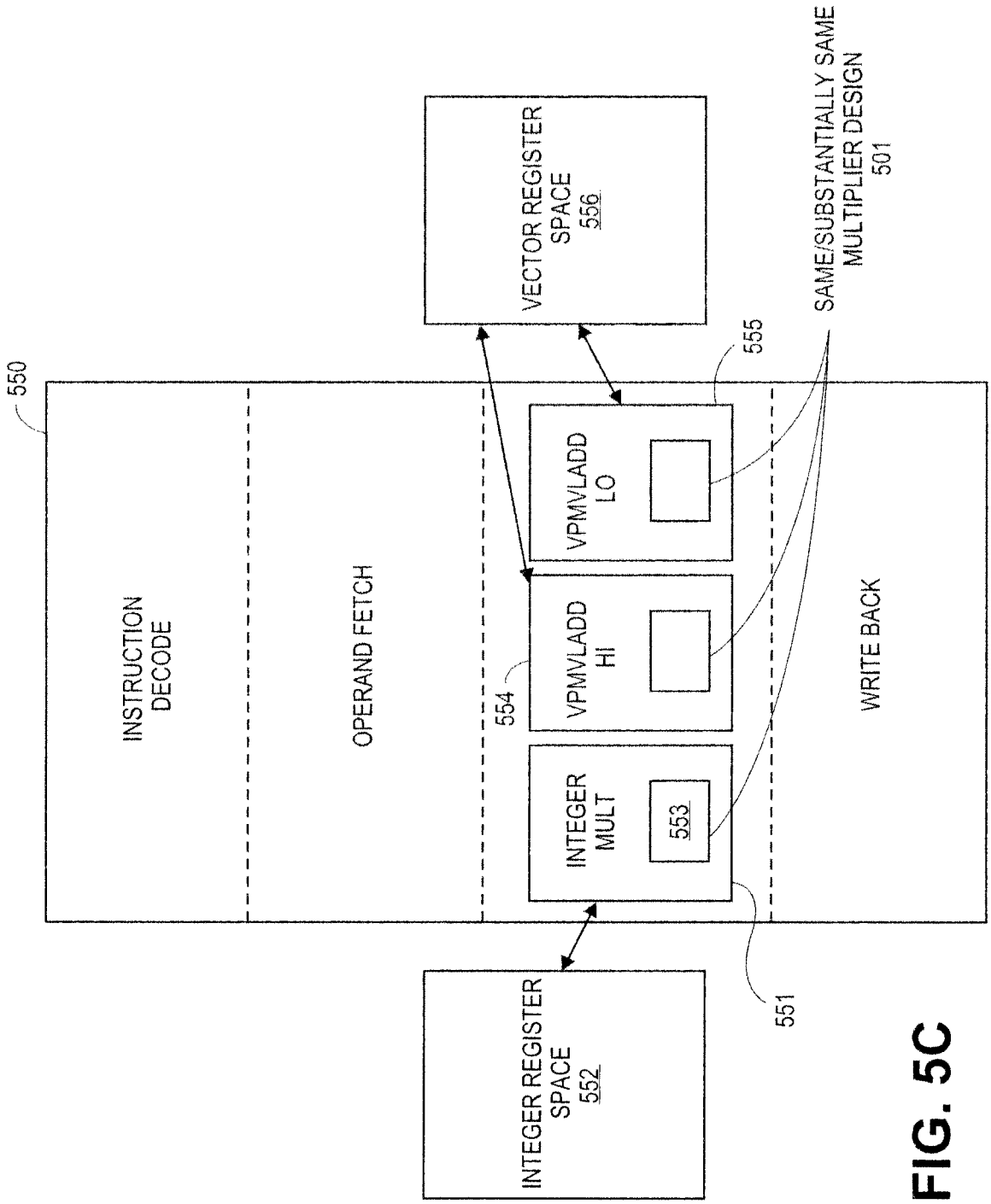

550
INSTRUCTION DECODE
OPERAND FETCH
554
INTEGER MULT
553
VPMVLADD HI
VPMVLADD LO
555
WRITE BACK
551
VECTOR REGISTER SPACE 556
INTEGER REGISTER SPACE 552
SAME/SUBSTANTIALLY SAME MULTIPLIER DESIGN 501

FIG. 5C

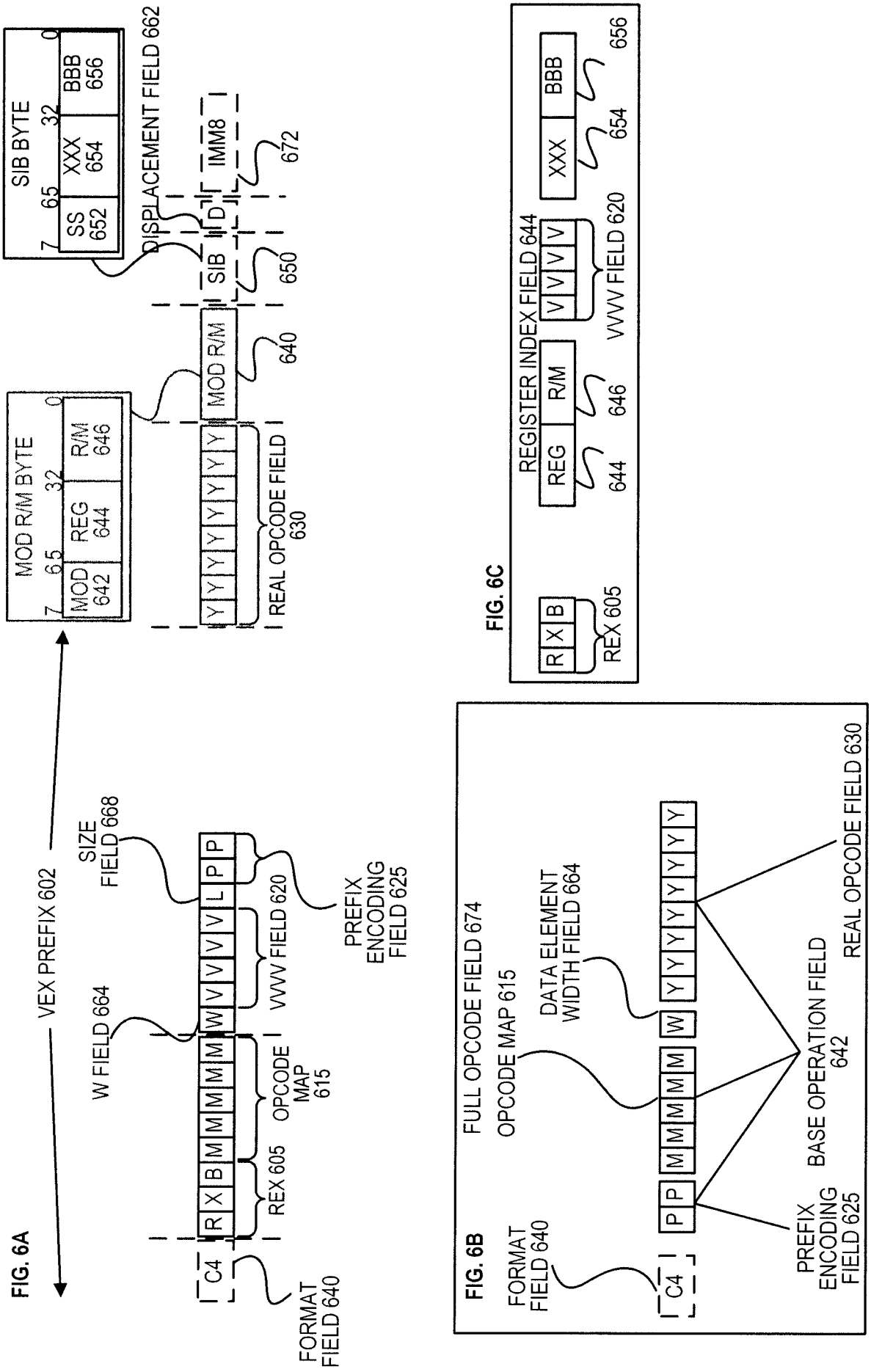


FIG. 6A
VEX PREFIX 602
SIB BYTE
7 6 5 3 2 0
SS 652
XXX 654
BBB 656
MOD R/M BYTE
MOD 642
REG 644
R/M 646
DISPLACEMENT FIELD 662
SIB
D
IMM8
672
650
640
MOD R/M
REAL OPCODE FIELD 630
Y Y Y Y Y Y Y Y
C4
R X B M M M M M
W V V V V L P P
REX 605
OPCODE MAP 615
W FIELD 664
SIZE FIELD 668
VVVV FIELD 620
PREFIX ENCODING FIELD 625
FORMAT FIELD 640
FIG. 6C
REGISTER INDEX FIELD 644
REG
R/M
V V V V
XXX
BBB
644
646
VVVV FIELD 620
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 M
W
Y Y Y Y Y Y Y Y
PREFIX ENCODING FIELD 625
BASE OPERATION FIELD 642
REAL OPCODE FIELD 630

