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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
04 January 2018 (04.01.2018)

WIPO | PCT

(10) International Publication Number
WO 2018/004946 A1

(51) International Patent Classification:
G06F 13/16 (2006.01) *G06F 9/46* (2006.01)

(21) International Application Number:
PCT/US2017/035168

(22) International Filing Date:
31 May 2017 (31.05.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
15/201,075 01 July 2016 (01.07.2016) US

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

(72) Inventors: **DEMENTIEV, Roman**; Bogenstr. 5, 69124 Heidelberg (DE). **DOSHI, Kshitij**; 2101 E Balboa Drive, Tempe, ZA 85282 (US).

(74) Agent: **WEBSTER, Thomas, C.**; Nicholson Devos 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, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK,

(54) **Title:** APPARATUS AND METHOD FOR REENTERING A TRANSACTIONAL SEQUENCE WITH HARDWARE TRANSACTIONAL MEMORY

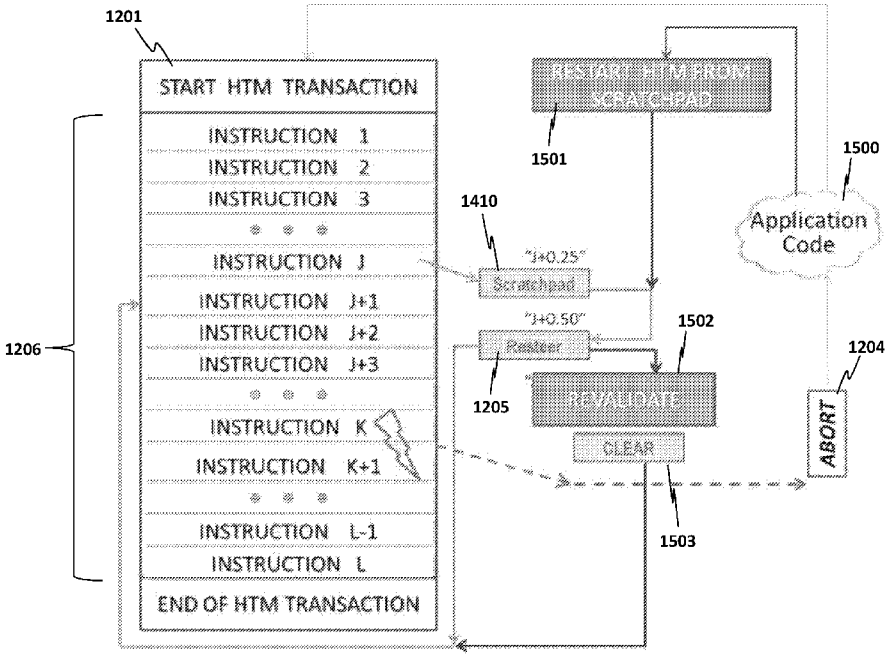


Fig. 15

(57) **Abstract:** An apparatus and method are described for reentering a transactional sequence for hardware transactional memory. For example, one embodiment of a processor comprises: one or more cores to execute instructions and process data; execution circuitry within at least one of the cores to execute a transactional sequence of instructions; a mask value to identify a specified set of architectural state to be saved upon reaching a particular instruction within the transactional sequence of instructions; and a scratchpad memory within the execution circuitry to store the specified set of architectural state upon reaching the particular instruction within the sequence of instructions.

WO 2018/004946 A1

[Continued on next page]

EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*
— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

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

APPARATUS AND METHOD FOR REENTERING A TRANSACTIONAL SEQUENCE WITH HARDWARE TRANSACTIONAL MEMORY

BACKGROUND

Field of the Invention

[0001] This invention relates generally to the field of computer processors. More particularly, the invention relates to a method and apparatus for reentering a transactional sequence with hardware transactional memory (HTM).

Description of the Related Art

1. Processor Microarchitectures

[0002] An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, including the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). It should be noted that the term "instruction" generally refers herein to macro-instructions – that is instructions that are provided to the processor for execution – as opposed to micro-instructions or micro-ops – that is the result of a processor's decoder decoding macro-instructions. The micro-instructions or micro-ops can be configured to instruct an execution unit on the processor to perform operations to implement the logic associated with the macro-instruction.

[0003] The ISA is distinguished from the microarchitecture, which is the set of processor design techniques used to implement 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). 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 distinction is required, 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 designate registers in a given microarchitecture (e.g., physical register, reorder buffer, retirement register, register pool).

2. Transactional Memory

[0004] Transactional memory systems attempt to simplify concurrent programming by allowing groups of load and store instructions to execute in an atomic manner. Hardware transactional memory systems may include modifications to processors, cache and bus protocol to support these atomic transactions. Software transactional memory systems provide transactional memory semantics in software runtime libraries or programming languages and typically require minimal hardware support. However, software implementations typically come with a performance penalty when compared with hardware solutions.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] A better understanding of the present invention can be obtained from the following detailed description in conjunction with the following drawings, in which:

[0006] FIGS. 1A and 1B are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention;

[0007] FIG. 2A-D is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention;

[0008] FIG. 3 is a block diagram of a register architecture according to one embodiment of the invention; and

[0009] FIG. 4A 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;

[0010] FIG. 4B 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;

[0011] FIG. 5A is a block diagram of a single processor core, along with its connection to an on-die interconnect network;

[0012] FIG. 5B illustrates an expanded view of part of the processor core in FIG 5A according to embodiments of the invention;

[0013] FIG. 6 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;

[0014] FIG. 7 illustrates a block diagram of a system in accordance with one embodiment of the present invention;

[0015] FIG. 8 illustrates a block diagram of a second system in accordance with an embodiment of the present invention;

[0016] FIG. 9 illustrates a block diagram of a third system in accordance with an embodiment of the present invention;

[0017] FIG. 10 illustrates a block diagram of a system on a chip (SoC) in accordance with an embodiment of the present invention;

[0018] FIG. 11 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;

[0019] FIG. 12 illustrates an exemplary transactional instruction sequence;

[0020] FIG. 13 illustrates another view of the exemplary transactional instruction sequence;

[0021] FIG. 14 illustrates a set of mask bits and an exemplary scratchpad memory implemented in one embodiment of the invention;

[0022] FIG. 15 illustrates one embodiment in which transactional state is read from the scratchpad memory and revalidated; and

[0023] FIGS. 16-17 illustrate a method in accordance with one embodiment of the invention.

DETAILED DESCRIPTION

[0024] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the embodiments of the invention described below. It will be apparent, however, to one skilled in the art that the embodiments of the invention may be practiced without some of these specific details. In other instances, well-known structures and devices are shown in block diagram form to avoid obscuring the underlying principles of the embodiments of the invention.

EXEMPLARY PROCESSOR ARCHITECTURES AND DATA TYPES

[0025] 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. A set of SIMD extensions referred to the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme, has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developers Manual, October 2011; and see Intel® Advanced Vector Extensions Programming Reference, June 2011).

Exemplary Instruction Formats

[0026] 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. Generic Vector Friendly Instruction Format

[0027] 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.

[0028] Figures 1A-1B are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments

of the invention. Figure 1A 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 1B 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 100 for which are defined class A and class B instruction templates, both of which include no memory access 105 instruction templates and memory access 120 instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

[0029] 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).

[0030] The class A instruction templates in Figure 1A include: 1) within the no memory access 105 instruction templates there is shown a no memory access, full round control type operation 110 instruction template and a no memory access, data transform type operation 115 instruction template; and 2) within the memory access 120 instruction templates there is shown a memory access, temporal 125 instruction template and a memory access, non-temporal 130 instruction template. The class B instruction templates in Figure 1B

include: 1) within the no memory access 105 instruction templates there is shown a no memory access, write mask control, partial round control type operation 112 instruction template and a no memory access, write mask control, vsize type operation 117 instruction template; and 2) within the memory access 120 instruction templates there is shown a memory access, write mask control 127 instruction template.

[0031] The generic vector friendly instruction format 100 includes the following fields listed below in the order illustrated in Figures 1A-1B.

[0032] Format field 140 – 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.

[0033] Base operation field 142 – its content distinguishes different base operations.

[0034] Register index field 144 – 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).

[0035] Modifier field 146 – 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 105 instruction templates and memory access 120 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.

[0036] Augmentation operation field 150 – 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 168, an alpha field 152, and a beta field 154. The augmentation operation field 150 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions.

[0037] Scale field 160 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses 2^{scale} * index + base).

[0038] Displacement Field 162A– its content is used as part of memory address generation (e.g., for address generation that uses 2^{scale} * index + base + displacement).

[0039] Displacement Factor Field 162B (note that the juxtaposition of displacement field 162A directly over displacement factor field 162B indicates one or the other is used) – its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N) – where N is the number of bytes in the memory access (e.g., for address generation that uses 2^{scale} * index + base + scaled displacement). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on

the full opcode field 174 (described later herein) and the data manipulation field 154C. The displacement field 162A and the displacement factor field 162B are optional in the sense that they are not used for the no memory access 105 instruction templates and/or different embodiments may implement only one or none of the two.

[0040] Data element width field 164 – 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.

[0041] Write mask field 170 – 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 170 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 170 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 170 content indirectly identifies that masking to be performed), alternative embodiments instead or additional allow the mask write field's 170 content to directly specify the masking to be performed.

[0042] Immediate field 172 – 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.

[0043] Class field 168 – its content distinguishes between different classes of instructions. With reference to **Figures 1A-B**, the contents of this field select between class A and class B instructions. In **Figures 1A-B**, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 168A and class B 168B for the class field 168 respectively in **Figures 1A-B**).

Instruction Templates of Class A

[0044] In the case of the non-memory access 105 instruction templates of class A, the alpha field 152 is interpreted as an RS field 152A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 152A.1 and data transform 152A.2 are respectively specified for the no memory access, round type operation 110 and the no memory access, data transform type operation 115 instruction templates), while the beta field 154 distinguishes which of the operations of the specified type is to be performed. In the no memory access 105 instruction templates, the scale field 160, the displacement field 162A, and the displacement scale filed 162B are not present.

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

[0045] In the no memory access full round control type operation 110 instruction template, the beta field 154 is interpreted as a round control field 154A, whose content(s) provide static rounding. While in the described

embodiments of the invention the round control field 154A includes a suppress all floating point exceptions (SAE) field 156 and a round operation control field 158, 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 158).

[0046] SAE field 156 – its content distinguishes whether or not to disable the exception event reporting; when the SAE field's 156 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.

[0047] Round operation control field 158 – 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 158 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 150 content overrides that register value.

No Memory Access Instruction Templates – Data Transform Type Operation

[0048] In the no memory access data transform type operation 115 instruction template, the beta field 154 is interpreted as a data transform field 154B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

[0049] In the case of a memory access 120 instruction template of class A, the alpha field 152 is interpreted as an eviction hint field 152B, whose content distinguishes which one of the eviction hints is to be used (in **Figure 1A**, temporal 152B.1 and non-temporal 152B.2 are respectively specified for the memory access, temporal 125 instruction template and the memory access, non-temporal 130 instruction template), while the beta field 154 is interpreted as a data manipulation field 154C, 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 120 instruction templates include the scale field 160, and optionally the displacement field 162A or the displacement scale field 162B.

[0050] Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

Memory Access Instruction Templates – Temporal

[0051] Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Memory Access Instruction Templates – Non-Temporal

[0052] Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

Instruction Templates of Class B

[0053] In the case of the instruction templates of class B, the alpha field 152 is interpreted as a write mask control (Z) field 152C, whose content distinguishes whether the write masking controlled by the write mask field 170 should be a merging or a zeroing.

[0054] In the case of the non-memory access 105 instruction templates of class B, part of the beta field 154 is interpreted as an RL field 157A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 157A.1 and vector length (VSIZE) 157A.2 are respectively specified for the no memory access, write mask control, partial round control type operation 112 instruction template and the no memory access, write mask control, VSIZE type operation 117 instruction template), while

the rest of the beta field 154 distinguishes which of the operations of the specified type is to be performed. In the no memory access 105 instruction templates, the scale field 160, the displacement field 162A, and the displacement scale filed 162B are not present.

[0055] In the no memory access, write mask control, partial round control type operation 110 instruction template, the rest of the beta field 154 is interpreted as a round operation field 159A 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).

[0056] Round operation control field 159A – just as round operation control field 158, 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 159A 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 150 content overrides that register value.

[0057] In the no memory access, write mask control, VSIZE type operation 117 instruction template, the rest of the beta field 154 is interpreted as a vector length field 159B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

[0058] In the case of a memory access 120 instruction template of class B, part of the beta field 154 is interpreted as a broadcast field 157B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 154 is interpreted the vector length field 159B. The memory access 120 instruction templates include the scale field 160, and optionally the displacement field 162A or the displacement scale field 162B.

[0059] With regard to the generic vector friendly instruction format 100, a full opcode field 174 is shown including the format field 140, the base

operation field 142, and the data element width field 164. While one embodiment is shown where the full opcode field 174 includes all of these fields, the full opcode field 174 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 174 provides the operation code (opcode).

[0060] The augmentation operation field 150, the data element width field 164, and the write mask field 170 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format.

[0061] 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.

[0062] The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance general purpose cores with out of order execution and register renaming intended for general-purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and

class B. Of course, features from one class may also be implement in the other class in different embodiments of the invention. Programs written in a high level language would be put (e.g., just in time compiled or statically compiled) into an variety of different executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently executing the code.

B. Exemplary Specific Vector Friendly Instruction Format

[0063] Figure 2 is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. Figure 2 shows a specific vector friendly instruction format 200 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 200 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 1 into which the fields from Figure 2 map are illustrated.

[0064] It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 200 in the context of the generic vector friendly instruction format 100 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 200 except where claimed. For example, the generic vector friendly instruction format 100 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 200 is shown as having fields of specific sizes. By way of specific example, while the data

element width field 164 is illustrated as a one bit field in the specific vector friendly instruction format 200, the invention is not so limited (that is, the generic vector friendly instruction format 100 contemplates other sizes of the data element width field 164).

[0065] The generic vector friendly instruction format 100 includes the following fields listed below in the order illustrated in Figure 2A.

[0066] EVEX Prefix (Bytes 0-3) 202 - is encoded in a four-byte form.

[0067] Format Field 140 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 140 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention).

[0068] The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

[0069] REX field 205 (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 157BEX byte 1, bit[5] – B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 1111B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

[0070] REX' field 110 – this is the first part of the REX' field 110 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.

[0071] Opcode map field 215 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3).

[0072] Data element width field 164 (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).

[0073] EVEX.vvvv 220 (EVEX Byte 2, bits [6:3]-vvvv)- the role of EVEX.vvvv may include the following: 1) EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) EVEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) EVEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. Thus, EVEX.vvvv field 220 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.

[0074] EVEX.U 168 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.

[0075] Prefix encoding field 225 (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.

[0076] Alpha field 152 (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.

[0077] Beta field 154 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s_{2-0}, EVEX.r_{2-0}, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with βββ) – as previously described, this field is context specific.

[0078] REX' field 110 – 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.

[0079] Write mask field 170 (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).

[0080] Real Opcode Field 230 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field.

[0081] MOD R/M Field 240 (Byte 5) includes MOD field 242, Reg field 244, and R/M field 246. As previously described, the MOD field's 242 content distinguishes between memory access and non-memory access operations. The role of Reg field 244 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 246 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.

[0082] Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 150 content is used for memory address generation. SIB.xxx 254 and SIB.bbb 256 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb.

[0083] Displacement field 162A (Bytes 7-10) – when MOD field 242 contains 10, bytes 7-10 are the displacement field 162A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

[0084] Displacement factor field 162B (Byte 7) – when MOD field 242 contains 01, byte 7 is the displacement factor field 162B. 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 162B is a reinterpretation of disp8; when using displacement factor field 162B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is based on the assumption that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field 162B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 162B is encoded the same way as an x86 instruction

set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset).

[0085] Immediate field 172 operates as previously described.

Full Opcode Field

[0086] Figure 2B is a block diagram illustrating the fields of the specific vector friendly instruction format 200 that make up the full opcode field 174 according to one embodiment of the invention. Specifically, the full opcode field 174 includes the format field 140, the base operation field 142, and the data element width (W) field 164. The base operation field 142 includes the prefix encoding field 225, the opcode map field 215, and the real opcode field 230.

