



- (51) International Patent Classification:
G06F 9/30 (2006.01) *G06F 7/487* (2006.01)
- (21) International Application Number:
PCT/US2015/062328
- (22) International Filing Date:
24 November 2015 (24.11.2015)
- (25) Filing Language: English
- (26) Publication Language: English
- (30) Priority Data:
14/583,046 24 December 2014 (24.12.2014) US
- (71) Applicant: INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, California 95054 (US).

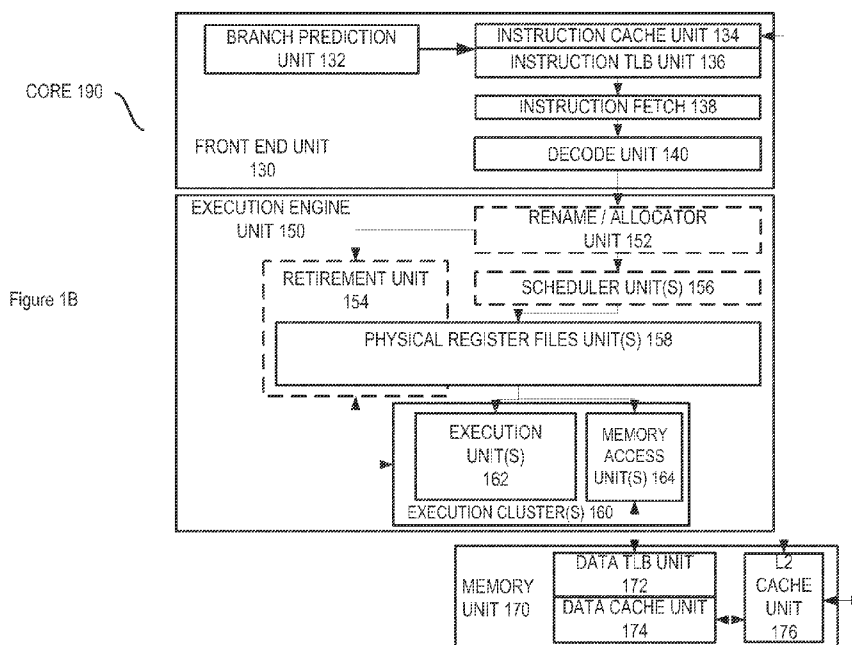
- (72) Inventors: CORBAL SAN ADRIAN, Jesus; 6723 NE Vinings Way #1725, Hillsboro, Oregon 97124 (US). VALENTINE, Robert; Rechov Hadganiot 33-5, Ha, 36054 Kiryat Tivon (IL). CHARNEY, Mark J.; 610 Waltham Street, Lexington, Massachusetts 02421 (US). OULD-AHMED-VALL, Elmoustapha; 5000 West Chandler Boulevard, M/s: Ch7-401, Chandler, Arizona 85226 (US). ESPASA, Roger; Muntaner 447, 4rt 1a, 08021 Barcelona (ES). SOLE, Guillem; Jerico 27, 3-4, 08035 Barcelona (ES). FERNANDEZ, Manel; C/ Catasus, 15-1, 08042 Barcelona (ES). HICKMAN, Brian; 14762 SW Smock Street, Sherwood, Oregon 97140 (US).

- (74) Agent: NICHOLSON, David F.; Nicholson De Vos Webster & Elliott, LLP, 217 High Street, Palo Alto, California 94301 (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, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

[Continued on next page]

- (54) Title: APPARATUS AND METHOD FOR FUSED MULTIPLY-MULTIPLY INSTRUCTIONS



- (57) Abstract: In one embodiment of the invention, a processor device including a storage location configured to store a set of source packed-data operands, each of the operands having a plurality of packed-data elements that are positive or negative according to an immediate bit value within one of the operands. The processor also including: a decoder to decode an instruction requiring an input of a plurality of source operands, and an execution unit to receive the decoded instructions and to generate a result that is a product of the source operands. In one embodiment, the result is stored back into one of the source operands or the result is stored into an operand that is independent of the source operands.



(84) Designated States (*unless otherwise indicated, for every kind of regional protection available*): ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE,

SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Published:

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

APPARATUS AND METHOD FOR FUSED MULTIPLY-MULTIPLY INSTRUCTIONS**FIELD**

[0001] This disclosure relates to microprocessors and, more particularly, to instructions for operations on data elements in the microprocessors.

BACKGROUND

[0002] To improve the efficiency of multimedia applications, as well as other applications with similar characteristics, Single Instruction, Multiple Data (SIMD) architectures have been implemented in microprocessor systems to enable one instruction to operate on several operands in parallel. In particular, SIMD architectures take advantage of packing many data elements within one register or contiguous memory location. With parallel hardware execution, multiple operations are performed on separate data elements by one instruction. This typically results in significant performance advantages, however, at a cost of increased required logic and thus greater power consumption.

BRIEF DESCRIPTION OF THE DRAWINGS

[0003] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings in which like references indicate similar elements.

[0004] **Figure 1A** is a block diagram illustrating both an exemplary in-order fetch, decode, retire pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention.

[0005] **Figure 1B** is a block diagram illustrating both an exemplary embodiment of an in-order fetch, decode, retire 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.

[0006] **Figure 2** is a block diagram of a single core processor and a multicore processor with integrated memory controller and graphics according to embodiments of the invention;

[0007] **Figure 3** illustrates a block diagram of a system in accordance with one embodiment of the present invention;

[0008] **Figure 4** illustrates a block diagram of a second system in accordance with an embodiment of the present invention;

[0009] **Figure 5** illustrates a block diagram of a third system in accordance with an embodiment of the present invention;

[0010] **Figure 6** illustrates a block diagram of a system on a chip (SoC) in accordance with an embodiment of the present invention;

[0011] **Figure 7** illustrates 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;

[0012] **Figures 8A and 8B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

[0013] **Figures 9A-D** are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention; and

[0014] **Figure 10** is a block diagram of a register architecture according to one embodiment of the invention;

[0015] **Figure 11A** is a block diagram of a single processor core, along with its connection to the on-die interconnect network and with its local subset of the Level 2 (L2) cache, according to embodiments of the invention; and

[0016] **Figure 11B** is an expanded view of part of the processor core in **Figure 9A** according to embodiments of the invention.

[0017] **Figures 12-15** are flow diagrams illustrating a fused multiply-multiply operation, according to the embodiments of the invention.

[0018] **Figure 16** is a flow diagram of the method of a fused multiply-multiply operation, according to the embodiments of the invention.

[0019] **Figure 17** is a flow diagram illustrating the data interface in a processing device.

[0020] **Figure 18** is a flow diagram illustrating a first alternative exemplary data flow for an implementation of a fused multiply-multiply operation in a processing device.

[0021] **Figure 19** is a flow diagram illustrating a second alternative exemplary data flow for an implementation of a fused multiply-multiply operation in a processing device.

DETAILED DESCRIPTION

[0022] When working with SIMD data, there are circumstances where it would be beneficial to reduce the total instructions count and improve power efficiency, especially for small cores. In particular, an instruction implementing a fused multiply-multiply operation for floating-point data types allows reduction in the total instruction count and reduced workload power requirements.

[0023] In the following description, numerous specific details are set forth. However, it is understood that embodiments of the invention may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description. It will be appreciated, however, by one skilled in the art that the invention may be practiced without such specific details. Those of ordinary skill in the art, with the included descriptions, will be able to implement appropriate functionality without undue experimentation.

[0024] References in the specification to “one embodiment,” “an embodiment,” “an example embodiment,” etc., indicate that the embodiment described may include a particular feature, structure, or characteristic, but every embodiment may not necessarily include the particular feature, structure, or characteristic. Moreover, such phrases are not necessarily referring to the same embodiment. Further, when a particular feature, structure, or characteristic is described in connection with an embodiment, it is submitted that it is within the knowledge of one skilled in the art to effect such feature, structure, or characteristic in connection with other embodiments whether or not explicitly described.

[0025] In the following description and claims, the terms “coupled” and “connected,” along with their derivatives, may be used. It should be understood that these terms are not intended as synonyms for each other. “Coupled” is used to indicate that two or more elements, which may or may not be in direct physical or electrical contact with each other, co-operate or interact with each other. “Connected” is used to indicate the establishment of communication between two or more elements that are coupled with each other.

[0026] Instruction Set

[0027] An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, and may include the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). The term instruction generally refers herein to macro-instructions – that is instructions that are provided to the processor (or instruction converter that

translates (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morphs, emulates, or otherwise converts an instruction to one or more other instructions to be processed by the processor) for execution – as opposed to micro-instructions or micro-operations (micro-ops) – that is the result of a processor's decoder decoding macro-instructions.

[0028] The ISA is distinguished from the microarchitecture, which is the internal design of the processor implementing the instruction set. Processors with different microarchitectures can share a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale CA implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. For example, the same register architecture of the ISA may be implemented in different ways in different microarchitectures using well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB), and a retirement register file; the use of multiple maps and a pool of registers), etc. Unless otherwise specified, the phrases register architecture, register file, and register are used herein to refer to that which is visible to the software/programmer and the manner in which instructions specify registers. Where a specificity is desired, the adjective logical, architectural, or software visible will be used to indicate registers/files in the register architecture, while different adjectives will be used to designation registers in a given microarchitecture (e.g., physical register, reorder buffer, retirement register, register pool).

[0029] An instruction set includes one or more instruction formats. A given instruction format defines various fields (number of bits, location of bits) to specify, among other things, the operation to be performed (opcode) and the operand(s) on which that operation is to be performed. Some instruction formats are further broken down though the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD

instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands.

[0030] Scientific, financial, auto-vectorized general purpose, RMS (recognition, mining, and synthesis), and visual and multimedia applications (e.g., 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation) often require the same operation to be performed on a large number of data items (referred to as "data parallelism"). Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data items. SIMD technology is especially suited to processors that can logically divide the bits in a register into a number of fixed-sized data elements, each of which represents a separate value. For example, the bits in a 256-bit register may be specified as a source operand to be operated on as four separate 64-bit packed data elements (quad-word (Q) size data elements), eight separate 32-bit packed data elements (double word (D) size data elements), sixteen separate 16-bit packed data elements (word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). This type of data is referred to as packed data type or vector data type, and operands of this data type are referred to as packed data operands or vector operands. In other words, a packed data item or vector refers to a sequence of packed data elements, and a packed data operand or a vector operand is a source or destination operand of a SIMD instruction (also known as a packed data instruction or a vector instruction).

[0031] By way of example, one type of SIMD instruction specifies a single vector operation to be performed on two source vector operands in a vertical fashion to generate a destination vector operand (also referred to as a result vector operand) of the same size, with the same number of data elements, and in the same data element order. The data elements in the source vector operands are referred to as source data elements, while the data elements in the destination vector operand are referred to a destination or result data elements. These source vector operands are of the same size and contain data elements of the same width, and thus they contain the same number of data elements. The source data elements in the same bit positions in the two source vector operands form pairs of data elements (also referred to as corresponding data elements; that is, the data element in data element position 0 of each source operand correspond, the data element in data element position 1 of each source operand correspond, and so on). The operation specified by that SIMD instruction is performed separately on each of these pairs of

source data elements to generate a matching number of result data elements, and thus each pair of source data elements has a corresponding result data element. Since the operation is vertical and since the result vector operand is the same size, has the same number of data elements, and the result data elements are stored in the same data element order as the source vector operands, the result data elements are in the same bit positions of the result vector operand as their corresponding pair of source data elements in the source vector operands. In addition to this exemplary type of SIMD instruction, there are a variety of other types of SIMD instructions (e.g., that has only one or has more than two source vector operands, that operate in a horizontal fashion, that generates a result vector operand that is of a different size, that has a different size data elements, and/or that has a different data element order). It should be understood that the term destination vector operand (or destination operand) is defined as the direct result of performing the operation specified by an instruction, including the storage of that destination operand at a location (be it a register or at a memory address specified by that instruction) so that it may be accessed as a source operand by another instruction (by specification of that same location by the another instruction).

[0032] The SIMD technology, such as that employed by the Intel® Core™ processors having an instruction set including x86, MMX™, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, and SSE4.2 instructions, has enabled a significant improvement in application performance. An additional set of SIMD extensions, referred to the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme, has been , has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developers Manual, October 2011; and see Intel® Advanced Vector Extensions Programming Reference, June 2011).

[0033] **Figure 1A** is a block diagram illustrating both an exemplary in-order fetch, decode, retire pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. **Figure 1B** is a block diagram illustrating both an exemplary embodiment of an in-order fetch, decode, retire 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 1A-B** illustrate the in-order portions of the pipeline and core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core.

[0034] In **Figure 1A**, a processor pipeline 100 includes a fetch stage 102, a length decode stage 104, a decode stage 106, an allocation stage 108, a renaming stage 110, a scheduling (also

known as a dispatch or issue) stage 112, a register read/memory read stage 114, an execute stage 116, a write back/memory write stage 118, an exception handling stage 122, and a commit stage 124. **Figure 1B** shows processor core 190 including a front end unit 130 coupled to an execution engine unit 150, and both are coupled to a memory unit 170. The core 190 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 190 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.

[0035] The front end unit 130 includes a branch prediction unit 132 coupled to an instruction cache unit 134, which is coupled to an instruction translation lookaside buffer (TLB) 136, which is coupled to an instruction fetch unit 138, which is coupled to a decode unit 140. The decode unit 140 (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 140 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 190 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 140 or otherwise within the front end unit 130). The decode unit 140 is coupled to a rename/allocator unit 152 in the execution engine unit 150.

[0036] The execution engine unit 150 includes the rename/allocator unit 152 coupled to a retirement unit 154 and a set of one or more scheduler unit(s) 156. The scheduler unit(s) 156 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 156 is coupled to the physical register file(s) unit(s) 158. Each of the physical register file(s) units 158 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 158 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) 158 is overlapped by the retirement unit 154 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.).

[0037] The retirement unit 154 and the physical register file(s) unit(s) 158 are coupled to the execution cluster(s) 160. The execution cluster(s) 160 includes a set of one or more execution units 162 and a set of one or more memory access units 164. The execution units 162 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) 156, physical register file(s) unit(s) 158, and execution cluster(s) 160 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) 164). 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.

[0038] The set of memory access units 164 is coupled to the memory unit 170, which includes a data TLB unit 172 coupled to a data cache unit 174 coupled to a level 2 (L2) cache unit 176. In one exemplary embodiment, the memory access units 164 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 172 in the memory unit 170. The instruction cache unit 134 is further coupled to a level 2 (L2) cache unit 176 in the memory unit 170. The L2 cache unit 176 is coupled to one or more other levels of cache and eventually to a main memory.

[0039] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 100 as follows: 1) the instruction fetch 138 performs the fetch and length decoding stages 102 and 104; 2) the decode unit 140 performs the decode stage 106; 3) the rename/allocator unit 152 performs the allocation stage 108 and renaming stage 110; 4) the scheduler unit(s) 156 performs the schedule stage 112; 5) the physical register file(s)

unit(s) 158 and the memory unit 170 perform the register read/memory read stage 114; the execution cluster 160 perform the execute stage 116; 6) the memory unit 170 and the physical register file(s) unit(s) 158 perform the write back/memory write stage 118; 7) various units may be involved in the exception handling stage 122; and 8) the retirement unit 154 and the physical register file(s) unit(s) 158 perform the commit stage 124.

The core 190 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 190 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), described below), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[0040] 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).

[0041] 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 134/174 and a shared L2 cache unit 176, 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.

Figure 2 is a block diagram of a processor 200 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 2** illustrate a processor 200 with a single core 202A, a system agent 210, a set of one or more bus controller units 216, while the optional addition of the dashed lined boxes illustrates an alternative processor 200 with multiple cores 202A-N, a set of one or more integrated memory controller unit(s) 214 in the system agent

unit 210, and special purpose logic 208.