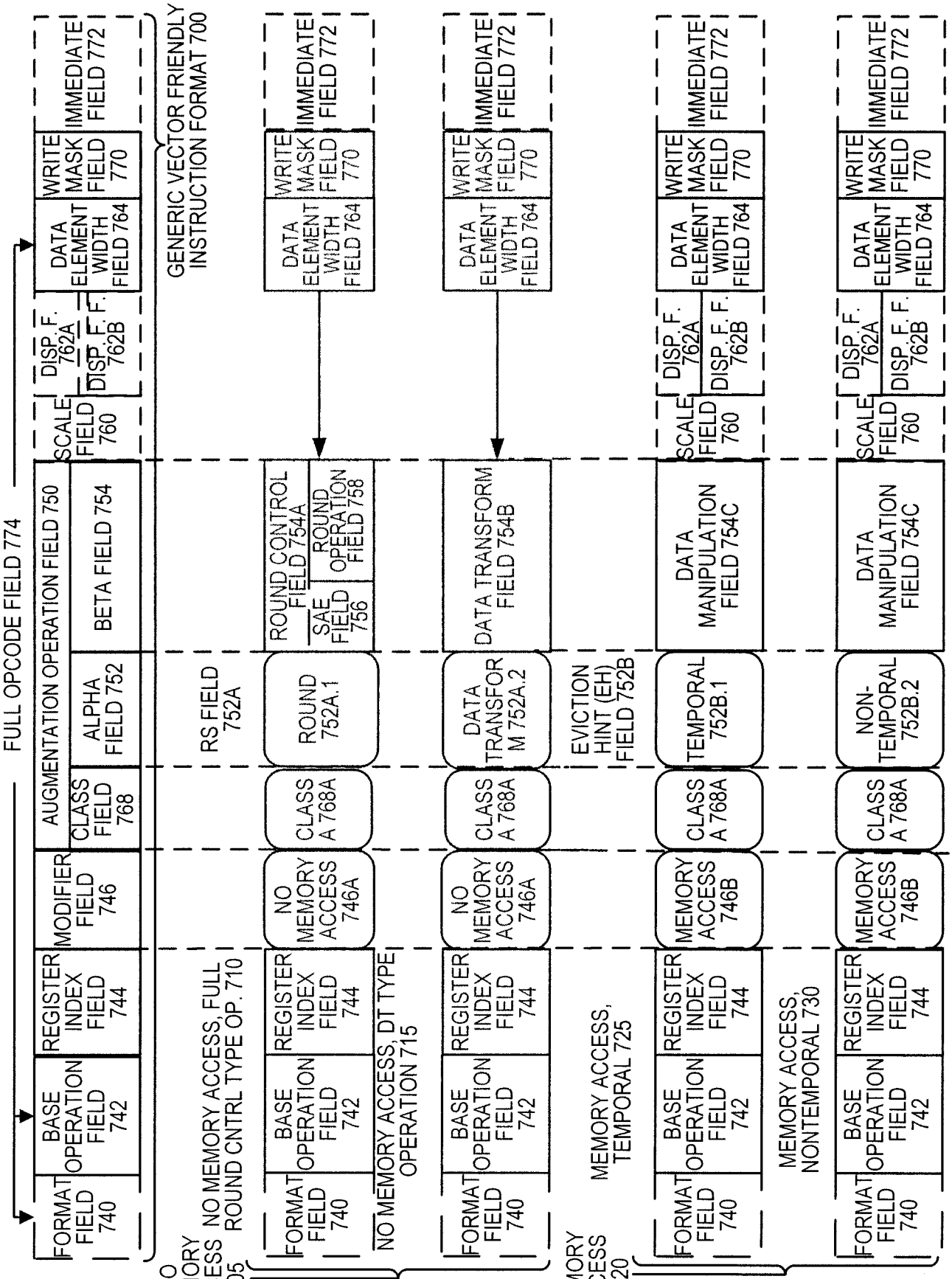


FIG.7A
FULL OPCODE FIELD 774
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
NO MEMORY ACCESS 705
NO MEMORY ACCESS, FULL ROUND CNTRL TYPE OP. 710
NO MEMORY ACCESS, DT TYPE OPERATION 715
MEMORY ACCESS 720
MEMORY ACCESS, TEMPORAL 725
MEMORY ACCESS, NONTEMPORAL 730
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD 746
NO MEMORY ACCESS 746A
MEMORY ACCESS 746B
AUGMENTATION OPERATION FIELD 750
CLASS FIELD 768
CLASS A 768A
ALPHA FIELD 752
RS FIELD 752A
ROUND 752A.1
DATA TRANSFORM 752A.2
EVICTION HINT (EH) FIELD 752B
TEMPORAL 752B.1
NON-TEMPORAL 752B.2
BETA FIELD 754
ROUND CONTROL FIELD 754A
SAE FIELD 756
ROUND OPERATION FIELD 758
DATA TRANSFORM FIELD 754B
DATA MANIPULATION FIELD 754C
SCALE FIELD 760
DISP. F. 762A
DISP. F. F. 762B
DATA ELEMENT WIDTH FIELD 764
WRITE MASK FIELD 770
IMMEDIATE FIELD 772