Register Index Field

[0087] Figure 2C is a block diagram illustrating the fields of the specific vector friendly instruction format 200 that make up the register index field 144 according to one embodiment of the invention. Specifically, the register index field 144 includes the REX field 205, the REX' field 210, the MODR/M.reg field 244, the MODR/M.r/m field 246, the VVVV field 220, xxx field 254, and the bbb field 256.

Augmentation Operation Field

[0088] Figure 2D is a block diagram illustrating the fields of the specific vector friendly instruction format 200 that make up the augmentation operation field 150 according to one embodiment of the invention. When the class (U) field 168 contains 0, it signifies EVEX.U0 (class A 168A); when it contains 1, it signifies EVEX.U1 (class B 168B). When U=0 and the MOD field 242 contains 11 (signifying a no memory access operation), the alpha field 152 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 152A. When the rs field 152A

contains a 1 (round 152A.1), the beta field 154 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the round control field 154A. The round control field 154A includes a one bit SAE field 156 and a two bit round operation field 158. When the rs field 152A contains a 0 (data transform 152A.2), the beta field 154 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data transform field 154B. When U=0 and the MOD field 242 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 152 (EVEX byte 3, bit [7] – EH) is interpreted as the eviction hint (EH) field 152B and the beta field 154 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 154C.

[0089] When U=1, the alpha field 152 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 152C. When U=1 and the MOD field 242 contains 11 (signifying a no memory access operation), part of the beta field 154 (EVEX byte 3, bit [4]- S_0) is interpreted as the RL field 157A; when it contains a 1 (round 157A.1) the rest of the beta field 154 (EVEX byte 3, bit [6-5]- S_{2-1}) is interpreted as the round operation field 159A, while when the RL field 157A contains a 0 (VSIZE 157.A2) the rest of the beta field 154 (EVEX byte 3, bit [6-5]- S_{2-1}) is interpreted as the vector length field 159B (EVEX byte 3, bit [6-5]- L_{1-0}). When U=1 and the MOD field 242 contains 00, 01, or 10 (signifying a memory access operation), the beta field 154 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 159B (EVEX byte 3, bit [6-5]- L_{1-0}) and the broadcast field 157B (EVEX byte 3, bit [4]- B).

C. Exemplary Register Architecture

[0090] Figure 3 is a block diagram of a register architecture 300 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 310 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 200 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 159B	A (Figure 1A; U=0)	110, 115, 125, 130	zmm registers (the vector length is 64 byte)
	B (Figure 1B; U=1)	112	zmm registers (the vector length is 64 byte)
Instruction templates that do include the vector length field 159B	B (Figure 1B; U=1)	117, 127	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 159B

[0091] In other words, the vector length field 159B 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 159B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 200 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.

[0092] Write mask registers 315 - 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 315 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.

[0093] General-purpose registers 325 - 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.

[0094] Scalar floating point stack register file (x87 stack) 345, on which is aliased the MMX packed integer flat register file 350 - 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.

[0095] 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.

D. Exemplary Core Architectures, Processors, and Computer Architectures

[0096] Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include:

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

[0097] Figure 4A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. Figure 4B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes in Figures 4A-B illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

[0098] In Figure 4A, a processor pipeline 400 includes a fetch stage 402, a length decode stage 404, a decode stage 406, an allocation stage 408, a renaming stage 410, a scheduling (also known as a dispatch or issue) stage 412, a register read/memory read stage 414, an execute stage 416, a write back/memory write stage 418, an exception handling stage 422, and a commit stage 424.

[0099] Figure 4B shows processor core 490 including a front end unit 430 coupled to an execution engine unit 450, and both are coupled to a memory unit 470. The core 490 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 490 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.

[00100] The front end unit 430 includes a branch prediction unit 432 coupled to an instruction cache unit 434, which is coupled to an instruction translation lookaside buffer (TLB) 436, which is coupled to an instruction fetch unit 438, which is coupled to a decode unit 440. The decode unit 440 (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 440 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 490 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 440 or otherwise within the front end unit 430). The decode unit 440 is coupled to a rename/allocator unit 452 in the execution engine unit 450.

[00101] The execution engine unit 450 includes the rename/allocator unit 452 coupled to a retirement unit 454 and a set of one or more scheduler unit(s) 456. The scheduler unit(s) 456 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 456 is coupled to the physical register file(s) unit(s) 458. Each of the physical register file(s) units 458 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 458 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) 458 is overlapped by the retirement unit 454 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit 454 and the physical register file(s) unit(s) 458 are coupled to the execution cluster(s) 460. The execution cluster(s) 460 includes a set of one or more execution units 462 and a set of one or more memory access units 464. The execution units 462 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) 456, physical register file(s) unit(s) 458, and execution cluster(s) 460 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) 464). 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.

[00102] The set of memory access units 464 is coupled to the memory unit 470, which includes a data TLB unit 472 coupled to a data cache unit 474 coupled to a level 2 (L2) cache unit 476. In one exemplary embodiment, the memory access units 464 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 472 in the memory unit 470. The instruction cache unit 434 is further coupled to a level 2 (L2) cache unit 476 in the memory unit 470. The L2 cache unit 476 is coupled to one or more other levels of cache and eventually to a main memory.

[00103] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 400 as follows: 1) the instruction fetch 438 performs the fetch and length decoding stages 402 and 404; 2) the decode unit 440 performs the decode stage 406; 3) the rename/allocator unit 452 performs the allocation stage 408 and renaming stage 410; 4) the scheduler unit(s) 456 performs the schedule stage 412; 5) the physical register file(s) unit(s) 458 and the memory unit 470 perform the register read/memory read stage 414; the execution cluster 460 perform the execute stage 416; 6) the memory unit 470 and the physical register file(s) unit(s) 458 perform the write back/memory write stage 418; 7) various units may be involved in the exception handling stage 422; and 8) the retirement unit 454 and the physical register file(s) unit(s) 458 perform the commit stage 424.

[00104] The core 490 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 490 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2), thereby allowing the operations used by many multimedia applications to be performed using packed data.

[00105] 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).

[00106] 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 434/474 and a shared L2 cache unit 476, 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.

[00107] Figures 5A-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.

[00108] Figure 5A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 502 and with its local subset of the Level 2 (L2) cache 504, according to embodiments of the invention. In one embodiment, an instruction decoder 500 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 506 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 508 and a vector unit

510 use separate register sets (respectively, scalar registers 512 and vector registers 514) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 506, 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).

[00109] The local subset of the L2 cache 504 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 504. Data read by a processor core is stored in its L2 cache subset 504 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 504 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.

[00110] **Figure 5B** is an expanded view of part of the processor core in **Figure 5A** according to embodiments of the invention. **Figure 5B** includes an L1 data cache 506A part of the L1 cache 504, as well as more detail regarding the vector unit 510 and the vector registers 514. Specifically, the vector unit 510 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 528), 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 520, numeric conversion with numeric convert units 522A-B, and replication with replication unit 524 on the memory input. Write mask registers 526 allow predicating resulting vector writes.

[00111] **Figure 6** is a block diagram of a processor 600 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 6** illustrate a processor 600 with a single core 602A, a system agent 610, a set of one or more bus controller units 616, while the optional addition of the dashed lined boxes illustrates an alternative processor 600 with multiple cores 602A-N, a set of one or more integrated memory controller unit(s) 614 in the system agent unit 610, and special purpose logic 608.

[00112] Thus, different implementations of the processor 600 may include: 1) a CPU with the special purpose logic 608 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 602A-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 602A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 602A-N being a large number of general purpose in-order cores. Thus, the processor 600 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 600 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.

[00113] The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 606, and external memory (not shown) coupled to the set of integrated memory controller units 614. The set of shared cache units 606 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 612 interconnects the integrated graphics logic 608, the set of shared cache units 606, and the system agent unit 610/integrated

memory controller unit(s) 614, 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 606 and cores 602-A-N.

[00114] In some embodiments, one or more of the cores 602A-N are capable of multi-threading. The system agent 610 includes those components coordinating and operating cores 602A-N. The system agent unit 610 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 602A-N and the integrated graphics logic 608. The display unit is for driving one or more externally connected displays.

[00115] The cores 602A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 602A-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.

[00116] Figures 7-10 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.

[00117] Referring now to Figure 7, shown is a block diagram of a system 700 in accordance with one embodiment of the present invention. The system 700 may include one or more processors 710, 715, which are coupled to a controller hub 720. In one embodiment the controller hub 720 includes a graphics memory controller hub (GMCH) 790 and an Input/Output Hub (IOH) 750 (which may be on separate chips); the GMCH 790 includes memory and

graphics controllers to which are coupled memory 740 and a coprocessor 745; the IOH 750 is couples input/output (I/O) devices 760 to the GMCH 790. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 740 and the coprocessor 745 are coupled directly to the processor 710, and the controller hub 720 in a single chip with the IOH 750.

[00118] The optional nature of additional processors 715 is denoted in **Figure 7** with broken lines. Each processor 710, 715 may include one or more of the processing cores described herein and may be some version of the processor 600.

[00119] The memory 740 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 720 communicates with the processor(s) 710, 715 via a multi-drop bus, such as a frontside bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 795.

[00120] In one embodiment, the coprocessor 745 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 720 may include an integrated graphics accelerator.

[00121] There can be a variety of differences between the physical resources 710, 715 in terms of a spectrum of metrics of merit including architectural, microarchitectural, thermal, power consumption characteristics, and the like.

[00122] In one embodiment, the processor 710 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 710 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 745. Accordingly, the processor 710 issues these coprocessor instructions (or control signals representing coprocessor

instructions) on a coprocessor bus or other interconnect, to coprocessor 745. Coprocessor(s) 745 accept and execute the received coprocessor instructions.

[00123] Referring now to **Figure 8**, shown is a block diagram of a first more specific exemplary system 800 in accordance with an embodiment of the present invention. As shown in **Figure 8**, multiprocessor system 800 is a point-to-point interconnect system, and includes a first processor 870 and a second processor 880 coupled via a point-to-point interconnect 850. Each of processors 870 and 880 may be some version of the processor 600. In one embodiment of the invention, processors 870 and 880 are respectively processors 710 and 715, while coprocessor 838 is coprocessor 745. In another embodiment, processors 870 and 880 are respectively processor 710 coprocessor 745.

[00124] Processors 870 and 880 are shown including integrated memory controller (IMC) units 872 and 882, respectively. Processor 870 also includes as part of its bus controller units point-to-point (P-P) interfaces 876 and 878; similarly, second processor 880 includes P-P interfaces 886 and 888. Processors 870, 880 may exchange information via a point-to-point (P-P) interface 850 using P-P interface circuits 878, 888. As shown in **Figure 8**, IMCs 872 and 882 couple the processors to respective memories, namely a memory 832 and a memory 834, which may be portions of main memory locally attached to the respective processors.

[00125] Processors 870, 880 may each exchange information with a chipset 890 via individual P-P interfaces 852, 854 using point to point interface circuits 876, 894, 886, 898. Chipset 890 may optionally exchange information with the coprocessor 838 via a high-performance interface 839. In one embodiment, the coprocessor 838 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.

[00126] 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.

[00127] Chipset 890 may be coupled to a first bus 816 via an interface 896. In one embodiment, first bus 816 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.

[00128] As shown in **Figure 8**, various I/O devices 814 may be coupled to first bus 816, along with a bus bridge 818 which couples first bus 816 to a second bus 820. In one embodiment, one or more additional processor(s) 815, 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 816. In one embodiment, second bus 820 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 820 including, for example, a keyboard and/or mouse 822, communication devices 827 and a storage unit 828 such as a disk drive or other mass storage device which may include instructions/code and data 830, in one embodiment. Further, an audio I/O 824 may be coupled to the second bus 820. Note that other architectures are possible. For example, instead of the point-to-point architecture of **Figure 8**, a system may implement a multi-drop bus or other such architecture.

[00129] Referring now to **Figure 9**, shown is a block diagram of a second more specific exemplary system 900 in accordance with an embodiment of the present invention. Like elements in **Figures 8 and 9** bear like reference numerals, and certain aspects of **Figure 8** have been omitted from **Figure 9** in order to avoid obscuring other aspects of **Figure 9**.

[00130] **Figure 9** illustrates that the processors 870, 880 may include integrated memory and I/O control logic ("CL") 872 and 882, respectively. Thus, the CL 872, 882 include integrated memory controller units and include I/O control logic. **Figure 9** illustrates that not only are the memories 832, 834 coupled to the CL 872, 882, but also that I/O devices 914 are also coupled to

the control logic 872, 882. Legacy I/O devices 915 are coupled to the chipset 890.

[00131] Referring now to **Figure 10**, shown is a block diagram of a SoC 1000 in accordance with an embodiment of the present invention. Similar elements in **Figure 6** bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In **Figure 10**, an interconnect unit(s) 1002 is coupled to: an application processor 1010 which includes a set of one or more cores 202A-N and shared cache unit(s) 606; a system agent unit 610; a bus controller unit(s) 616; an integrated memory controller unit(s) 614; a set or one or more coprocessors 1020 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 1030; a direct memory access (DMA) unit 1032; and a display unit 1040 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 1020 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.

[00132] 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.

[00133] Program code, such as code 830 illustrated in **Figure 8**, 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.

[00134] 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.

[00135] 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.

[00136] 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.

[00137] 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.

[00138] 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.

[00139] Figure 11 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 11 shows a program in a high level language 1102 may be compiled using an x86 compiler 1104 to generate x86 binary code 1106 that may be natively executed by a processor with at least one x86 instruction set core 1116. The processor with at least one x86 instruction set core 1116 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 1104 represents a compiler that is operable to generate x86 binary code 1106 (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 1116. Similarly, Figure 11 shows the program in the high level language 1102 may be

compiled using an alternative instruction set compiler 1108 to generate alternative instruction set binary code 1110 that may be natively executed by a processor without at least one x86 instruction set core 1114 (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 1112 is used to convert the x86 binary code 1106 into code that may be natively executed by the processor without an x86 instruction set core 1114. This converted code is not likely to be the same as the alternative instruction set binary code 1110 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 1112 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 1106.

Apparatus and Method for Reentering a Transactional Sequence With Hardware Transactional Memory

[00140] In many situations, a hardware transactional memory (HTM) transaction aborts after having performed much useful work, and thus a good deal of useful work is lost due to the all-or-nothing guarantee semantic attached to HTM implementations like transactional synchronization extensions (TSX). One approach to mitigate this is to support non-transactional loads and stores, but this approach is very expensive in hardware complexity. The embodiments of the invention described below include a much more efficient hardware extension that permits software to achieve efficient transactional exclusion while preserving the ability to benefit from partial transactions.

[00141] In general, to implement a volatile memory transaction, one has to (a) find some collection *C* of items that satisfy some criterion individually or collectively, and (b) perform some updates to state *S*, where the updates

depends on *C*. In some cases (a) may be trivial, but in general, significant work may be needed in (a) to search, filter through some large and continuously changing dataset to generate *C*. To provide Atomicity and Consistency, transactional software generally uses mutual exclusion among concurrent threads in shared memory to protect one transaction on one CPU from either obtaining inconsistent views of *C* or generating inconsistent views of *S* in other CPUs. Such mutual exclusion spans from the start of (a) to the end of (b). A common way of achieving mutual exclusion is locking.

[00142] Hardware Transactional Memory (HTM) is a more recent capability that provides for efficient and scalable mutual exclusion based on detecting and acting on concurrent accesses that break consistency of transactions. In current HTM implementations, hardware treats *any* memory conflict (e.g., reads racing with writes, or writes racing with writes) at a cacheline granularity as a reason to abort a transaction. Hardware may also abort a transaction for capacity reasons. When an HTM transaction is aborted, *all changes* to both memory and machine state (i.e., architectural registers) *are discarded*, so that the thread performing the HTM transaction is repositioned at the start with a restored architectural state as of the point when it had previously started the ill-fated transaction. The one change that is different is that in a restricted transactional memory (RTM), software is positioned at an exception vector so that it can handle the abort condition in an appropriate manner. Usually software reattempts the same transaction or backs off to do something else, or backs off and then reattempts. Thus, a program using transactional extensions can lose efficiency. The latency cost of redoing all of the work lost due to an abort can be high, and so latency-critical algorithms become problematic to deploy with transactional extensions.