[0042] Thus, different implementations of the processor 200 may include: 1) a CPU with the special purpose logic 208 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 202A-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 202A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 202A-N being a large number of general purpose in-order cores. Thus, the processor 200 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 200 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.

[0043] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 206, and external memory (not shown) coupled to the set of integrated memory controller units 214. The set of shared cache units 206 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 212 interconnects the integrated graphics logic 208, the set of shared cache units 206, and the system agent unit 210/integrated memory controller unit(s) 214, 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 206 and cores 202A-N.

[0044] In some embodiments, one or more of the cores 202A-N are capable of multi-threading. The system agent 210 includes those components coordinating and operating cores 202A-N. The system agent unit 210 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 202A-N and the integrated graphics logic 208. The display unit is for driving one or more externally connected displays. The cores 202A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 202A-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. In one embodiment, the cores 202A-N are heterogeneous and include both the “small” cores and “big” cores described below.

[0045] **Figures 3-6** 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.

[0046] Referring now to **Figure 3**, shown is a block diagram of a system 300 in accordance with one embodiment of the present invention. The system 300 may include one or more processors 310, 315, which are coupled to a controller hub 320. In one embodiment the controller hub 320 includes a graphics memory controller hub (GMCH) 390 and an Input/Output Hub (IOH) 350 (which may be on separate chips); the GMCH 390 includes memory and graphics controllers to which are coupled memory 340 and a coprocessor 345; the IOH 350 is couples input/output (I/O) devices 360 to the GMCH 390. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 340 and the coprocessor 345 are coupled directly to the processor 310, and the controller hub 320 in a single chip with the IOH 350.

[0047] The optional nature of additional processors 315 is denoted in **Figure 3** with broken lines. Each processor 310, 315 may include one or more of the processing cores described herein and may be some version of the processor 200. The memory 340 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 320 communicates with the processor(s) 310, 315 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 395. In one embodiment, the coprocessor 345 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 320 may include an integrated graphics accelerator. There can be a variety of differences between the physical resources 310, 315 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

[0048] In one embodiment, the processor 310 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 310 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 345. Accordingly, the processor 310 issues these coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 345. Coprocessor(s) 345 accept and execute the received coprocessor instructions.

[0049] Referring now to **Figure 4**, shown is a block diagram of a first more specific exemplary system 400 in accordance with an embodiment of the present invention. As shown in **Figure 4**, multiprocessor system 400 is a point-to-point interconnect system, and includes a first processor 470 and a second processor 480 coupled via a point-to-point interconnect 450. Each of processors 470 and 480 may be some version of the processor 200. In one embodiment of the invention, processors 470 and 480 are respectively processors 310 and 315, while coprocessor 438 is coprocessor 345. In another embodiment, processors 470 and 480 are respectively processor 310 coprocessor 345.

[0050] Processors 470 and 480 are shown including integrated memory controller (IMC) units 472 and 482, respectively. Processor 470 also includes as part of its bus controller units point-to-point (P-P) interfaces 476 and 478; similarly, second processor 480 includes P-P interfaces 486 and 488. Processors 470, 480 may exchange information via a point-to-point (P-P) interface 450 using P-P interface circuits 478, 488. As shown in **Figure 4**, IMCs 472 and 482 couple the processors to respective memories, namely a memory 432 and a memory 434, which may be portions of main memory locally attached to the respective processors. Processors 470, 480 may each exchange information with a chipset 490 via individual P-P interfaces 452, 454 using point to point interface circuits 476, 494, 486, 498. Chipset 490 may optionally exchange information with the coprocessor 438 via a high-performance interface 439. In one embodiment, the coprocessor 438 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.

[0051] 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 490 may be coupled to a first bus 416 via an interface 496. In one embodiment, first bus 416 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.

[0052] As shown in **Figure 4**, various I/O devices 414 may be coupled to first bus 416, along with a bus bridge 418 which couples first bus 416 to a second bus 420. In one embodiment, one or more additional processor(s) 415, 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 416. In one embodiment, second bus 420 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 420 including, for example, a keyboard and/or mouse 422, communication devices 427 and a storage unit 428 such as a disk drive or other mass storage device which may include instructions/code and data 430, in one embodiment. Further, an audio I/O 424 may be coupled to the second bus 420. Note that other architectures are possible. For example, instead of the point-to-point architecture of **Figure 4**, a system may implement a multi-drop bus or other such architecture.

[0053] Referring now to **Figure 5**, shown is a block diagram of a second more specific exemplary system 500 in accordance with an embodiment of the present invention. Like elements in **Figures 4 and 5** bear like reference numerals, and certain aspects of **Figure 4** have been omitted from **Figure 5** in order to avoid obscuring other aspects of **Figure 5**. **Figure 5** illustrates that the processors 470, 480 may include integrated memory and I/O control logic ("CL") 472 and 482, respectively. Thus, the CL 472, 482 include integrated memory controller units and include I/O control logic. **Figure 5** illustrates that not only are the memories 432, 434 coupled to the CL 472, 482, but also that I/O devices 514 are also coupled to the control logic 472, 482. Legacy I/O devices 515 are coupled to the chipset 490.

[0054] Referring now to **Figure 6**, shown is a block diagram of a SoC 600 in accordance with an embodiment of the present invention. Similar elements in **Figure 2** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 6**, an interconnect unit(s) 602 is coupled to: an application processor 610 which includes a set of one or more cores 202A-N and shared cache unit(s) 206; a system agent unit 210; a bus controller unit(s) 216; an integrated memory controller unit(s) 214; a set or one or more coprocessors 620 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; a static random access memory (SRAM) unit 630; a direct memory access (DMA) unit 632; and a display unit 640 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 620 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.

[0055] 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 430 illustrated in **Figure 4**, 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.

[0056] 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.

[0057] 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. 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.

[0058] **Figure 7** 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 7** shows a program in a high level language 702 may be compiled using an x86 compiler 704 to generate x86 binary code 706 that may be natively executed by a processor with at least one x86 instruction set core 716. The processor with at least one x86 instruction set core 716 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 704 represents a compiler that is operable to generate x86 binary code 706 (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 716.

[0059] Similarly, **Figure 7** shows the program in the high level language 702 may be compiled using an alternative instruction set compiler 708 to generate alternative instruction set binary code 710 that may be natively executed by a processor without at least one x86 instruction set core 714 (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 712 is used to convert the x86 binary

code 706 into code that may be natively executed by the processor without an x86 instruction set core 714. This converted code is not likely to be the same as the alternative instruction set binary code 710 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 712 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 706.

[0060] Exemplary Instruction Formats

[0061] Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed. 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.

[0062] **Figures 8A-8B** are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. **Figure 8A** 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 8B** 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 800 for which are defined class A and class B instruction templates, both of which include no memory access 805 instruction templates and memory access 820 instruction templates.

[0063] 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).

[0064] The class A instruction templates in **Figure 8A** include: 1) within the no memory access 805 instruction templates there is shown a no memory access, full round control type operation 810 instruction template and a no memory access, data transform type operation 815 instruction template; and 2) within the memory access 820 instruction templates there is shown a memory access, temporal 825 instruction template and a memory access, non-temporal 830 instruction template. The class B instruction templates in **Figure 8B** include: 1) within the no memory access 805 instruction templates there is shown a no memory access, write mask control, partial round control type operation 812 instruction template and a no memory access, write mask control, vsize type operation 817 instruction template; and 2) within the memory access 820 instruction templates there is shown a memory access, write mask control 827 instruction template. The generic vector friendly instruction format 800 includes the following fields listed below in the order illustrated in **Figures 8A-8B**.

[0065] Format field 840 – 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.

[0066] Base operation field 842 – its content distinguishes different base operations.

[0067] Register index field 844 – 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).

[0068] Modifier field 846 – 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 805 instruction templates and memory access 820 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.

[0069] Augmentation operation field 850 – 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 868, an alpha field 852, and a beta field 854. The augmentation operation field 850 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

[0070] Scale field 860 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base}$).

[0071] Displacement Field 862A – its content is used as part of memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{displacement}$).

[0072] Displacement Factor Field 862B (note that the juxtaposition of displacement field 862A directly over displacement factor field 862B indicates one or the other is used) – its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N) – where N is the number of bytes in the memory access (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{scaled displacement}$).

Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field 874 (described herein) and the data manipulation field 854C. The displacement field 862A and the displacement factor field 862B are optional in the sense that they are not used for the no memory access 805 instruction templates and/or different embodiments may implement only one or none of the two.

[0073] Data element width field 864 – 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.

[0074] Write mask field 870 – 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 870 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 870 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 870 content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's 870 content to directly specify the masking to be performed.

[0075] Immediate field 872 – 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.

[0076] Class field 868 – its content distinguishes between different classes of instructions. With reference to **Figures 8A-B**, the contents of this field select between class A and class B instructions. In **Figures 8A-B**, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 868A and class B 868B for the class field 868 respectively in **Figures 8A-B**).

[0077] **Instruction Templates of Class A**

[0078] In the case of the non-memory access 805 instruction templates of class A, the alpha field 852 is interpreted as an RS field 852A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 852A.1 and data

transform 852A.2 are respectively specified for the no memory access, round type operation 810 and the no memory access, data transform type operation 815 instruction templates), while the beta field 854 distinguishes which of the operations of the specified type is to be performed. In the no memory access 805 instruction templates, the scale field 860, the displacement field 862A, and the displacement scale field 862B are not present.

[0079] No-Memory Access Instruction Templates – Full Round Control Type Operation

[0080] In the no memory access full round control type operation 810 instruction template, the beta field 854 is interpreted as a round control field 854A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field 854A includes a suppress all floating point exceptions (SAE) field 856 and a round operation control field 858, 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 858).

[0081] SAE field 856 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 856 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.

[0082] Round operation control field 858 – 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 858 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 850 content overrides that register value.

[0083] No Memory Access Instruction Templates – Data Transform Type Operation

[0084] In the no memory access data transform type operation 815 instruction template, the beta field 854 is interpreted as a data transform field 854B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

[0085] In the case of a memory access 820 instruction template of class A, the alpha field 852 is interpreted as an eviction hint field 852B, whose content distinguishes which one of the eviction hints is to be used (in **Figure 8A**, temporal 852B.1 and non-temporal 852B.2 are respectively specified for the memory access, temporal 825 instruction template and the memory access, non-temporal 830 instruction template), while the beta field 854 is interpreted as a data

manipulation field 854C, 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 820 instruction templates include the scale field 860, and optionally the displacement field 862A or the displacement scale field 862B. 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.

[0086] Memory Access Instruction Templates – Temporal

[0087] 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.

[0088] Memory Access Instruction Templates – Non-Temporal

[0089] 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.

[0090] Instruction Templates of Class B

[0091] In the case of the instruction templates of class B, the alpha field 852 is interpreted as a write mask control (Z) field 852C, whose content distinguishes whether the write masking controlled by the write mask field 870 should be a merging or a zeroing. In the case of the non-memory access 805 instruction templates of class B, part of the beta field 854 is interpreted as an RL field 857A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 857A.1 and vector length (VSIZE) 857A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 812 instruction template and the no memory access, write mask control, VSIZE type operation 817 instruction template), while the rest of the beta field 854 distinguishes which of the operations of the specified type is to be performed. In the no memory access 805 instruction templates, the scale field 860, the displacement field 862A, and the displacement scale field 862B are not present. In the no memory access, write mask control, partial round control type operation 810 instruction template, the rest of the beta field 854 is interpreted as a round operation field 859A 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).

[0092] Round operation control field 859A – just as round operation control field 858, 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 859A 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 850 content overrides that register value. In the no memory access, write mask control, VSIZE type operation 817 instruction template, the rest of the beta field 854 is interpreted as a vector length field 859B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

[0093] In the case of a memory access 820 instruction template of class B, part of the beta field 854 is interpreted as a broadcast field 857B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 854 is interpreted the vector length field 859B. The memory access 820 instruction templates include the scale field 860, and optionally the displacement field 862A or the displacement scale field 862B.

[0094] In the case of a memory access 820 instruction template of class B, part of the beta field 854 is interpreted as a broadcast field 857B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 854 is interpreted the vector length field 859B. The memory access 820 instruction templates include the scale field 860, and optionally the displacement field 862A or the displacement scale field 862B. With regard to the generic vector friendly instruction format 800, a full opcode field 874 is shown including the format field 840, the base operation field 842, and the data element width field 864. While one embodiment is shown where the full opcode field 874 includes all of these fields, the full opcode field 874 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 874 provides the operation code (opcode). The augmentation operation field 850, the data element width field 864, and the write mask field 870 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.

[0095] 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.

[0096] 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 implemented 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.

[0097] **Figure 9A-D** are block diagrams illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. **Figure 9** shows a specific vector friendly instruction format 900 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 900 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 8** into which the fields from **Figure 9** map are illustrated.

[0098] It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 900 in the context of the generic

vector friendly instruction format 800 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 900 except where claimed. For example, the generic vector friendly instruction format 800 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 900 is shown as having fields of specific sizes. By way of specific example, while the data element width field 864 is illustrated as a one bit field in the specific vector friendly instruction format 900, the invention is not so limited (that is, the generic vector friendly instruction format 800 contemplates other sizes of the data element width field 864). The generic vector friendly instruction format 800 includes the following fields listed below in the order illustrated in **Figure 9A**.

[0099] EVEX Prefix (Bytes 0-3) 902 - is encoded in a four-byte form.

[00100] Format Field 840 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 840 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.

[00101] REX field 905 (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 857BEX 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 811B, 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.

[00102] REX' field 810 – this is the first part of the REX' field 810 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.

[00103] Opcode map field 915 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

[00104] Data element width field 864 (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).

[00105] EVEX.vvvv 920 (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 811b. Thus, EVEX.vvvv field 920 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.

[00106] EVEX.U 868 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.

[00107] Prefix encoding field 925 (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.

[00108] Alpha field 852 (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.

[00109] Beta field 854 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s₂₋₀, EVEX.r₂₋₀, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with $\beta\beta\beta$) – as previously described, this field is context specific.

[00110] REX' field 810 – 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.

[00111] Write mask field 870 (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).

[00112] Real Opcode Field 930 (Byte 4) – it is also known as the opcode byte. Part of the opcode is specified in this field.

[00113] MOD R/M Field 940 (Byte 5) includes MOD field 942, Reg field 944, and R/M field 946. As previously described, the MOD field's 942 content distinguishes between memory access and non-memory access operations. The role of Reg field 944 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 946 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.

[00114] Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 850 content is used for memory address generation. SIB.xxx 954 and SIB.bbb 956 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[00115] Displacement field 862A (Bytes 7-10) – when MOD field 942 contains 10, bytes 7-10 are the displacement field 862A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

[00116] Displacement factor field 862B (Byte 7) – when MOD field 942 contains 01, byte 7 is the displacement factor field 862B. 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 862B is a reinterpretation of disp8; when using displacement factor field 862B, 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 is 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 862B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 862B 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 $\text{disp8} \times N$. 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 872 operates as previously described.

[00117] Full Opcode Field

[00118] Figure 9B is a block diagram illustrating the fields of the specific vector friendly instruction format 900 that make up the full opcode field 874 according to one embodiment of the invention. Specifically, the full opcode field 874 includes the format field 840, the base operation field 842, and the data element width (W) field 864. The base operation field 842 includes the prefix encoding field 925, the opcode map field 915, and the real opcode field 930.

[00119] Register Index Field

[00120] Figure 9C is a block diagram illustrating the fields of the specific vector friendly instruction format 900 that make up the register index field 844 according to one embodiment of the invention. Specifically, the register index field 844 includes the REX field 905, the REX' field 910, the MODR/M.reg field 944, the MODR/M.r/m field 946, the VVVV field 920, xxx field 954, and the bbb field 956.