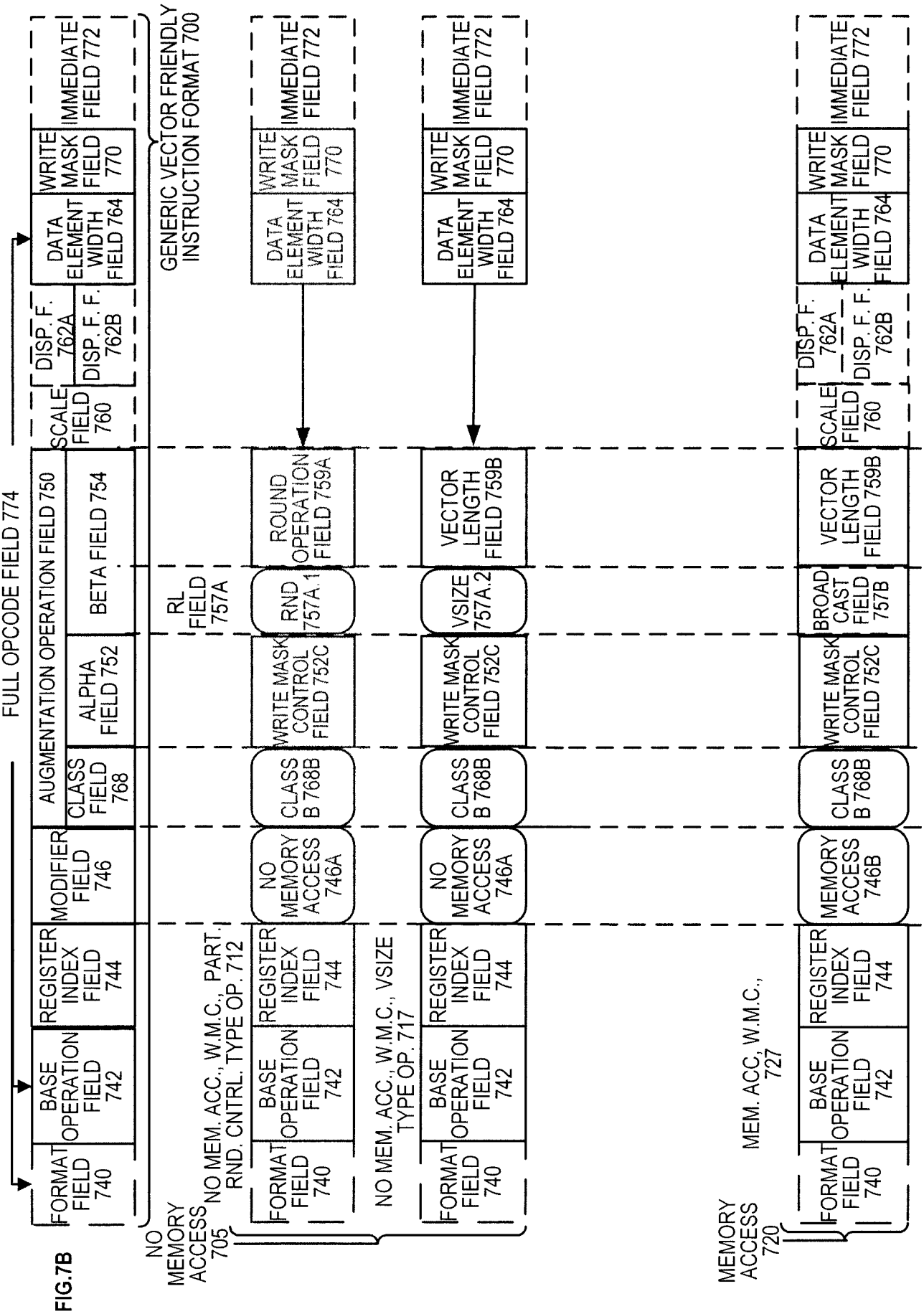

FULL OPCODE FIELD 774
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 700
NO MEMORY ACCESS 705
NO MEM. ACC., W.M.C., PART. RND. CNTRL. TYPE OP. 712
NO MEM. ACC., W.M.C., VSIZE TYPE OP. 717
MEMORY ACCESS 720
MEM. ACC, W.M.C., 727
FORMAT FIELD 740
BASE OPERATION FIELD 742
REGISTER INDEX FIELD 744
MODIFIER FIELD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

FIG.7B

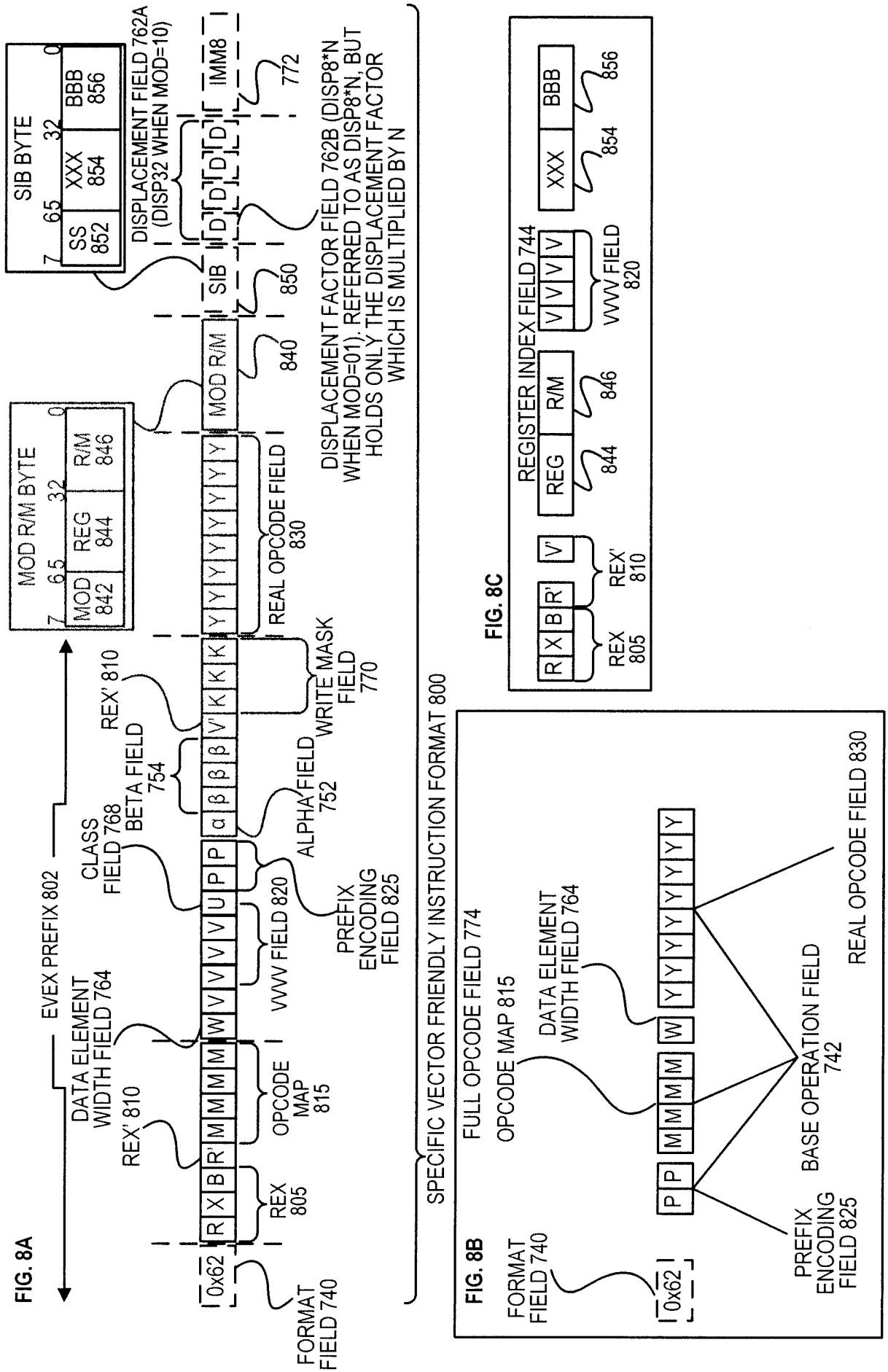
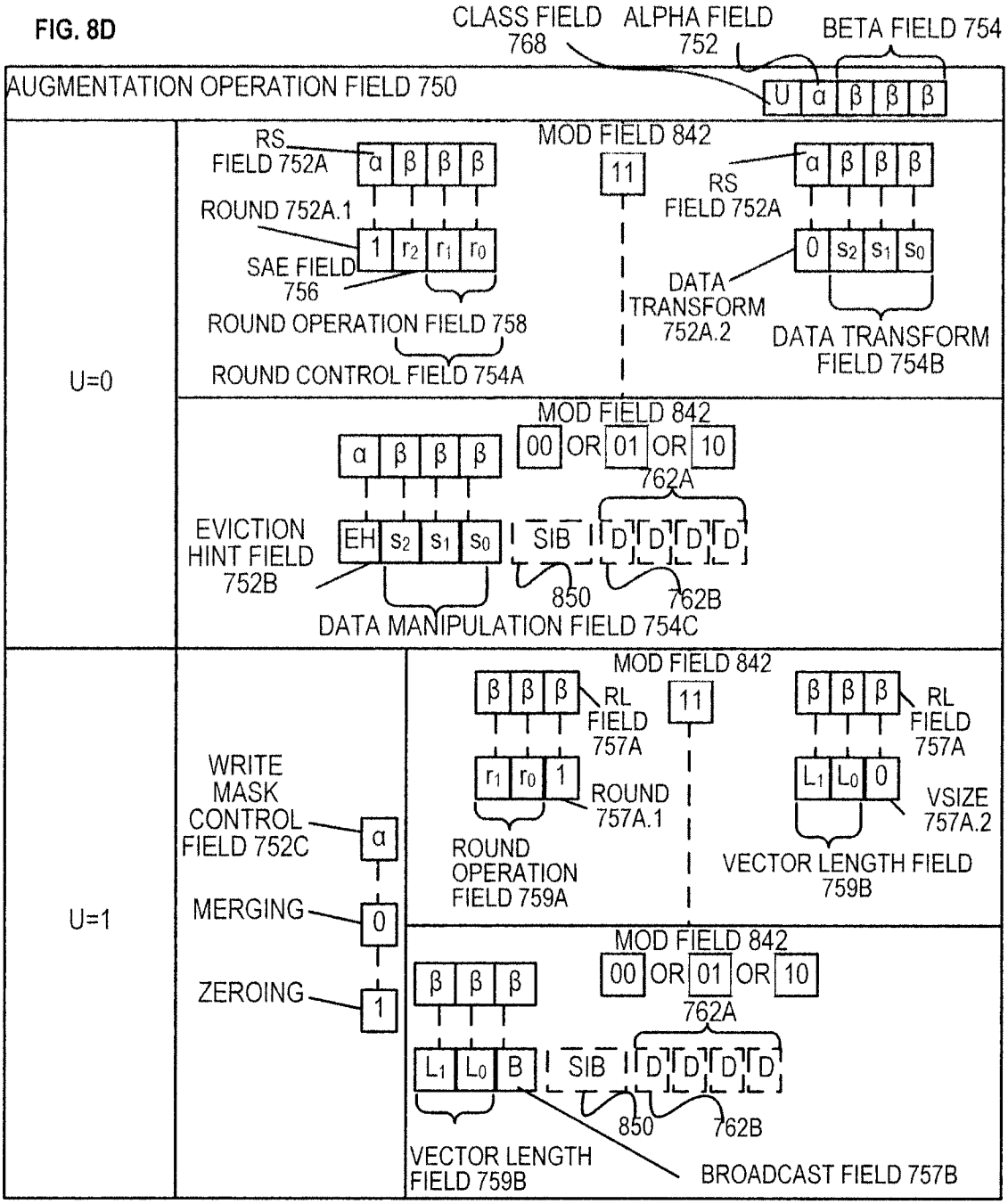


