



- (51) International Patent Classification:
G06F 9/30 (2006.01) *G06F 12/0802* (2016.01)
- (21) International Application Number:
PCT/US2017/039339
- (22) International Filing Date:
26 June 2017 (26.06.2017)
- (25) Filing Language: English
- (26) Publication Language: English
- (30) Priority Data:
15/201,405 02 July 2016 (02.07.2016) US
- (71) Applicant: **INTEL CORPORATION** [US/US]; 2200 Mission College Boulevard, Santa Clara, California 95054-1549 (US).
- (72) Inventors: **STRONG, Beeman C.**; 3841 NE Flanders, Portland, Oregon 97232 (US). **MERTEN, Matthew C.**; 12900 SW Whitmore Road, Hillsboro, Oregon 97123 (US). **BAUGH, Lee W.**; 832 NW 53rd Street, Unit A., Seattle, Washington 98107 (US).

(74) Agent: **CRANDALL, Sean C.**; Patent Capital Group, 2816 Lago Vista Lane, Rockwall, Texas 75032 (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, JO, 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, 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).

(54) Title: COOPERATIVE TRIGGERING

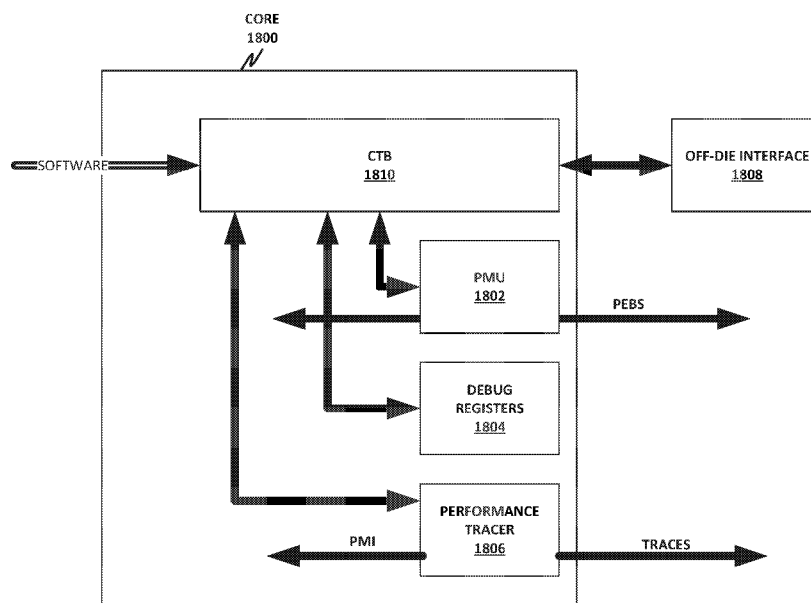


Fig. 18

(57) Abstract: There is disclosed in an example a processor, having: a front end including circuitry to decode instructions from an instruction stream; a data cache unit including circuitry to cache data for the processor; and a core triggering block (CTB) to provide integration between two or more different debug capabilities.

Declarations under Rule 4.17:

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

COOPERATIVE TRIGGERING

FIELD OF THE SPECIFICATION

[0001] This disclosure relates in general to the field of semiconductor devices, and more particularly, though not exclusively to, a system and method for cooperative triggering.

BACKGROUND

[0002] Multiprocessor systems are becoming more and more common. Applications of multiprocessor systems include dynamic domain partitioning all the way down to desktop computing. In order to take advantage of multiprocessor systems, code to be executed may be separated into multiple threads for execution by various processing entities. Each thread may be executed in parallel with one another. To increase the utility of a processing entity, out-of-order execute may be employed. Out-of-order execution may execute instructions as input to such instructions is made available. Thus, an instruction that appears later in a code sequence may be executed before an instruction appearing earlier in a code sequence.

BRIEF DESCRIPTION OF THE DRAWINGS

[0003] The present disclosure is best understood from the following detailed description when read with the accompanying figures. It is emphasized that, in accordance with the standard practice in the industry, various features are not necessarily drawn to scale, and are used for illustration purposes only. Where a scale is shown, explicitly or implicitly, it provides only one illustrative example. In other embodiments, the dimensions of the various features may be arbitrarily increased or reduced for clarity of discussion.

[0004] FIGURE 1A is a block diagram of an exemplary computer system formed with a processor that may include execution units to execute an instruction, in accordance with embodiments of the present disclosure.

[0005] FIGURE 1B illustrates a data processing system, in accordance with embodiments of the present disclosure.

[0006] FIGURE 1C illustrates other embodiments of a data processing system for performing text string comparison operations.

[0007] FIGURE 2 is a block diagram of the micro-architecture for a processor that may include logic circuits to perform instructions, in accordance with embodiments of the present disclosure.

[0008] FIGURE 3A illustrates various packed data type representations in multimedia registers, in accordance with embodiments of the present disclosure.

[0009] FIGURE 3B illustrates possible in-register data storage formats, in accordance with embodiments of the present disclosure.

[0010] FIGURE 3C illustrates various signed and unsigned packed data type representations in multimedia registers, in accordance with embodiments of the present disclosure.

[0011] FIGURE 3D illustrates an embodiment of an operation encoding format.

[0012] FIGURE 3E illustrates another possible operation encoding format having forty or more bits, in accordance with embodiments of the present disclosure.

[0013] FIGURE 3F illustrates yet another possible operation encoding format, in accordance with embodiments of the present disclosure.

[0014] FIGURE 4A is a block diagram illustrating an in-order pipeline and a register renaming stage, out-of-order issue/execution pipeline, in accordance with embodiments of the present disclosure.

[0015] FIGURE 4B is a block diagram illustrating an in-order architecture core and a register renaming logic, out-of-order issue/execution logic to be included in a processor, in accordance with embodiments of the present disclosure.

[0016] FIGURE 5A is a block diagram of a processor, in accordance with embodiments of the present disclosure.

[0017] FIGURE 5B is a block diagram of an example implementation of a core, in accordance with embodiments of the present disclosure;

[0018] FIGURE 6 is a block diagram of a system, in accordance with embodiments of the present disclosure.

[0019] FIGURE 7 is a block diagram of a second system, in accordance with embodiments of the present disclosure.

[0020] FIGURE 8 is a block diagram of a third system in accordance with embodiments of the present disclosure.

[0021] FIGURE 9 is a block diagram of a system-on-a-chip, in accordance with embodiments of the present disclosure.

[0022] FIGURE 10 illustrates a processor containing a central processing unit and a graphics processing unit which may perform at least one instruction, in accordance with embodiments of the present disclosure.

[0023] FIGURE 11 is a block diagram illustrating the development of IP cores, in accordance with embodiments of the present disclosure.

[0024] FIGURE 12 illustrates how an instruction of a first type may be emulated by a processor of a different type, in accordance with embodiments of the present disclosure.

[0025] FIGURE 13 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, in accordance with embodiments of the present disclosure.

[0026] FIGURE 14 is a block diagram of an instruction set architecture of a processor, in accordance with embodiments of the present disclosure.

[0027] FIGURE 15 is a more detailed block diagram of an instruction set architecture of a processor, in accordance with embodiments of the present disclosure.

[0028] FIGURE 16 is a block diagram of an execution pipeline for an instruction set architecture of a processor, in accordance with embodiments of the present disclosure.

[0029] FIGURE 17 is a block diagram of an electronic device for utilizing a processor, in accordance with embodiments of the present disclosure.

[0030] FIGURE 18 is a block diagram of processor visibility features, in accordance with embodiments of the present disclosure.

[0031] FIGURE 19 is a block diagram of performance monitoring interrupts, in accordance with embodiments of the present disclosure.

[0032] FIGURE 20 is a flow chart of performance monitoring interrupt handling, in accordance with embodiments of the present disclosure.

[0033] FIGURE 21 is a block diagram of PEBS, in accordance with embodiments of the present disclosure.

[0034] FIGURE 22 is a block diagram of a PEBS buffer, in accordance with embodiments of the present disclosure.

[0035] FIGURE 23 is a flow chart of a PEBS method, in accordance with embodiments of the present disclosure.

[0036] FIGURE 24 is a flow chart of a cooperative visibility feature, in accordance with embodiments of the present disclosure.

SUMMARY

[0037] There is disclosed in an example a processor, having: a front end including circuitry to decode instructions from an instruction stream; a data cache unit including circuitry to cache data for the processor; and a core triggering block (CTB) to provide integration between two or more different debug capabilities.

EMBODIMENTS OF THE DISCLOSURE

[0038] The following disclosure provides many different embodiments, or examples, for implementing different features of the present disclosure. Specific examples of components and arrangements are described below to simplify the present disclosure. These are, of course, merely examples and are not intended to be limiting. Further, the present disclosure may repeat reference numerals and/or letters in the various examples. This repetition is for the purpose of simplicity and clarity and does not in itself dictate a relationship between the various embodiments and/or configurations discussed. Different embodiments may have different advantages, and no particular advantage is necessarily required of any embodiment.

[0039] Computer hardware and software do not exist in a vacuum, or in an idealized theoretical environment. Rather, computer hardware and software exists within real-world complexities that can affect performance in sometimes unpredictable ways. This can lead to non-optimal program performance, or to processors that do not function at their theoretical maximum capability.

[0040] So-called “observability features” may be provided to help hardware and software designers optimize their designs. As used in this specification, an observability feature is any hardware and/or software feature that provides a designer with real-time or post-mortem visibility into relevant data such as program state, program flow, branching, variable states, memory usage, data structures, or other metrics that may be relevant to a

program's performance. Observability features can take many forms, and each feature may provide different information.

[0041] Observability features may be ranked loosely according to the interference they cause to a program. In some cases, interference with the program's flow is unimportant, and it is more important for the programmer to have maximum visibility into the program's state. In other cases, minimizing interference with the program's actual operations is critical, and a programmer or system administrator may prefer "leaner" data so as to minimize such interference. To provide some nonlimiting examples, several observability features are discussed below. Each feature is assigned a "weight" according to the interference that it causes to a program.