[00143] This "loss of memory" described above is an inefficiency since the CPU work that is done in the course of an aborted transactions is lost. So long as HTM aborts are rare, it is not an issue, but it is generally difficult to guarantee that HTM aborts will be rare as this depends on many factors.

Consider the analogy of students who take practice exams to improve their scores on the real exam. Often this is an artifact of being able to recall speedily which alternatives were already considered and rejected during the practice session. Likewise, in software algorithms there are situations where it is considerably more expensive to compute or find a result, but very trivial to determine that a "hinted" solution is indeed correct. For example:

· Equation Solving: It takes a lot more iterations and work to find the roots of an equation such as,

$$0 = \sum_{i=0}^{i=N} \alpha_i x^i \sin\left(\frac{ipx}{N}\right)$$

than it takes to determine whether a given value of x is (or is not) a root.

· Decoding: Many comparisons are needed to find some anagram of a string *str*, that is a permitted word in a dictionary, while it is straightforward to determine whether a particular anagram is a dictionary word.

· Linearizability: Determining whether there is any partial or total order from among N operations that does not violate a set of invariants is difficult, but checking that a chosen order does not violate the same invariants is straightforward.

[00144] In many data structures, one needs to stabilize a data structure such as a B-tree with a lock, identify a particular node where a new item may be inserted or deleted, and then in some cases, perform some corrective operation (such as rebalancing the tree). When HTM is used to remove the locking, and a lot of work has gone into identifying the particular insertion/deletion point, a transaction abort becomes costly. Suppose a designer could protect the effort by remembering the path to the "found node" in one part of the transaction. Thus, if the transaction were to abort after that point, the designer can verify that the transaction can be resumed and joined at a safe point to proceed to completion.

[00145] Arguably, in the above example, it is possible to just split the original transaction into two HTM transactions at all times, with the first transaction producing values that are used in the second transaction (after validating). The problem with this approach is that now the price of revalidation has to be paid every time, and subtracts from the benefit of using HTM. Instead, an approach that requires revalidation only for the aborting subset of transactions gives the best of both worlds: high performance and scalability when there are no aborts, and a limited amount of extra work (to revalidate) for the subset of transactions that abort.

[00146] The embodiments of the invention address the case when it is not possible to obtain a reasonable guarantee that HTM aborts will be rare. One embodiment provides for a way for circuitry and/or software to salvage a chunk of the work already performed up to the point of the abort, and to reuse the results with a high probability of success when restarting the transaction. Hardware changes and new instructions may be used, but these techniques are specifically aimed at keeping both the hardware complexity and the software burden of reusing salvaged information very low.

[00147] One embodiment of the invention achieves the same benefit as non-transactional stores, but avoids the complexity and cache overheads that go into non-transactional stores. Non-transactional stores are hard to implement because various parts of the out of order machine have to be specialized for treating non-transactional stores differently from transactional stores. Further, it is difficult to bound all corner cases such as those arising from one instruction performing a transactional store to value A and another instruction performing a non-transactional store to value B, where, A and B happen to share a cache line.

[00148] Instead, one embodiment provides a benefit over non-transactional stores by checkpointing (e.g., saving) specific portions of state in a lightweight HTM scratchpad:

[1] a subset of the application state which may include, for example, values of GPRs, ZMM registers, RFLAGS, mask registers (k), and the instruction pointer register (RIP). However, not everything is saved; a 64-bit mask value is supplied by software, which tells hardware at a particular point inside an HTM transaction those and only those values to checkpoint.

[2] a specified (potentially small) number of values that the application writes into the scratchpad, to supplement the above architectural state. What this is, is entirely up to the application software. It can be nothing, if the application so chooses.

[00149] In one embodiment, instructions that can be thought of as lightweight versions of XSAVE/XRESTORE effectively checkpoint the indicated state and restore it into architectural registers. The values that the application specifically writes into the scratchpad can be read by the application at any time. In this sense, they are like non-temporal stores, but they are not in memory space at all and have no bearing on cache alignment. In one embodiment, the application does not write or read these values by ordinary MOV instructions, but uses a different pair of instructions.

[00150] In one embodiment, the scratchpad is implemented within the execution engine; consequently, there is no cache/coherence/store ordering overhead. The amount of architectural state saved and restored is relatively small and controlled by software. Thus, only a handful of architectural registers have to be saved and these data transfers are expected to complete quickly without burdening the load/store ports or the cache subsystem.

[00151] For architectures that do not allow transactions to survive a context switch (e.g., TSX) the saving/restoring of the additional architectural state of the scratchpad across a context switch can be made optional to avoid increasing the overhead of context switches.

[00152] Existing processors that support HTM execution support scratchpads with non-transactional stores and transaction pause/resume instructions. Those instructions have a high complexity to implement in hardware or/and have very

high latency. As mentioned, the embodiments of the invention avoid complexity by taking a snapshot only as directed by software and only for the subset of architectural state that is designated as interesting by software. This allows the designer of a software transaction and the compiler to work together to minimize the amount of work necessary to determine whether or not to continue a transaction. For example, the compiler can apply value-flow analysis to limit what machine state needs to be saved for resumption, depending on where a snapshot is taken, while allowing the software designer to decide when it is worthwhile to take a snapshot. The software designer also has the additional flexibility of passing values meaningful to him/her at any point through the USRSTATE fields (see description below) and not just at the points where a snapshot of the state has been taken. The HTM scratchpad may be lightweight, not requiring saving/restoring the additional architecture state upon a context switch, in contrast to existing HTM implementations and proposals.

[00153] Figure 12 illustrates the flow of an exemplary transaction comprising a sequence of instructions 1206 utilizing transactional synchronization extensions (TSX). When an HTM transaction begins at 1201, the CPU implicitly saves the architectural state at 1202 and then in case it aborts partway at 1204, the CPU resteers the transaction at 1205 to a designated RIP (which may or may not be the same as the starting RIP) but with the architectural state restored from the HTM start point 1201. The abort code is not illustrated in Figure 12 for simplicity.

[00154] Figure 13 illustrates the same transaction a little differently, to highlight the manner in which the embodiments of the invention will change the flow. In particular, an abort 1204 is illustrated partway through a transaction 1206 that consists of L instructions in total. This abort is shown occurring at some instruction numbered K inside the transaction. When the abort happens, application code may execute an abort handler (with the same architectural state as the point at which the HTM transaction was entered), and then retry

the same transaction some time later or soon after, or choose to do something else.

[00155] Assume now that in **Figure 13**, a lot of work was completed up to instruction J and a few key results were derived by the application. Suppose the application designer knows that at instruction J, he/she would like to save certain registers (e.g., RCX, ZMM5), and a few other pieces of data he/she has computed, in a very light-weight fashion just as insurance against the possible loss of work in case the transaction aborted.

[00156] In one embodiment, the programmer would do so by using a 64 bit "MASK" value to tell the CPU what to save. **Figure 14** illustrates a scratchpad 1410 within an execution engine 1415 (e.g., an execution unit of a processor core) with an exemplary mask value comprising a GPR component 1401, AVX components 1402-1403 and an “Other” component 1404 which may be application-specific. The programmer may use some space set aside in the scratchpad 1410 and covered by the "Other" portion 1404 in the mask.

[00157] In the illustrated embodiment, of the 64 bits in the mask, 16 bits 1401 are used for 16 GPRs, another 32 bits 1402-1403 for AVX512 vector registers, and another 16 bits 1404 for additional things such as RFLAGS, RIP, and the k[] registers. In addition, in one embodiment, some number of quadwords are set aside for discretionary use by the user, who may use them to capture values of interest. In **Figure 14**, for example, the second quadword's mask in the USRSTATE portion 1420 is 1, which means that a value will be saved at the corresponding offset inside the scratchpad 1410 by the user. All other state for which the scratchpad mask bits are 0 is undefined. In **Figure 14**, the RIP 1421 and RFLAGS 1422 (whose mask bits 48 and 49 are both 1) are also saved, though not shown explicitly in the scratchpad 1410.

[00158] **Figure 15** illustrates a similar start of the HTM transaction 1201 and set of instructions 1206 to be executed transactionally as in **Figure 13**, but with the ability to save and continue from a known good state through execution of a “restart HTM from scratchpad” operation 1501 and a revalidation operation

1502. In the specific example illustrated in **Figure 15**, at instruction J, application software saves the necessary pieces of state in the scratchpad 1410. Now, when there an abort 1204 is encountered at instruction K, control passes back to application code 1500 outside the transaction via the illustrated abort path. The application code 1500 now has the option to re-enter the HTM transaction at instruction J+1 instead of at instruction 1, and with the state it needs restored from the scratchpad 1410. In the forward path (i.e., before the abort occurs), a checkpointing instruction saves the state to the scratchpad 1410 as of instruction J (i.e., next instruction = J+1) and hardware automatically resteers it to that instruction. The checkpointing may be implemented by checkpointing circuitry within the memory execution unit (MEU). But in the return path, the hardware resteers the flow to a software designated path first with the HTM in control. In that path, initiated by a restart of HTM from scratchpad operation 1501, the software does its own validation 1502 of the saved state (i.e., whatever data it cared about and saved in the USRSTATE in addition to what it saved in the architectural state), then clears the scratchpad 1410, and resumes at instruction J+1. Not shown in **Figure 15** is an alternative embodiment where software decides that the validation failed (i.e., revalidation operation 1502 returns a failure) and thus backs out and repeats the entire transaction from the start (instruction 1). That eventuality is expected to be very rare else SW would not have bothered to try to save and continue.

[00159] A method in accordance with one embodiment of the invention is illustrated in **Figures 16-17**. The method may be implemented within the context of the processor architectures described above but is not limited to any particular processor architecture. For example, the operations set forth in **Figures 16-17** may be implemented by circuitry within a translational memory unit (e.g., within a memory execution unit that supports transactional memory operations), by program code executed by a general purpose execution unit, or by any combination of program code and circuitry.

[00160] At 1601, a transactional sequence of instructions are initiated (e.g., from instructions 1 to L in **Figure 15**). At 1602, a point is detected within the instruction sequence at which a checkpoint operation is desired. As mentioned, this point may be identified by the software designer and/or compiler. At 1603, mask values stored in a mask register are read to determine the particular register values and (optionally) other application-specific data to be saved as a checkpoint. At 1604, the register values and application-specific data are stored a scratchpad memory. If an abort condition is detected, determined at 1605, then the process proceeds to **Figure 17**. If not, then execution of the transactional instruction sequence completes at 1606.

[00161] At 1701, in response to a detected abort condition, control is passed to circuitry and/or program code outside of the transactional sequence of instructions. For example, the program code may comprise a handler for processing an abort condition. At 1702 the HTM is restarted using state stored in the scratchpad memory (e.g., designated register values, application-specific state data, etc). At 1703, revalidation of the state retrieved from the scratchpad is performed. If successful, determined at 1704, then at 1705 the execution of the transactional instruction sequence is restarted at the instruction following the checkpoint (e.g., instruction J+1 in the example above) using the validated state retrieved from the scratchpad memory. The checkpoint is then cleared from the scratchpad memory at 1707. If revalidation is not successful at 1704, then at 1706 execution of the transaction is restarted from the beginning of the transactional instruction sequence (e.g., instruction 1 in the example above).

[00162] In the foregoing specification, the embodiments of invention have been described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

[00163] 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.

[00164] 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.). 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. 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:

one or more cores to execute instructions and process data;

execution circuitry within at least one of the cores to execute a transactional sequence of instructions;

one or more mask registers to store a mask value identifying a specified set of architectural state to be saved upon reaching a particular instruction within the transactional sequence of instructions; and

a scratchpad memory within the execution circuitry to store the specified set of architectural state upon reaching the particular instruction within the sequence of instructions.

2. The processor as in claim 1 wherein the particular instruction is to be identified by a compiler or an application designer.

3. The processor as in claim 1 wherein the specified set of architectural state includes a specified set of register values and/or application-specific data.

4. The processor as in claim 3 further comprising:

abort detection logic to detect an abort condition associated with the transactional sequence of instructions; and

resteering logic to restore the architectural state from the scratchpad memory and to identify a next instruction within the transactional sequence of instructions from which transactional execution is to resume.

5. The processor as in claim 4 wherein the next instruction directly follows the particular instruction in the transactional sequence.

6. The processor as in claim 5 further comprising:

circuitry to clear the scratchpad memory upon the resteering logic restoring the architectural state.

7. The processor as in claim 4 further comprising:

revalidation logic to revalidate the architectural state prior to executing the next instruction in the transactional sequence, wherein if revalidation fails, the execution circuitry is to reexecute the transactional sequence from a first instruction of the transactional sequence.

8. The processor as in claim 7 wherein the revalidation logic comprises revalidation program code executed by the processor to perform the revalidation of the architectural state.

9. A method comprising:

generating a mask value to identify a specified set of architectural state to be saved upon reaching a particular instruction within a transactional sequence of instructions;

storing the specified set of architectural state upon reaching the particular instruction within the sequence of instructions.

10. The method as in claim 9 wherein the particular instruction is to be identified by a compiler or an application designer.

11. The method as in claim 9 wherein the specified set of architectural state includes a specified set of register values and/or application-specific data.

12. The method as in claim 11 further comprising:

detecting an abort condition associated with the transactional sequence of instructions; and

restoring the architectural state from the scratchpad memory and to identify a next instruction within the transactional sequence of instructions from which transactional execution is to resume.

13. The method as in claim 12 wherein the next instruction directly follows the particular instruction in the transactional sequence.

14. The method as in claim 13 further comprising:

clearing the scratchpad memory upon restoring the architectural state.

15. The method as in claim 12 further comprising:

revalidating the architectural state prior to executing the next instruction in the transactional sequence, wherein if revalidation fails, then reexecuting the transactional sequence from a first instruction of the transactional sequence.

16. An apparatus comprising:

means for generating a mask value to identify a specified set of architectural state to be saved upon reaching a particular instruction within a transactional sequence of instructions;

means for storing the specified set of architectural state upon reaching the particular instruction within the sequence of instructions.

17. The apparatus as in claim 16 wherein the particular instruction is to be identified by a compiler or an application designer.

18. The apparatus as in claim 16 wherein the specified set of architectural state includes a specified set of register values and/or application-specific data.

19. The apparatus as in claim 18 further comprising:

means for detecting an abort condition associated with the transactional sequence of instructions; and

means for restoring the architectural state from the scratchpad memory and to identify a next instruction within the transactional sequence of instructions from which transactional execution is to resume.

20. The apparatus as in claim 19 wherein the next instruction directly follows the particular instruction in the transactional sequence.