[00121] Augmentation Operation Field

[00122] Figure 9D is a block diagram illustrating the fields of the specific vector friendly instruction format 900 that make up the augmentation operation field 850 according to one embodiment of the invention. When the class (U) field 868 contains 0, it signifies EVEX.U0 (class A 868A); when it contains 1, it signifies EVEX.U1 (class B 868B). When $U=0$ and the MOD field 942 contains 11 (signifying a no memory access operation), the alpha field 852 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 852A. When the rs field 852A contains a 1 (round 852A.1), the beta field 854 (EVEX byte 3, bits [6:4] – SSS) is interpreted as the round control field 854A. The round control field 854A includes a one bit SAE field 856 and a two bit round operation field 858. When the rs field 852A contains a 0 (data transform 852A.2), the

beta field 854 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data transform field 854B. When U=0 and the MOD field 942 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 852 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 852B and the beta field 854 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 854C.

[00123] When U=1, the alpha field 852 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 852C. When U=1 and the MOD field 942 contains 11 (signifying a no memory access operation), part of the beta field 854 (EVEX byte 3, bit [4]- S₀) is interpreted as the RL field 857A; when it contains a 1 (round 857A.1) the rest of the beta field 854 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the round operation field 859A, while when the RL field 857A contains a 0 (VSIZE 857.A2) the rest of the beta field 854 (EVEX byte 3, bit [6-5]- S₂₋₁) is interpreted as the vector length field 859B (EVEX byte 3, bit [6-5]- L₁₋₀). When U=1 and the MOD field 1442 contains 00, 01, or 10 (signifying a memory access operation), the beta field 854 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 859B (EVEX byte 3, bit [6-5]- L₁₋₀) and the broadcast field 857B (EVEX byte 3, bit [4]- B).

[00124] **Figure 10** is a block diagram of a register architecture 1000 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 1010 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 900 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 859B	A (Figure 8A; U=0)	810, 815, 825, 830	zmm registers (the vector length is 64 byte)
	B (Figure 8B; U=1)	812	zmm registers (the vector length is 64 byte)

Instruction Templates that do include the vector length field 859B	B (Figure 8B; U=1)	817, 827	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 859B
--	--------------------	----------	--

[00125] In other words, the vector length field 859B 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 859B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 900 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.

[00126] Write mask registers 1015 - 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 1015 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.

[00127] General-purpose registers 1025 - 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.

[00128] Scalar floating point stack register file (x87 stack) 1045, on which is aliased the MMX packed integer flat register file 1050 - 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.

[00129] 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.

[00130] 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).

[00131] 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.

[00132] 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.

[00133] Embodiments of the invention may include various steps, which have been described above. The steps may be embodied in machine-executable instructions which may be used to cause a general-purpose or special-purpose processor to perform the steps. Alternatively, these steps may be performed by specific hardware components that contain hardwired logic for performing the steps, or by any combination of programmed computer components and custom hardware components.

[00134] As described herein, instructions may refer to specific configurations of hardware such as application specific integrated circuits (ASICs) configured to perform certain operations or having a predetermined functionality or software instructions stored in memory embodied in a non-transitory computer readable medium. Thus, the techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices (e.g., an end station, a network element, etc.). Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer machine-readable media, such as non-transitory computer machine-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer machine-readable communication media (e.g., electrical, optical, acoustical or other form of propagated signals – such as carrier waves, infrared signals, digital signals, etc.).

[00135] In addition, such electronic devices typically include a set of one or more processors coupled to one or more other components, such as one or more storage devices (non-transitory machine-readable storage media), user input/output devices (e.g., a keyboard, a touchscreen, and/or a display), and network connections. The coupling of the set of processors and other components is typically through one or more busses and bridges (also termed as bus controllers). The storage device and signals carrying the network traffic respectively represent one or more machine-readable storage media and machine-readable communication media. Thus, the storage device of a given electronic device typically stores code and/or data for execution on the set of one or more processors of that electronic device. Of course, one or more parts of an embodiment of the invention may be implemented using different combinations of software, firmware, and/or hardware.

[00136] Apparatus And Method For Performing Fused Multiply-Multiply Operation

[00137] As mentioned above, when working with vector/SIMD data, there are circumstances

where it would be beneficial to reduce the total instructions count and improve power efficiency, especially for small cores. In particular, an instruction implementing a fused multiply-multiply operation for floating-point data types allows reduction in the total instruction count and reduced workload power requirements.

[00138] **Figures 12-15** illustrate embodiments of a fused multiply-multiply operation on 512-bit vector/SIMD operands each to be operated on as 8 separate 64-bit packed-data elements containing single-precision floating point values. It should be noted, however, that the specific vector and packed-data element sizes illustrated in **Figures 12-15** are used merely for the purposes of illustration. The underlying principles of the invention may be implemented using any vector or packed-data element size. Referring to figures 12-15, the source 1 and source 2 operands (1205-1505 and 1201-1501, respectively) may be SIMD packed-data registers and the source 3 operands 1203-1503 may be SIMD packed-data registers or locations in memory. Responsive to a fused multiply-multiply operation, rounding control is set depending on the vector format. In the embodiment described herein, rounding control may be set according to the instruction templates of Class A **Figure 8A** (including a no memory access, round type operation 810) or the instruction templates of Class B **Figure 8B** (including a no memory access, write mask control, partial round control type operation 812).

[00139] As illustrated in **Figure 12**, an initial packed-data element occupying the least significant 64-bits of the source 2 operand (e.g., the packed-data element having the value 7 in 1201) is multiplied by a corresponding packed-data element from the source 3 operand (e.g., the packed-data element having the value 15 in 1203), generating a first result data element. The first result data element is rounded and multiplied by the corresponding packed-data element of the source 1/destination operand (e.g., the packed-data element having the value 8 in 1205), generating a second result data element. The second result data element is rounded and written back into the same packed-data element position of the source 1/destination operand 1207 (e.g., packed-data element having the value 840 1215). In one embodiment, an immediate byte value is encoded in the source 3 operand, wherein the least significant 3 bits 1209 each contain either a one or a zero, assigning a positive or negative value to each of the respective packed-data elements of each operand for the fused multiply-multiply operation. Immediate bits [7:3] 1211 of the immediate byte encode a register or location in memory of source 3. The fused multiply-multiply operation repeats for each respective packed-data element of corresponding source operands, wherein each source operand includes a plurality of packed-data elements (e.g., for a corresponding set of operands, each having 8 packed-data elements with vector operand lengths

of 512 bits, where each packed-data element is 64 bits wide).

[00140] Another embodiment involves four packed-data operands. Similar to **Figure 12**, **Figure 13** illustrates an initial packed-data element occupying the least significant 64-bits of the source 2 operand 1301. The initial packed-data element is multiplied by a corresponding packed-data element from the source 3 operand 1303, generating a first result data element. The first result data element is rounded and multiplied by the corresponding packed-data element of the source 1 operand 1305, generating a second result data element. In contrast to **Figure 12**, the second result data element, after being rounded, is written into a corresponding packed-data element of a fourth packed-data operand, the destination operand 1307 (e.g., packed-data element having the value 840 1315). In one embodiment, an immediate byte value is encoded in the source 3 operand, wherein the least significant 3 bits 1309 each contain either a one or a zero, assigning a positive or negative value, respectively, to each of the packed-data elements of each operand for the fused multiply-multiply operation. Immediate bits [7:3] 1311 of the immediate byte encode a register or location in memory of source 3. The fused multiply-multiply operation repeats for each respective packed-data element of corresponding source operands, wherein each source operand includes a plurality of packed-data elements (e.g., for a corresponding set of operands, each having 8 packed-data elements with vector operand lengths of 512 bits, where each packed-data element is 64 bits wide).

[00141] **Figure 14** illustrates an alternative embodiment including the addition of a write-mask register K1 1419 having packed-data element widths of 64 bits. The lower 8 bits of the write-mask register K1 include a mix of ones and zeros. Each of the lower 8 bit positions in the write-mask register K1 corresponds to one of the packed-data element positions. For each packed-data element position in the source 1/destination operand 1407, it contains the content of that packed-data element position in the source 1/destination operand 1405 (e.g., packed-data element having the value 6 1421) or the result of the operation (e.g., packed-data element having the value 840 1415) depending on whether the corresponding bit position in the write-mask register K1 is a zero or one, respectively. In another embodiment, as illustrated in **Figure 15**, the source 1/destination operand 1405 is replaced with an additional source operand, the source 1 operand 1505 (e.g., for embodiments having four packed-data operands). In those embodiments, the destination operand 1507 contains the contents of the source 1 operand from before the operation in those of the packed-data element positions in which the corresponding bit positions of the mask register K1 are zero (e.g., the packed-data element having the value 6 1521) and contains the result of the operation in those of the packed-data element positions in which the

corresponding bit positions of the mask register K1 are 1 (e.g., packed-data element having the value 840 1515).

[00142] According to the embodiments of the fused multiply-multiply instruction described above, the operands may be encoded as follows with reference to **Figures 12-15 and 9A**. The destination operand 1207-1507 (also the source 1/destination operand in **Figures 12 and 14**) is a packed-data register and encoded in Reg field 944. The source 2 operand 1201-1501 is a packed-data register and encoded in VVVV field 920. In one embodiment, the source 3 operand 1203-1503 is a packed-data register and in another embodiment, it is a 64-bit floating point packed-data memory location. The source 3 operand may be encoded in Immediate field 872 or in R/M field 946.

[00143] **Figure 16** is a flow diagram illustrating exemplary steps followed by a processor while performing a fused multiply-multiply operation according to one embodiment. The method may be implemented within the context of the architectures described above but is not limited to any specific architectures. At step 1601, a decode unit (e.g., decode unit 140) receives an instruction and decodes the instruction to determine that a fused multiply-multiply operation is to be performed. The instruction may specify a set of three or four source packed-data operands, each having an array of N packed-data elements. The value of each packed-data element within each of the packed-data operands is positive or negative in accordance with a corresponding value in a bit position with an immediate byte (e.g., the least significant 3 bits in an immediate byte within the source 3 operand each containing either a one or a zero, assigning a positive or negative value, respectively, to each of the packed-data elements of each operand for the fused multiply-multiply operation). In some embodiments, the decoded fused multiply-multiply instruction is translated into micro-code for use on independent multiplication units.

[00144] At step 1603, decode unit 140 accesses registers (e.g., registers in physical register files unit 158), or a locations within memory (e.g., memory unit 170). Registers in physical register files unit 158, or memory locations in memory unit 170, may be accessed depending on the register address specified in the instruction. For example, a fused multiply-multiply operation may include SRC1, SRC2, SRC3, and DEST register addresses, where SRC1 is the address of the first source register, SRC2 is the address of the second source register, and SRC3 is the address of the third source register. DEST is the address of the destination register where the result data is stored. In some implementations, the storage location referenced by SRC1 is also used to store the result and is referred to as SRC1/DEST. In some implementations, anyone, or all, of SRC1, SRC2, SRC3, and DEST, defines a memory location in the addressable memory

space of a processor. For example, SRC3 may identify a memory location in memory unit 170, while SRC2 and SRC1/DEST identify a first and second register, respectively, in physical register files unit 158. For simplicity of the description herein, the embodiments will be described in relation to accessing physical register files. However, these accesses could be made to memory instead.

[00145] At step 1605, an execution unit (e.g., execution engine unit 150) is enabled to perform the fused multiply-multiply operation on the accessed data. In accordance with a fused multiply-multiply operation, an initial packed-data element of the source 2 operand is multiplied by a corresponding packed-data element from the source 3 operand, generating a first result data element. The first result data element is rounded and multiplied by the corresponding packed-data element of the source 1/destination operand, generating a second result data element. The second result data element is rounded and written back into the same packed-data element position of the source 1/destination operand. For an embodiment involving four packed-data operands, the second result data element, after being rounded, is written into a corresponding packed-data element of a fourth packed-data operand, the destination operand. In one embodiment, an immediate byte value is encoded in the source 3 operand, wherein the least significant 3 bits each contain either a one or a zero, assigning a positive or negative value to each of the respective packed-data elements of each operand for the fused multiply-multiply operation. Immediate bits [7:3] encode a register of source 3.

[00146] For embodiments including a write-mask register, each packed-data element position in the source 1/destination operand contains either the content of that packed-data element position in the source 1/destination or the result of the operation in accordance with the corresponding bit position in the write-mask register being a zero or one, respectively. The fused multiply-multiply operation repeats for each respective packed-data element of corresponding source operands, wherein each source operand includes a plurality of packed-data elements. According to the requirements of the instruction, the source 1/destination operand or destination operand may specify a register in the physical register files unit 158 where the result of the fused multiply-multiply operation is stored. At step 1607, the result of the fused multiply-multiply operation may be stored back into physical register files unit 158 or in a location in memory unit 170, according to the requirements of the instruction.

[00147] **Figure 17** illustrates an exemplary data flow for an implementation of a fused multiply-multiply operation. In one embodiment, execution unit 1705 of processing unit 1701 is a fused multiply-multiply unit 1705 and is coupled to physical register file unit 1703 to receive

the source operands from respective source registers. In one embodiment, the fused multiply-multiply unit is operable to perform the fused multiply-multiply operation on packed-data elements stored in registers specified by the first, second, and third source operands.

[00148] The fused multiply-multiply unit further includes a sub-circuit(s) for operating on packed-data elements from each of the source operands. Each sub-circuit multiplies one packed-data element from the source 2 operand (1201-1501) by a corresponding packed-data element of the source 3 operand (1203-1503), generating a first result data element. The first result data element is rounded and multiplied by the corresponding packed-data element of the source 1/destination operand or the source 1 operand (1205-1505), according to the instruction having three or four source operands, respectively, generating a second result data element. The second result data element is rounded and written back into the corresponding packed-data element position of the source 1/destination operand or destination operand (1207-1507). After the completion of the operation, the result within the source 1/destination operand or the destination operand may be written back to physical register files unit 1703, for example, in a write back or retirement stage.

[00149] **Figure 18** illustrates an alternative data flow for an implementation of a fused multiply-multiply operation. Similar to **Figure 17**, the execution unit 1807 of processing unit 1801 is a fused multiply-multiply unit 1807 and is operable to perform the fused multiply-multiply operation on packed-data elements stored in registers specified by the first, second, and third source operands. In one embodiment, a scheduler 1805 is coupled to the physical register files unit 1803 to receive the source operands from respective source registers and the scheduler is coupled to the fused multiply-multiply unit 1807. The scheduler 1805 receives the source operands from respective source registers in the physical register files unit 1803 and dispatches the source operands to the fused multiply-multiply unit 1807 for execution of the fused multiply-multiply operation.

[00150] In one embodiment, where there are not two fused multiply-multiply units nor two sub-circuits available for performing a single fused multiply-multiply instruction, a scheduler 1805 dispatches the instruction twice to the fused multiply-multiply unit, not dispatching the second instruction until the first instruction has completed (i.e., the scheduler 1805 dispatches the fused multiply-multiply instruction and waits for the one packed-data element from the source 2 operand (1201-1501) to be multiplied by a corresponding packed-data element of the source 3 operand (1203-1503), generating a first result data element; the scheduler then dispatches the fused multiply-multiply instruction a second time and the first result data element is rounded and

multiplied by the corresponding packed-data element of the source 1/destination operand or the source 1 operand (1205-1505), according to the instruction having three or four source operands, respectively, generating a second result data element). The second result data element is rounded and written back into the corresponding packed-data element position of the source1/destination operand or destination operand (1207-1507). After the completion of the operation, the result within the source 1/destination operand or the destination operand may be written back to physical register files unit 1803, for example, in a write back or retirement stage.