[0042] Debugging breakpoints and variable watch lists or inspectors may be considered very high interference observability features. These are most useful for a programmer trying to debug a program that is behaving in unexpected ways. The programmer may set breakpoints, and add certain variables and other data to a watchlist. In a modern context, the programmer can then run the program from within an integrated development environment (IDE) and interact with the program in real-time as it runs. When the program reaches a breakpoint, execution pauses, and control is passed to the programmer, who can then inspect the program state. The programmer can take time to inspect the value of certain variables, observe the current state of program flow, and otherwise analyze the program to determine what bugs may be causing the program to behave unexpectedly. Because these breakpoints includes human interaction, in a modern gigahertz-class microprocessor, program flow is interrupted for, at a minimum, several billions of cycles as the programmer inspects the program state and decides how to proceed.

[0043] Certain modern processors also provide built-in performance monitoring ("perfmon") capabilities that are less obtrusive than programmer breakpoints, but may still be considered "heavy weight" monitoring. These are not interactive, but may still be visible in the software space. For example, modern Intel® architectures provide a performance tracer, which enables a programmer or administrator to set a performance monitoring interrupt (PMI) on the occurrence of certain triggering conditions (e.g., every n cache misses or branch mispredicts), which is driven by an on-chip processor tracer (PT). PT may log certain limited information about software execution without any interrupts.

[0044] Upon triggering an event, a debug utility may also optionally send a PMI. The PT may also provide tracing, which occurs on every instance of a particular event of interest, such as every time a branch is taken or not taken, or every load or store on a data structure of interest. The programmer can define a PMI handler routine that then performs whatever logging or does any other work the programmer defines for the PMI handler. In common practice, the PMI handler is not interactive, so it will not interrupt the program flow for several billions of cycles like a breakpoint. But because control is passed out to a software handler routine, program flow may be interrupted on the order of hundreds of thousands or millions of cycles.

[0045] Triggering includes the architecture for providing multiple observability features, such as performance counters, breakpoints, and trace, and allowing them to trigger new actions. For instance, a performance counter overflow could trigger in-depth tracing to start, and hence the debugger can log detailed software behavior as performance begins to lag.

[0046] In the case of a PMI or breakpoint, software execution is interrupted. An address match in the debug breakpoint hardware may also, for example, trigger a trace to log the Instruction Pointer (IP) and Data Linear Address (DLA) of the access, so that all accesses to a variable or data structure can be logged without requiring software to stop for breakpoint events.

[0047] Precise Event-based Sampling (PEBS) may be considered a “medium-weight” observability feature. In modern Intel® architecture processors, PEBS is provided by an on-chip performance monitoring unit (PMU). PEBS is a special counting mode in which counters can be configured to overflow, interrupt the processor, and capture the machine state at that point. The PMU has access to a dedicated PEBS buffer, which may have limited memory space and flexibility relative to a PMI handler. Values in the PEBS buffer may be analyzed offline, so the analysis does not interfere with program flow. PEBS also operates in microcode rather than in user-space software, meaning that it may interrupt software execution for only several hundreds or a few thousands of cycles as it writes out to the PEBS buffer.

[0048] However, as CPUs increase in complexity, so does the process of debugging and tuning the software that runs on them. Modern CPUs typically support a combination of debug and performance analysis features, including tracing, sampling, and address-based

breakpoints, but in some embodiments these features operate in separate silos. For example, breakpoint address matching hardware may be used to trigger a breakpoint, and performance counters can only be used to trigger interrupts or PEBS.

[0049] It is advantageous to provide a more efficient approach to leverage the existing monitoring capabilities, allowing them to work together to support a broader set of usage models. This empowers debuggers and developers to find and fix problems in their software more quickly, improving time-to-market, software health, and customer satisfaction. These may be provided by the core triggering block (CTB) described herein.

[0050] In an example, very high rate sampling may be supported by leveraging performance counter overflows to trigger tracing hardware to log data such as the time and IP. Without requiring an expensive interrupt or microarchitecture event (such as PEBS), the slowdown may be minimal, but the granularity of performance data very high. This results in a substantial increase in available debug and profiling usage models, with limited additional hardware. This may also provide an extensible framework for expanding the suite of capabilities that can interact via the CTB.

[0051] New features and hardware may be added to serve new usage models targeted here. For instance, a filtered data address trace capability may be added to track accesses to variables of interest. Advantageously, certain embodiments of the present specification leverage existing hardware to realize the disclosed features. For example, breakpoint address match hardware may be used to identify and accesses a variable of interest and trigger existing control flow trace mechanisms to log the time and IP.

[0052] Debug breakpoints, performance counters, and a tracing infrastructure are observability features found in some modern CPUs. These features may be used in conjunction with each other, but in some embodiments, software is required to implement interactions. For example, a code breakpoint may be programmed to stop execution so that tracing can be initiated by a handler. However, the nature of some debug or profiling scenarios makes running such “glue” software undesirable, as the interruption may change software behavior and make it difficult to reproduce (and debug) the issue, particularly if the bug is related to timing.

[0053] Triggering provides an architecture and hardware infrastructure by which many tasks commonly performed in software can instead be done in hardware, with no

interruption of native software execution. The hardware may include simple logic to connect trigger sources or inputs to the software-configured triggering actions.

[0054] A system and method for cooperative triggering will now be described with more particular reference to the attached FIGURES. It should be noted that throughout the FIGURES, certain reference numerals may be repeated to indicate that a particular device or block is wholly or substantially consistent across the FIGURES. This is not, however, intended to imply any particular relationship between the various embodiments disclosed.

[0055] Although the below examples describe instruction handling and distribution in the context of execution units and logic circuits, other embodiments of the present disclosure may be accomplished by way of a data or instructions stored on a machine-readable, tangible medium, which when performed by a machine cause the machine to perform functions consistent with at least one embodiment of the disclosure. In one embodiment, functions associated with embodiments of the present disclosure are embodied in machine-executable instructions. The instructions may be used to cause a general-purpose or special-purpose processor that may be programmed with the instructions to perform the steps of the present disclosure. Embodiments of the present disclosure may be provided as a computer program product or software which may include a machine or computer-readable medium having stored thereon instructions which may be used to program a computer (or other electronic devices) to perform one or more operations according to embodiments of the present disclosure. Furthermore, steps of embodiments of the present disclosure might be performed by specific hardware components that contain fixed-function logic for performing the steps, or by any combination of programmed computer components and fixed-function hardware components.

[0056] Instructions used to program logic to perform embodiments of the present disclosure may be stored within a memory in the system, such as DRAM, cache, flash memory, or other storage. Furthermore, the instructions may be distributed via a network or by way of other computer-readable media. Thus a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable

Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the computer-readable medium may include any type of tangible machine-readable medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer).

[0057] A design may go through various stages, from creation to simulation to fabrication. Data representing a design may represent the design in a number of manners. First, as may be useful in simulations, the hardware may be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. Furthermore, designs, at some stage, may reach a level of data representing the physical placement of various devices in the hardware model. In cases wherein some semiconductor fabrication techniques are used, the data representing the hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data may be stored in any form of a machine-readable medium. A memory or a magnetic or optical storage such as a disc may be the machine-readable medium to store information transmitted via optical or electrical wave modulated or otherwise generated to transmit such information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or retransmission of the electrical signal is performed, a new copy may be made. Thus, a communication provider or a network provider may store on a tangible, machine-readable medium, at least temporarily, an article, such as information encoded into a carrier wave, embodying techniques of embodiments of the present disclosure.

[0058] In modern processors, a number of different execution units may be used to process and execute a variety of code and instructions. Some instructions may be quicker to complete while others may take a number of clock cycles to complete. The faster the throughput of instructions, the better the overall performance of the processor. Thus it would be advantageous to have as many instructions execute as fast as possible. However, there may be certain instructions that have greater complexity and require more in terms of

execution time and processor resources, such as floating point instructions, load/store operations, data moves, etc.

[0059] As more computer systems are used in internet, text, and multimedia applications, additional processor support has been introduced over time. In one embodiment, an instruction set may be associated with one or more computer architectures, including data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O).

[0060] In one embodiment, the instruction set architecture (ISA) may be implemented by one or more micro-architectures, which may include processor logic and circuits used to implement one or more instruction sets. Accordingly, processors with different micro-architectures may share at least a portion of 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. Similarly, processors designed by other processor development companies, such as ARM Holdings, Ltd., MIPS, or their licensees or adopters, may share at least a portion a common instruction set, but may include different processor designs. For example, the same register architecture of the ISA may be implemented in different ways in different micro-architectures using new or 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. In one embodiment, registers may include one or more registers, register architectures, register files, or other register sets that may or may not be addressable by a software programmer.

[0061] An instruction may include one or more instruction formats. In one embodiment, an instruction format may indicate various fields (number of bits, location of bits, etc.) to specify, among other things, the operation to be performed and the operands on which that operation will be performed. In a further embodiment, some instruction formats may be further defined by instruction templates (or sub-formats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format's fields and/or defined to have a given field interpreted differently.

In one embodiment, an instruction may be expressed using an instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and specifies or indicates the operation and the operands upon which the operation will operate.

[0062] Scientific, financial, auto-vectorized general purpose, RMS (recognition, mining, and synthesis), and visual and multimedia applications (e.g., 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation) may require the same operation to be performed on a large number of data items. In one embodiment, Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data elements. SIMD technology may be used in processors that may logically divide the bits in a register into a number of fixed-sized or variable-sized data elements, each of which represents a separate value. For example, in one embodiment, the bits in a 64-bit register may be organized as a source operand containing four separate 16-bit data elements, each of which represents a separate 16-bit value. This type of data may be referred to as 'packed' data type or 'vector' data type, and operands of this data type may be referred to as packed data operands or vector operands. In one embodiment, a packed data item or vector may be a sequence of packed data elements stored within a single register, and a packed data operand or a vector operand may be a source or destination operand of a SIMD instruction (or 'packed data instruction' or a 'vector instruction'). In one embodiment, a SIMD instruction specifies a single vector operation to be performed on two source vector operands to generate a destination vector operand (also referred to as a result vector operand) of the same or different size, with the same or different number of data elements, and in the same or different data element order.

[0063] SIMD technology, such as that employed by the Intel® Core™ processors having an instruction set including x86, MMX™, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, and SSE4.2 instructions, ARM processors, such as the ARM Cortex® family of processors having an instruction set including the Vector Floating Point (VFP) and/or NEON instructions, and MIPS processors, such as the Loongson family of processors developed by the Institute of Computing Technology (ICT) of the Chinese Academy of Sciences, has enabled a significant improvement in application performance (Core™ and MMX™ are registered trademarks or trademarks of Intel Corporation of Santa Clara, Calif.).