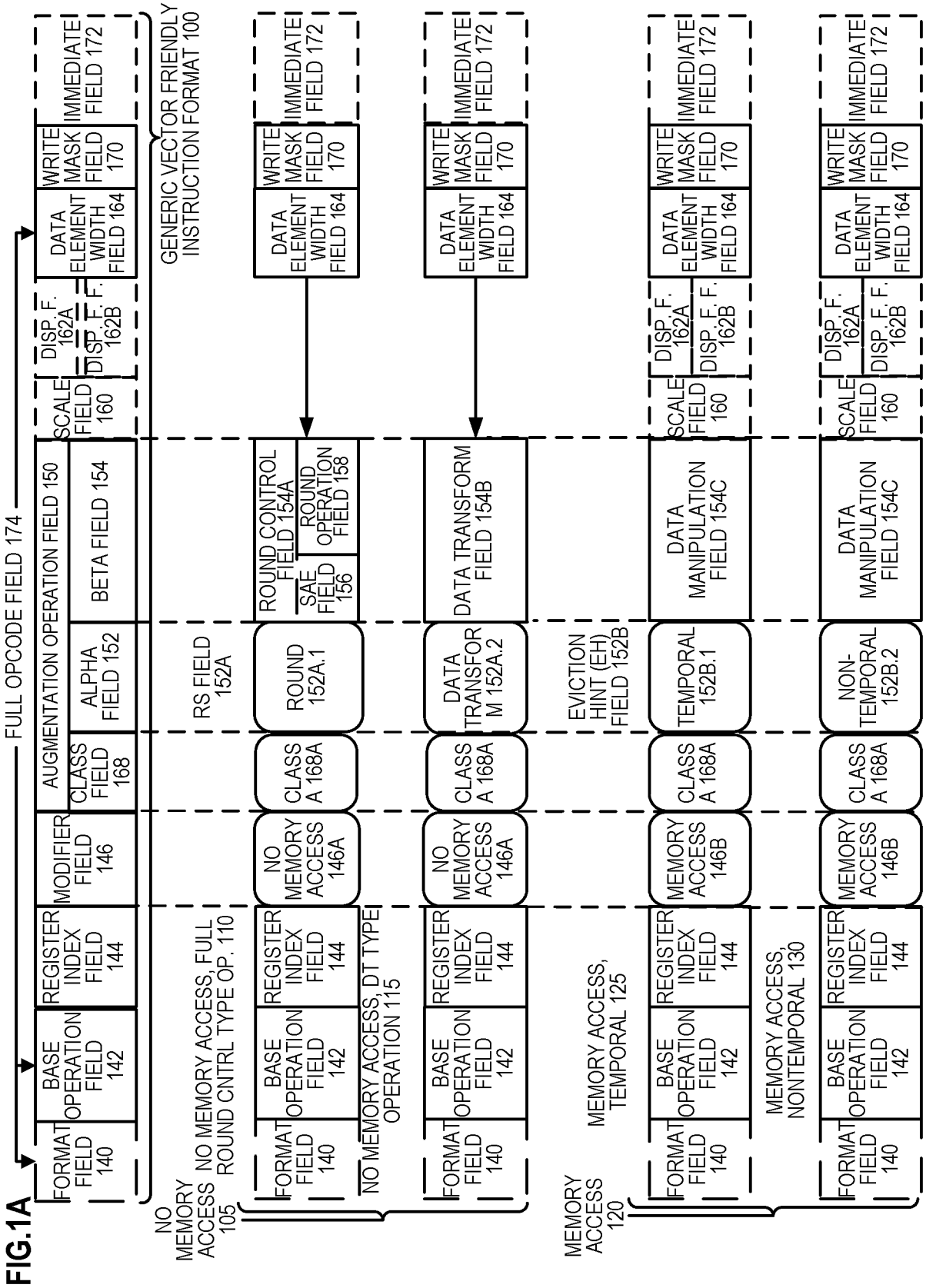
21. The apparatus as in claim 20 further comprising:

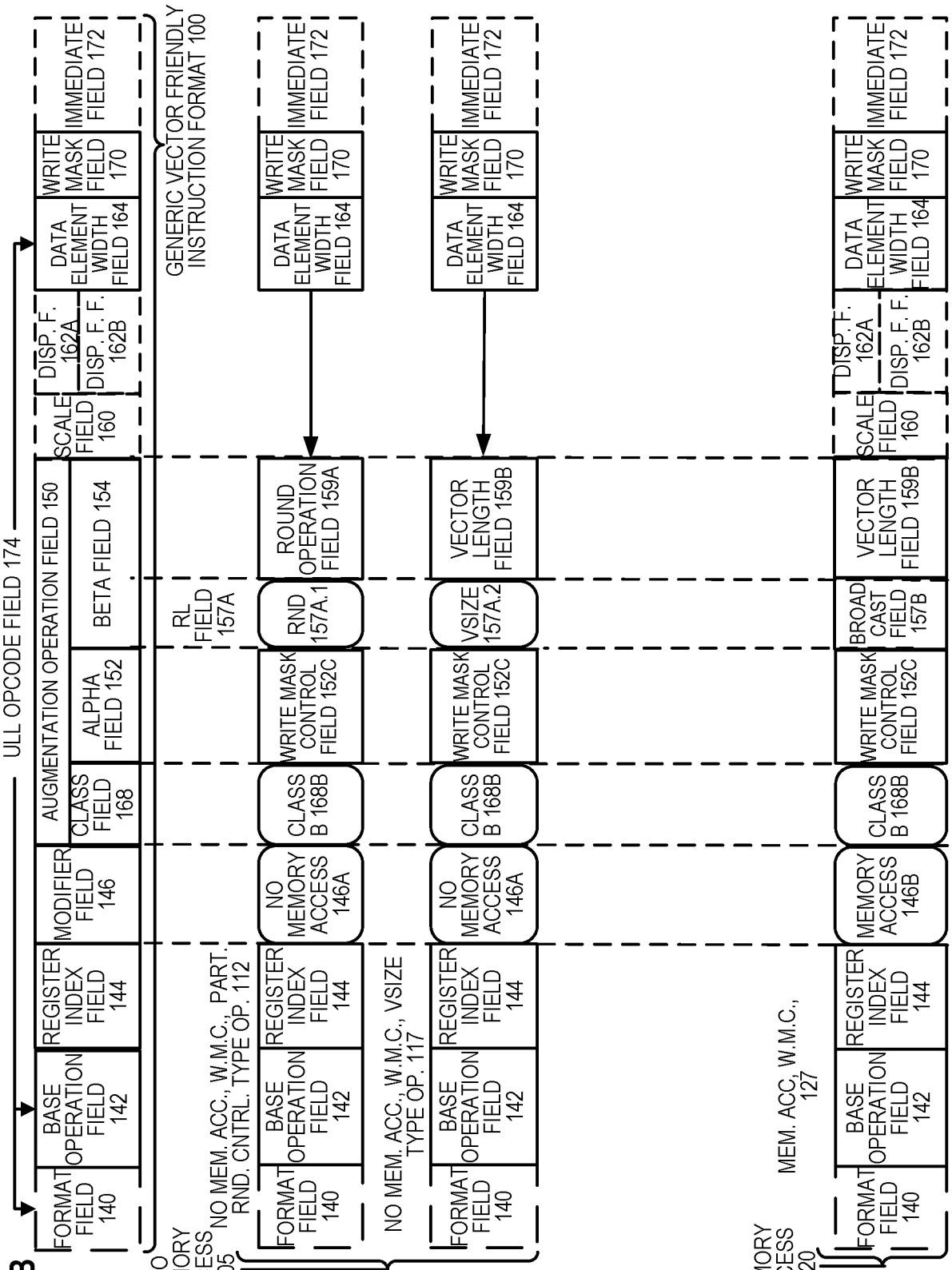
clearing the scratchpad memory upon restoring the architectural state.

22. The apparatus as in claim 19 further comprising:

means for revalidating the architectural state prior to executing the next instruction in the transactional sequence, wherein if revalidation fails, then reexecuting the transactional sequence from a first instruction of the transactional sequence.

FIG.1A


FULL OPCODE FIELD 174
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 100
FORMAT FIELD 140
BASE OPERATION FIELD 142
REGISTER INDEX FIELD 144
MODIFIER FIELD 146
AUGMENTATION OPERATION FIELD 150
CLASS FIELD 168
ALPHA FIELD 152
BETA FIELD 154
SCALE FIELD 160
DISP. F. 162A
DISP. F. F. 162B
DATA ELEMENT WIDTH FIELD 164
WRITE MASK FIELD 170
IMMEDIATE FIELD 172
NO MEMORY ACCESS 105
NO MEMORY ACCESS, FULL ROUND CNTRL TYPE OP. 110
NO MEMORY ACCESS, DT TYPE OPERATION 115
RS FIELD 152A
NO MEMORY ACCESS 146A
CLASS A 168A
ROUND 152A.1
ROUND CONTROL FIELD 154A
SAE FIELD 156
ROUND OPERATION FIELD 158
DATA TRANSFORM 152A.2
DATA TRANSFORM FIELD 154B
MEMORY ACCESS 120
MEMORY ACCESS, TEMPORAL 125
MEMORY ACCESS, NONTEMPORAL 130
EVICTION HINT (EH) FIELD 152B
MEMORY ACCESS 146B
TEMPORAL 152B.1
NON-TEMPORAL 152B.2
DATA MANIPULATION FIELD 154C


FIG.1B
ULL OPCODE FIELD 174
GENERIC VECTOR FRIENDLY INSTRUCTION FORMAT 100
NO MEMORY ACCESS 105
MEMORY ACCESS 120
FORMAT FIELD 140
BASE OPERATION FIELD 142
REGISTER INDEX FIELD 144
MODIFIER FIELD 146
AUGMENTATION OPERATION FIELD 150
CLASS FIELD 168
ALPHA FIELD 152
BETA FIELD 154
SCALE FIELD 160
DISP. F. 162A
DISP. F. F. 162B
DATA ELEMENT WIDTH FIELD 164
WRITE MASK FIELD 170
IMMEDIATE FIELD 172
NO MEM. ACC., W.M.C., PART. RND. CNTRL. TYPE OP. 112
NO MEMORY ACCESS 146A
CLASS B 168B
WRITE MASK CONTROL FIELD 152C
RL FIELD 157A
RND 157A.1
ROUND OPERATION FIELD 159A
NO MEM. ACC., W.M.C., VSIZE TYPE OP. 117
VSIZE 157A.2
VECTOR LENGTH FIELD 159B
MEM. ACC., W.M.C., 127
MEMORY ACCESS 146B
BROAD CAST FIELD 157B

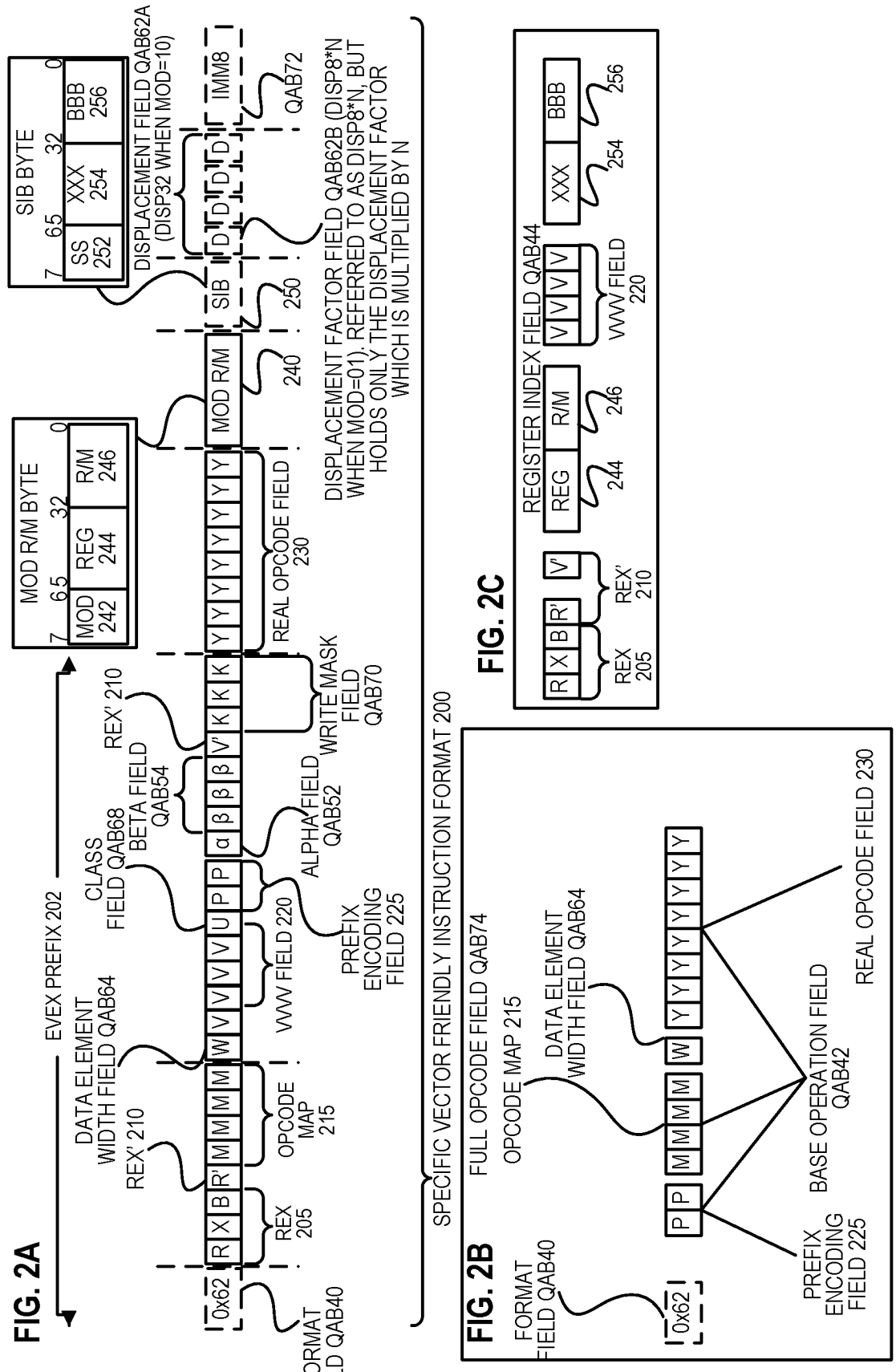

FIG. 2A
EVEX PREFIX 202
0x62
FORMAT FIELD QAB40
REX 205
REX' 210
OPCODE MAP 215
DATA ELEMENT WIDTH FIELD QAB64
CLASS FIELD QAB68
BETA FIELD QAB54
REX' 210
VVV FIELD 220
ALPHA FIELD QAB52
PREFIX ENCODING FIELD 225
WRITE MASK FIELD QAB70
REAL OPCODE FIELD 230
MOD R/M BYTE
MOD 242
REG 244
R/M 246
SIB BYTE
SS 252
XXX 254
BBB 256
DISPLACEMENT FIELD QAB62A (DISP32 WHEN MOD=10)
DISPLACEMENT FACTOR FIELD QAB62B (DISP8*N WHEN MOD=01), REFERRED TO AS DISP8*N, BUT HOLDS ONLY THE DISPLACEMENT FACTOR WHICH IS MULTIPLIED BY N
SIB
IMM8
QAB72
SPECIFIC VECTOR FRIENDLY INSTRUCTION FORMAT 200
FIG. 2B
FULL OPCODE FIELD QAB74
FORMAT FIELD QAB40
OPCODE MAP 215
DATA ELEMENT WIDTH FIELD QAB64
BASE OPERATION FIELD QAB42
PREFIX ENCODING FIELD 225
REAL OPCODE FIELD 230
FIG. 2C
REGISTER INDEX FIELD QAB44
REX 205
REX' 210
REG 244
R/M 246
VVV FIELD 220
XXX 254
BBB 256