FIG. 8A
EVEX PREFIX 802
FORMAT FIELD 740
0x62
REX 805
REX' 810
OPCODE MAP 815
DATA ELEMENT WIDTH FIELD 764
VVV FIELD 820
CLASS FIELD 768
BETA FIELD 754
REX' 810
ALPHA FIELD 752
PREFIX ENCODING FIELD 825
WRITE MASK FIELD 770
REAL OPCODE FIELD 830
MOD R/M BYTE
MOD 842
REG 844
R/M 846
MOD R/M
SIB
DISPLACEMENT FIELD 762A (DISP32 WHEN MOD=10)
SIB BYTE
SS 852
XXX 854
BBB 856
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
IMM8
772
840
850
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
REG
R/M
V'
VVV FIELD 820
REX 805
REX' 810
844
846
854
856

FIG. 8D


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

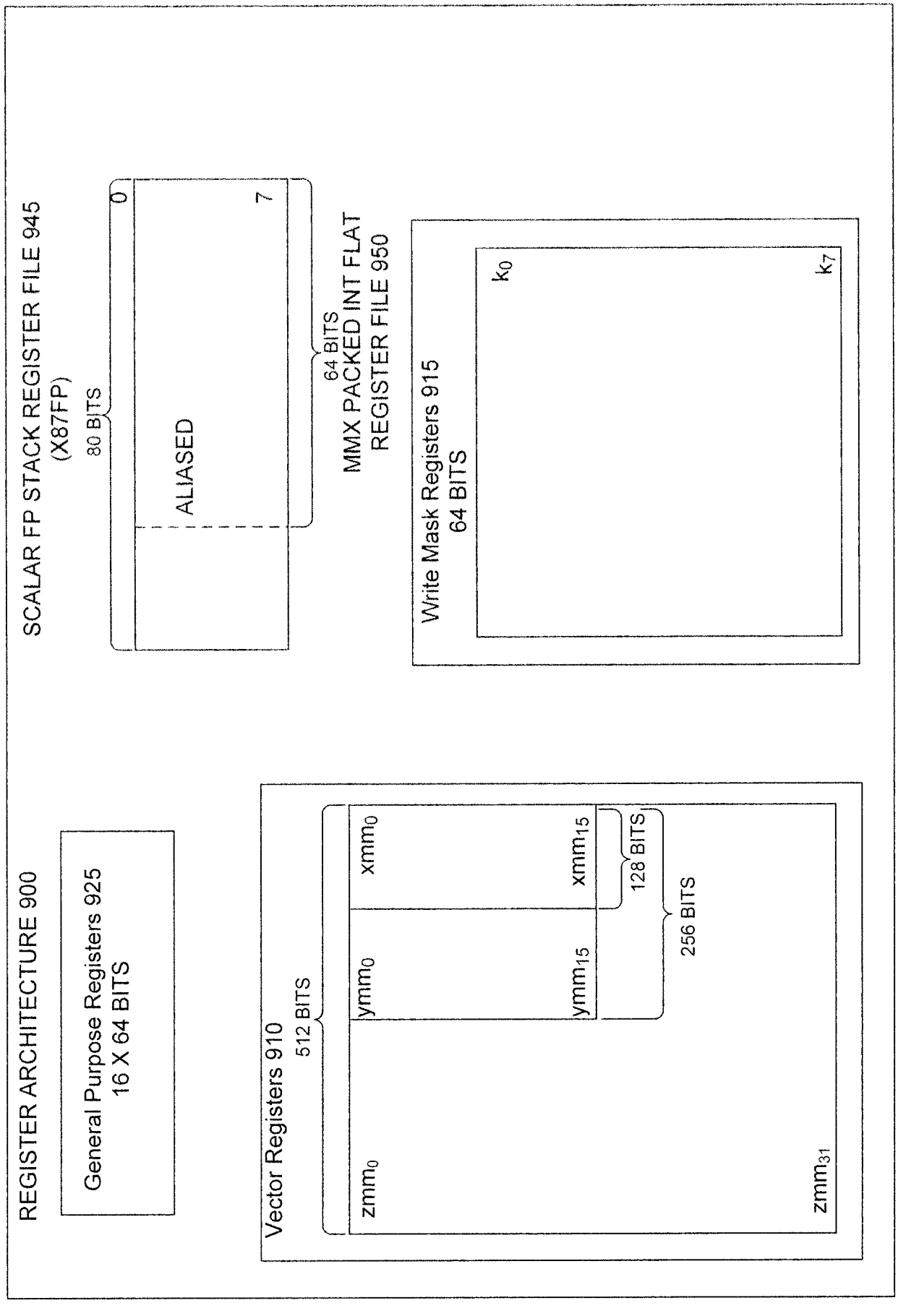

REGISTER ARCHITECTURE 900
General Purpose Registers 925
16 X 64 BITS
SCALAR FP STACK REGISTER FILE 945
(X87FP)
80 BITS
0
7
ALIASED
64 BITS
MMX PACKED INT FLAT
REGISTER FILE 950
Write Mask Registers 915
64 BITS
k0
k7
Vector Registers 910
512 BITS
zmm0
ymm0
xmm0
ymm15
xmm15
128 BITS
256 BITS
zmm31