[0064] In one embodiment, destination and source registers/data may be generic terms to represent the source and destination of the corresponding data or operation. In some embodiments, they may be implemented by registers, memory, or other storage areas having other names or functions than those depicted. For example, in one embodiment, “DEST1” may be a temporary storage register or other storage area, whereas “SRC1” and “SRC2” may be a first and second source storage register or other storage area, and so forth. In other embodiments, two or more of the SRC and DEST storage areas may correspond to different data storage elements within the same storage area (e.g., a SIMD register). In one embodiment, one of the source registers may also act as a destination register by, for example, writing back the result of an operation performed on the first and second source data to one of the two source registers serving as a destination registers.

[0065] FIGURE 1A is a block diagram of an exemplary computer system formed with a processor that may include execution units to execute an instruction, in accordance with embodiments of the present disclosure. System 100 may include a component, such as a processor 102 to employ execution units including circuits with logic to perform algorithms for process data, in accordance with the present disclosure, such as in the embodiment described herein. System 100 may be representative of processing systems based on the PENTIUM® III, PENTIUM® 4, Xeon™, Itanium®, XScale™ and/or StrongARM™ microprocessors available from Intel Corporation of Santa Clara, California, although other systems (including PCs having other microprocessors, engineering workstations, set-top boxes and the like) may also be used. In one embodiment, sample system 100 may execute a version of the WINDOWS™ operating system available from Microsoft Corporation of Redmond, Washington, although other operating systems (UNIX and Linux for example), embedded software, and/or graphical user interfaces, may also be used. Thus, embodiments of the present disclosure are not limited to any specific combination of hardware circuitry, programmable circuitry, and software.

[0066] Embodiments are not limited to computer systems. Embodiments of the present disclosure may be used in other devices such as handheld devices and embedded applications. Some examples of handheld devices include cellular phones, Internet Protocol devices, digital cameras, personal digital assistants (PDAs), and handheld PCs. Embedded applications may include a micro controller, a digital signal processor (DSP), system on a chip,

network computers (NetPC), set-top boxes, network hubs, wide area network (WAN) switches, or any other system that may perform one or more instructions in accordance with at least one embodiment.

[0067] Computer system 100 may include a processor 102 that may include one or more execution units 108 to perform an algorithm to perform at least one instruction in accordance with one embodiment of the present disclosure. One embodiment may be described in the context of a single processor desktop or server system, but other embodiments may be included in a multiprocessor system. System 100 may be an example of a 'hub' system architecture. System 100 may include a processor 102 for processing data signals. Processor 102 may include a complex instruction set computer (CISC) microprocessor, a reduced instruction set computing (RISC) microprocessor, a very long instruction word (VLIW) microprocessor, a processor implementing a combination of instruction sets, or any other processor device, such as a digital signal processor, for example. In one embodiment, processor 102 may be coupled to a processor bus 110 that may transmit data signals between processor 102 and other components in system 100. The elements of system 100 may perform conventional functions that are well known to those familiar with the art.

[0068] In one embodiment, processor 102 may include a Level 1 (L1) internal cache memory 104. Depending on the architecture, the processor 102 may have a single internal cache or multiple levels of internal cache. In another embodiment, the cache memory may reside external to processor 102. Other embodiments may also include a combination of both internal and external caches depending on the particular implementation and needs. Register file 106 may store different types of data in various registers including integer registers, floating point registers, status registers, and instruction pointer register.

[0069] Execution unit 108, including circuits with logic to perform integer and floating point operations, also resides in processor 102. Processor 102 may also include a microcode (ucode) ROM that stores microcode for certain macroinstructions. In one embodiment, execution unit 108 may include circuits with logic to handle a packed instruction set 109. By including the packed instruction set 109 in the instruction set of a general-purpose processor 102, along with associated circuitry to execute the instructions, the operations used by many multimedia applications may be performed using packed data in

a general-purpose processor 102. Thus, many multimedia applications may be accelerated and executed more efficiently by using the full width of a processor's data bus for performing operations on packed data. This may eliminate the need to transfer smaller units of data across the processor's data bus to perform one or more operations one data element at a time.

[0070] Embodiments of an execution unit 108 may also be used in micro controllers, embedded processors, graphics devices, DSPs, and other types of logic circuits. System 100 may include a memory 120. Memory 120 may be implemented as a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or other memory device. Memory 120 may store instructions and/or data represented by data signals that may be executed by processor 102.

[0071] A system logic chip 116 may be coupled to processor bus 110 and memory 120. System logic chip 116 may include a memory controller hub (MCH). Processor 102 may communicate with MCH 116 via a processor bus 110. MCH 116 may provide a high bandwidth memory path 118 to memory 120 for instruction and data storage and for storage of graphics commands, data and textures. MCH 116 may direct data signals between processor 102, memory 120, and other components in system 100 and to bridge the data signals between processor bus 110, memory 120, and system I/O 122. In some embodiments, the system logic chip 116 may provide a graphics port for coupling to a graphics controller 112. MCH 116 may be coupled to memory 120 through a memory interface 118. Graphics card 112 may be coupled to MCH 116 through an Accelerated Graphics Port (AGP) interconnect 114.

[0072] System 100 may use a proprietary hub interface bus 122 to couple MCH 116 to I/O controller hub (ICH) 130. In one embodiment, ICH 130 may provide direct connections to some I/O devices via a local I/O bus. The local I/O bus may include a high-speed I/O bus for connecting peripherals to memory 120, chipset, and processor 102. Examples may include the audio controller, firmware hub (flash BIOS) 128, wireless transceiver 126, data storage 124, legacy I/O controller containing user input and keyboard interfaces, a serial expansion port such as Universal Serial Bus (USB), and a network controller 134. Data storage device 124 may comprise a hard disk drive, a floppy disk drive, a CD-ROM device, a flash memory device, or other mass storage device.

[0073] For another embodiment of a system, an instruction in accordance with one embodiment may be used with a system on a chip. One embodiment of a system on a chip comprises of a processor and a memory. The memory for one such system may include a flash memory. The flash memory may be located on the same die as the processor and other system components. Additionally, other logic blocks such as a memory controller or graphics controller may also be located on a system on a chip.

[0074] FIGURE 1B illustrates a data processing system 140 which implements the principles of embodiments of the present disclosure. It will be readily appreciated by one of skill in the art that the embodiments described herein may operate with alternative processing systems without departure from the scope of embodiments of the disclosure.

[0075] Computer system 140 comprises a processing core 159 for performing at least one instruction in accordance with one embodiment. In one embodiment, processing core 159 represents a processing unit of any type of architecture, including but not limited to a CISC, a RISC or a VLIW type architecture. Processing core 159 may also be suitable for manufacture in one or more process technologies and by being represented on a machine-readable media in sufficient detail, may be suitable to facilitate said manufacture.

[0076] Processing core 159 comprises an execution unit 142, a set of register files 145, and a decoder 144. Processing core 159 may also include additional circuitry (not shown) which may be unnecessary to the understanding of embodiments of the present disclosure. Execution unit 142 may execute instructions received by processing core 159. In addition to performing typical processor instructions, execution unit 142 may perform instructions in packed instruction set 143 for performing operations on packed data formats. Packed instruction set 143 may include instructions for performing embodiments of the disclosure and other packed instructions. Execution unit 142 may be coupled to register file 145 by an internal bus. Register file 145 may represent a storage area on processing core 159 for storing information, including data. As previously mentioned, it is understood that the storage area may store the packed data might not be critical. Execution unit 142 may be coupled to decoder 144. Decoder 144 may decode instructions received by processing core 159 into control signals and/or microcode entry points. In response to these control signals and/or microcode entry points, execution unit 142 performs the appropriate operations. In one embodiment, the decoder may interpret the opcode of the instruction, which will

indicate what operation should be performed on the corresponding data indicated within the instruction.

[0077] Processing core 159 may be coupled with bus 141 for communicating with various other system devices, which may include but are not limited to, for example, synchronous dynamic random access memory (SDRAM) control 146, static random access memory (SRAM) control 147, burst flash memory interface 148, personal computer memory card international association (PCMCIA)/compact flash (CF) card control 149, liquid crystal display (LCD) control 150, direct memory access (DMA) controller 151, and alternative bus master interface 152. In one embodiment, data processing system 140 may also comprise an I/O bridge 154 for communicating with various I/O devices via an I/O bus 153. Such I/O devices may include but are not limited to, for example, universal asynchronous receiver/transmitter (UART) 155, universal serial bus (USB) 156, Bluetooth wireless UART 157 and I/O expansion interface 158.

[0078] One embodiment of data processing system 140 provides for mobile, network and/or wireless communications and a processing core 159 that may perform SIMD operations including a text string comparison operation. Processing core 159 may be programmed with various audio, video, imaging and communications algorithms including discrete transformations such as a Walsh-Hadamard transform, a fast Fourier transform (FFT), a discrete cosine transform (DCT), and their respective inverse transforms; compression/decompression techniques such as color space transformation, video encode motion estimation or video decode motion compensation; and modulation/demodulation (MODEM) functions such as pulse coded modulation (PCM).

[0079] FIGURE 1C illustrates other embodiments of a data processing system that performs SIMD text string comparison operations. In one embodiment, data processing system 160 may include a main processor 166, a SIMD coprocessor 161, a cache memory 167, and an input/output system 168. Input/output system 168 may optionally be coupled to a wireless interface 169. SIMD coprocessor 161 may perform operations including instructions in accordance with one embodiment. In one embodiment, processing core 170 may be suitable for manufacture in one or more process technologies and by being represented on a machine-readable media in sufficient detail, may be suitable to facilitate the manufacture of all or part of data processing system 160 including processing core 170.

[0080] In one embodiment, SIMD coprocessor 161 comprises an execution unit 162 and a set of register files 164. One embodiment of main processor 165 comprises a decoder 165 to recognize instructions of instruction set 163 including instructions in accordance with one embodiment for execution by execution unit 162. In other embodiments, SIMD coprocessor 161 also comprises at least part of decoder 165 to decode instructions of instruction set 163. Processing core 170 may also include additional circuitry (not shown) which may be unnecessary to the understanding of embodiments of the present disclosure.