FIG. 2D

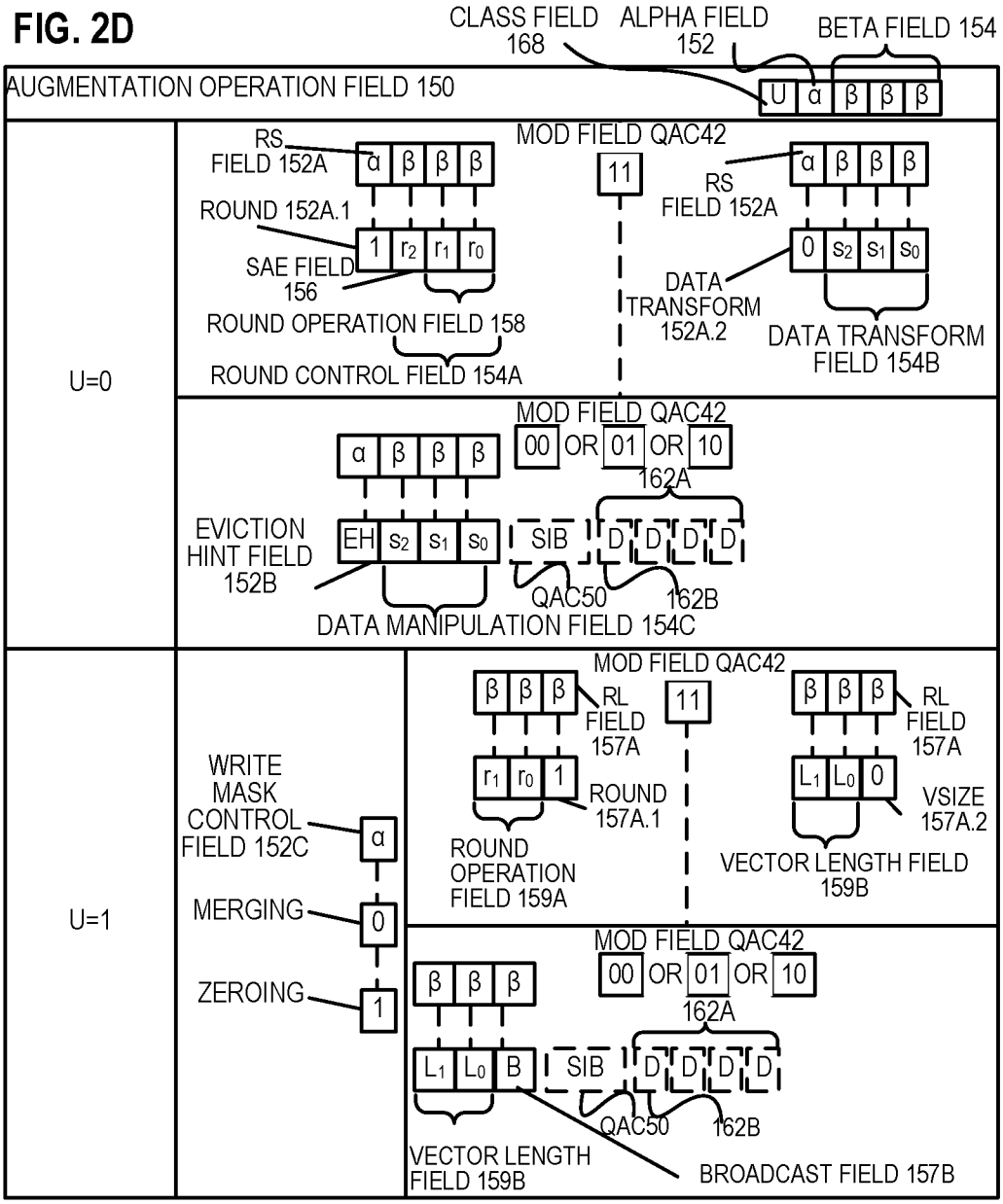
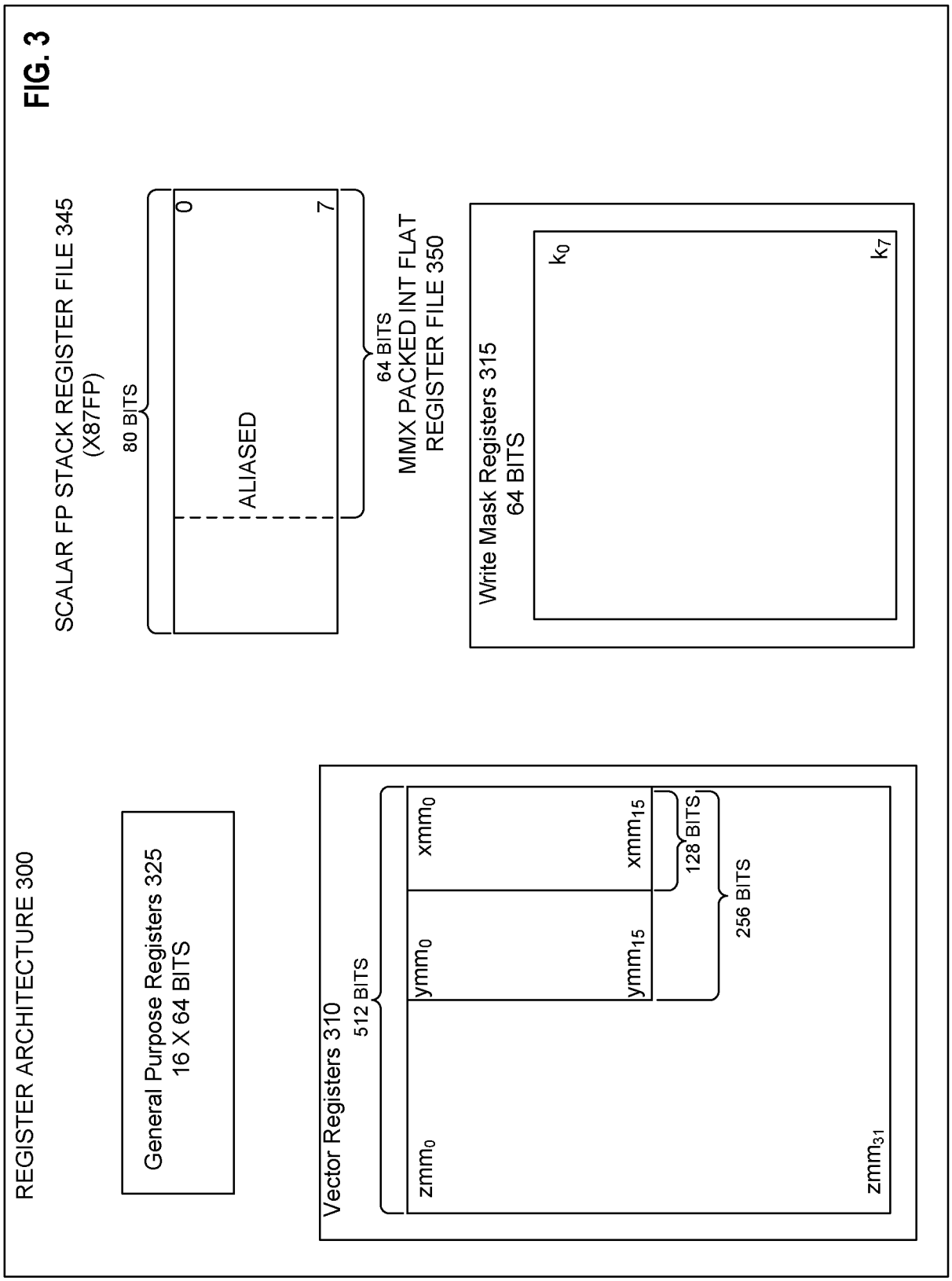

CLASS FIELD 168
ALPHA FIELD 152
BETA FIELD 154
AUGMENTATION OPERATION FIELD 150
U α β β β
U=0
RS FIELD 152A
α β β β
ROUND 152A.1
1 r2 r1 r0
SAE FIELD 156
ROUND OPERATION FIELD 158
ROUND CONTROL FIELD 154A
MOD FIELD QAC42
11
RS FIELD 152A
α β β β
0 s2 s1 s0
DATA TRANSFORM 152A.2
DATA TRANSFORM FIELD 154B
MOD FIELD QAC42
00 OR 01 OR 10
162A
α β β β
EVICTION HINT FIELD 152B
EH s2 s1 s0
SIB
D D D D
QAC50
162B
DATA MANIPULATION FIELD 154C
U=1
WRITE MASK CONTROL FIELD 152C
α
MERGING
0
ZEROING
1
MOD FIELD QAC42
β β β
RL FIELD 157A
11
r1 r0 1
ROUND 157A.1
ROUND OPERATION FIELD 159A
β β β
RL FIELD 157A
L1 L0 0
VSIZE 157A.2
VECTOR LENGTH FIELD 159B
MOD FIELD QAC42
00 OR 01 OR 10
162A
β β β
L1 L0 B
SIB
D D D D
QAC50
162B
VECTOR LENGTH FIELD 159B
BROADCAST FIELD 157B

FIG. 3


REGISTER ARCHITECTURE 300
General Purpose Registers 325
16 X 64 BITS
SCALAR FP STACK REGISTER FILE 345
(X87FP)
80 BITS
0
7
ALIASED
64 BITS
MMX PACKED INT FLAT
REGISTER FILE 350
Write Mask Registers 315
64 BITS
k0
k7
Vector Registers 310
512 BITS
zmm0
ymm0
xmm0
ymm15
xmm15
128 BITS
256 BITS
zmm31

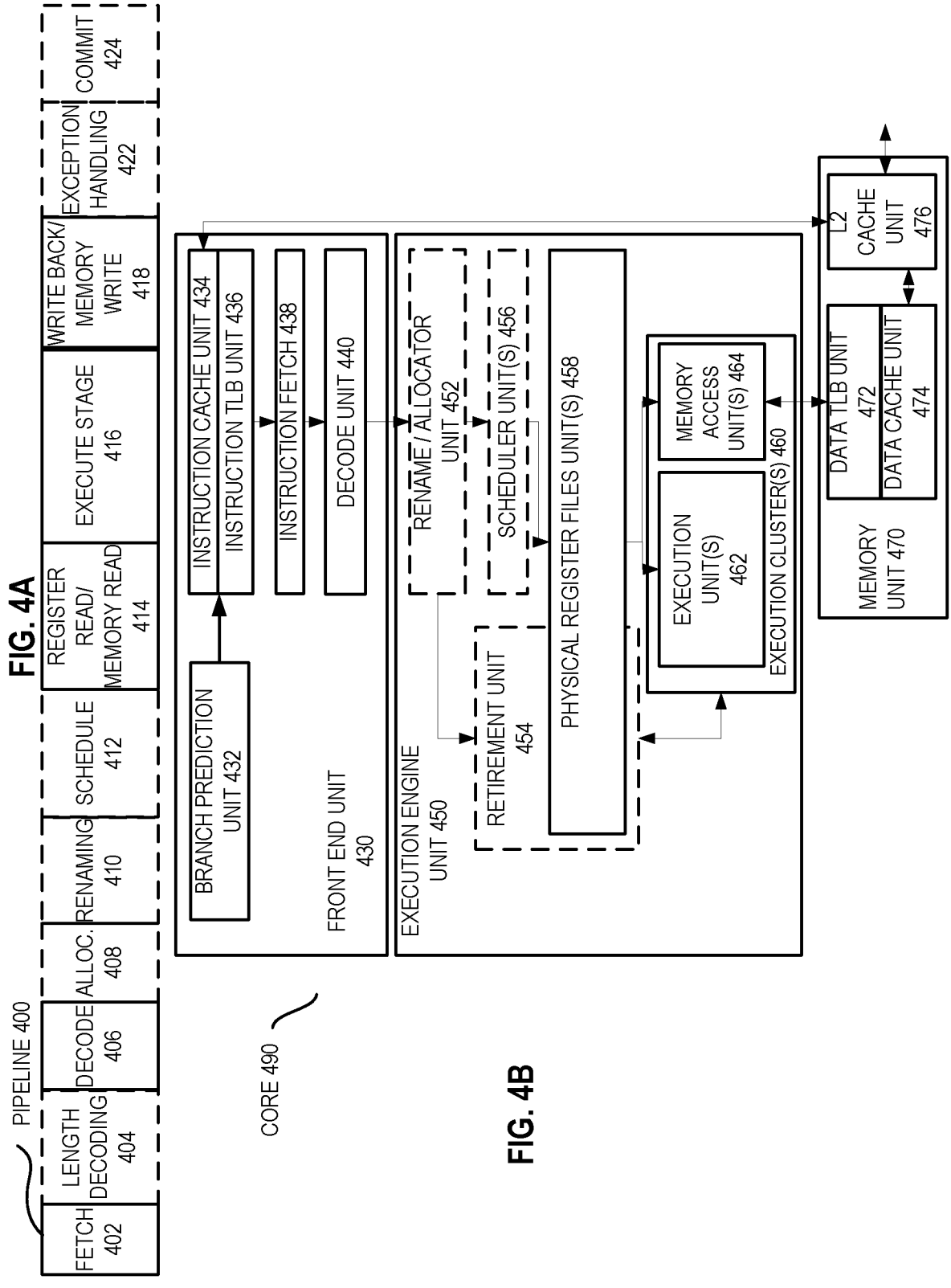

PIPELINE 400
FETCH 402
LENGTH DECODING 404
DECODE 406
ALLOC. 408
RENAMING 410
SCHEDULE 412
REGISTER READ/ MEMORY READ 414
EXECUTE STAGE 416
WRITE BACK/ MEMORY WRITE 418
EXCEPTION HANDLING 422
COMMIT 424
CORE 490
FRONT END UNIT 430
BRANCH PREDICTION UNIT 432
INSTRUCTION CACHE UNIT 434
INSTRUCTION TLB UNIT 436
INSTRUCTION FETCH 438
DECODE UNIT 440
EXECUTION ENGINE UNIT 450
RENAME / ALLOCATOR UNIT 452
RETIREMENT UNIT 454
SCHEDULER UNIT(S) 456
PHYSICAL REGISTER FILES UNIT(S) 458
EXECUTION UNIT(S) 462
MEMORY ACCESS UNIT(S) 464
EXECUTION CLUSTER(S) 460
MEMORY UNIT 470
DATA TLB UNIT 472
DATA CACHE UNIT 474
L2 CACHE UNIT 476

FIG. 4A

FIG. 4B

FIG. 5B

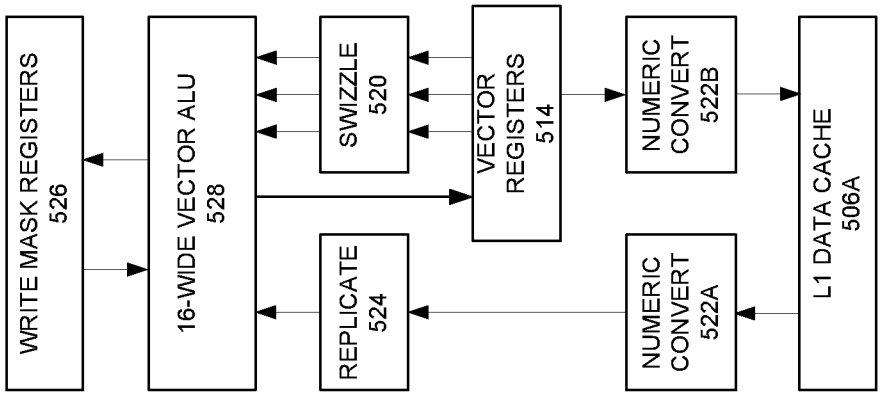
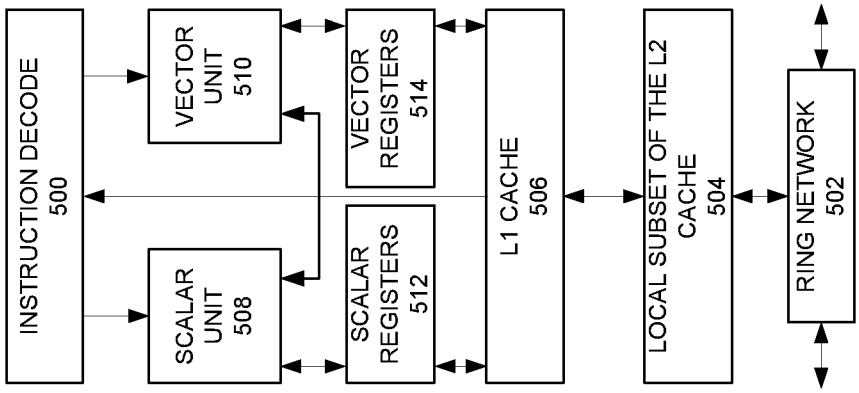

WRITE MASK REGISTERS 526
16-WIDE VECTOR ALU 528
REPLICATE 524
SWIZZLE 520
VECTOR REGISTERS 514
NUMERIC CONVERT 522A
NUMERIC CONVERT 522B
L1 DATA CACHE 506A

FIG. 5A


INSTRUCTION DECODE 500
SCALAR UNIT 508
VECTOR UNIT 510
SCALAR REGISTERS 512
VECTOR REGISTERS 514
L1 CACHE 506
LOCAL SUBSET OF THE L2 CACHE 504
RING NETWORK 502