[00151] **Figure 19** illustrates another alternative data flow for an implementation of a fused multiply-multiply operation. Similar to **Figure 18**, the execution unit 1907 of processing unit 1901 is a fused multiply-multiply unit 1907 and is operable to perform the fused multiply-multiply operation on packed-data elements stored in registers specified by the first, second, and third source operands. In one embodiment, the physical register files unit 1903 is coupled to an additional execution unit that is also a fused multiply-multiply unit 1905 (also operable to perform the fused multiply-multiply operation on packed-data elements stored in registers specified by the first, second, and third source operands) and the two fused multiply-multiply units are in series (i.e., the output of fused multiply-multiply unit 1905 is coupled to the input of fused multiply-multiply unit 1907).

[00152] In one embodiment, a first fused multiply-multiply unit (i.e., fused multiply-multiply unit 1905) performs a multiplication of one packed-data element from the source 2 operand (1201-1501) by a corresponding packed-data element of the source 3 operand (1203-1503), generating a first result data element. In one embodiment, after the first result data element is rounded, a second fused multiply-multiply unit (i.e., fused multiply-multiply unit 1907) performs multiplication of the first result data element by the corresponding packed-data element of the source 1/destination operand or the source 1 operand (1205-1505), according to the instruction having three or four source operands, respectively, generating a second result data element. The second result data element is rounded and written back into the corresponding packed-data element position of the source1/destination operand or destination operand (1207-1507). After the completion of the operation, the result within the source 1/destination operand or the destination operand may be written back to physical register files unit 1903, for example, in a write back or retirement stage.

[00153] Throughout this detailed description, for the purposes of explanation, numerous specific details were set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the invention may be

practiced without some of these specific details. In certain instances, well known structures and functions were not described in elaborate detail in order to avoid obscuring the subject matter of the present invention. Accordingly, the scope and spirit of the invention should be judged in terms of the claims which follow.

CLAIMS

What is claimed is:

1. A processor comprising:
 - a first source register to store a first operand comprising a first plurality of packed data elements;
 - a second source register to store a second operand comprising a second plurality of packed data elements;
 - a third source register to store a third operand comprising a third plurality of packed data elements; and
 - fused multiply-multiply circuitry to interpret the plurality of packed data elements as positive or negative in accordance with a corresponding value in a bit position within an immediate value, the fused multiply-multiply circuitry to multiply a corresponding data element of the first plurality by a first result data element comprising a product of corresponding data elements of the second plurality and the third plurality, to generate a second result data element, the fused multiply-multiply circuitry to store the second result data element in a destination.
2. The processor of claim 1, wherein the fused multiply-multiply circuitry comprises a decode unit to decode a fused multiply-multiply instruction and an execution unit to execute the fused multiply-multiply instruction.
3. The processor as in claim 2, wherein the decode unit is to decode a single fused multiply-multiply instruction into a plurality of microoperations to be executed by the execution unit.
4. The processor as in claim 3, wherein the execution unit, having a plurality of sub-circuits, is to use the microoperations to interpret the plurality of packed data elements as positive or negative in accordance with a corresponding value in a bit position within an immediate value, multiply a corresponding data element of the first plurality to a first result data element comprising a product of corresponding data elements of the second plurality and the third plurality, generating a second result data element, and to store the

second result data element in a destination.

5. The processor of claim 1, wherein the first operand and the destination are a single register where the second result data element is stored.
6. The processor of claim 1, wherein the second result data element is written to the destination based on a value of a write-mask register of the processor.
7. The processor of claim 1, wherein to interpret the plurality of packed data elements as positive or negative, the fused multiply-multiply circuitry is to read a bit value in a first bit position of the immediate value corresponding to the first plurality of packed data elements to determine whether the first plurality of packed data elements are positive or negative, to read a bit value in a second bit position of the immediate value corresponding to the second plurality of packed data elements to determine whether the second plurality of packed data elements are positive or negative, and to read a bit value in a third bit position of the immediate value corresponding to the third plurality of packed data elements to determine whether the third plurality of packed data elements are positive or negative.
8. The processor as in claim 7, wherein the fused multiply-multiply circuitry is to further read a set of one or more bits other than the bits in the first, second, and third bit positions to determine a register or memory location of at least one of the operands.
9. A method comprising:
 - storing a first operand comprising a first plurality of packed data elements in a first source register;
 - storing a second operand comprising a second plurality of packed data elements in a second source register;
 - storing a third operand comprising a third plurality of packed data elements in a third source register;
 - interpreting the plurality of packed data elements as positive or negative in accordance with a corresponding value in a bit position within an immediate value of an instruction; and

multiplying a corresponding data element of the first plurality to a first result data element comprising a product of corresponding data elements of the second plurality and the third plurality, generating a second result data element, and storing the second result data element in a destination.

10. The method of claim 9, further comprising:

decoding by a decoder in a processor, the instruction specifying the first source register, the second source register, and the third source register; and

executing by an execution unit in the processor the instruction by interpreting the plurality of packed data elements as positive or negative in accordance with the corresponding value in bit positions within the immediate value.

11. The method as in claim 10, wherein the decoder is to decode a single instruction into a plurality of microoperations to be executed by the execution unit.

12. The method as in claim 11, further comprising:

interpreting using the microoperations by the execution unit, having a plurality of sub-circuits, the plurality of packed data elements as positive or negative in accordance with a corresponding value in a bit position within an immediate value, multiplying a corresponding data element of the first plurality to a first result data element comprising a product of corresponding data elements of the second plurality and the third plurality, generating a second result data element, and storing the second result data element in a destination.

13. The method of claim 9, wherein the first operand and the destination are a single register where the second result data element is stored.

14. The method of claim 9, wherein the second result data element is written to the destination based on a value of a write-mask register of the processor.

15. The method of claim 9, further comprising:

interpreting the plurality of packed data elements as positive or negative, by the

fused multiply-multiply circuitry reading a bit value in a first bit position of the immediate value corresponding to the first plurality of packed data elements, to determine whether the first plurality of packed data elements are positive or negative, reading a bit value in a second bit position of the immediate value corresponding to the second plurality of packed data elements to determine whether the second plurality of packed data elements are positive or negative, and reading a bit value in a third bit position of the immediate value corresponding to the third plurality of packed data elements to determine whether the third plurality of packed data elements are positive or negative.

16. The method as in claim 15, further comprising:

reading by the fused multiply-multiply circuitry a set of one or more bits other than the bits in the first, second, and third bit positions to determine a register or memory location of at least one of the operands.

17. A system comprising:

a memory unit coupled to a first storage location configured to store a first plurality of packed data elements; and

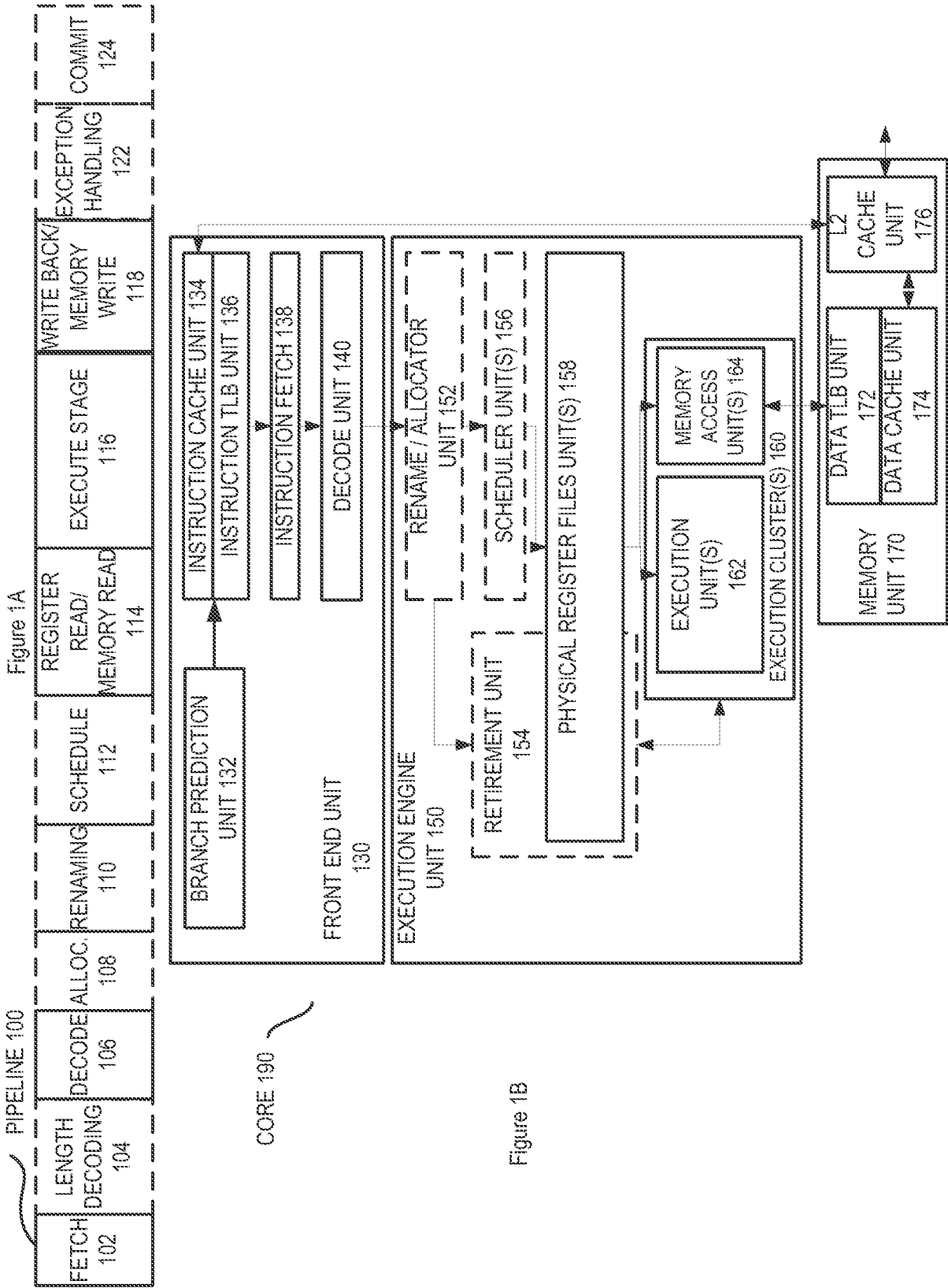
a processor coupled to the memory unit, the processor comprising:

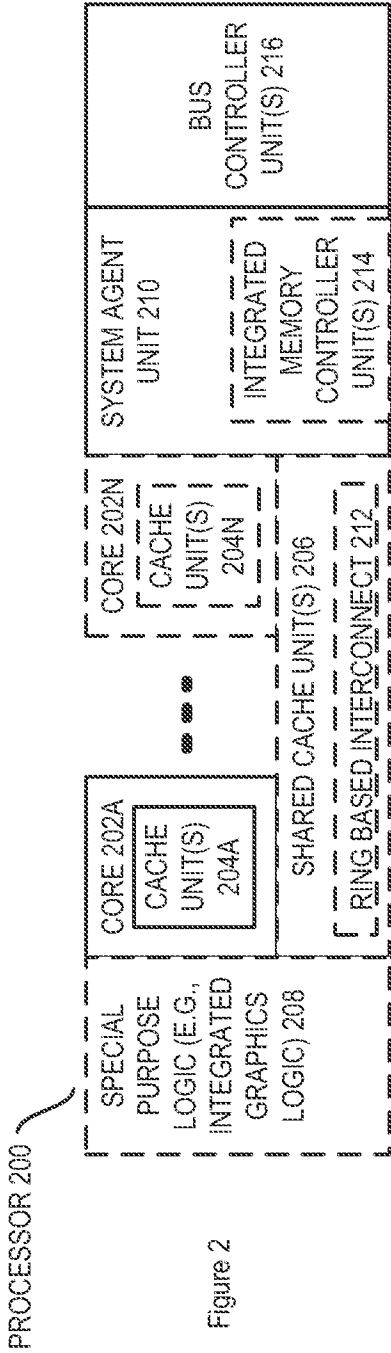
a register file unit configured to store a plurality of packed data operands, including a first source register to store a first operand comprising a first plurality of packed data elements, a second source register to store a second operand comprising a second plurality of packed data elements, and a third source register to store a third operand comprising a third plurality of packed data elements;

fused multiply-multiply circuitry to interpret the plurality of packed data elements as positive or negative in accordance with a corresponding value in a bit position within an immediate value, the fused multiply-multiply circuitry to multiply a corresponding data element of the first plurality by a first result data element comprising a product of corresponding data elements of the second plurality and the third plurality, to generate a second result data element, the fused multiply-multiply circuitry to store the second result data element in a destination.

18. The system of claim 17, wherein the fused multiply-multiply circuitry comprises a decode unit to decode a fused multiply-multiply instruction and an execution unit to execute the fused multiply-multiply instruction.
19. The system as in claim 18, wherein the decode unit is to decode a single fused multiply-multiply instruction into a plurality of microoperations to be executed by the execution unit.
20. The system as in claim 19, wherein the execution unit, having a plurality of sub-circuits, is to use the microoperations to interpret the plurality of packed data elements as positive or negative in accordance with a corresponding value in a bit position within an immediate value, multiply a corresponding data element of the first plurality by a first result data element comprising a product of corresponding data elements of the second plurality and the third plurality, generating a second result data element, and to store the second result data element in a destination.
21. The system of claim 17, wherein the first operand and the destination are a single register where the second result data element is stored.
22. The system of claim 17, wherein the second result data element is written to the destination based on a value of a write-mask register of the processor.
23. The system of claim 17, wherein to interpret the plurality of packed data elements as positive or negative, the fused multiply-multiply circuitry is to read a bit value in a first bit position of the immediate value corresponding to the first plurality of packed data elements to determine whether the first plurality of packed data elements are positive or negative, to read a bit value in a second bit position of the immediate value corresponding to the second plurality of packed data elements to determine whether the second plurality of packed data elements are positive or negative, and to read a bit value in a third bit position of the immediate value corresponding to the third plurality of packed data elements to determine whether the third plurality of packed data elements are positive or negative.

24. The system as in claim 23, wherein the fused multiply-multiply circuitry is to further read a set of one or more bits other than the bits in the first, second, and third bit positions to determine a register or memory location of at least one of the operands.