FIG. 9

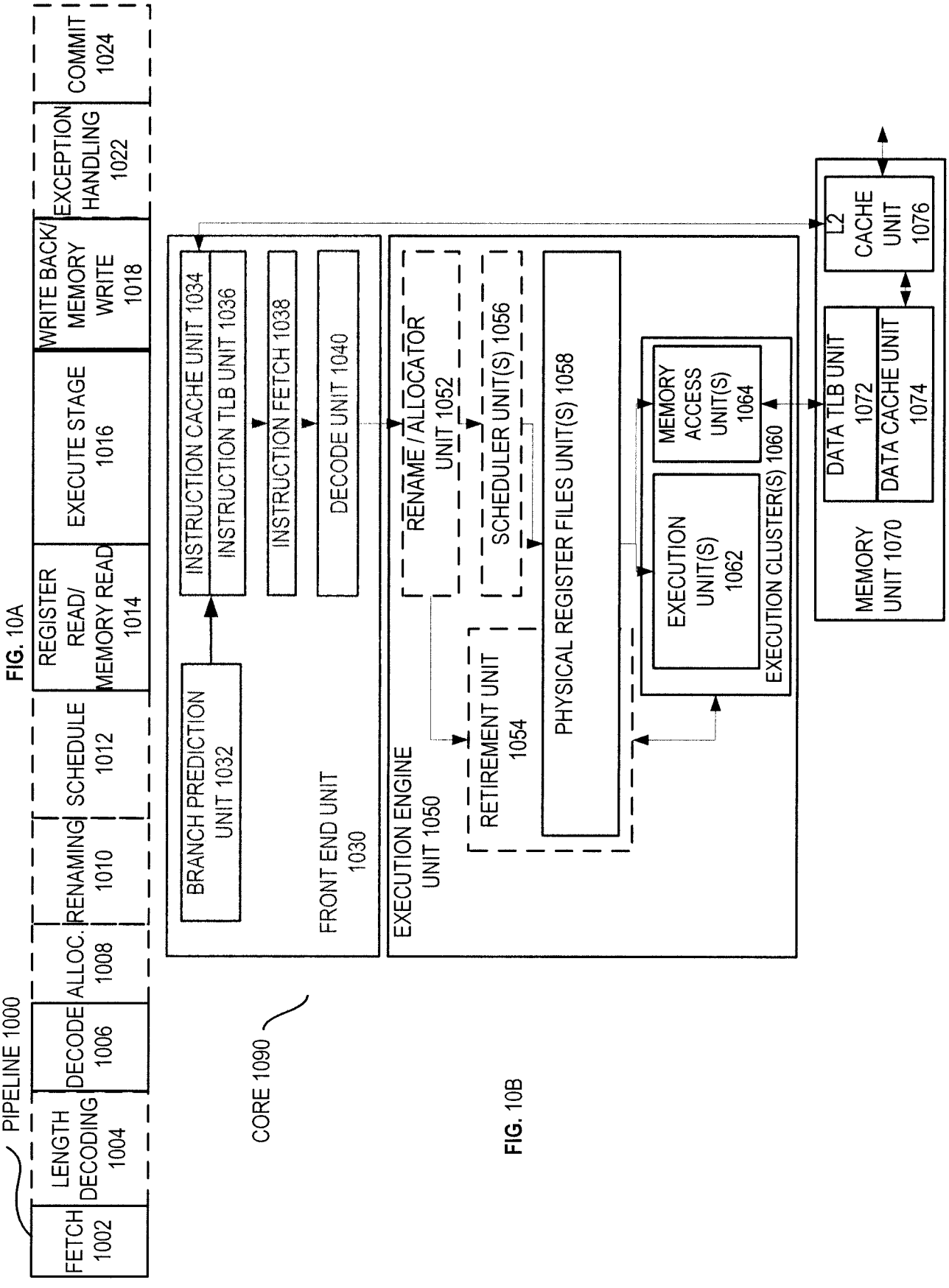


FIG. 10A

FIG. 10B

FIG. 11B

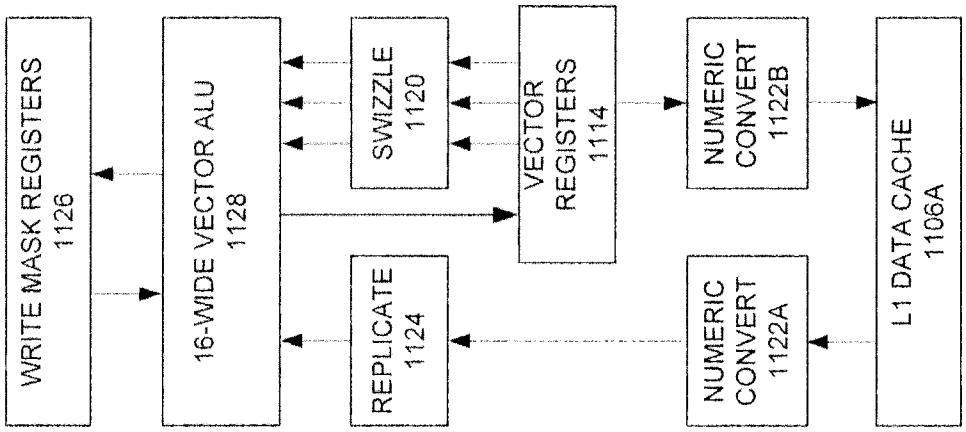
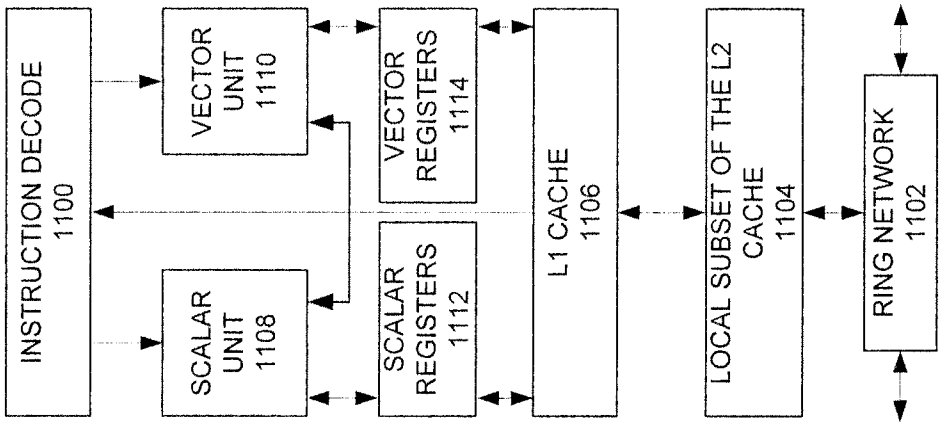


FIG. 11A



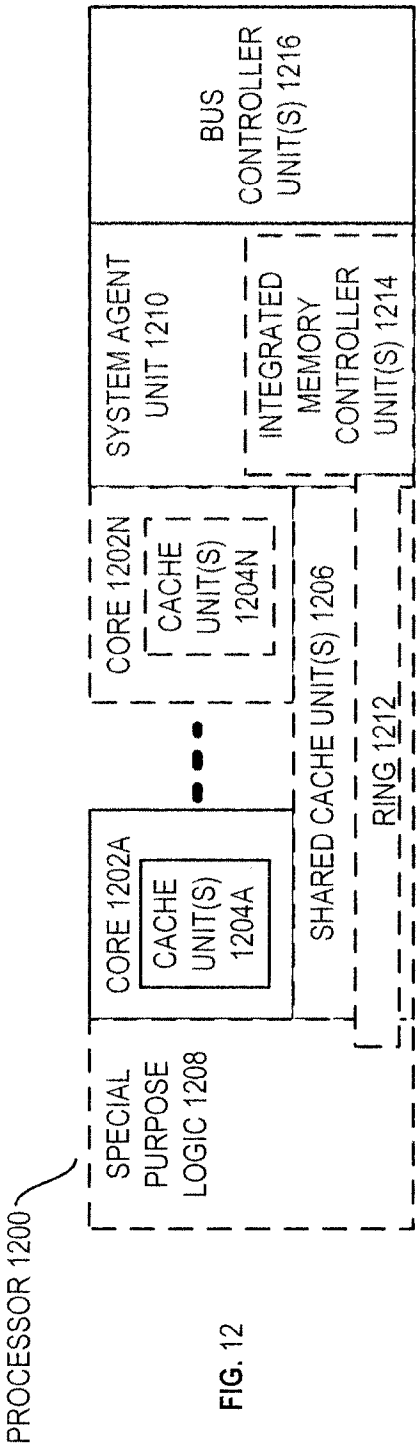

PROCESSOR 1200
SPECIAL PURPOSE LOGIC 1208
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
RING 1212
SYSTEM AGENT UNIT 1210
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214
BUS CONTROLLER UNIT(S) 1216