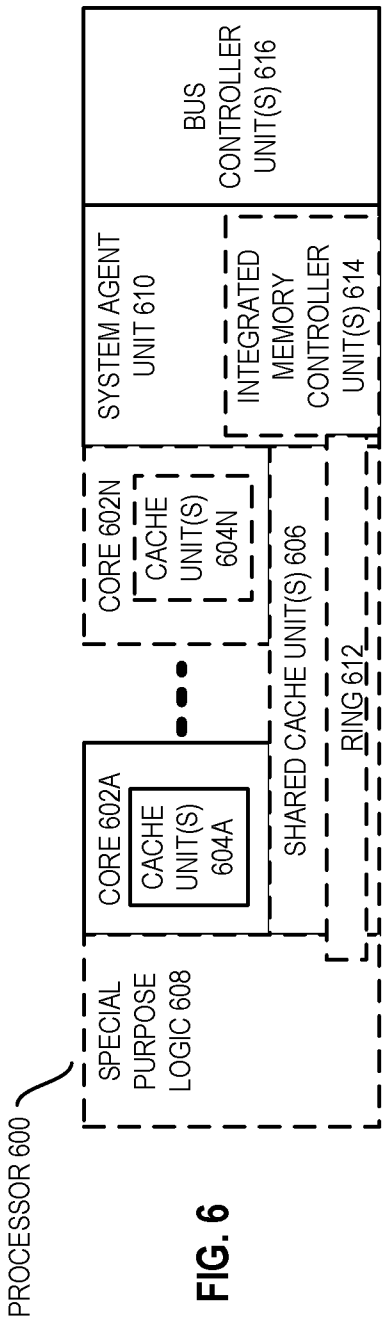

PROCESSOR 600
SPECIAL PURPOSE LOGIC 608
CORE 602A
CACHE UNIT(S) 604A
CORE 602N
CACHE UNIT(S) 604N
SHARED CACHE UNIT(S) 606
RING 612
SYSTEM AGENT UNIT 610
INTEGRATED MEMORY CONTROLLER UNIT(S) 614
BUS CONTROLLER UNIT(S) 616

FIG. 6

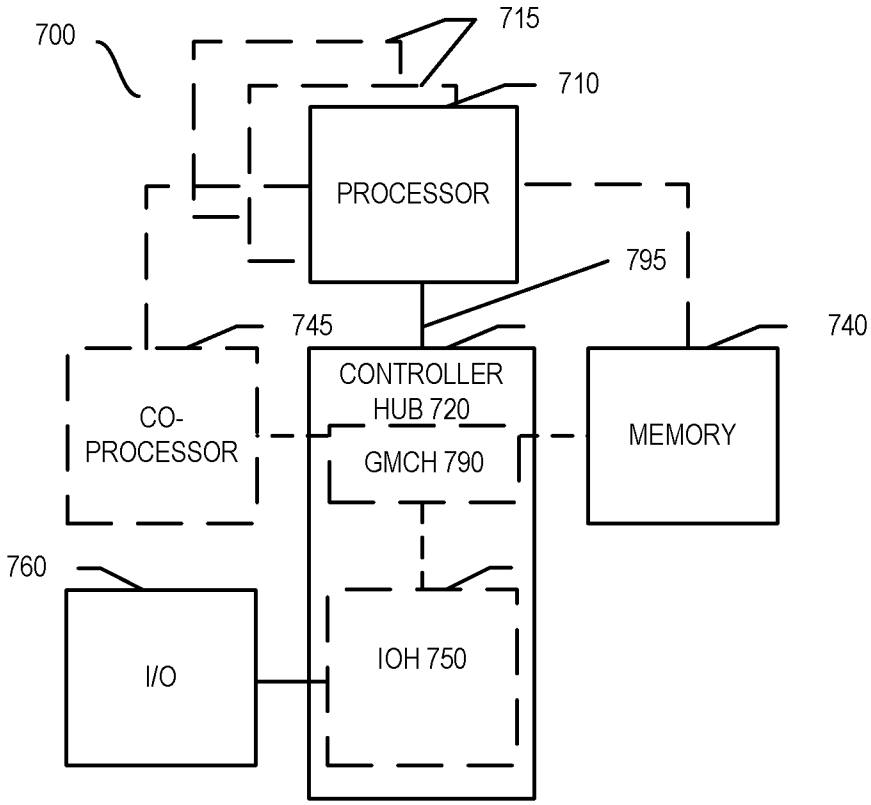

700
715
710
PROCESSOR
795
745
740
CONTROLLER
HUB 720
CO-
PROCESSOR
MEMORY
GMCH 790
760
I/O
IOH 750

FIG. 7

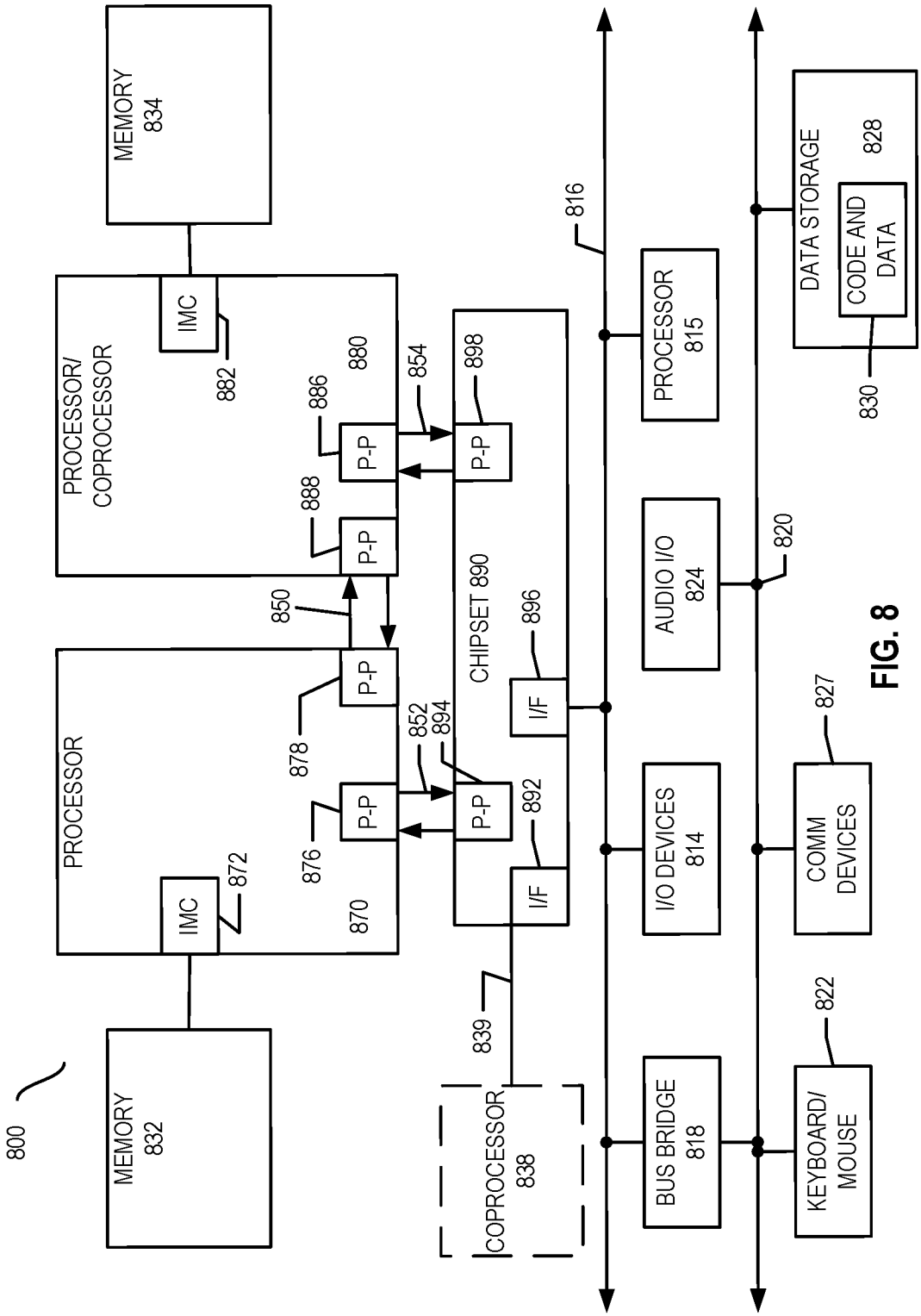

800
MEMORY 832
MEMORY 834
PROCESSOR 870
IMC 872
PROCESSOR/ COPROCESSOR 880
IMC 882
P-P 876
P-P 878
P-P 886
P-P 888
850
852
854
CHIPSET 890
P-P 892
P-P 894
P-P 898
I/F 896
I/F 892
839
COPROCESSOR 838
816
PROCESSOR 815
AUDIO I/O 824
820
I/O DEVICES 814
827
COMM DEVICES
BUS BRIDGE 818
822
KEYBOARD/ MOUSE
DATA STORAGE 828
CODE AND DATA
830

FIG. 8

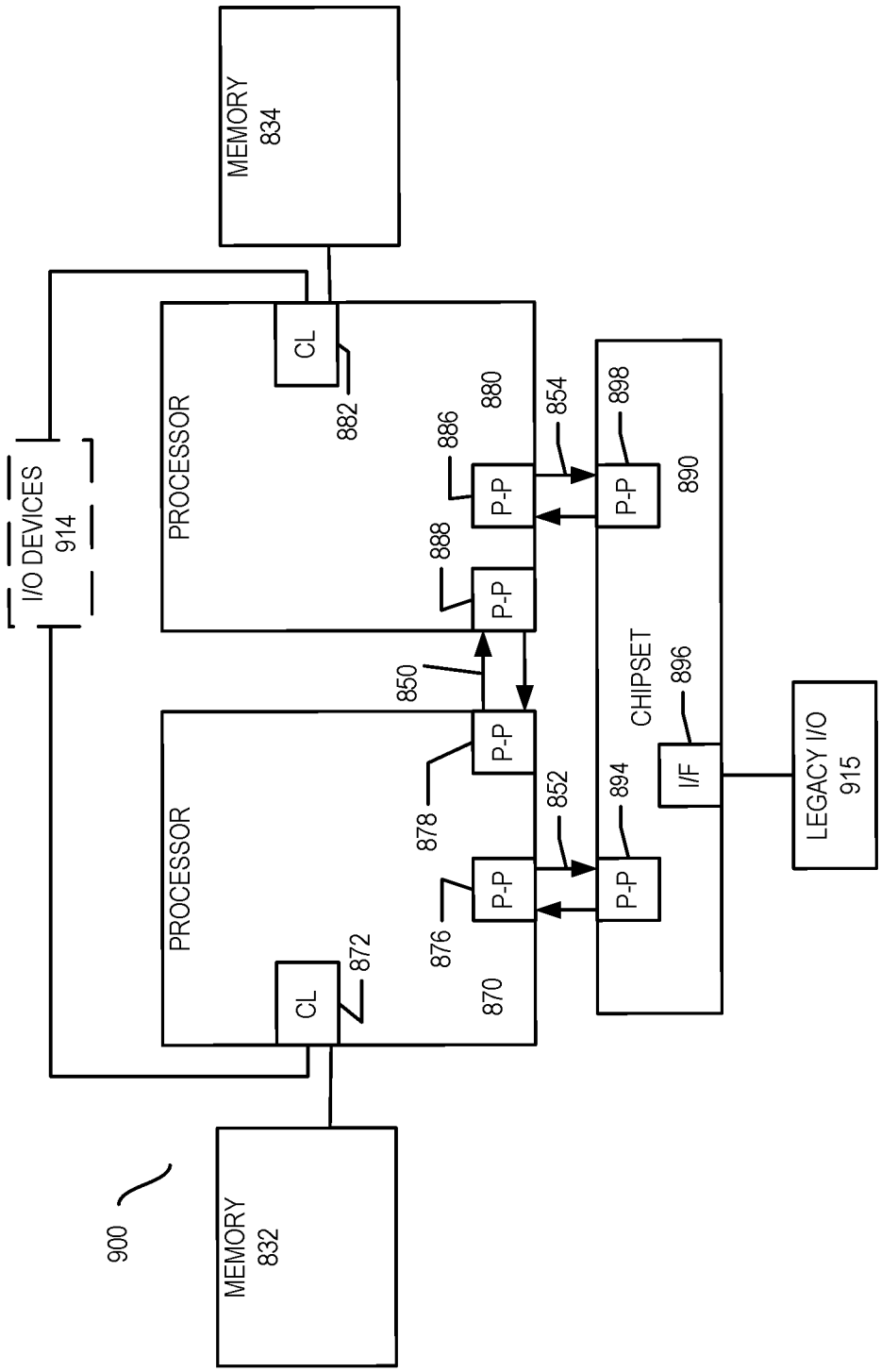

MEMORY
834
CL
882
PROCESSOR
880
886
P-P
888
P-P
854
898
P-P
890
I/O DEVICES
914
850
P-P
878
852
894
P-P
CHIPSET
896
I/F
LEGACY I/O
915
876
P-P
872
CL
PROCESSOR
870
900
MEMORY
832

FIG. 9

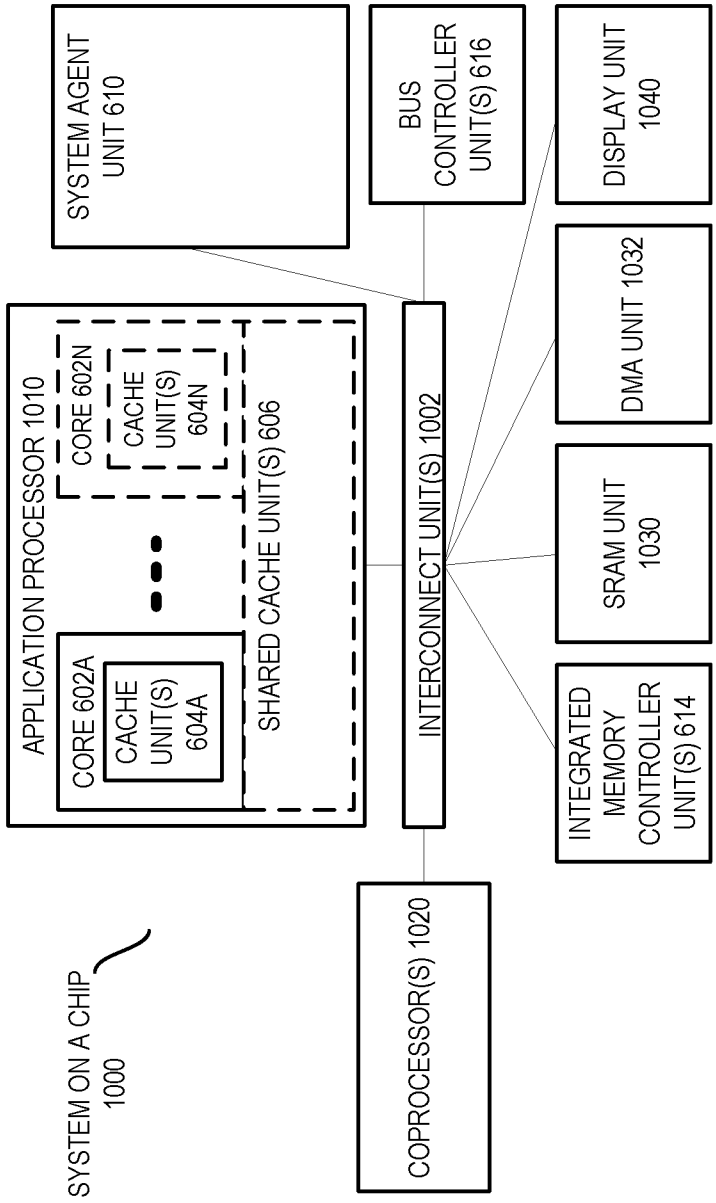

SYSTEM ON A CHIP
1000
APPLICATION PROCESSOR 1010
CORE 602A
CACHE UNIT(S) 604A
CORE 602N
CACHE UNIT(S) 604N
SHARED CACHE UNIT(S) 606
INTERCONNECT UNIT(S) 1002
SYSTEM AGENT UNIT 610
BUS CONTROLLER UNIT(S) 616
DISPLAY UNIT 1040
DMA UNIT 1032
SRAM UNIT 1030
INTEGRATED MEMORY CONTROLLER UNIT(S) 614
COPROCESSOR(S) 1020