[0081] In operation, main processor 166 executes a stream of data processing instructions that control data processing operations of a general type including interactions with cache memory 167, and input/output system 168. Embedded within the stream of data processing instructions may be SIMD coprocessor instructions. Decoder 165 of main processor 166 recognizes these SIMD coprocessor instructions as being of a type that should be executed by an attached SIMD coprocessor 161. Accordingly, main processor 166 issues these SIMD coprocessor instructions (or control signals representing SIMD coprocessor instructions) on the coprocessor bus 166. From coprocessor bus 166, these instructions may be received by any attached SIMD coprocessors. In this case, SIMD coprocessor 161 may accept and execute any received SIMD coprocessor instructions intended for it.

[0082] Data may be received via wireless interface 169 for processing by the SIMD coprocessor instructions. For one example, voice communication may be received in the form of a digital signal, which may be processed by the SIMD coprocessor instructions to regenerate digital audio samples representative of the voice communications. For another example, compressed audio and/or video may be received in the form of a digital bit stream, which may be processed by the SIMD coprocessor instructions to regenerate digital audio samples and/or motion video frames. In one embodiment of processing core 170, main processor 166, and a SIMD coprocessor 161 may be integrated into a single processing core 170 comprising an execution unit 162, a set of register files 164, and a decoder 165 to recognize instructions of instruction set 163 including instructions in accordance with one embodiment.

[0083] FIGURE 2 is a block diagram of the micro-architecture for a processor 200 that may include logic circuits to perform instructions, in accordance with embodiments of the present disclosure. In some embodiments, an instruction in accordance with one

embodiment may be implemented to operate on data elements having sizes of byte, word, doubleword, quadword, etc., as well as datatypes, such as single and double precision integer and floating point datatypes. In one embodiment, in-order front end 201 may implement a part of processor 200 that may fetch instructions to be executed and prepares the instructions to be used later in the processor pipeline. Front end 201 may include several units. In one embodiment, instruction prefetcher 226 fetches instructions from memory and feeds the instructions to an instruction decoder 228 which in turn decodes or interprets the instructions. For example, in one embodiment, the decoder decodes a received instruction into one or more operations called “micro-instructions” or “micro-operations” (also called micro op or uops) that the machine may execute. In other embodiments, the decoder parses the instruction into an opcode and corresponding data and control fields that may be used by the micro-architecture to perform operations in accordance with one embodiment. In one embodiment, trace cache 230 may assemble decoded uops into program ordered sequences or traces in uop queue 234 for execution. When trace cache 230 encounters a complex instruction, microcode ROM 232 provides the uops needed to complete the operation.

[0084] Some instructions may be converted into a single micro-op, whereas others need several micro-ops to complete the full operation. In one embodiment, if more than four micro-ops are needed to complete an instruction, decoder 228 may access microcode ROM 232 to perform the instruction. In one embodiment, an instruction may be decoded into a small number of micro ops for processing at instruction decoder 228. In another embodiment, an instruction may be stored within microcode ROM 232 should a number of micro-ops be needed to accomplish the operation. Trace cache 230 refers to an entry point programmable logic array (PLA) to determine a correct micro-instruction pointer for reading the micro-code sequences to complete one or more instructions in accordance with one embodiment from micro-code ROM 232. After microcode ROM 232 finishes sequencing micro-ops for an instruction, front end 201 of the machine may resume fetching micro-ops from trace cache 230.

[0085] Out-of-order execution engine 203 may prepare instructions for execution. The out-of-order execution logic has a number of buffers to smooth out and re-order the flow of instructions to optimize performance as they go down the pipeline and get scheduled for execution. The allocator logic allocates the machine buffers and resources that each uop

needs in order to execute. The register renaming logic renames logic registers onto entries in a register file. The allocator also allocates an entry for each uop in one of the two uop queues, one for memory operations and one for non-memory operations, in front of the instruction schedulers: memory scheduler, fast scheduler 202, slow/general floating point scheduler 204, and simple floating point scheduler 206. Uop schedulers 202, 204, 206, determine when a uop is ready to execute based on the readiness of their dependent input register operand sources and the availability of the execution resources the uops need to complete their operation. Fast scheduler 202 of one embodiment may schedule on each half of the main clock cycle while the other schedulers may only schedule once per main processor clock cycle. The schedulers arbitrate for the dispatch ports to schedule uops for execution.

[0086] Register files 208, 210 may be arranged between schedulers 202, 204, 206, and execution units 212, 214, 216, 218, 220, 222, 224 in execution block 211. Each of register files 208, 210 perform integer and floating point operations, respectively. Each register file 208, 210, may include a bypass network that may bypass or forward just completed results that have not yet been written into the register file to new dependent uops. Integer register file 208 and floating point register file 210 may communicate data with the other. In one embodiment, integer register file 208 may be split into two separate register files, one register file for low-order thirty-two bits of data and a second register file for high order thirty-two bits of data. Floating point register file 210 may include 128-bit wide entries because floating point instructions typically have operands from 64 to 128 bits in width.

[0087] Execution block 211 may contain execution units 212, 214, 216, 218, 220, 222, 224. Execution units 212, 214, 216, 218, 220, 222, 224 may execute the instructions. Execution block 211 may include register files 208, 210 that store the integer and floating point data operand values that the micro-instructions need to execute. In one embodiment, processor 200 may comprise a number of execution units: address generation unit (AGU) 212, AGU 214, fast ALU 216, fast ALU 218, slow ALU 220, floating point ALU 222, floating point move unit 224. In another embodiment, floating point execution blocks 222, 224, may execute floating point, MMX, SIMD, and SSE, or other operations. In yet another embodiment, floating point ALU 222 may include a 64-bit by 64-bit floating point divider to execute divide, square root, and remainder micro-ops. In various embodiments, instructions

involving a floating point value may be handled with the floating point hardware. In one embodiment, ALU operations may be passed to high-speed ALU execution units 216, 218. High-speed ALUs 216, 218 may execute fast operations with an effective latency of half a clock cycle. In one embodiment, most complex integer operations go to slow ALU 220 as slow ALU 220 may include integer execution hardware for long-latency type of operations, such as a multiplier, shifts, flag logic, and branch processing. Memory load/store operations may be executed by AGUs 212, 214. In one embodiment, integer ALUs 216, 218, 220 may perform integer operations on 64-bit data operands. In other embodiments, ALUs 216, 218, 220 may be implemented to support a variety of data bit sizes including sixteen, thirty-two, 128, 256, etc. Similarly, floating point units 222, 224 may be implemented to support a range of operands having bits of various widths. In one embodiment, floating point units 222, 224, may operate on 128-bit wide packed data operands in conjunction with SIMD and multimedia instructions.

[0088] In one embodiment, uops schedulers 202, 204, 206, dispatch dependent operations before the parent load has finished executing. As uops may be speculatively scheduled and executed in processor 200, processor 200 may also include circuits with logic to handle memory misses. If a data load misses in the data cache, there may be dependent operations in flight in the pipeline that have left the scheduler with temporarily incorrect data. A replay mechanism tracks and re-executes instructions that use incorrect data. Only the dependent operations might need to be replayed and the independent ones may be allowed to complete. The schedulers and replay mechanism of one embodiment of a processor may also be designed to catch instruction sequences for text string comparison operations.

[0089] The term “registers” may refer to the on-board processor storage locations that may be used as part of instructions to identify operands. In other words, registers may be those that may be usable from the outside of the processor (from a programmer’s perspective). However, in some embodiments registers might not be limited to a particular type of circuit. Rather, a register may store data, provide data, and perform the functions described herein. The registers described herein may be implemented by circuitry within a processor using any number of different techniques, such as dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated

and dynamically allocated physical registers, etc. In one embodiment, integer registers store 32-bit integer data. A register file of one embodiment also contains eight multimedia SIMD registers for packed data. For the discussions below, the registers may be understood to be data registers designed to hold packed data, such as 64-bit wide MMX™ registers (also referred to as ‘mm’ registers in some instances) in microprocessors enabled with MMX technology from Intel Corporation of Santa Clara, California. These MMX registers, available in both integer and floating point forms, may operate with packed data elements that accompany SIMD and SSE instructions. Similarly, 128-bit wide XMM registers relating to SSE2, SSE3, SSE4, or beyond (referred to generically as “SSEx”) technology may hold such packed data operands. In one embodiment, in storing packed data and integer data, the registers do not need to differentiate between the two data types. In one embodiment, integer and floating point may be contained in the same register file or different register files. Furthermore, in one embodiment, floating point and integer data may be stored in different registers or the same registers.

[0090] In the examples of the following figures, a number of data operands may be described. FIGURE 3A illustrates various packed data type representations in multimedia registers, in accordance with embodiments of the present disclosure. FIGURE 3A illustrates data types for a packed byte 310, a packed word 320, and a packed doubleword (dword) 330 for 128-bit wide operands. Packed byte format 310 of this example may be 128 bits long and contains sixteen packed byte data elements. A byte may be defined, for example, as eight bits of data. Information for each byte data element may be stored in bit 7 through bit 0 for byte 0, bit 15 through bit 8 for byte 1, bit 23 through bit 16 for byte 2, and finally bit 120 through bit 127 for byte 15. Thus, all available bits may be used in the register. This storage arrangement increases the storage efficiency of the processor. As well, with sixteen data elements accessed, one operation may now be performed on sixteen data elements in parallel.

[0091] Generally, a data element may include an individual piece of data that is stored in a single register or memory location with other data elements of the same length. In packed data sequences relating to SSEx technology, the number of data elements stored in a XMM register may be 128 bits divided by the length in bits of an individual data element. Similarly, in packed data sequences relating to MMX and SSE technology, the number of data

elements stored in an MMX register may be 64 bits divided by the length in bits of an individual data element. Although the data types illustrated in FIGURE 3A may be 128 bits long, embodiments of the present disclosure may also operate with 64-bit wide or other sized operands. Packed word format 320 of this example may be 128 bits long and contains eight packed word data elements. Each packed word contains sixteen bits of information. Packed doubleword format 330 of FIGURE 3A may be 128 bits long and contains four packed doubleword data elements. Each packed doubleword data element contains thirty-two bits of information. A packed quadword may be 128 bits long and contain two packed quad-word data elements.