FIG. 12

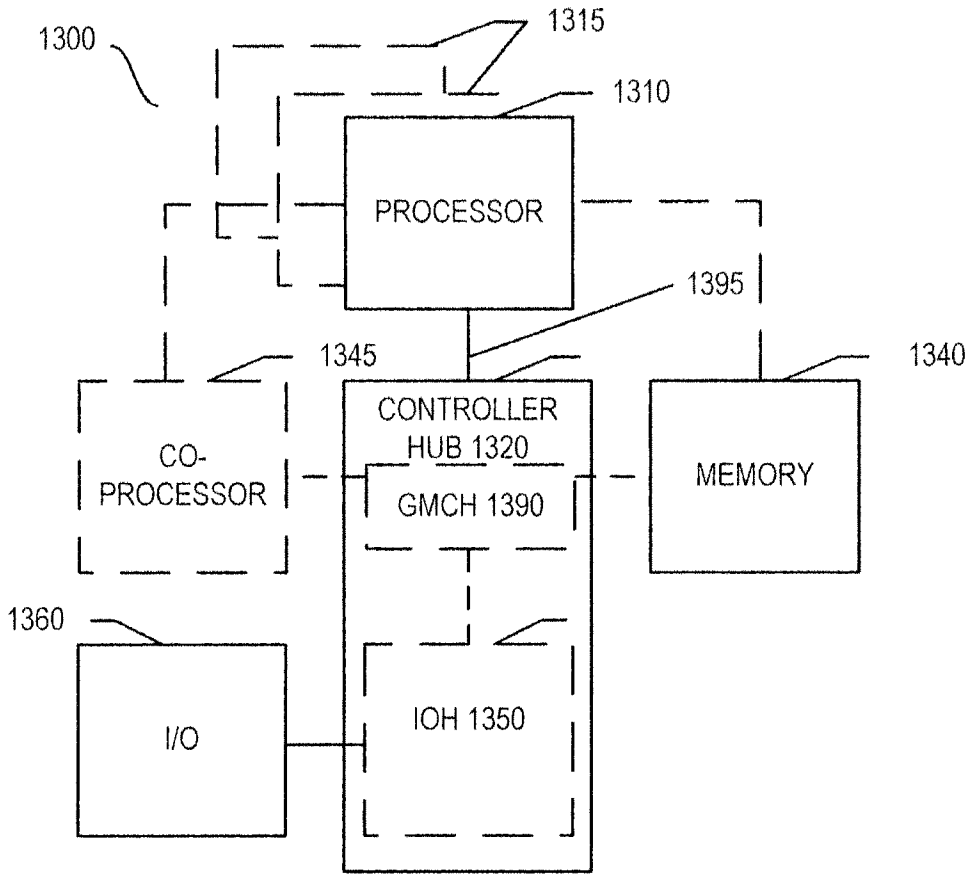

1300
1315
1310
PROCESSOR
1395
1345
1340
CONTROLLER HUB 1320
CO-PROCESSOR
GMCH 1390
MEMORY
1360
I/O
IOH 1350

FIG. 13

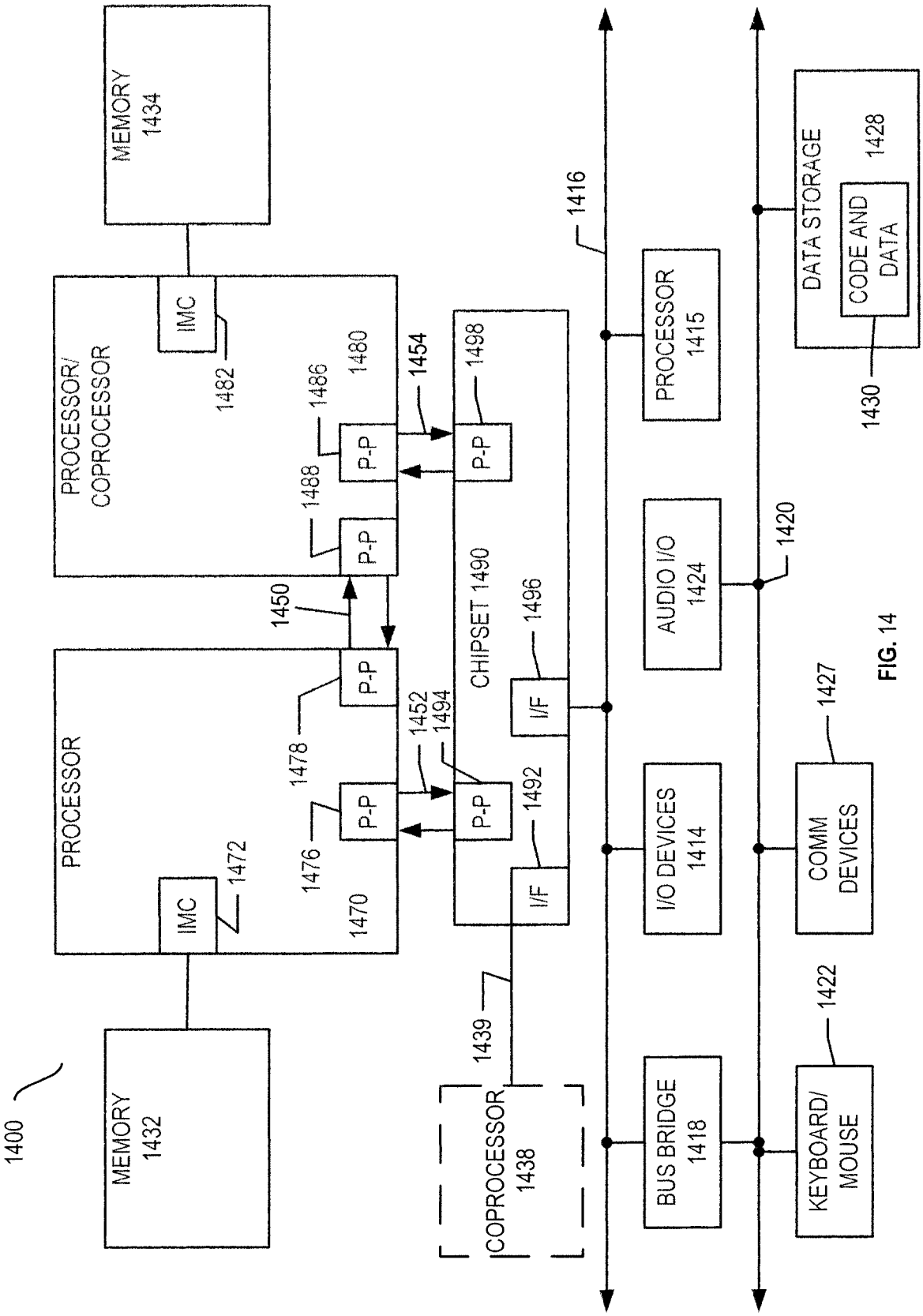

1400
MEMORY 1432
MEMORY 1434
PROCESSOR
IMC
1472
1470
1476
1478
P-P
1450
PROCESSOR/ COPROCESSOR
IMC
1482
1480
1486
1488
P-P
1454
1452
1494
CHIPSET 1490
1498
1496
1492
I/F
1439
COPROCESSOR 1438
1416
PROCESSOR 1415
AUDIO I/O 1424
1420
I/O DEVICES 1414
BUS BRIDGE 1418
DATA STORAGE
CODE AND DATA
1428
1430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