3/21

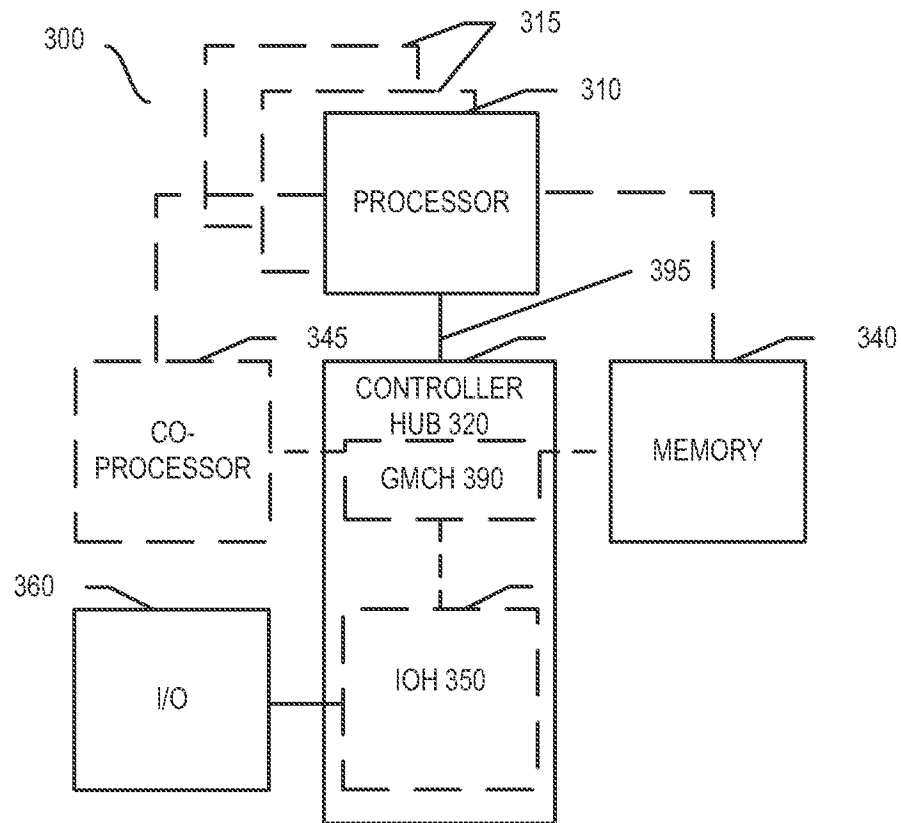


Figure 3

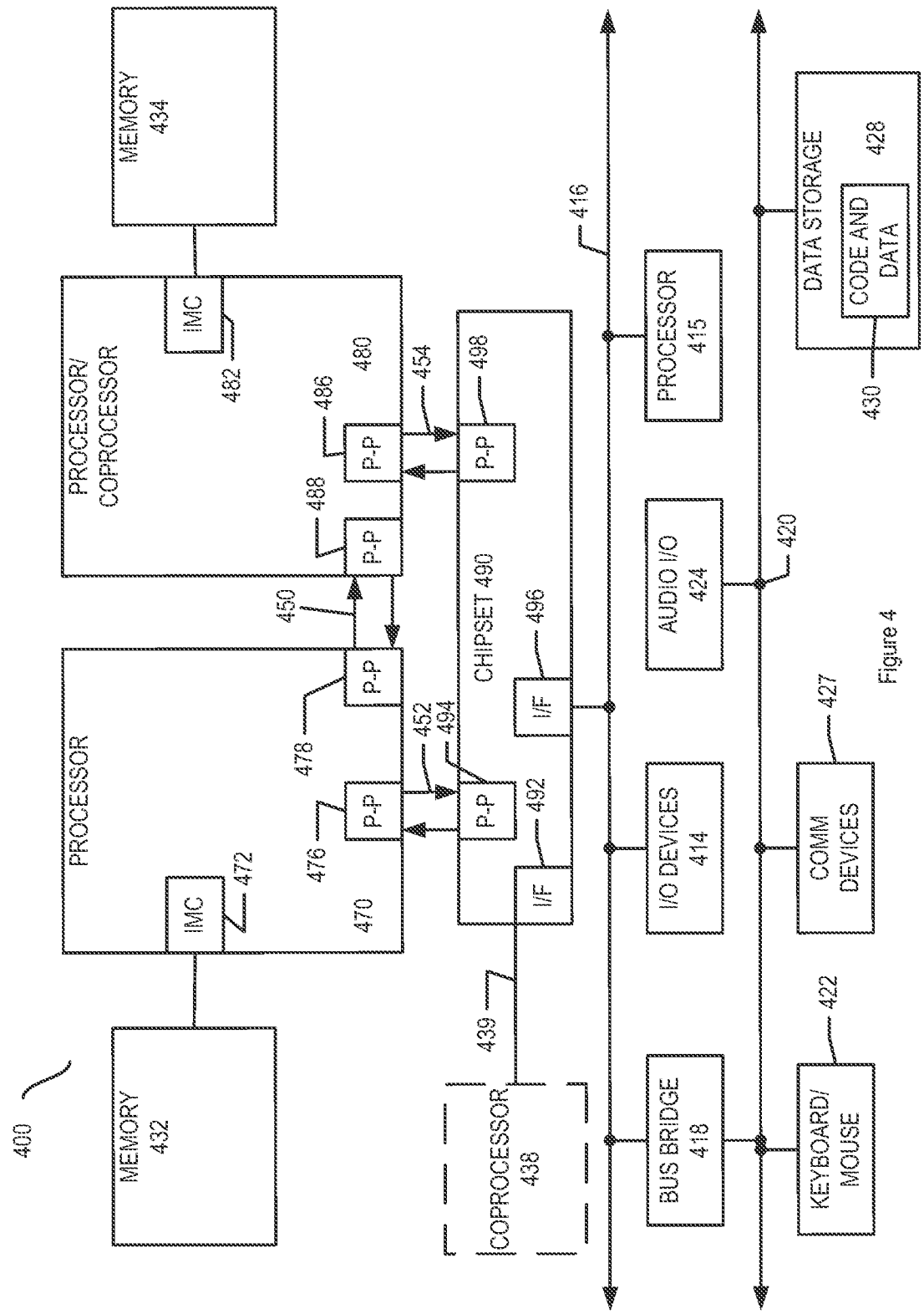


Figure 4

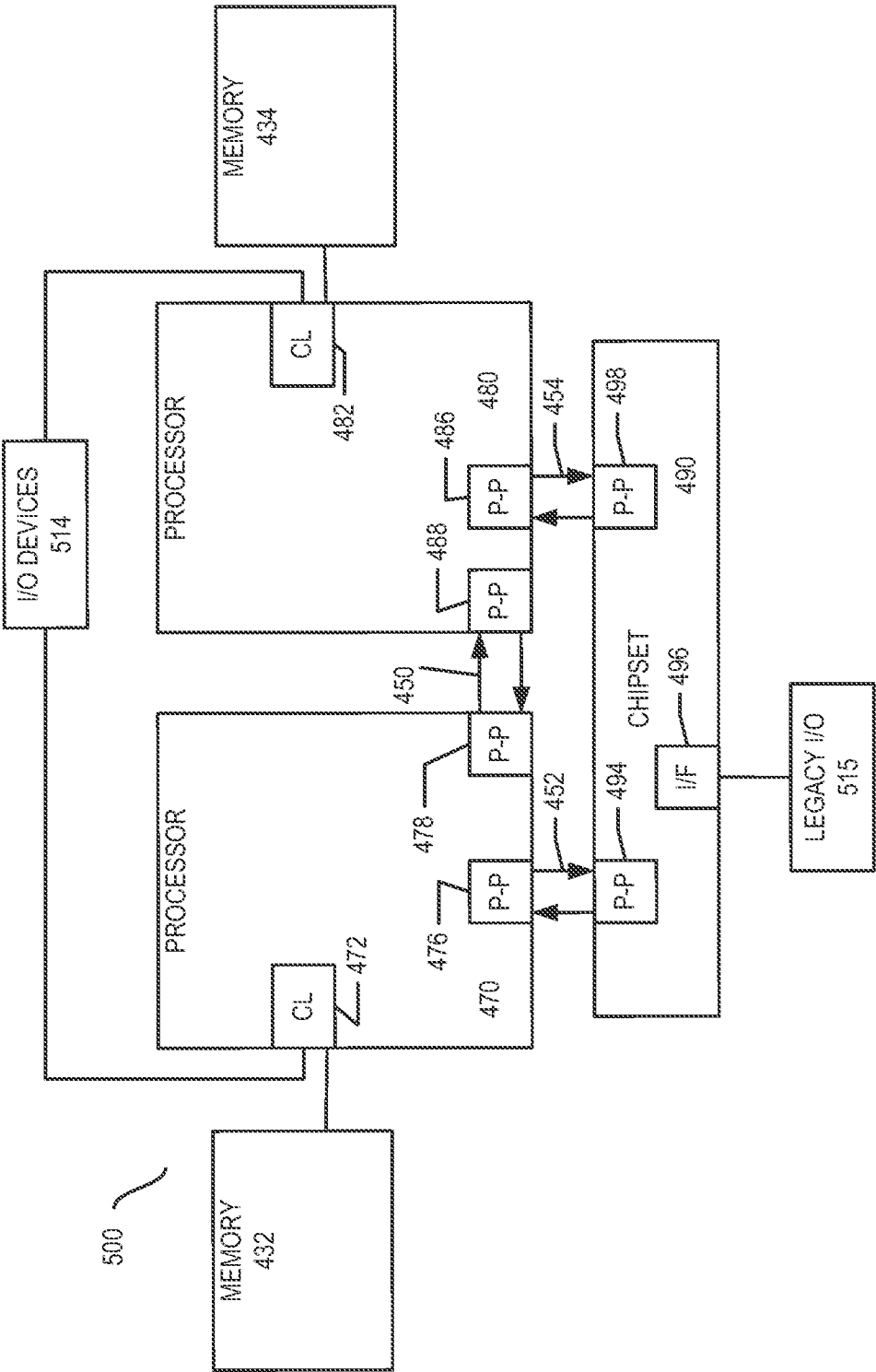


Figure 5

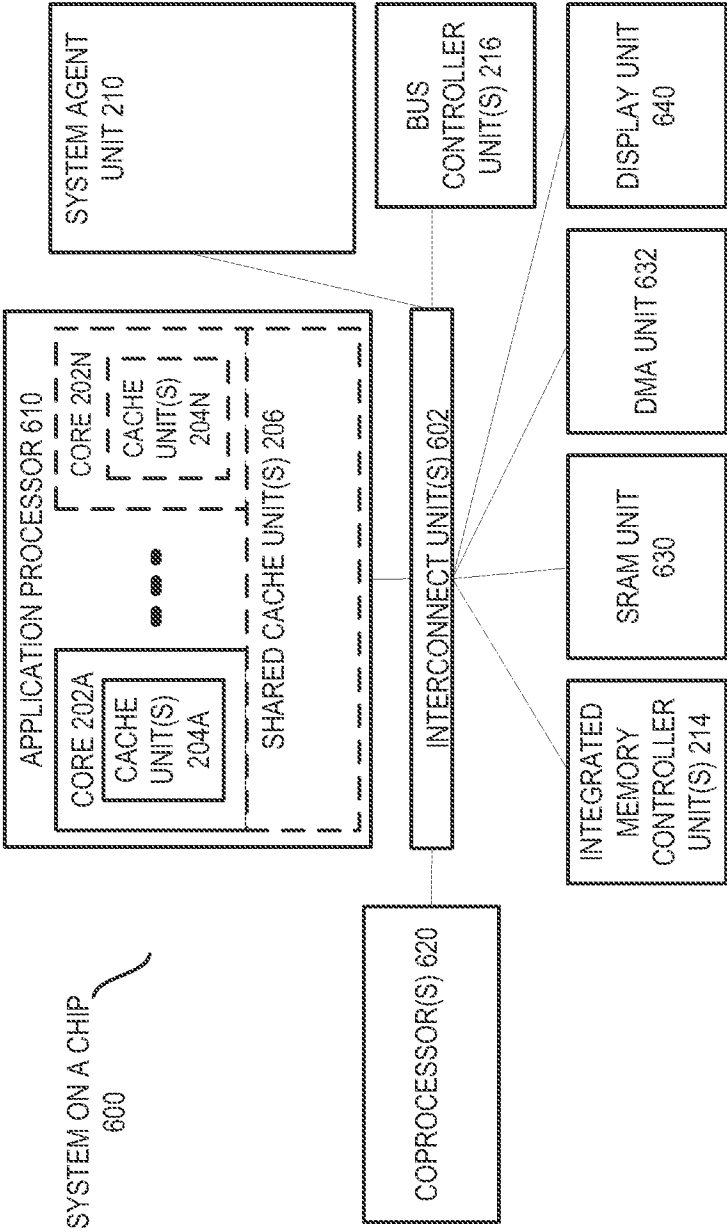
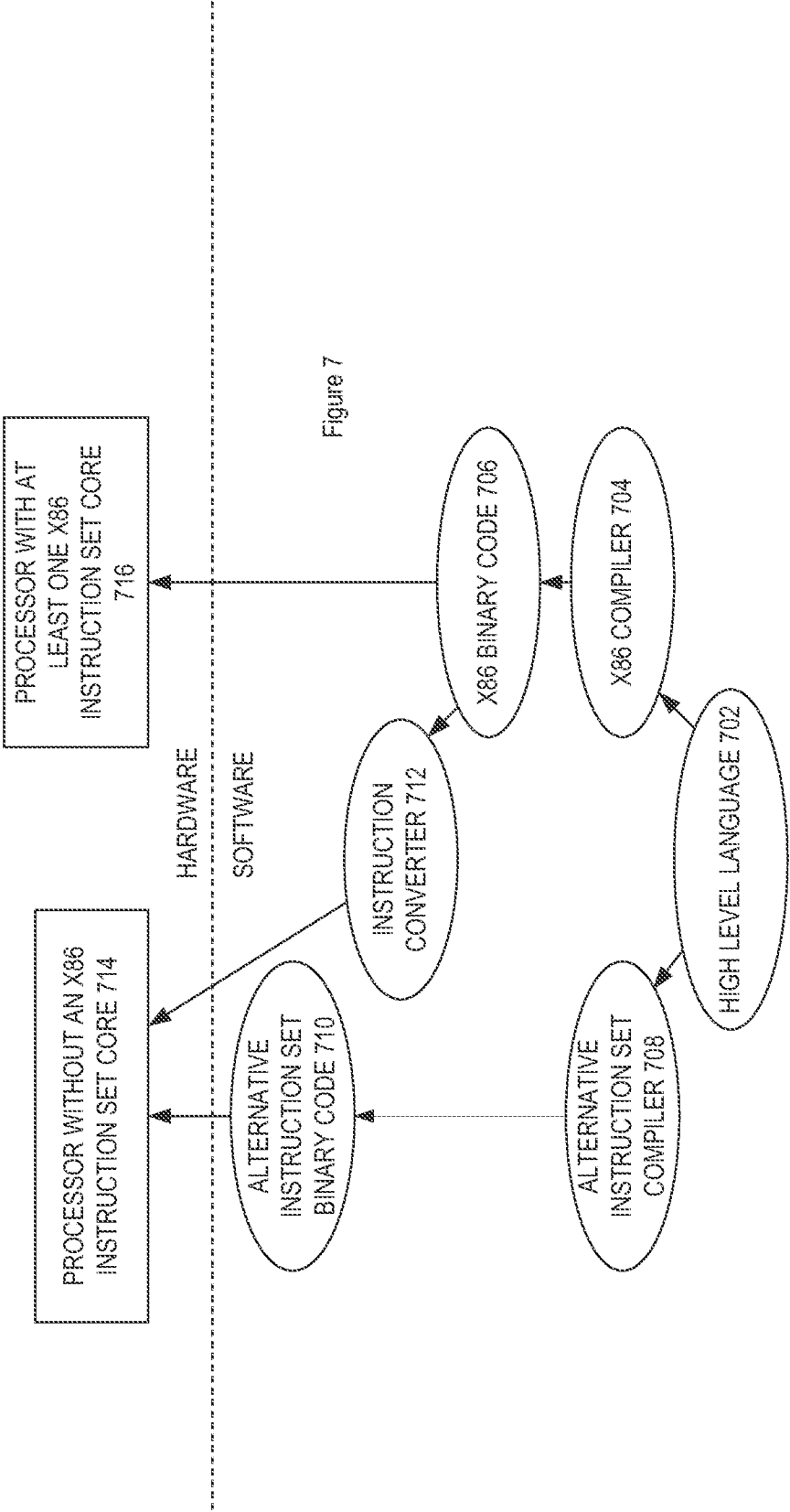
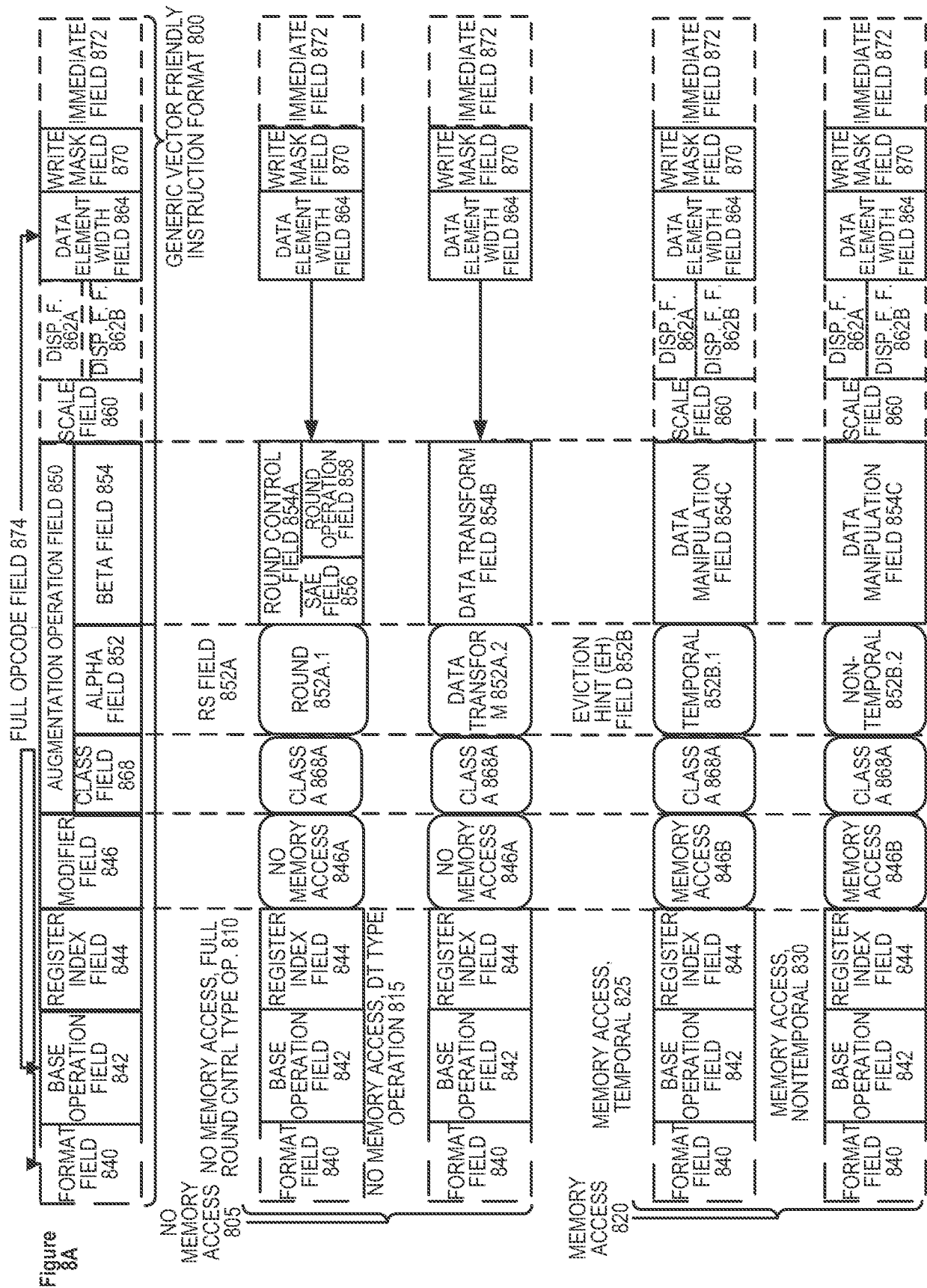