[0092] FIGURE 3B illustrates possible in-register data storage formats, in accordance with embodiments of the present disclosure. Each packed data may include more than one independent data element. Three packed data formats are illustrated; packed half 341, packed single 342, and packed double 343. One embodiment of packed half 341, packed single 342, and packed double 343 contain fixed-point data elements. For another embodiment one or more of packed half 341, packed single 342, and packed double 343 may contain floating-point data elements. One embodiment of packed half 341 may be 128 bits long containing eight 16-bit data elements. One embodiment of packed single 342 may be 128 bits long and contains four 32-bit data elements. One embodiment of packed double 343 may be 128 bits long and contains two 64-bit data elements. It will be appreciated that such packed data formats may be further extended to other register lengths, for example, to 96-bits, 160-bits, 192-bits, 224-bits, 256-bits, 512-bits or more.

[0093] FIGURE 3C illustrates various signed and unsigned packed data type representations in multimedia registers, in accordance with embodiments of the present disclosure. Unsigned packed byte representation 344 illustrates the storage of an unsigned packed byte in a SIMD register. Information for each byte data element may be stored in bit 7 through bit 0 for byte 0, bit 15 through bit 8 for byte 1, bit 23 through bit 16 for byte 2, and finally bit 120 through bit 127 for byte 15. Thus, all available bits may be used in the register. This storage arrangement may increase the storage efficiency of the processor. As well, with sixteen data elements accessed, one operation may now be performed on sixteen data elements in a parallel fashion. Signed packed byte representation 345 illustrates the storage of a signed packed byte. Note that the eighth bit of every byte data element may be the sign

indicator. Unsigned packed word representation 346 illustrates how word seven through word zero may be stored in a SIMD register. Signed packed word representation 347 may be similar to the unsigned packed word in-register representation 346. Note that the sixteenth bit of each word data element may be the sign indicator. Unsigned packed doubleword representation 348 shows how doubleword data elements are stored. Signed packed doubleword representation 349 may be similar to unsigned packed doubleword in-register representation 348. Note that the necessary sign bit may be the thirty-second bit of each doubleword data element.

[0094] FIGURE 3D illustrates an embodiment of an operation encoding (opcode). Furthermore, format 360 may include register/memory operand addressing modes corresponding with a type of opcode format described in the “IA-32 Intel Architecture Software Developer’s Manual Volume 2: Instruction Set Reference,” which is available from Intel Corporation, Santa Clara, CA on the world-wide-web (www) at intel.com/design/litcentr. In one embodiment, an instruction may be encoded by one or more of fields 361 and 362. Up to two operand locations per instruction may be identified, including up to two source operand identifiers 364 and 365. In one embodiment, destination operand identifier 366 may be the same as source operand identifier 364, whereas in other embodiments they may be different. In another embodiment, destination operand identifier 366 may be the same as source operand identifier 365, whereas in other embodiments they may be different. In one embodiment, one of the source operands identified by source operand identifiers 364 and 365 may be overwritten by the results of the text string comparison operations, whereas in other embodiments identifier 364 corresponds to a source register element and identifier 365 corresponds to a destination register element. In one embodiment, operand identifiers 364 and 365 may identify 32-bit or 64-bit source and destination operands.

[0095] FIGURE 3E illustrates another possible operation encoding (opcode) format 370, having forty or more bits, in accordance with embodiments of the present disclosure. Opcode format 370 corresponds with opcode format 360 and comprises an optional prefix byte 378. An instruction according to one embodiment may be encoded by one or more of fields 378, 371, and 372. Up to two operand locations per instruction may be identified by source operand identifiers 374 and 375 and by prefix byte 378. In one embodiment, prefix byte 378 may be used to identify 32-bit or 64-bit source and destination operands. In one

embodiment, destination operand identifier 376 may be the same as source operand identifier 374, whereas in other embodiments they may be different. For another embodiment, destination operand identifier 376 may be the same as source operand identifier 375, whereas in other embodiments they may be different. In one embodiment, an instruction operates on one or more of the operands identified by operand identifiers 374 and 375 and one or more operands identified by operand identifiers 374 and 375 may be overwritten by the results of the instruction, whereas in other embodiments, operands identified by identifiers 374 and 375 may be written to another data element in another register. Opcode formats 360 and 370 allow register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing specified in part by MOD fields 363 and 373 and by optional scale-index-base and displacement bytes.

[0096] FIGURE 3F illustrates yet another possible operation encoding (opcode) format, in accordance with embodiments of the present disclosure. 64-bit single instruction multiple data (SIMD) arithmetic operations may be performed through a coprocessor data processing (CDP) instruction. Operation encoding (opcode) format 380 depicts one such CDP instruction having CDP opcode fields 382 and 389. The type of CDP instruction, for another embodiment, operations may be encoded by one or more of fields 383, 384, 387, and 388. Up to three operand locations per instruction may be identified, including up to two source operand identifiers 385 and 390 and one destination operand identifier 386. One embodiment of the coprocessor may operate on eight, sixteen, thirty-two, and 64-bit values. In one embodiment, an instruction may be performed on integer data elements. In some embodiments, an instruction may be executed conditionally, using condition field 381. For some embodiments, source data sizes may be encoded by field 383. In some embodiments, Zero (Z), negative (N), carry (C), and overflow (V) detection may be done on SIMD fields. For some instructions, the type of saturation may be encoded by field 384.

[0097] FIGURE 4A is a block diagram illustrating an in-order pipeline and a register renaming stage, out-of-order issue/execution pipeline, in accordance with embodiments of the present disclosure. FIGURE 4B is a block diagram illustrating an in-order architecture core and a register renaming logic, out-of-order issue/execution logic to be included in a processor, in accordance with embodiments of the present disclosure. The solid lined boxes

in FIGURE 4A illustrate the in-order pipeline, while the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline. Similarly, the solid lined boxes in FIGURE 4B illustrate the in-order architecture logic, while the dashed lined boxes illustrates the register renaming logic and out-of-order issue/execution logic.

[0098] In FIGURE 4A, a processor pipeline 400 may include 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] In FIGURE 4B, arrows denote a coupling between two or more units and the direction of the arrow indicates a direction of data flow between those units. FIGURE 4B shows processor core 490 including a front end unit 430 coupled to an execution engine unit 450, and both may be coupled to a memory unit 470.

[0100] 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. In one embodiment, core 490 may be a special-purpose core, such as, for example, a network or communication core, compression engine, graphics core, or the like.

[0101] Front end unit 430 may include a branch prediction unit 432 coupled to an instruction cache unit 434. Instruction cache unit 434 may be coupled to an instruction translation lookaside buffer (TLB) 436. TLB 436 may be coupled to an instruction fetch unit 438, which is coupled to a decode unit 440. Decode unit 440 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 may be decoded from, or which otherwise reflect, or may be derived from, the original instructions. The decoder 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, instruction cache unit 434 may be further coupled to a level 2 (L2) cache unit 476 in memory unit 470. Decode unit 440 may be coupled to a rename/allocator unit 452 in execution engine unit 450.

[0102] Execution engine unit 450 may include rename/allocator unit 452 coupled to a retirement unit 454 and a set of one or more scheduler units 456. Scheduler units 456 represent any number of different schedulers, including reservations stations, central instruction window, etc. Scheduler units 456 may be coupled to physical register file units 458. Each of physical register file 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, etc., status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. Physical register file units 458 may be overlapped by retirement unit 154 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using one or more reorder buffers and one or more retirement register files, using one or more future files, one or more history buffers, and one or more retirement register files; using register maps and a pool of registers; etc.). Generally, the architectural registers may be visible from the outside of the processor or from a programmer's perspective. The registers might not be limited to any known particular type of circuit. Various different types of registers may be suitable as long as they store and provide data as described herein. Examples of suitable registers include, but might not be limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. Retirement unit 454 and physical register file units 458 may be coupled to execution clusters 460. Execution clusters 460 may include a set of one or more execution units 462 and a set of one or more memory access units 464. 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. Scheduler units 456, physical register file units 458, and execution clusters 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 unit, and/or execution cluster – and in the case of a separate memory access pipeline, certain embodiments may be implemented in which only the execution cluster of this pipeline has memory access units 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.

[0103] The set of memory access units 464 may be coupled to memory unit 470, which may include 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, memory access units 464 may include a load unit, a store address unit, and a store data unit, each of which may be coupled to data TLB unit 472 in memory unit 470. L2 cache unit 476 may be coupled to one or more other levels of cache and eventually to a main memory.

[0104] By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement pipeline 400 as follows: 1) instruction fetch 438 may perform fetch and length decoding stages 402 and 404; 2) decode unit 440 may perform decode stage 406; 3) rename/allocator unit 452 may perform allocation stage 408 and renaming stage 410; 4) scheduler units 456 may perform schedule stage 412; 5) physical register file units 458 and memory unit 470 may perform register read/memory read stage 414; execution cluster 460 may perform execute stage 416; 6) memory unit 470 and physical register file units 458 may perform write-back/memory-write stage 418; 7) various units may be involved in the performance of exception handling stage 422; and 8) retirement unit 454 and physical register file units 458 may perform commit stage 424.

[0105] 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).

[0106] It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads) in a variety of manners. Multithreading support may be performed by, for example, 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. Such a combination may include, for example, time sliced fetching and

decoding and simultaneous multithreading thereafter such as in the Intel® Hyperthreading technology.

[0107] While register renaming may be 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 may also include a separate instruction and data cache units 434/474 and a shared L2 cache unit 476, other 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 may be external to the core and/or the processor. In other embodiments, all of the cache may be external to the core and/or the processor.

[0108] FIGURE 5A is a block diagram of a processor 500, in accordance with embodiments of the present disclosure. In one embodiment, processor 500 may include a multicore processor. Processor 500 may include a system agent 510 communicatively coupled to one or more cores 502. Furthermore, cores 502 and system agent 510 may be communicatively coupled to one or more caches 506. Cores 502, system agent 510, and caches 506 may be communicatively coupled via one or more memory control units 552. Furthermore, cores 502, system agent 510, and caches 506 may be communicatively coupled to a graphics module 560 via memory control units 552.

[0109] Processor 500 may include any suitable mechanism for interconnecting cores 502, system agent 510, and caches 506, and graphics module 560. In one embodiment, processor 500 may include a ring-based interconnect unit 508 to interconnect cores 502, system agent 510, and caches 506, and graphics module 560. In other embodiments, processor 500 may include any number of well-known techniques for interconnecting such units. Ring-based interconnect unit 508 may utilize memory control units 552 to facilitate interconnections.