FIG. 10

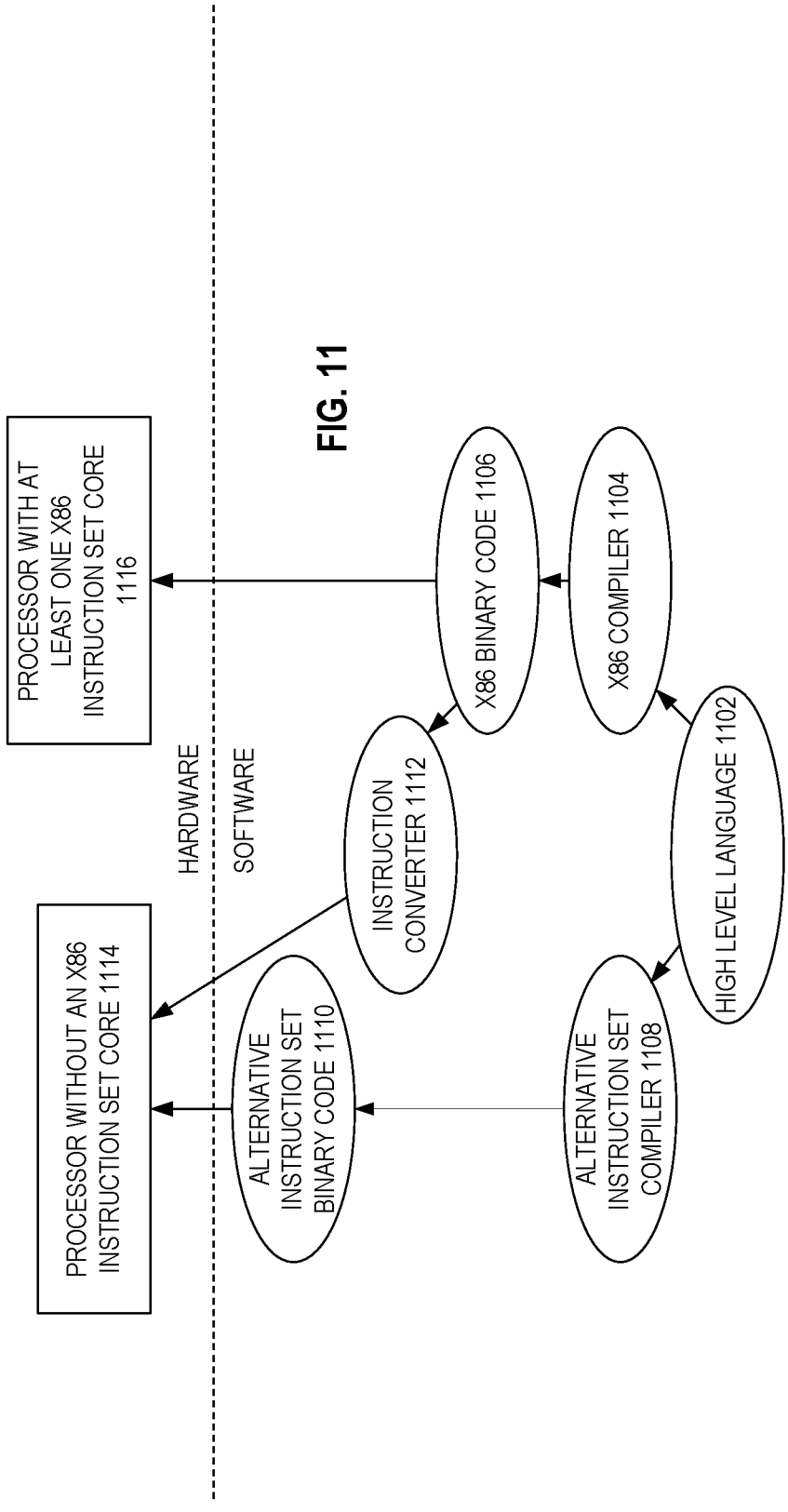

PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE 1116
X86 BINARY CODE 1106
X86 COMPILER 1104
INSTRUCTION CONVERTER 1112
HIGH LEVEL LANGUAGE 1102
HARDWARE
SOFTWARE
PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE 1114
ALTERNATIVE INSTRUCTION SET BINARY CODE 1110
ALTERNATIVE INSTRUCTION SET COMPILER 1108

FIG. 11

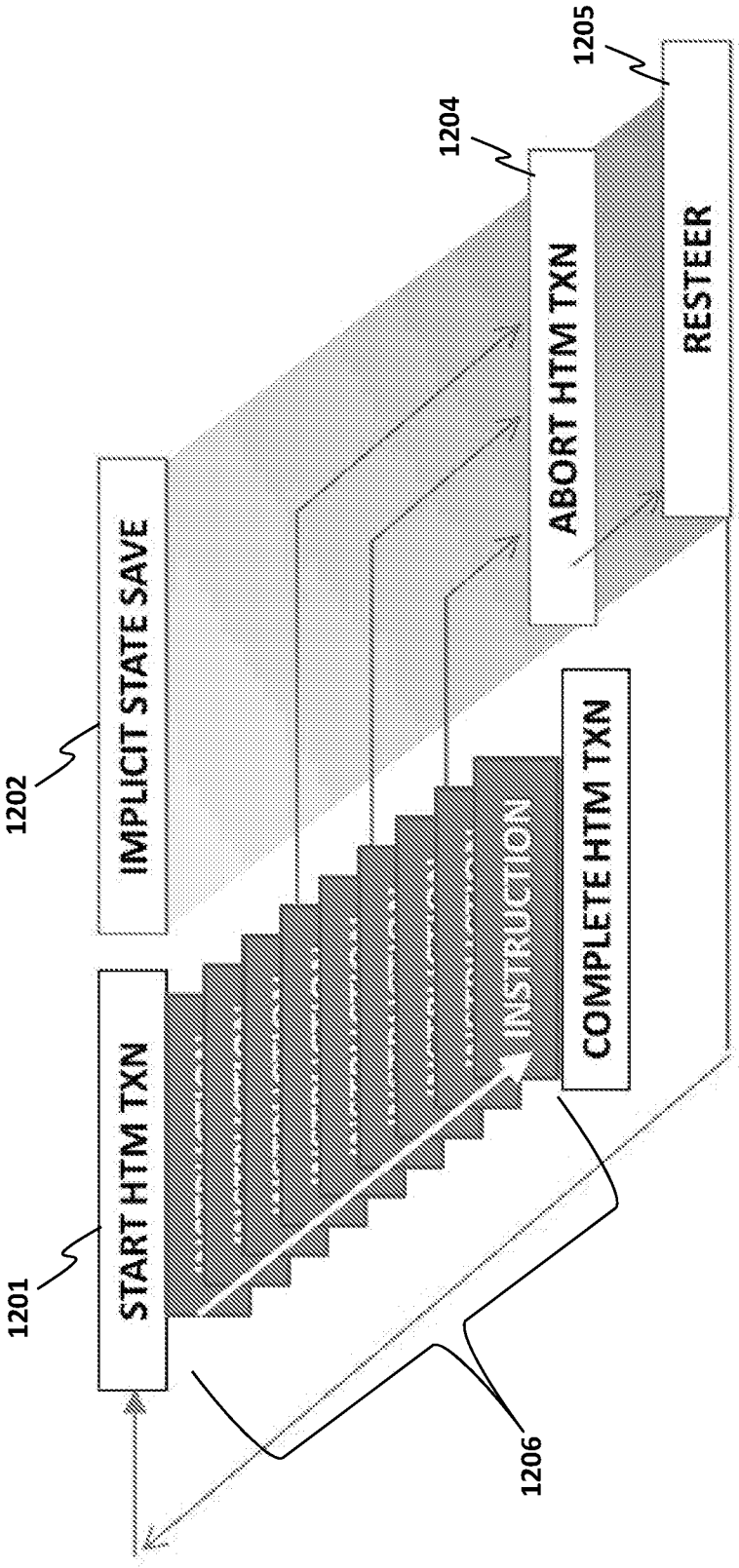

1205
1204
1202
1201
1206
START HTM TXN
IMPLICIT STATE SAVE
INSTRUCTION
COMPLETE HTM TXN
ABORT HTM TXN
RESTEER

Fig. 12

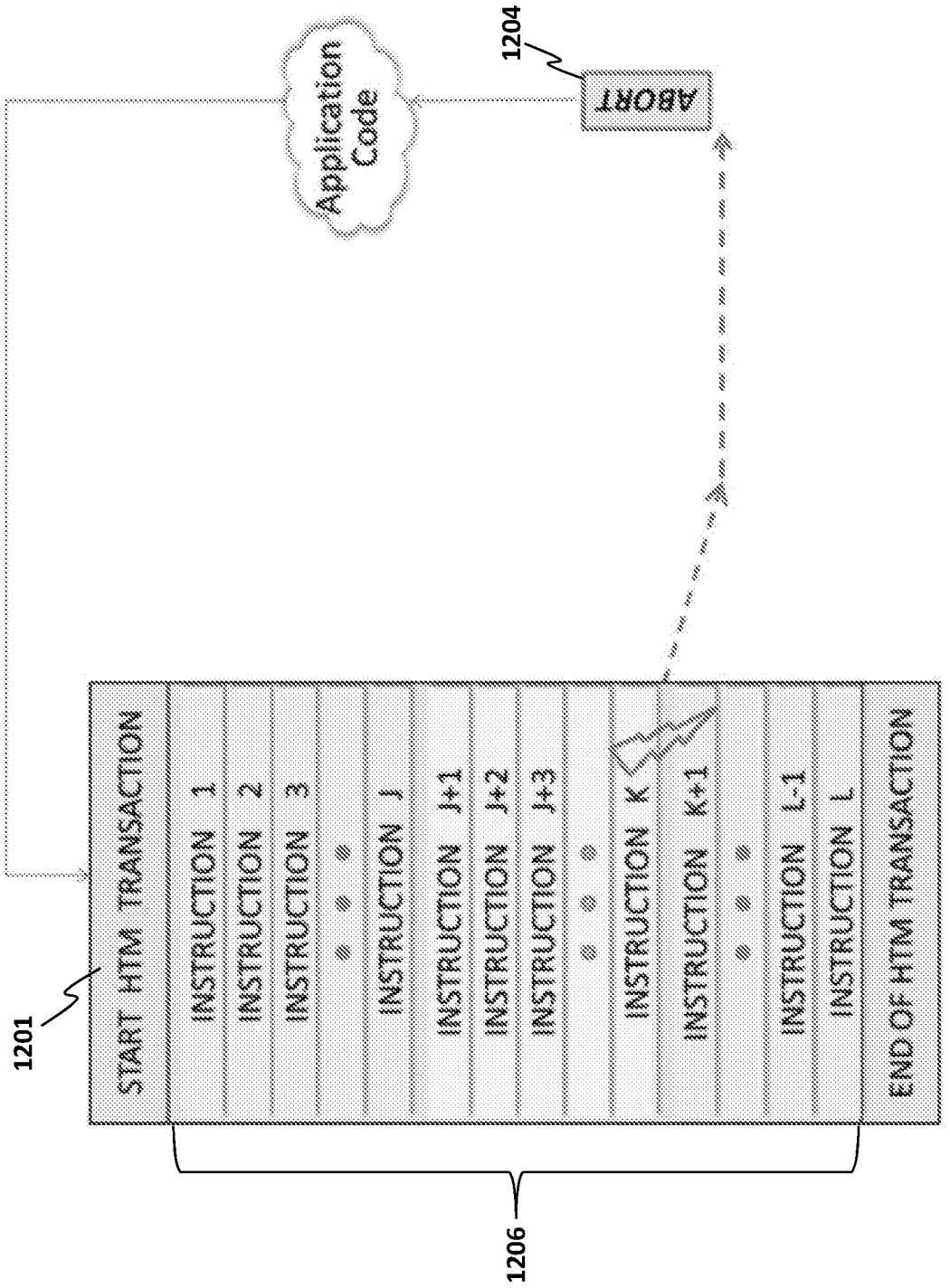

Application Code
1204
ABORT
START HTM TRANSACTION
INSTRUCTION 1
INSTRUCTION 2
INSTRUCTION 3
INSTRUCTION J
INSTRUCTION J+1
INSTRUCTION J+2
INSTRUCTION J+3
INSTRUCTION K
INSTRUCTION K+1
INSTRUCTION L-1
INSTRUCTION L
END OF HTM TRANSACTION
1201
1206

Fig. 13

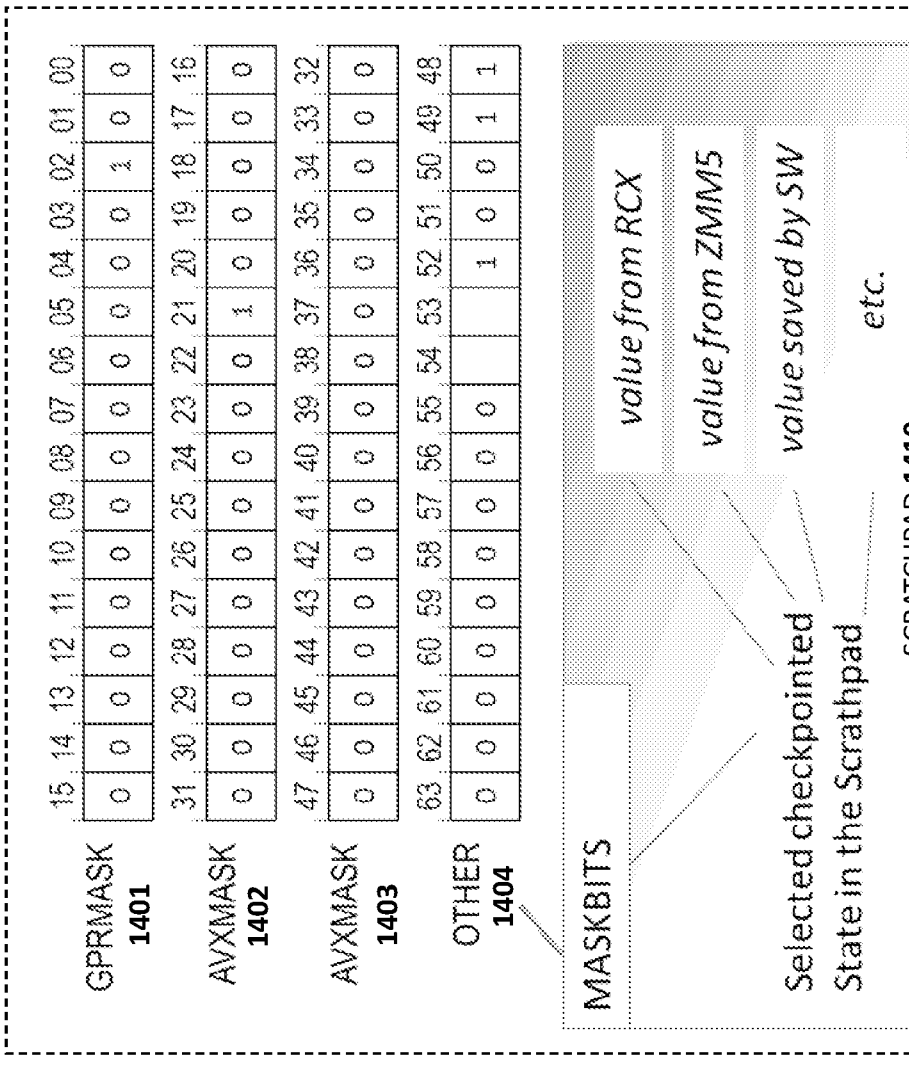

EXECUTION ENGINE 1415
GPRMASK 1401
AVXMASK 1402
AVXMASK 1403
OTHER 1404
MASKBITS
value from RCX
value from ZMM5
value saved by SW
etc.
Selected checkpointed State in the Scrathpad
SCRATCHPAD 1410
RFLAGS 1422
RIP 1421
k[]
XSAVE_1
1420
XSAVE_M
RESERVED_1
RESERVED_M
OTHER 1405