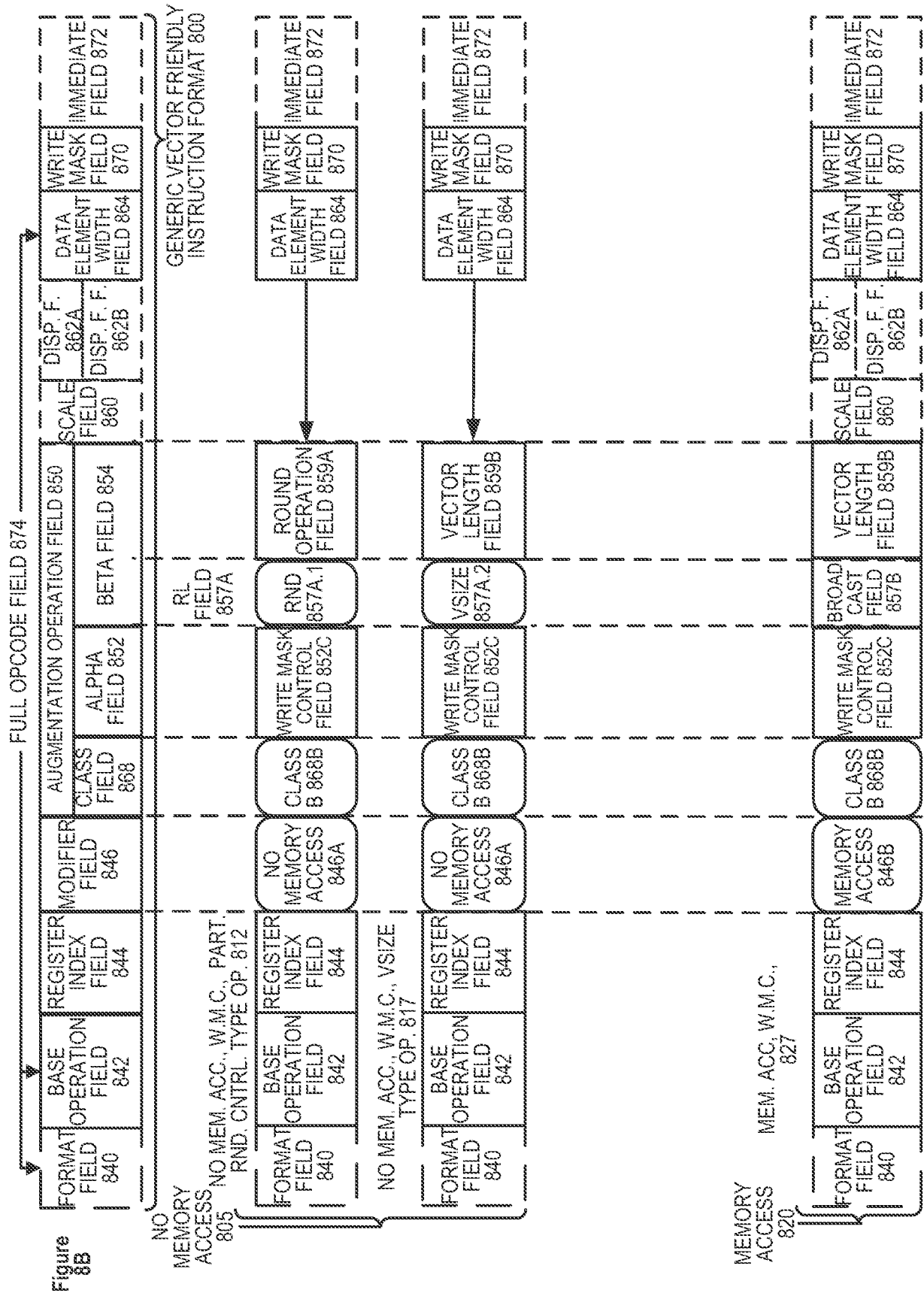
Figure 6



8/21



9/21



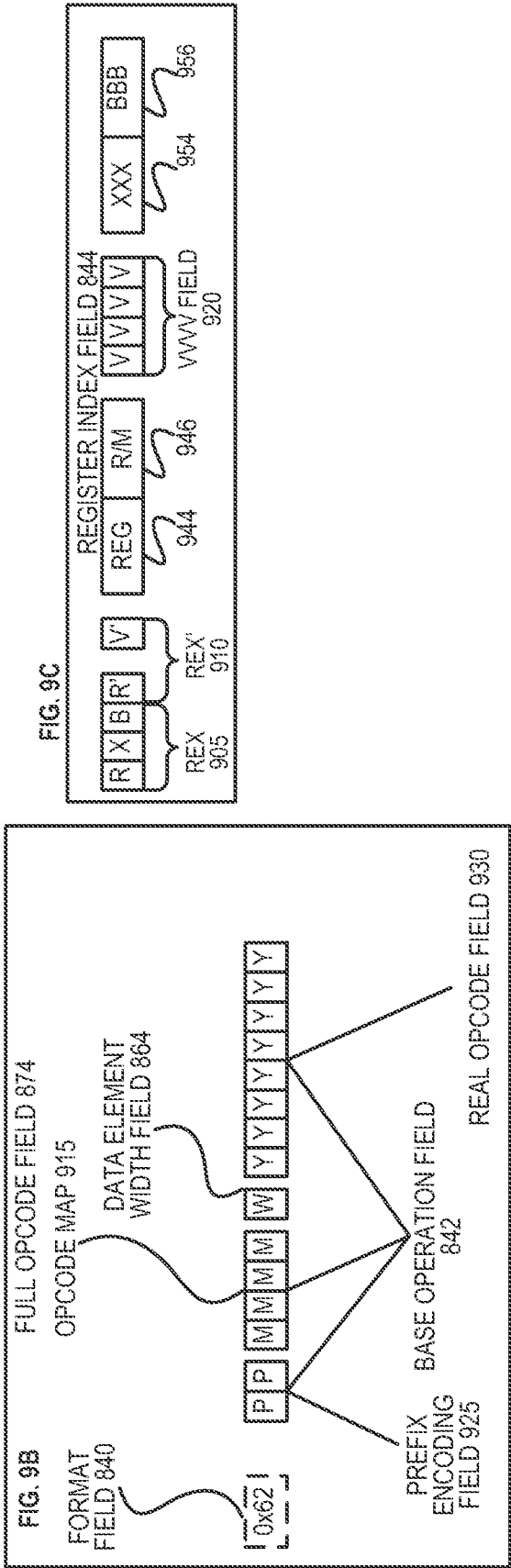
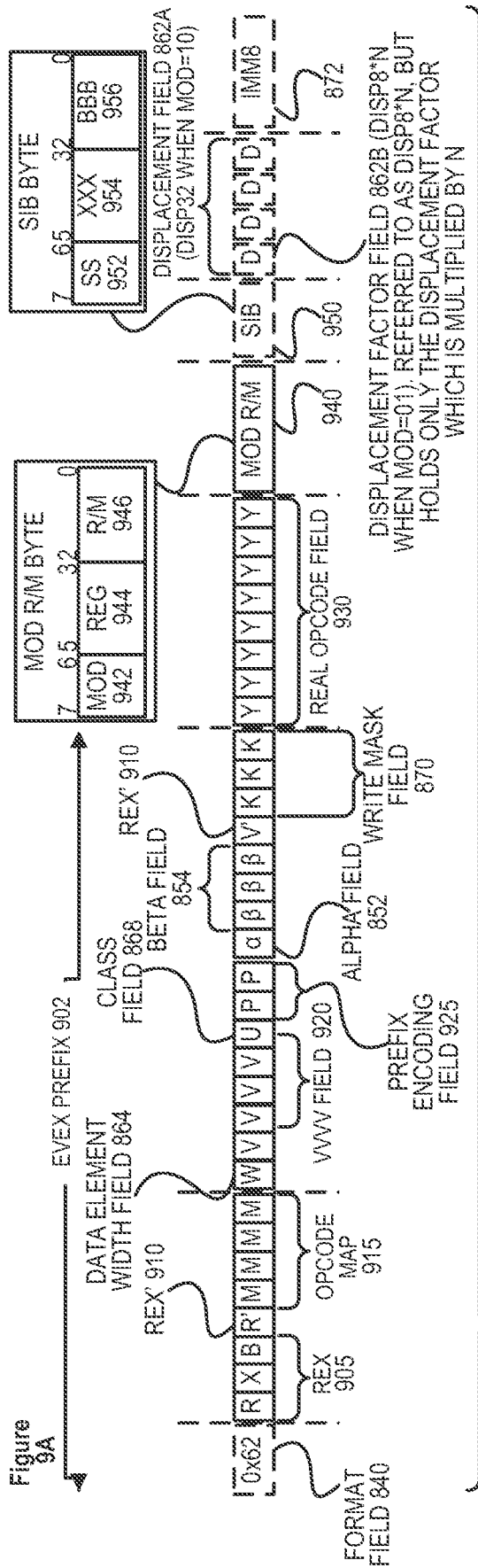
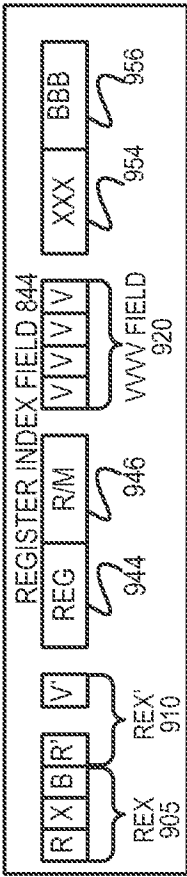
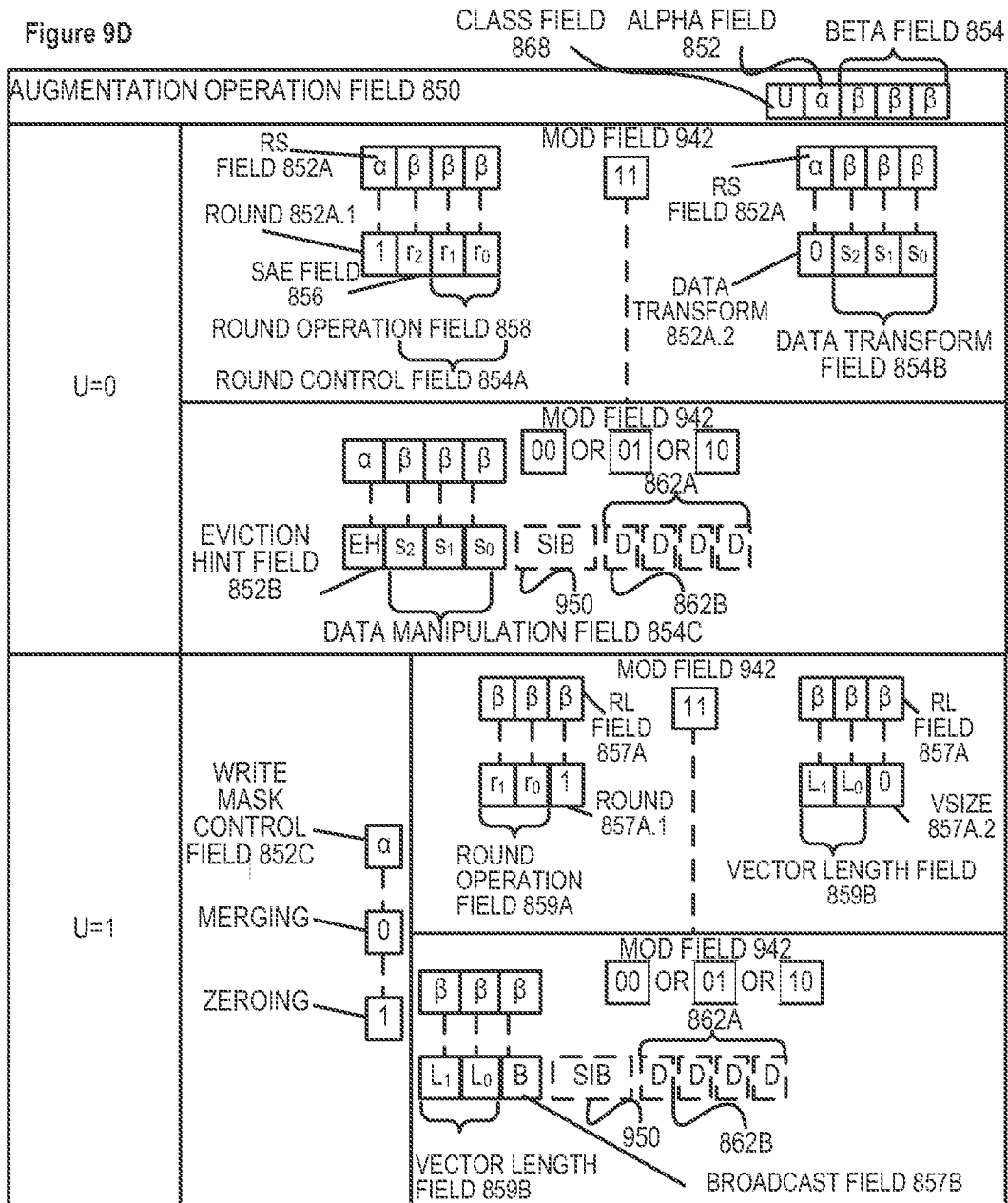