FIG. 14

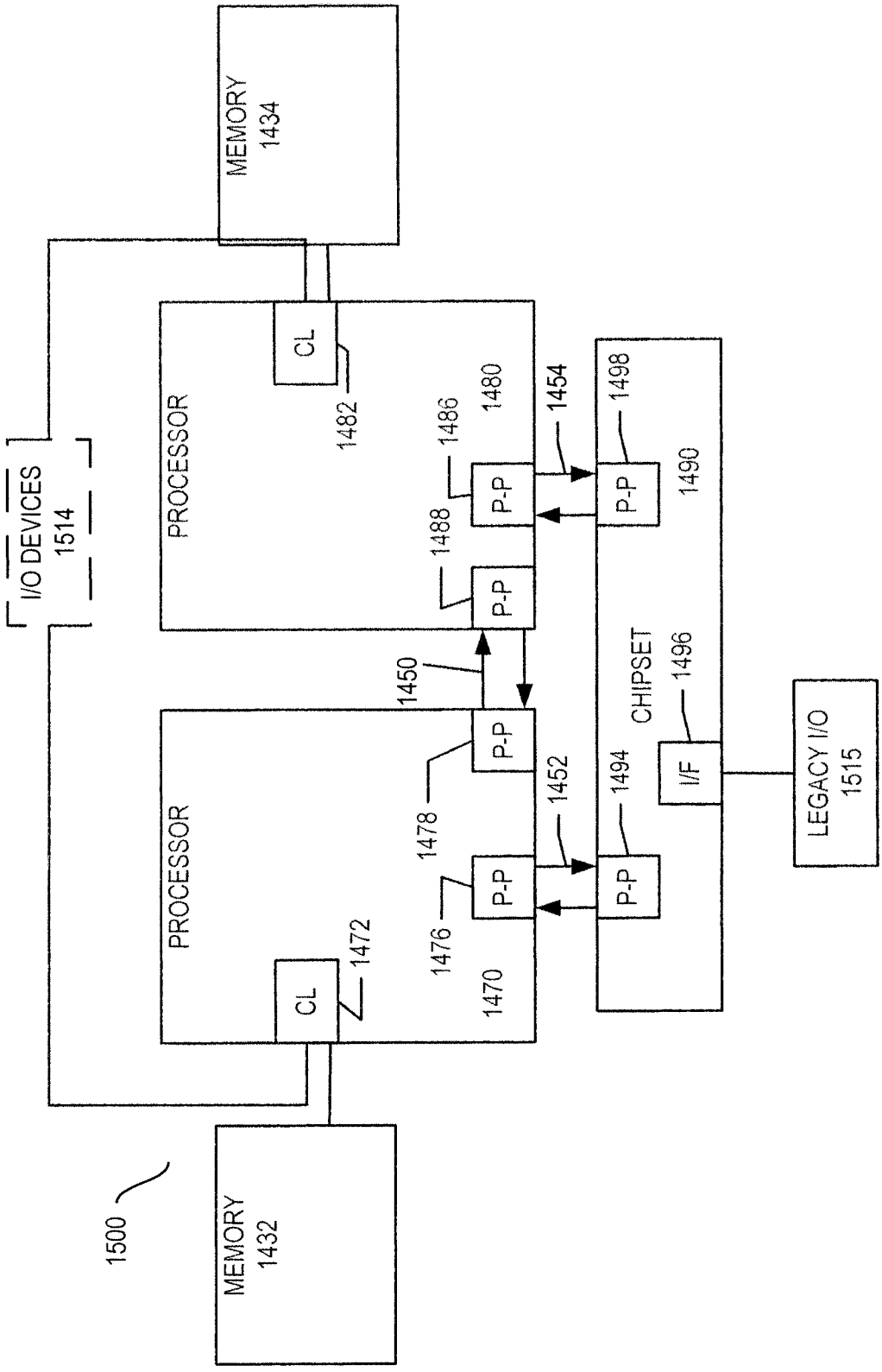

MEMORY
1434
I/O DEVICES
1514
PROCESSOR
CL
1482
1480
1486
1488
P-P
P-P
1454
1498
P-P
1490
CHIPSET
1496
I/F
LEGACY I/O
1515
1450
1478
P-P
1452
1494
P-P
1476
P-P
1470
PROCESSOR
CL
1472
1500
MEMORY
1432

FIG. 15

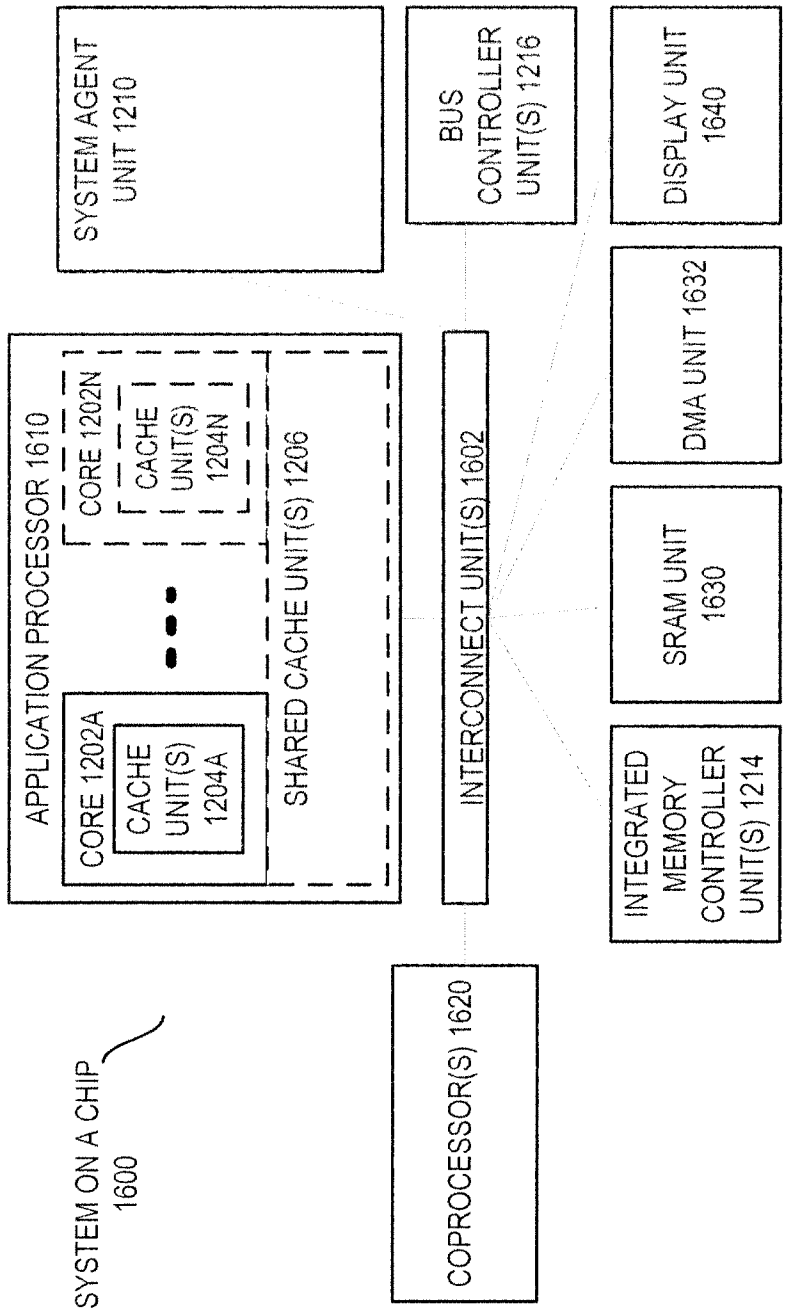

SYSTEM ON A CHIP
1600
APPLICATION PROCESSOR 1610
CORE 1202A
CACHE UNIT(S) 1204A
CORE 1202N
CACHE UNIT(S) 1204N
SHARED CACHE UNIT(S) 1206
SYSTEM AGENT UNIT 1210
BUS CONTROLLER UNIT(S) 1216
INTERCONNECT UNIT(S) 1602
COPROCESSOR(S) 1620
INTEGRATED MEMORY CONTROLLER UNIT(S) 1214
SRAM UNIT 1630
DMA UNIT 1632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