[0110] Processor 500 may include a memory hierarchy comprising one or more levels of caches within the cores, one or more shared cache units such as caches 506, or external memory (not shown) coupled to the set of integrated memory controller units 552. Caches 506 may include any suitable cache. In one embodiment, caches 506 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.

[0111] In various embodiments, one or more of cores 502 may perform multi-threading. System agent 510 may include components for coordinating and operating cores 502. System agent unit 510 may include for example a power control unit (PCU). The PCU may be or include logic and components needed for regulating the power state of cores 502. System agent 510 may include a display engine 512 for driving one or more externally connected displays or graphics module 560. System agent 510 may include an interface for communications busses for graphics. In one embodiment, the interface may be implemented by PCI Express (PCIe). In a further embodiment, the interface may be implemented by PCI Express Graphics (PEG) 514. System agent 510 may include a direct media interface (DMI) 516. DMI 516 may provide links between different bridges on a motherboard or other portion of a computer system. System agent 510 may include a PCIe bridge 518 for providing PCIe links to other elements of a computing system. PCIe bridge 518 may be implemented using a memory controller 520 and coherence logic 522.

[0112] Cores 502 may be implemented in any suitable manner. Cores 502 may be homogenous or heterogeneous in terms of architecture and/or instruction set. In one embodiment, some of cores 502 may be in-order while others may be out-of-order. In another embodiment, two or more of cores 502 may execute the same instruction set, while others may execute only a subset of that instruction set or a different instruction set.

[0113] Processor 500 may include a general-purpose processor, such as a Core™ i3, i5, i7, 2 Duo and Quad, Xeon™, Itanium™, XScale™ or StrongARM™ processor, which may be available from Intel Corporation, of Santa Clara, Calif. Processor 500 may be provided from another company, such as ARM Holdings, Ltd, MIPS, etc. Processor 500 may be a special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, co-processor, embedded processor, or the like. Processor 500 may be implemented on one or more chips. Processor 500 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.

[0114] In one embodiment, a given one of caches 506 may be shared by multiple ones of cores 502. In another embodiment, a given one of caches 506 may be dedicated to

one of cores 502. The assignment of caches 506 to cores 502 may be handled by a cache controller or other suitable mechanism. A given one of caches 506 may be shared by two or more cores 502 by implementing time-slices of a given cache 506.

[0115] Graphics module 560 may implement an integrated graphics processing subsystem. In one embodiment, graphics module 560 may include a graphics processor. Furthermore, graphics module 560 may include a media engine 565. Media engine 565 may provide media encoding and video decoding.

[0116] FIGURE 5B is a block diagram of an example implementation of a core 502, in accordance with embodiments of the present disclosure. Core 502 may include a front end 570 communicatively coupled to an out-of-order engine 580. Core 502 may be communicatively coupled to other portions of processor 500 through cache hierarchy 503.

[0117] Front end 570 may be implemented in any suitable manner, such as fully or in part by front end 201 as described above. In one embodiment, front end 570 may communicate with other portions of processor 500 through cache hierarchy 503. In a further embodiment, front end 570 may fetch instructions from portions of processor 500 and prepare the instructions to be used later in the processor pipeline as they are passed to out-of-order execution engine 580.

[0118] Out-of-order execution engine 580 may be implemented in any suitable manner, such as fully or in part by out-of-order execution engine 203 as described above. Out-of-order execution engine 580 may prepare instructions received from front end 570 for execution. Out-of-order execution engine 580 may include an allocate module 1282. In one embodiment, allocate module 1282 may allocate resources of processor 500 or other resources, such as registers or buffers, to execute a given instruction. Allocate module 1282 may make allocations in schedulers, such as a memory scheduler, fast scheduler, or floating point scheduler. Such schedulers may be represented in FIGURE 5B by resource schedulers 584. Allocate module 1282 may be implemented fully or in part by the allocation logic described in conjunction with FIGURE 2. Resource schedulers 584 may determine when an instruction is ready to execute based on the readiness of a given resource's sources and the availability of execution resources needed to execute an instruction. Resource schedulers 584 may be implemented by, for example, schedulers 202, 204, 206 as discussed above. Resource schedulers 584 may schedule the execution of instructions upon one or more

resources. In one embodiment, such resources may be internal to core 502, and may be illustrated, for example, as resources 586. In another embodiment, such resources may be external to core 502 and may be accessible by, for example, cache hierarchy 503. Resources may include, for example, memory, caches, register files, or registers. Resources internal to core 502 may be represented by resources 586 in FIGURE 5B. As necessary, values written to or read from resources 586 may be coordinated with other portions of processor 500 through, for example, cache hierarchy 503. As instructions are assigned resources, they may be placed into a reorder buffer 588. Reorder buffer 588 may track instructions as they are executed and may selectively reorder their execution based upon any suitable criteria of processor 500. In one embodiment, reorder buffer 588 may identify instructions or a series of instructions that may be executed independently. Such instructions or a series of instructions may be executed in parallel from other such instructions. Parallel execution in core 502 may be performed by any suitable number of separate execution blocks or virtual processors. In one embodiment, shared resources—such as memory, registers, and caches—may be accessible to multiple virtual processors within a given core 502. In other embodiments, shared resources may be accessible to multiple processing entities within processor 500.

[0119] Cache hierarchy 503 may be implemented in any suitable manner. For example, cache hierarchy 503 may include one or more lower or mid-level caches, such as caches 572, 574. In one embodiment, cache hierarchy 503 may include an LLC 595 communicatively coupled to caches 572, 574. In another embodiment, LLC 595 may be implemented in a module 590 accessible to all processing entities of processor 500. In a further embodiment, module 590 may be implemented in an uncore module of processors from Intel, Inc. Module 590 may include portions or subsystems of processor 500 necessary for the execution of core 502 but might not be implemented within core 502. Besides LLC 595, Module 590 may include, for example, hardware interfaces, memory coherency coordinators, interprocessor interconnects, instruction pipelines, or memory controllers. Access to RAM 599 available to processor 500 may be made through module 590 and, more specifically, LLC 595. Furthermore, other instances of core 502 may similarly access module 590. Coordination of the instances of core 502 may be facilitated in part through module 590.

[0120] FIGURES 6–8 may illustrate exemplary systems suitable for including processor 500, while FIGURE 9 may illustrate an exemplary system on a chip (SoC) that may include one or more of cores 502. Other system designs and implementations 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, may also be suitable. In general, a huge variety of systems or electronic devices that incorporate a processor and/or other execution logic as disclosed herein may be generally suitable.

[0121] FIGURE 6 illustrates a block diagram of a system 600, in accordance with embodiments of the present disclosure. System 600 may include one or more processors 610, 615, which may be coupled to graphics memory controller hub (GMCH) 620. The optional nature of additional processors 615 is denoted in FIGURE 6 with broken lines.

[0122] Each processor 610,615 may be some version of processor 500. However, it should be noted that integrated graphics logic and integrated memory control units might not exist in processors 610,615. FIGURE 6 illustrates that GMCH 620 may be coupled to a memory 640 that may be, for example, a dynamic random access memory (DRAM). The DRAM may, for at least one embodiment, be associated with a non-volatile cache.

[0123] GMCH 620 may be a chipset, or a portion of a chipset. GMCH 620 may communicate with processors 610, 615 and control interaction between processors 610, 615 and memory 640. GMCH 620 may also act as an accelerated bus interface between the processors 610, 615 and other elements of system 600. In one embodiment, GMCH 620 communicates with processors 610, 615 via a multi-drop bus, such as a frontside bus (FSB) 695.

[0124] Furthermore, GMCH 620 may be coupled to a display 645 (such as a flat panel display). In one embodiment, GMCH 620 may include an integrated graphics accelerator. GMCH 620 may be further coupled to an input/output (I/O) controller hub (ICH) 650, which may be used to couple various peripheral devices to system 600. External graphics device 660 may include be a discrete graphics device coupled to ICH 650 along with another peripheral device 670.

[0125] In other embodiments, additional or different processors may also be present in system 600. For example, additional processors 610, 615 may include additional processors that may be the same as processor 610, additional processors that may be heterogeneous or asymmetric to processor 610, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor. There may be a variety of differences between the physical resources 610, 615 in terms of a spectrum of metrics of merit including architectural, micro-architectural, thermal, power consumption characteristics, and the like. These differences may effectively manifest themselves as asymmetry and heterogeneity amongst processors 610, 615. For at least one embodiment, various processors 610, 615 may reside in the same die package.

[0126] FIGURE 7 illustrates a block diagram of a second system 700, in accordance with embodiments of the present disclosure. As shown in FIGURE 7, multiprocessor system 700 may include a point-to-point interconnect system, and may include a first processor 770 and a second processor 780 coupled via a point-to-point interconnect 750. Each of processors 770 and 780 may be some version of processor 500 as one or more of processors 610, 615.

[0127] While FIGURE 7 may illustrate two processors 770, 780, it is to be understood that the scope of the present disclosure is not so limited. In other embodiments, one or more additional processors may be present in a given processor.

[0128] Processors 770 and 780 are shown including integrated memory controller units 772 and 782, respectively. Processor 770 may also include as part of its bus controller units point-to-point (P-P) interfaces 776 and 778; similarly, second processor 780 may include P-P interfaces 786 and 788. Processors 770, 780 may exchange information via a point-to-point (P-P) interface 750 using P-P interface circuits 778, 788. As shown in FIGURE 7, IMCs 772 and 782 may couple the processors to respective memories, namely a memory 732 and a memory 734, which in one embodiment may be portions of main memory locally attached to the respective processors.

[0129] Processors 770, 780 may each exchange information with a chipset 790 via individual P-P interfaces 752, 754 using point to point interface circuits 776, 794, 786, 798. In one embodiment, chipset 790 may also exchange information with a high-performance graphics circuit 738 via a high-performance graphics interface 739.

[0130] 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.

[0131] Chipset 790 may be coupled to a first bus 716 via an interface 796. In one embodiment, first bus 716 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 disclosure is not so limited.

[0132] As shown in FIGURE 7, various I/O devices 714 may be coupled to first bus 716, along with a bus bridge 718 which couples first bus 716 to a second bus 720. In one embodiment, second bus 720 may be a low pin count (LPC) bus. Various devices may be coupled to second bus 720 including, for example, a keyboard and/or mouse 722, communication devices 727 and a storage unit 728 such as a disk drive or other mass storage device which may include instructions/code and data 730, in one embodiment. Further, an audio I/O 724 may be coupled to second bus 720. Note that other architectures may be possible. For example, instead of the point-to-point architecture of FIGURE 7, a system may implement a multi-drop bus or other such architecture.