FIG. 9C



11/21



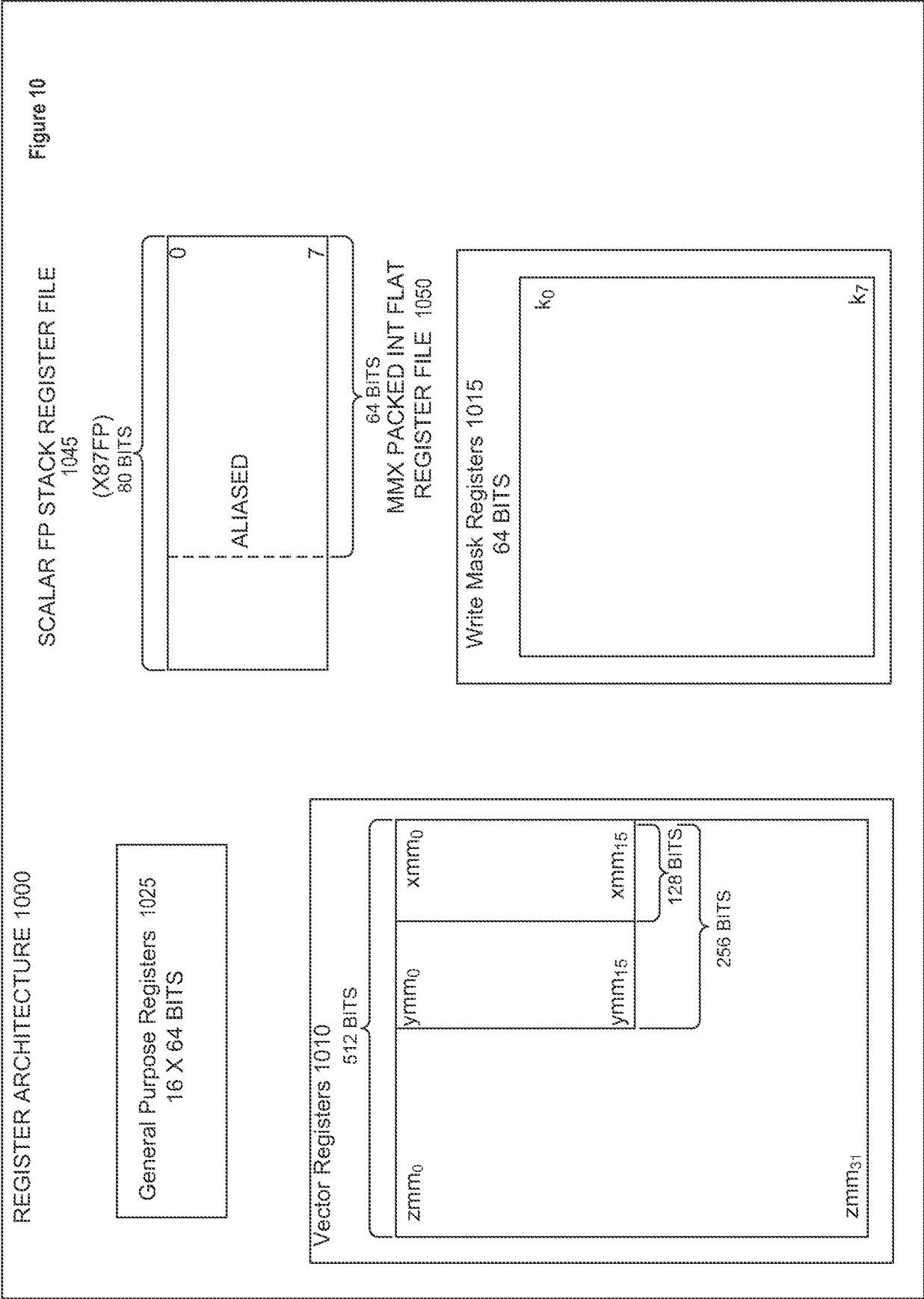


Figure 11A

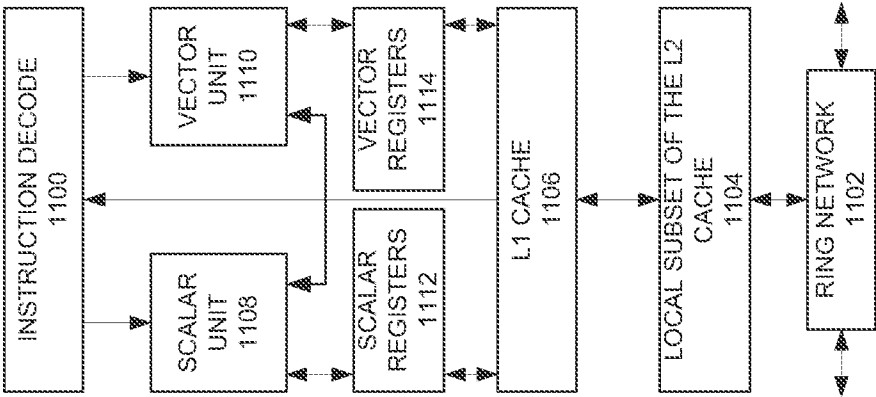


Figure 11B

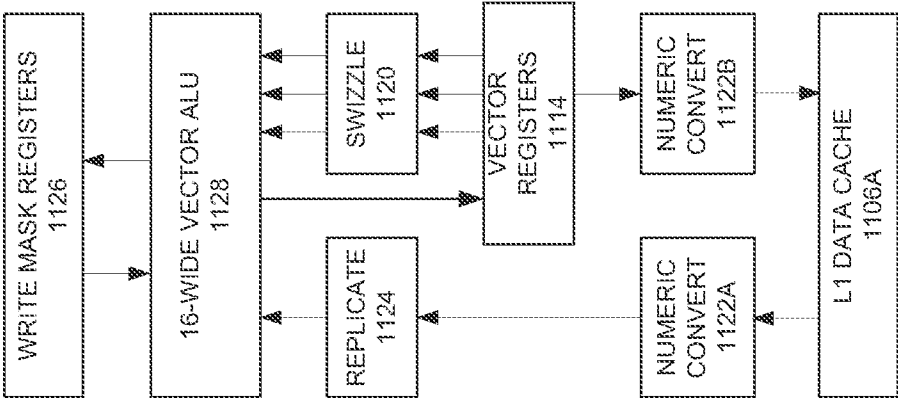


Figure 13

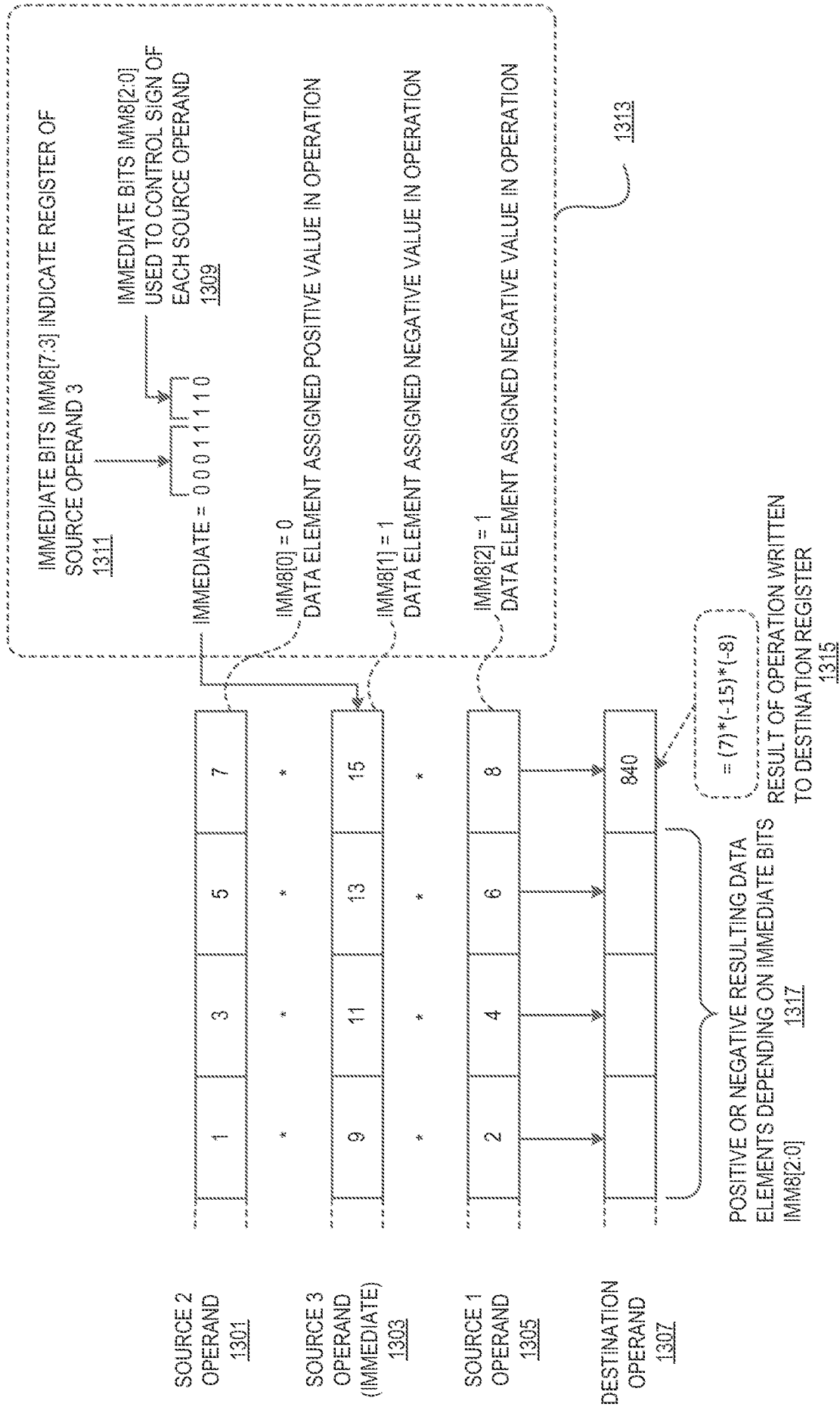


Figure 14

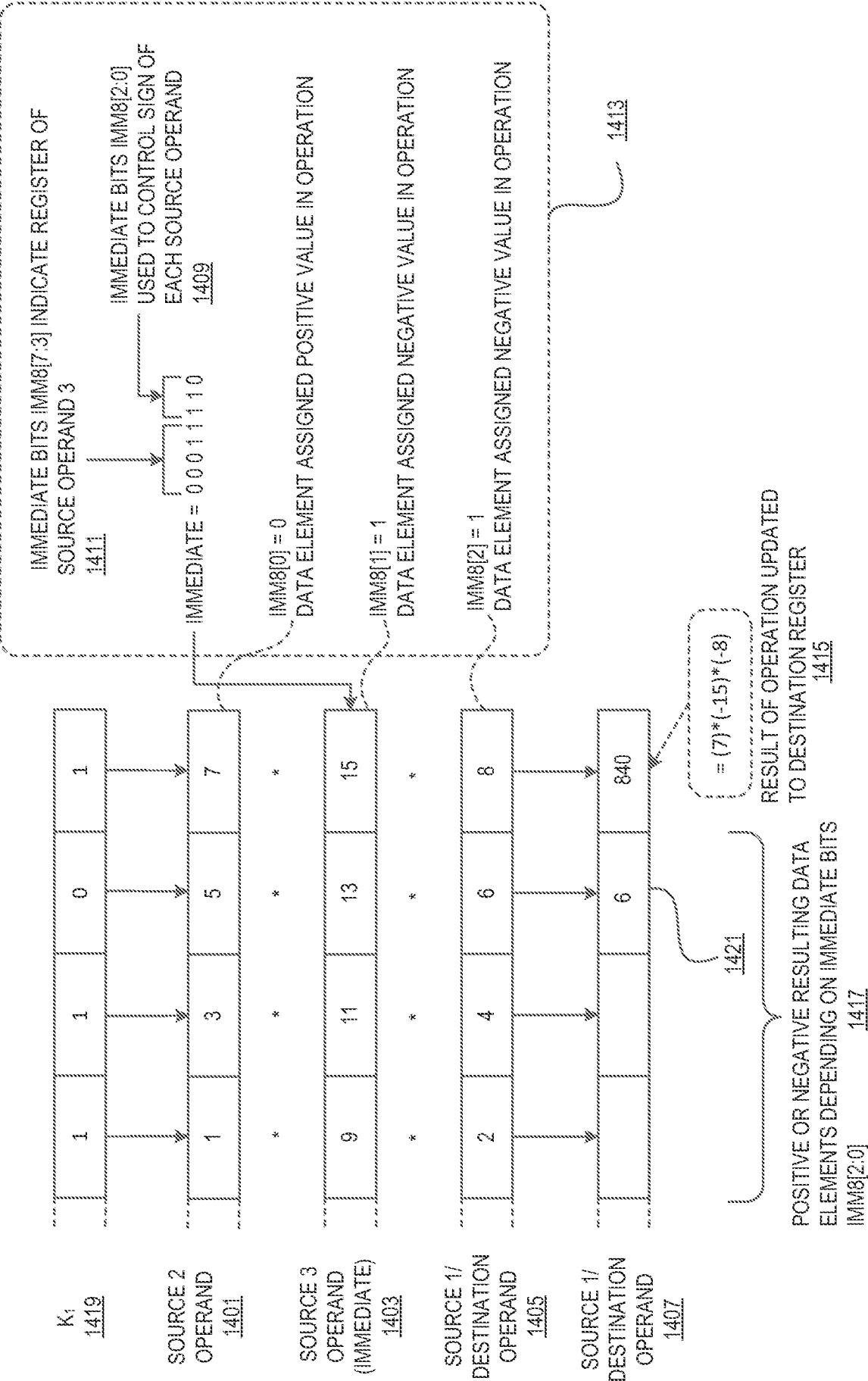


Figure 15

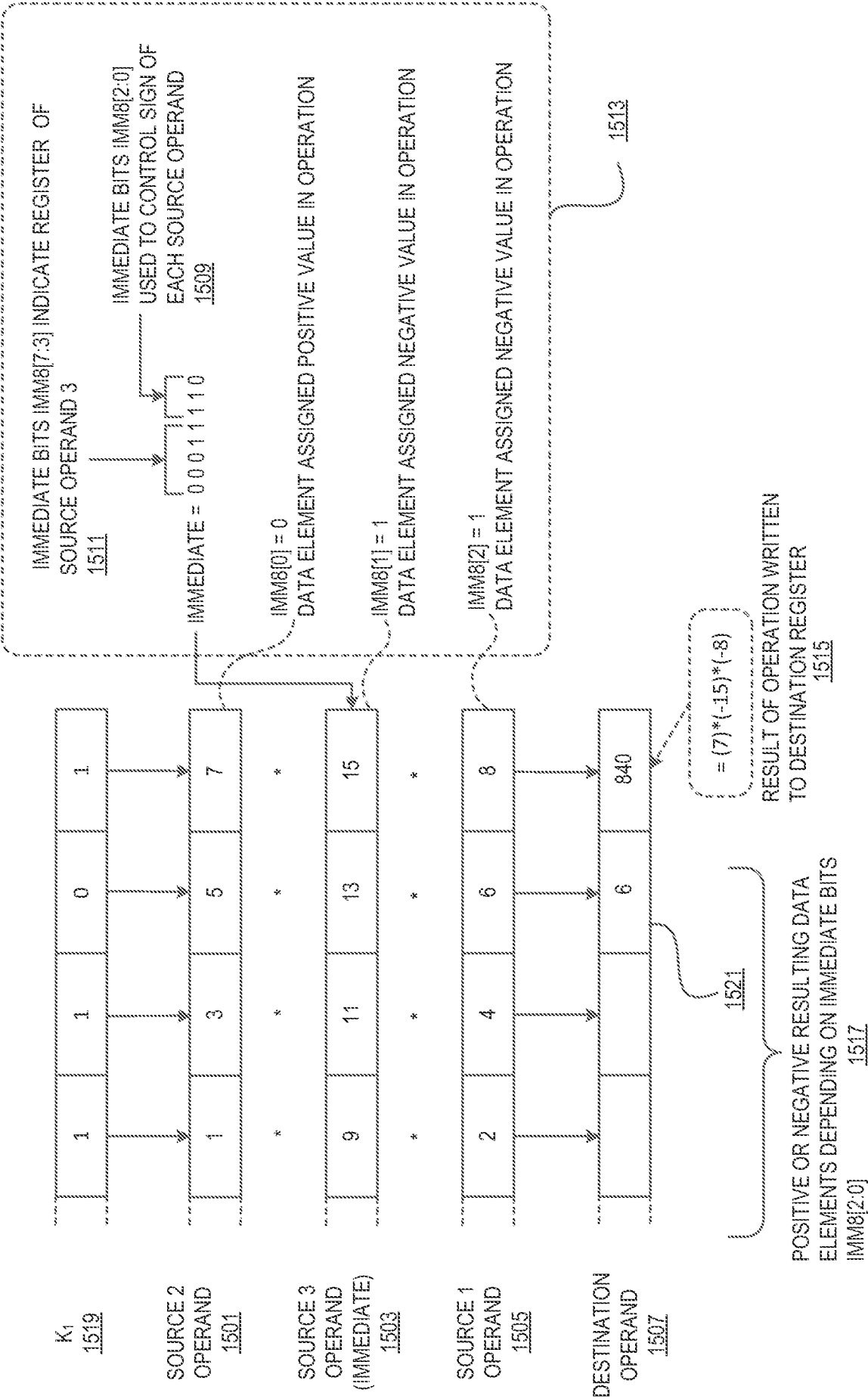
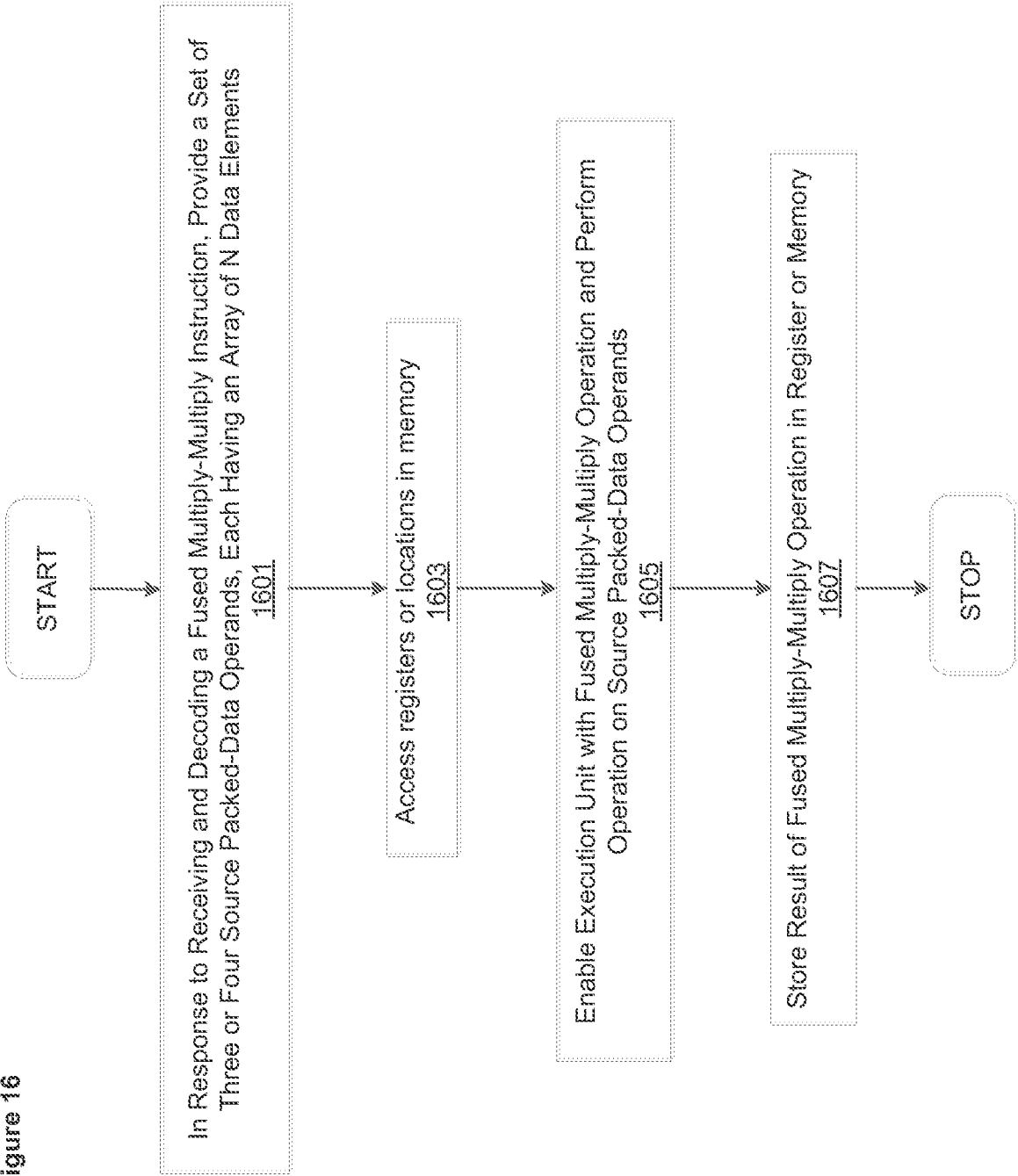


Figure 16



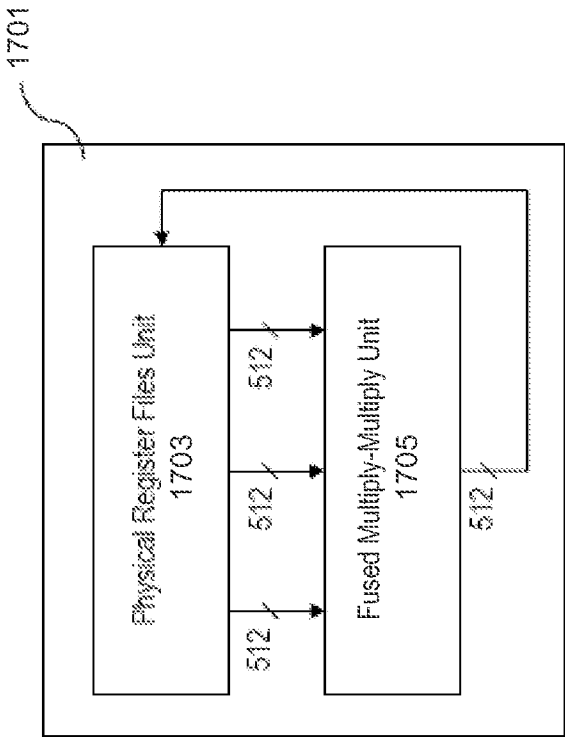


Figure 17

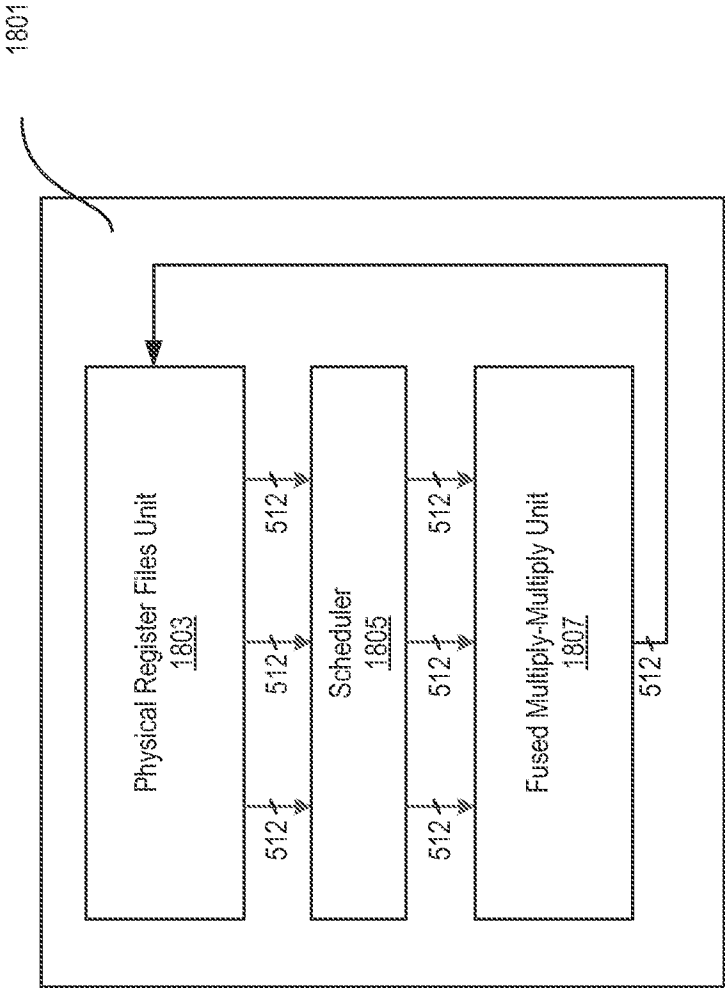


Figure 18

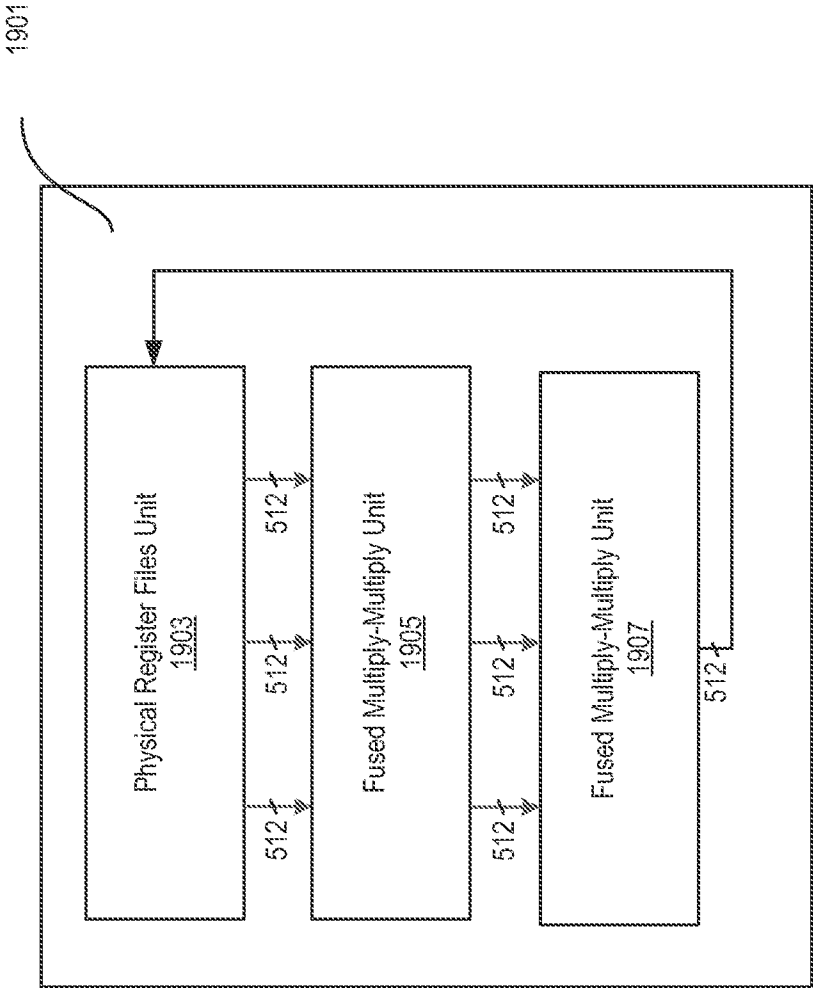


Figure 19

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2015/062328**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i, G06F 7/487(2006.01)i**

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 9/30; G06F 17/10; G06F 7/52; G06F 7/487; G06F 7/00; G06F 7/38

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: multiply-multiply, packed data, source register, immediate value, 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.
A	US 2013-0007075 A1 (DAVID OLIVER et al.) 03 January 2013 See paragraphs [0047]-[0059]; and figure 1.	1-24
A	US 2014-0149720 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 29 May 2014 See paragraphs [0064]-[0094]; and figures 5-6.	1-24
A	US 2014-0052968 A1 (JESUS CORBAL et al.) 20 February 2014 See paragraphs [0029]-[0041]; and figures 3A-3B.	1-24
A	US 6557022 B1 (GILBERT C. SIH et al.) 29 April 2003 See column 2, line 58 - column 3, line 43; and figure 2.	1-24
A	US 05675526 A (ALEXANDER PELEG et al.) 07 October 1997 See column 9, line 20 - column 11, line 15; claim 1; and figure 7.	1-24



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 another citation or other special reason (as specified)