FIG. 16

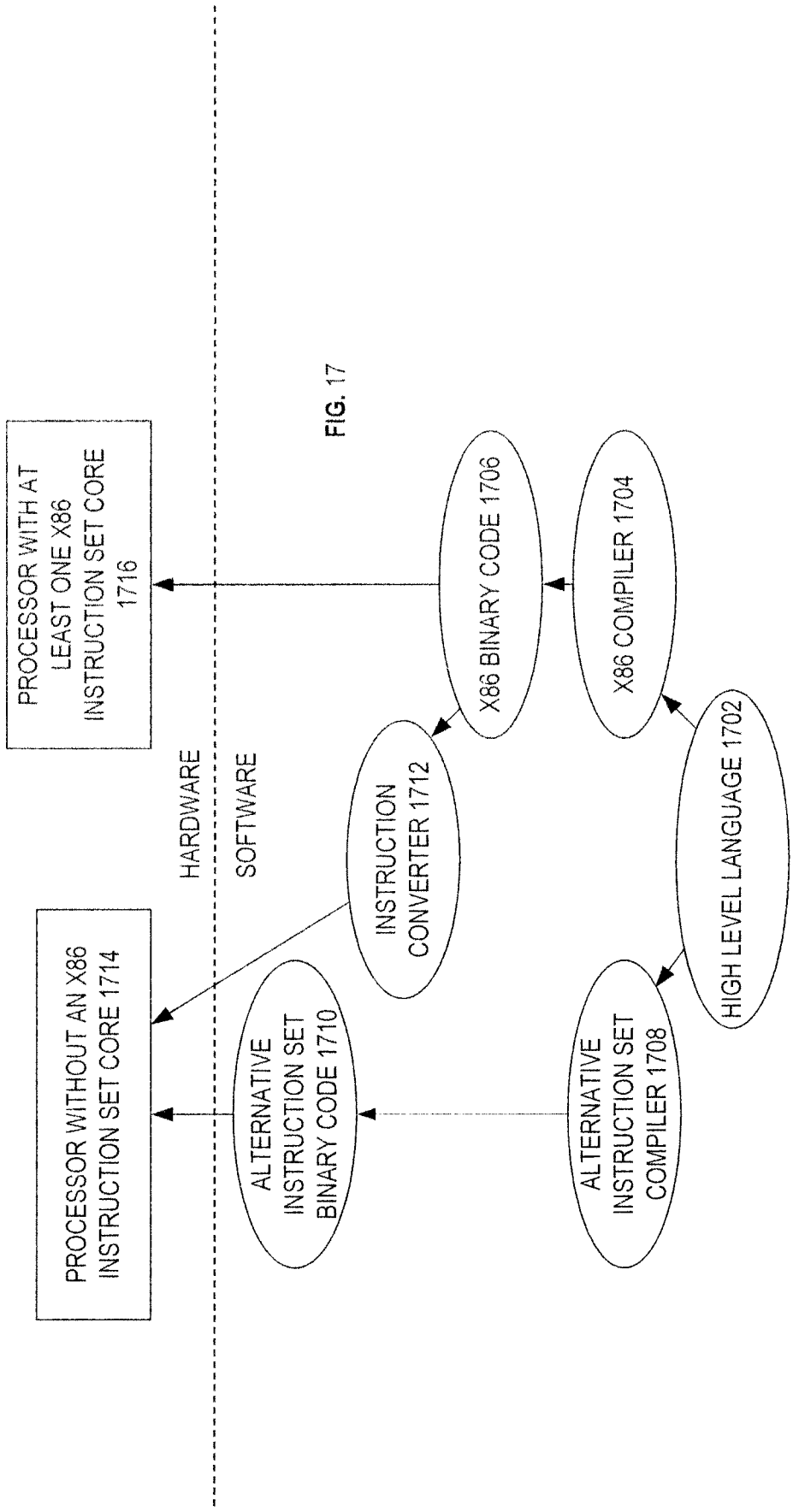

HIGH LEVEL LANGUAGE 1702
X86 COMPILER 1704
X86 BINARY CODE 1706
ALTERNATIVE INSTRUCTION SET COMPILER 1708
INSTRUCTION CONVERTER 1712
ALTERNATIVE INSTRUCTION SET BINARY CODE 1710
PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE 1714
PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE 1716
HARDWARE
SOFTWARE

FIG. 17

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2013/047393

A. CLASSIFICATION OF SUBJECT MATTER

G06F 17/16(2006.01)i, G06F 12/00(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 17/16; G06F 7/52; G06F 9/44; G06F 7/496; G06F 9/302; G06F 7/44; G06F 7/485; G06F 15/76; G06F 12/00

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: execution, pipeline, vector, multiply, add, accumulator, operand, and similar terms.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2012-0078992 A1 (JEFF WIEDEMEIER et al.) 29 March 2012 See paragraphs [0001]-[0005], [0018]-[0023], [0029]-[0032], [0042], and [0050]-[0051]; figure 3 and 4B; and claims 1-2.	11
Y		1-10,12-20
Y	US 2010-0274990 A1 (MLADEN WILDER et al.) 28 October 2010 See paragraphs [0014]-[0030], [0037], [0046]-[0048], [0062], [0065], [0069]-[0070], [0073]-[0074], [0082], [0084], [0087], [0090], [0095], and [0107]-[0108]; and figures 2 and 9.	1-10,12-20
A	US 2009-0248779 A1 (JEFFREY S. BROOKS et al.) 01 October 2009 See paragraphs [0012]-[0016], [0036]-[0040], and [0053]-[0058]; figures 1-3; and claims 1 and 14.	1-20
A	US 2006-0253520 A1 (CHINH N. TRAN) 09 November 2006 See paragraphs [0004]-[0005], [0018]-[0019], [0023], and [0052] ; figures 1 and 3; and claims 8 and 19.	1-20
A	KR 10-1005718 B1 (SANDBRIDGE TECHNOLOGIES, INC.) 10 January 2011 See paragraphs [0015]-[0019], [0027]-[0040], and [0067]-[0074]; and figures 1 and 4.	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
17 October 2013 (17.10.2013)	**18 October 2013 (18.10.2013)**

Name and mailing address of the ISA/KR
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701, Republic of Korea
Facsimile No. +82-42-472-7140

Authorized officer
NHO Ji Myong
Telephone No. +82-42-481-8528

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2013/047393

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2006-0129787 A1 (TIMOTHY VAN HOOK et al.) 15 June 2006 See paragraphs [0005]-[0008], [0013], [0034]-[0037], and [0053]-[0058]; and figures 3-5.	1-20

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2013/047393

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2012-0078992 A1	29/03/2012	CN 103119579 A	22/05/2013
		DE 112011103196 T5	19/09/2013
		GB 201303473 D0	10/04/2013
		GB 2497450 A	12/06/2013
		KR 10-2013-0063532 A	14/06/2013
		TW 201237743 A	16/09/2012
		WO 2012-040545 A2	29/03/2012
		WO 2012-040545 A3	14/06/2012
		WO 2012-040545 A9	04/10/2012
US 2010-0274990 A1	28/10/2010	CN 102197369 A	21/09/2011
		EP 2350813 A1	03/08/2011
		GB 0818491 D0	12/11/2008
		GB 0915208 D0	07/10/2009
		GB 2464178 A	14/04/2010
		GB 2464178 B	08/08/2012
		GB 2464292 A	14/04/2010
		IL 211767 D0	30/06/2011
		JP 2012-505455 A	01/03/2012
		KR 10-2011-0090915 A	10/08/2011
		TW 201020805 A	01/06/2010
		US 8443170 B2	14/05/2013
		WO 2010-040977 A1	15/04/2010
US 2009-0248779 A1	01/10/2009	US 2012-221614 A1	30/08/2012
		US 8239440 B2	07/08/2012
US 2006-0253520 A1	09/11/2006	US 8234326 B2	31/07/2012
KR 10-1005718 B1	10/01/2011	CN 1820246 A	16/08/2006
		EP 1623307 A2	08/02/2006
		EP 1735694 A2	27/12/2006
		EP 2290525 A2	02/03/2011
		EP 2290525 A3	20/04/2011
		JP 2006-529043 A	28/12/2006
		JP 2007-533009 A	15/11/2007
		KR 10-2006-0133086 A	22/12/2006
		US 2005-0071413 A1	31/03/2005
		US 2006-0041610 A1	23/02/2006
		US 7593978 B2	22/09/2009
		US 7797363 B2	14/09/2010
		WO 2004-103056 A2	02/12/2004
		WO 2004-103056 A3	29/12/2005
		WO 2005-101190 A2	27/10/2005
		WO 2005-101190 A3	22/03/2007
US 2006-0129787 A1	15/06/2006	None	