[0133] FIGURE 8 illustrates a block diagram of a third system 800 in accordance with embodiments of the present disclosure. Like elements in FIGURES 7 and 8 bear like reference numerals, and certain aspects of FIGURE 7 have been omitted from FIGURE 8 in order to avoid obscuring other aspects of FIGURE 8.

[0134] FIGURE 8 illustrates that processors 870, 880 may include integrated memory and I/O control logic ("CL") 872 and 882, respectively. For at least one embodiment, CL 872, 882 may include integrated memory controller units such as that described above in connection with FIGURES 5 and 7. In addition, CL 872, 882 may also include I/O control logic. FIGURE 8 illustrates that not only memories 832, 834 may be coupled to CL 872, 882, but also that I/O devices 814 may also be coupled to control logic 872, 882. Legacy I/O devices 815 may be coupled to chipset 890.

[0135] FIGURE 9 illustrates a block diagram of a SoC 900, in accordance with embodiments of the present disclosure. Similar elements in FIGURE 5 bear like reference numerals. Also, dashed lined boxes may represent optional features on more advanced SoCs.

An interconnect units 902 may be coupled to: an application processor 910 which may include a set of one or more cores 902A-N and shared cache units 906; a system agent unit 910; a bus controller units 916; an integrated memory controller units 914; a set or one or more media processors 920 which may include integrated graphics logic 908, an image processor 924 for providing still and/or video camera functionality, an audio processor 926 for providing hardware audio acceleration, and a video processor 928 for providing video encode/decode acceleration; an static random access memory (SRAM) unit 930; a direct memory access (DMA) unit 932; and a display unit 940 for coupling to one or more external displays.

[0136] FIGURE 10 illustrates a processor containing a central processing unit (CPU) and a graphics processing unit (GPU), which may perform at least one instruction, in accordance with embodiments of the present disclosure. In one embodiment, an instruction to perform operations according to at least one embodiment could be performed by the CPU. In another embodiment, the instruction could be performed by the GPU. In still another embodiment, the instruction may be performed through a combination of operations performed by the GPU and the CPU. For example, in one embodiment, an instruction in accordance with one embodiment may be received and decoded for execution on the GPU. However, one or more operations within the decoded instruction may be performed by a CPU and the result returned to the GPU for final retirement of the instruction. Conversely, in some embodiments, the CPU may act as the primary processor and the GPU as the co-processor.

[0137] In some embodiments, instructions that benefit from highly parallel, throughput processors may be performed by the GPU, while instructions that benefit from the performance of processors that benefit from deeply pipelined architectures may be performed by the CPU. For example, graphics, scientific applications, financial applications and other parallel workloads may benefit from the performance of the GPU and be executed accordingly, whereas more sequential applications, such as operating system kernel or application code may be better suited for the CPU.

[0138] In FIGURE 10, processor 1000 includes a CPU 1005, GPU 1010, image processor 1015, video processor 1020, USB controller 1025, UART controller 1030, SPI/SDIO controller 1035, display device 1040, memory interface controller 1045, MIPI controller 1050,

flash memory controller 1055, dual data rate (DDR) controller 1060, security engine 1065, and I²S/I²C controller 1070. Other logic and circuits may be included in the processor of FIGURE 10, including more CPUs or GPUs and other peripheral interface controllers.

[0139] One or more aspects of at least one embodiment may be implemented by representative data 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 (“tape”) and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor. For example, IP cores, such as the Cortex™ family of processors developed by ARM Holdings, Ltd. and Loongson IP cores developed the Institute of Computing Technology (ICT) of the Chinese Academy of Sciences may be licensed or sold to various customers or licensees, such as Texas Instruments, Qualcomm, Apple, or Samsung and implemented in processors produced by these customers or licensees.

[0140] FIGURE 11 illustrates a block diagram illustrating the development of IP cores, in accordance with embodiments of the present disclosure. Storage 1130 may include simulation software 1120 and/or hardware or software model 1110. In one embodiment, the data representing the IP core design may be provided to storage 1130 via memory 1140 (e.g., hard disk), wired connection (e.g., internet) 1150 or wireless connection 1160. The IP core information generated by the simulation tool and model may then be transmitted to a fabrication facility where it may be fabricated by a 3rd party to perform at least one instruction in accordance with at least one embodiment.

[0141] In some embodiments, one or more instructions may correspond to a first type or architecture (e.g., x86) and be translated or emulated on a processor of a different type or architecture (e.g., ARM). An instruction, according to one embodiment, may therefore be performed on any processor or processor type, including ARM, x86, MIPS, a GPU, or other processor type or architecture.

[0142] FIGURE 12 illustrates how an instruction of a first type may be emulated by a processor of a different type, in accordance with embodiments of the present disclosure. In FIGURE 12, program 1205 contains some instructions that may perform the same or substantially the same function as an instruction according to one embodiment. However the

instructions of program 1205 may be of a type and/or format that is different from or incompatible with processor 1215, meaning the instructions of the type in program 1205 may not be able to execute natively by the processor 1215. However, with the help of emulation logic, 1210, the instructions of program 1205 may be translated into instructions that may be natively be executed by the processor 1215. In one embodiment, the emulation logic may be embodied in hardware. In another embodiment, the emulation logic may be embodied in a tangible, machine-readable medium containing software to translate instructions of the type in program 1205 into the type natively executable by processor 1215. In other embodiments, emulation logic may be a combination of fixed-function or programmable hardware and a program stored on a tangible, machine-readable medium. In one embodiment, the processor contains the emulation logic, whereas in other embodiments, the emulation logic exists outside of the processor and may be provided by a third party. In one embodiment, the processor may load the emulation logic embodied in a tangible, machine-readable medium containing software by executing microcode or firmware contained in or associated with the processor.

[0143] FIGURE 13 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, in accordance with embodiments of the present disclosure. In the illustrated embodiment, the instruction converter may be a software instruction converter, although the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIGURE 13 shows a program in a high level language 1302 may be compiled using an x86 compiler 1304 to generate x86 binary code 1306 that may be natively executed by a processor with at least one x86 instruction set core 1316. The processor with at least one x86 instruction set core 1316 represents any processor that may 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. x86 compiler 1304 represents a compiler that may be operable to generate x86 binary code 1306 (e.g., object code) that may, with or without

additional linkage processing, be executed on the processor with at least one x86 instruction set core 1316. Similarly, FIGURE 13 shows the program in high level language 1302 may be compiled using an alternative instruction set compiler 1308 to generate alternative instruction set binary code 1310 that may be natively executed by a processor without at least one x86 instruction set core 1314 (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). Instruction converter 1312 may be used to convert x86 binary code 1306 into code that may be natively executed by the processor without an x86 instruction set core 1314. This converted code might not be the same as alternative instruction set binary code 1310; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, instruction converter 1312 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 x86 binary code 1306.

[0144] FIGURE 14 is a block diagram of an instruction set architecture 1400 of a processor, in accordance with embodiments of the present disclosure. Instruction set architecture 1400 may include any suitable number or kind of components.

[0145] For example, instruction set architecture 1400 may include processing entities such as one or more cores 1406, 1407 and a graphics processing unit 1415. Cores 1406, 1407 may be communicatively coupled to the rest of instruction set architecture 1400 through any suitable mechanism, such as through a bus or cache. In one embodiment, cores 1406, 1407 may be communicatively coupled through an L2 cache control 1408, which may include a bus interface unit 1409 and an L2 cache 1410. Cores 1406, 1407 and graphics processing unit 1415 may be communicatively coupled to each other and to the remainder of instruction set architecture 1400 through interconnect 1410. In one embodiment, graphics processing unit 1415 may use a video code 1420 defining the manner in which particular video signals will be encoded and decoded for output.

[0146] Instruction set architecture 1400 may also include any number or kind of interfaces, controllers, or other mechanisms for interfacing or communicating with other portions of an electronic device or system. Such mechanisms may facilitate interaction with,

for example, peripherals, communications devices, other processors, or memory. In the example of FIGURE 14, instruction set architecture 1400 may include a liquid crystal display (LCD) video interface 1425, a subscriber interface module (SIM) interface 1430, a boot ROM interface 1435, a synchronous dynamic random access memory (SDRAM) controller 1440, a flash controller 1445, and a serial peripheral interface (SPI) master unit 1450. LCD video interface 1425 may provide output of video signals from, for example, GPU 1415 and through, for example, a mobile industry processor interface (MIPI) 1490 or a high-definition multimedia interface (HDMI) 1495 to a display. Such a display may include, for example, an LCD. SIM interface 1430 may provide access to or from a SIM card or device. SDRAM controller 1440 may provide access to or from memory such as an SDRAM chip or module. Flash controller 1445 may provide access to or from memory such as flash memory or other instances of RAM. SPI master unit 1450 may provide access to or from communications modules, such as a Bluetooth module 1470, high-speed 3G modem 1475, global positioning system module 1480, or wireless module 1485 implementing a communications standard such as 802.11.

[0147] FIGURE 15 is a more detailed block diagram of an instruction set architecture 1500 of a processor, in accordance with embodiments of the present disclosure. Instruction architecture 1500 may implement one or more aspects of instruction set architecture 1400. Furthermore, instruction set architecture 1500 may illustrate modules and mechanisms for the execution of instructions within a processor.

[0148] Instruction architecture 1500 may include a memory system 1540 communicatively coupled to one or more execution entities 1565. Furthermore, instruction architecture 1500 may include a caching and bus interface unit such as unit 1510 communicatively coupled to execution entities 1565 and memory system 1540. In one embodiment, loading of instructions into execution entities 1564 may be performed by one or more stages of execution. Such stages may include, for example, instruction prefetch stage 1530, dual instruction decode stage 1550, register rename stage 155, issue stage 1560, and writeback stage 1570.