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

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

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

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

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

"&" document member of the same patent family

Date of the actual completion of the international search

04 March 2016 (04.03.2016)

Date of mailing of the international search report

07 March 2016 (07.03.2016)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea



Facsimile No. +82-42-472-7140

Authorized officer

CHIN, Sang Bum

Telephone No. +82-42-481-8398



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2015/062328

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2013-0007075 A1	03/01/2013	US 8838664 B2	16/09/2014
US 2014-0149720 A1	29/05/2014	None	
US 2014-0052968 A1	20/02/2014	CN 104011665 A TW 201346739 A WO 2013-095614 A1	27/08/2014 16/11/2013 27/06/2013
US 6557022 B1	29/04/2003	AT 316264 T AU 2001-39844 A1 CA 2400647 A1 CA 2400647 C CN 1439126 A DE 60116742 D1 DE 60116742 T2 EP 1259876 A2 EP 1259876 B1 ES 2259322 T3 JP 2004-501416 A JP 4477279 B2 KR 10-2003-0011071 A WO 01-63398 A2 WO 01-63398 A3	15/02/2006 03/09/2001 30/08/2001 29/06/2010 27/08/2003 06/04/2006 26/10/2006 27/11/2002 18/01/2006 01/10/2006 15/01/2004 09/06/2010 06/02/2003 30/08/2001 21/02/2002
US 05675526 A	07/10/1997	EP 0795155 A1 EP 0795155 B1 EP 1302848 A2 EP 1302848 A3 EP 1302848 B1 JP 11-500547 A US 05677862 A WO 96-17293 A1	26/02/2003 19/03/2003 16/04/2003 14/05/2003 02/11/2006 12/01/1999 14/10/1997 06/06/1996