Fig. 14

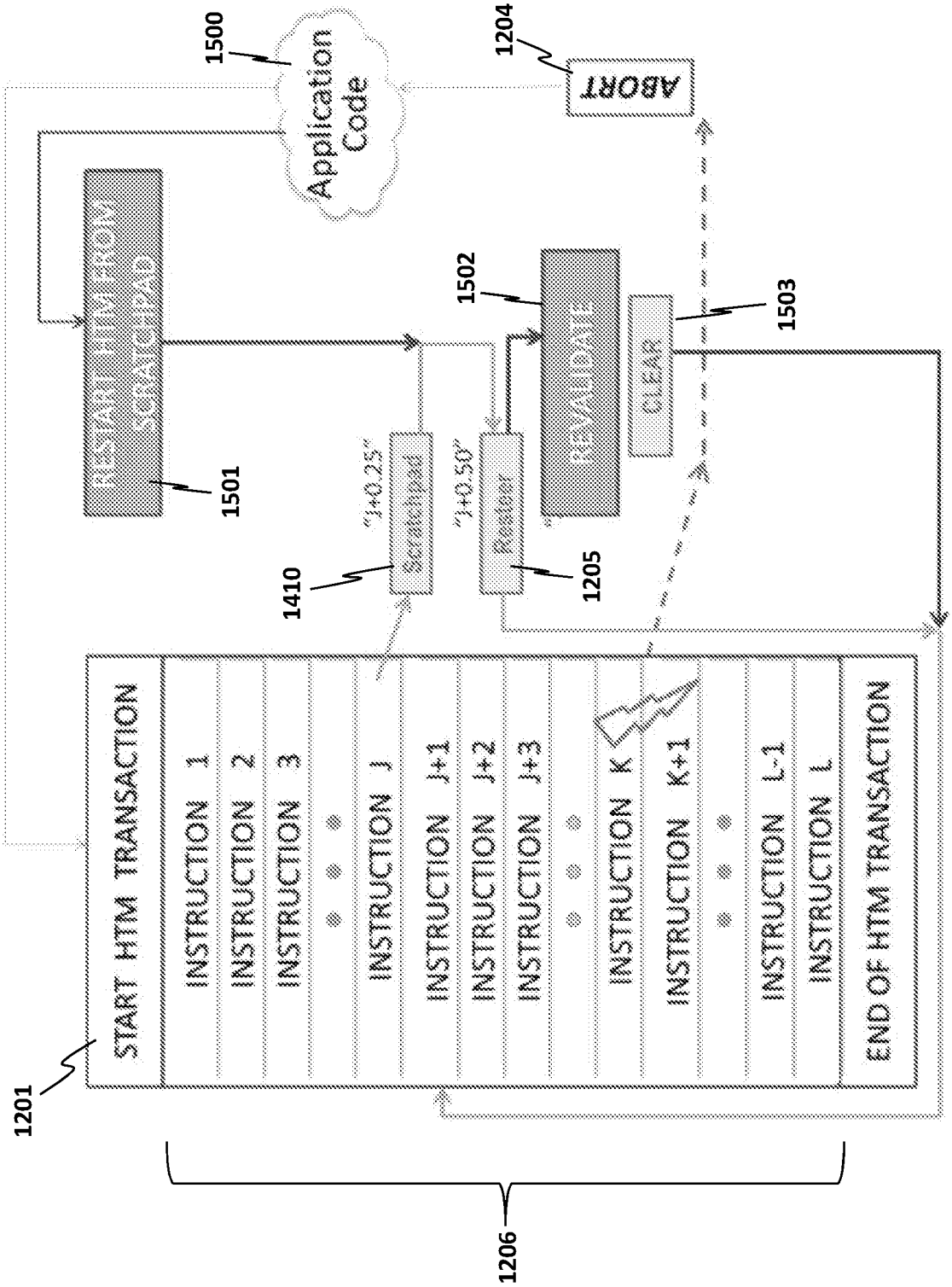

1500
Application Code
1204
ABORT
RESTART HTM FROM SCRATCHPAD
1501
1502
INVALIDATE
CLEAR
1503
"J+0.25"
Scratchpad
"J+0.50"
Register
1410
1205
START HTM TRANSACTION
INSTRUCTION 1
INSTRUCTION 2
INSTRUCTION 3
INSTRUCTION J
INSTRUCTION J+1
INSTRUCTION J+2
INSTRUCTION J+3
INSTRUCTION K
INSTRUCTION K+1
INSTRUCTION L-1
INSTRUCTION L
END OF HTM TRANSACTION
1201
1206

Fig. 15

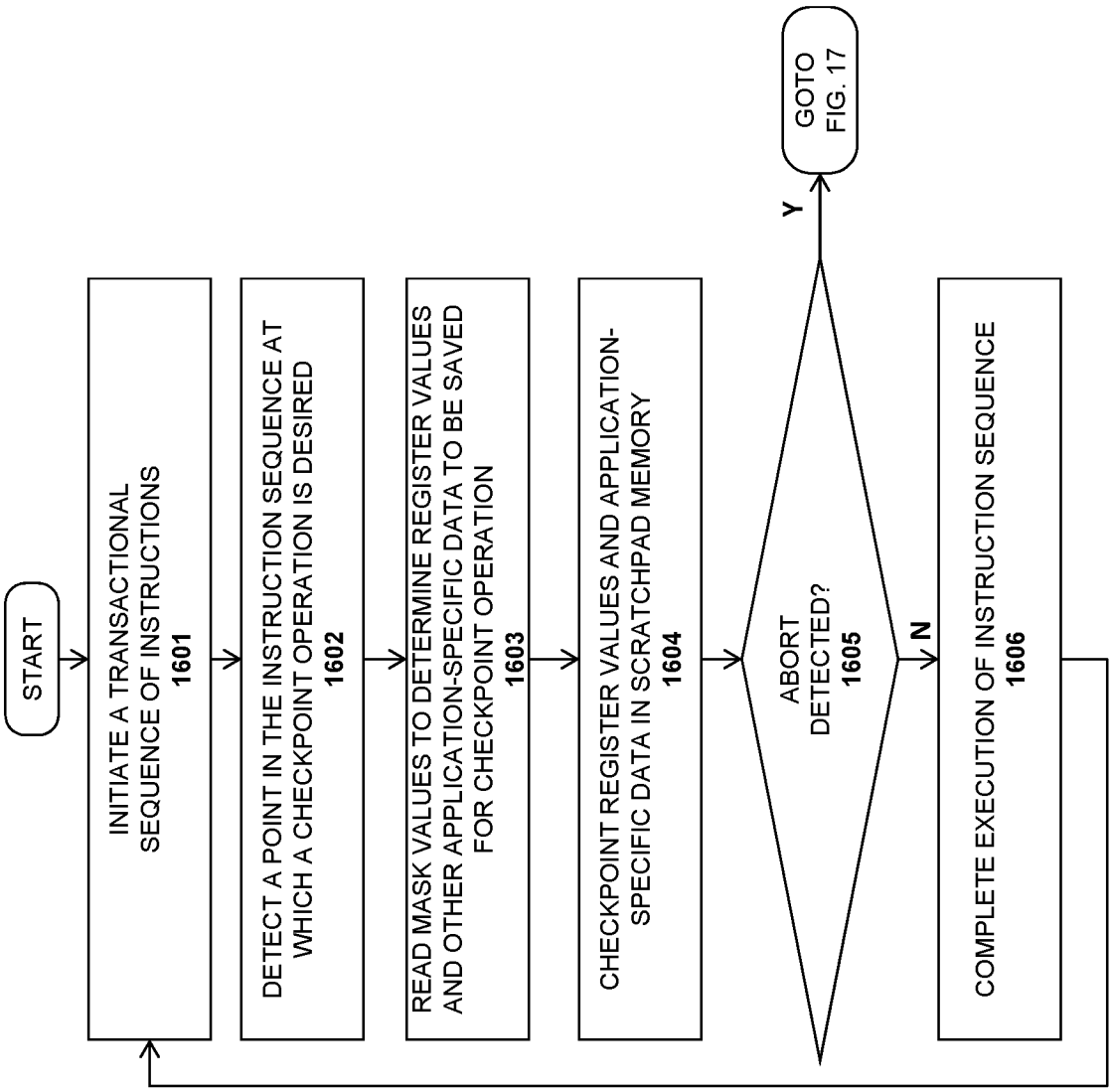

START
INITIATE A TRANSACTIONAL SEQUENCE OF INSTRUCTIONS
1601
DETECT A POINT IN THE INSTRUCTION SEQUENCE AT WHICH A CHECKPOINT OPERATION IS DESIRED
1602
READ MASK VALUES TO DETERMINE REGISTER VALUES AND OTHER APPLICATION-SPECIFIC DATA TO BE SAVED FOR CHECKPOINT OPERATION
1603
CHECKPOINT REGISTER VALUES AND APPLICATION-SPECIFIC DATA IN SCRATCHPAD MEMORY
1604
ABORT DETECTED?
1605
Y
GOTO FIG. 17
N
COMPLETE EXECUTION OF INSTRUCTION SEQUENCE
1606

Fig. 16

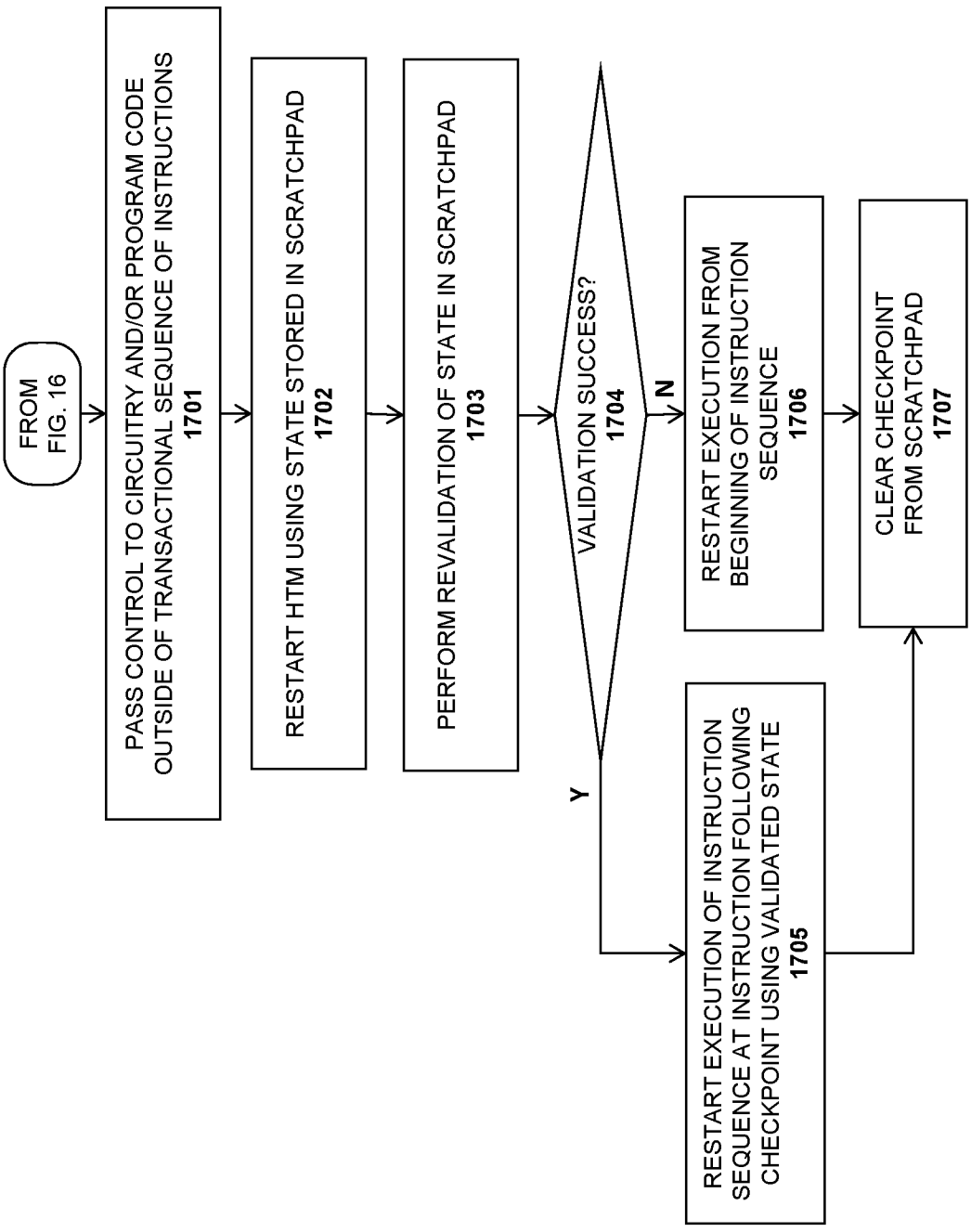

FROM FIG. 16
PASS CONTROL TO CIRCUITRY AND/OR PROGRAM CODE OUTSIDE OF TRANSACTIONAL SEQUENCE OF INSTRUCTIONS
1701
RESTART HTM USING STATE STORED IN SCRATCHPAD
1702
PERFORM REVALIDATION OF STATE IN SCRATCHPAD
1703
VALIDATION SUCCESS?
1704
Y
N
RESTART EXECUTION OF INSTRUCTION SEQUENCE AT INSTRUCTION FOLLOWING CHECKPOINT USING VALIDATED STATE
1705
RESTART EXECUTION FROM BEGINNING OF INSTRUCTION SEQUENCE
1706
CLEAR CHECKPOINT FROM SCRATCHPAD
1707

Fig. 17

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/035168

A. CLASSIFICATION OF SUBJECT MATTER

G06F 13/16(2006.01)i, G06F 9/46(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 13/16; G06F 9/46; G06F 9/38; G06F 9/312

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: hardware transaction memory (HTM), storing, architectural state, mask value, abort, and similar terms.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2013-0339704 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 19 December 2013 See paragraphs [0038], [0065], [0077], [0130], [0412], [0451]-[0452], [0502], and [0545]; claims 1-2; and figure 2B.	1-6,9-14,16-21
Y		7-8,15,22
X	CHRISTIAN JACOBI et al., 'Transactional Memory Architecture and Implementation for IBM System z,' In: 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 25-36, 01-05 December 2012 See page 27.	1-6,9-14,16-21
Y		7-8,15,22
Y	MAURICE HERLIHY et al., 'Transactional Memory: Architectural Support for Lock-Free Data Structures,' In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289-300, 16-19 May 1993 See page 290.	7-8,15,22
A	US 2012-0246658 A1 (RICHARD L. ARNDT et al.) 27 September 2012 See paragraphs [0042]-[0054] and [0061]-[0072]; and figures 1 and 3.	1-22

☒ 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	Date of mailing of the international search report
08 September 2017 (08.09.2017)	**08 September 2017 (08.09.2017)**

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. +82-42-481-8578

Authorized officer
Lee, Eun Kyu
Telephone No. +82-42-481-3580

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/035168

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2011-0296148 A1 (HAROLD W. CAIN, III et al.) 01 December 2011 See paragraphs [0056]-[0070]; claims 1-2 and 11; and figure 3.	1-22

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/035168

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2013-0339704 A1	19/12/2013	AU 2012-382778 A1	11/12/2014
		AU 2012-382778 B2	18/08/2016
		CA 2874179 A1	19/12/2013
		CN 104364778 A	18/02/2015
		CN 104364778 B	08/02/2017
		EP 2862092 A1	22/04/2015
		EP 2862092 A4	14/06/2017
		HK 1207700 A1	05/02/2016
		IL 236250 D0	29/01/2015
		JP 2015-526790 A	10/09/2015
		JP 6086406 B2	01/03/2017
		KR 10-1625323 B1	27/05/2016
		KR 10-2015-0016234 A	11/02/2015
		MX 2014015351 A	06/07/2015
		RU 2012148585 A	20/05/2014
		RU 2562424 C2	10/09/2015
		SG 11201407471 A	29/01/2015
		TW 201413581 A	01/04/2014
		TW I559225 B	21/11/2016
		US 2013-0339642 A1	19/12/2013
		US 2016-0266907 A1	15/09/2016
		US 9361115 B2	07/06/2016
		US 9367324 B2	14/06/2016
		WO 2013-186603 A1	19/12/2013
US 2012-0246658 A1	27/09/2012	US 2012-0084477 A1	05/04/2012
		US 8424015 B2	16/04/2013
		US 8544022 B2	24/09/2013
US 2011-0296148 A1	01/12/2011	US 9626187 B2	18/04/2017