[0149] In one embodiment, memory system 1540 may include an executed instruction pointer 1580. Executed instruction pointer 1580 may store a value identifying the oldest, undispatched instruction within a batch of instructions. The oldest instruction may

correspond to the lowest Program Order (PO) value. A PO may include a unique number of an instruction. Such an instruction may be a single instruction within a thread represented by multiple strands. A PO may be used in ordering instructions to ensure correct execution semantics of code. A PO may be reconstructed by mechanisms such as evaluating increments to PO encoded in the instruction rather than an absolute value. Such a reconstructed PO may be known as an “RPO.” Although a PO may be referenced herein, such a PO may be used interchangeably with an RPO. A strand may include a sequence of instructions that are data dependent upon each other. The strand may be arranged by a binary translator at compilation time. Hardware executing a strand may execute the instructions of a given strand in order according to PO of the various instructions. A thread may include multiple strands such that instructions of different strands may depend upon each other. A PO of a given strand may be the PO of the oldest instruction in the strand which has not yet been dispatched to execution from an issue stage. Accordingly, given a thread of multiple strands, each strand including instructions ordered by PO, executed instruction pointer 1580 may store the oldest—illustrated by the lowest number—PO in the thread.

[0150] In another embodiment, memory system 1540 may include a retirement pointer 1582. Retirement pointer 1582 may store a value identifying the PO of the last retired instruction. Retirement pointer 1582 may be set by, for example, retirement unit 454. If no instructions have yet been retired, retirement pointer 1582 may include a null value.

[0151] Execution entities 1565 may include any suitable number and kind of mechanisms by which a processor may execute instructions. In the example of FIGURE 15, execution entities 1565 may include ALU/multiplication units (MUL) 1566, ALUs 1567, and floating point units (FPU) 1568. In one embodiment, such entities may make use of information contained within a given address 1569. Execution entities 1565 in combination with stages 1530, 1550, 1555, 1560, 1570 may collectively form an execution unit.

[0152] Unit 1510 may be implemented in any suitable manner. In one embodiment, unit 1510 may perform cache control. In such an embodiment, unit 1510 may thus include a cache 1525. Cache 1525 may be implemented, in a further embodiment, as an L2 unified cache with any suitable size, such as zero, 128k, 256k, 512k, 1M, or 2M bytes of memory. In another, further embodiment, cache 1525 may be implemented in error-correcting code memory. In another embodiment, unit 1510 may perform bus interfacing to other portions

of a processor or electronic device. In such an embodiment, unit 1510 may thus include a bus interface unit 1520 for communicating over an interconnect, intraprocessor bus, interprocessor bus, or other communication bus, port, or line. Bus interface unit 1520 may provide interfacing in order to perform, for example, generation of the memory and input/output addresses for the transfer of data between execution entities 1565 and the portions of a system external to instruction architecture 1500.

[0153] To further facilitate its functions, bus interface unit 1520 may include an interrupt control and distribution unit 1511 for generating interrupts and other communications to other portions of a processor or electronic device. In one embodiment, bus interface unit 1520 may include a snoop control unit 1512 that handles cache access and coherency for multiple processing cores. In a further embodiment, to provide such functionality, snoop control unit 1512 may include a cache-to-cache transfer unit that handles information exchanges between different caches. In another, further embodiment, snoop control unit 1512 may include one or more snoop filters 1514 that monitors the coherency of other caches (not shown) so that a cache controller, such as unit 1510, does not have to perform such monitoring directly. Unit 1510 may include any suitable number of timers 1515 for synchronizing the actions of instruction architecture 1500. Also, unit 1510 may include an AC port 1516.

[0154] Memory system 1540 may include any suitable number and kind of mechanisms for storing information for the processing needs of instruction architecture 1500. In one embodiment, memory system 1504 may include a load store unit 1530 for storing information such as buffers written to or read back from memory or registers. In another embodiment, memory system 1504 may include a translation lookaside buffer (TLB) 1545 that provides look-up of address values between physical and virtual addresses. In yet another embodiment, bus interface unit 1520 may include a memory management unit (MMU) 1544 for facilitating access to virtual memory. In still yet another embodiment, memory system 1504 may include a prefetcher 1543 for requesting instructions from memory before such instructions are actually needed to be executed, in order to reduce latency.

[0155] The operation of instruction architecture 1500 to execute an instruction may be performed through different stages. For example, using unit 1510 instruction prefetch

stage 1530 may access an instruction through prefetcher 1543. Instructions retrieved may be stored in instruction cache 1532. Prefetch stage 1530 may enable an option 1531 for fast-loop mode, wherein a series of instructions forming a loop that is small enough to fit within a given cache are executed. In one embodiment, such an execution may be performed without needing to access additional instructions from, for example, instruction cache 1532. Determination of what instructions to prefetch may be made by, for example, branch prediction unit 1535, which may access indications of execution in global history 1536, indications of target addresses 1537, or contents of a return stack 1538 to determine which of branches 1557 of code will be executed next. Such branches may be possibly prefetched as a result. Branches 1557 may be produced through other stages of operation as described below. Instruction prefetch stage 1530 may provide instructions as well as any predictions about future instructions to dual instruction decode stage.

[0156] Dual instruction decode stage 1550 may translate a received instruction into microcode-based instructions that may be executed. Dual instruction decode stage 1550 may simultaneously decode two instructions per clock cycle. Furthermore, dual instruction decode stage 1550 may pass its results to register rename stage 1555. In addition, dual instruction decode stage 1550 may determine any resulting branches from its decoding and eventual execution of the microcode. Such results may be input into branches 1557.

[0157] Register rename stage 1555 may translate references to virtual registers or other resources into references to physical registers or resources. Register rename stage 1555 may include indications of such mapping in a register pool 1556. Register rename stage 1555 may alter the instructions as received and send the result to issue stage 1560.

[0158] Issue stage 1560 may issue or dispatch commands to execution entities 1565. Such issuance may be performed in an out-of-order fashion. In one embodiment, multiple instructions may be held at issue stage 1560 before being executed. Issue stage 1560 may include an instruction queue 1561 for holding such multiple commands. Instructions may be issued by issue stage 1560 to a particular processing entity 1565 based upon any acceptable criteria, such as availability or suitability of resources for execution of a given instruction. In one embodiment, issue stage 1560 may reorder the instructions within instruction queue 1561 such that the first instructions received might not be the first instructions executed. Based upon the ordering of instruction queue 1561, additional branching information may be

provided to branches 1557. Issue stage 1560 may pass instructions to executing entities 1565 for execution.

[0159] Upon execution, writeback stage 1570 may write data into registers, queues, or other structures of instruction set architecture 1500 to communicate the completion of a given command. Depending upon the order of instructions arranged in issue stage 1560, the operation of writeback stage 1570 may enable additional instructions to be executed. Performance of instruction set architecture 1500 may be monitored or debugged by trace unit 1575.

[0160] FIGURE 16 is a block diagram of an execution pipeline 1600 for an instruction set architecture of a processor, in accordance with embodiments of the present disclosure. Execution pipeline 1600 may illustrate operation of, for example, instruction architecture 1500 of FIGURE 15.

[0161] Execution pipeline 1600 may include any suitable combination of steps or operations. In 1605, predictions of the branch that is to be executed next may be made. In one embodiment, such predictions may be based upon previous executions of instructions and the results thereof. In 1610, instructions corresponding to the predicted branch of execution may be loaded into an instruction cache. In 1615, one or more such instructions in the instruction cache may be fetched for execution. In 1620, the instructions that have been fetched may be decoded into microcode or more specific machine language. In one embodiment, multiple instructions may be simultaneously decoded. In 1625, references to registers or other resources within the decoded instructions may be reassigned. For example, references to virtual registers may be replaced with references to corresponding physical registers. In 1630, the instructions may be dispatched to queues for execution. In 1640, the instructions may be executed. Such execution may be performed in any suitable manner. In 1650, the instructions may be issued to a suitable execution entity. The manner in which the instruction is executed may depend upon the specific entity executing the instruction. For example, at 1655, an ALU may perform arithmetic functions. The ALU may utilize a single clock cycle for its operation, as well as two shifters. In one embodiment, two ALUs may be employed, and thus two instructions may be executed at 1655. At 1660, a determination of a resulting branch may be made. A program counter may be used to designate the destination to which the branch will be made. 1660 may be executed within a single clock cycle. At 1665,

floating point arithmetic may be performed by one or more FPUs. The floating point operation may require multiple clock cycles to execute, such as two to ten cycles. At 1670, multiplication and division operations may be performed. Such operations may be performed in four clock cycles. At 1675, loading and storing operations to registers or other portions of pipeline 1600 may be performed. The operations may include loading and storing addresses. Such operations may be performed in four clock cycles. At 1680, write-back operations may be performed as required by the resulting operations of 1655-1675.

[0162] FIGURE 17 is a block diagram of an electronic device 1700 for utilizing a processor 1710, in accordance with embodiments of the present disclosure. Electronic device 1700 may include, for example, a notebook, an ultrabook, a computer, a tower server, a rack server, a blade server, a laptop, a desktop, a tablet, a mobile device, a phone, an embedded computer, or any other suitable electronic device.

[0163] Electronic device 1700 may include processor 1710 communicatively coupled to any suitable number or kind of components, peripherals, modules, or devices. Such coupling may be accomplished by any suitable kind of bus or interface, such as I²C bus, system management bus (SMBus), low pin count (LPC) bus, SPI, high definition audio (HDA) bus, Serial Advance Technology Attachment (SATA) bus, USB bus (versions 1, 2, 3), or Universal Asynchronous Receiver/Transmitter (UART) bus.

[0164] Such components may include, for example, a display 1724, a touch screen 1725, a touch pad 1730, a near field communications (NFC) unit 1745, a sensor hub 1740, a thermal sensor 1746, an express chipset (EC) 1735, a trusted platform module (TPM) 1738, BIOS/firmware/flash memory 1722, a digital signal processor 1760, a drive 1720 such as a solid state disk (SSD) or a hard disk drive (HDD), a wireless local area network (WLAN) unit 1750, a Bluetooth unit 1752, a wireless wide area network (WWAN) unit 1756, a global positioning system (GPS), a camera 1754 such as a USB 3.0 camera, or a low power double data rate (LPDDR) memory unit 1715 implemented in, for example, the LPDDR3 standard. These components may each be implemented in any suitable manner.

[0165] Furthermore, in various embodiments other components may be communicatively coupled to processor 1710 through the components discussed above. For example, an accelerometer 1741, ambient light sensor (ALS) 1742, compass 1743, and gyroscope 1744 may be communicatively coupled to sensor hub 1740. A thermal sensor

1739, fan 1737, keyboard 1746, and touch pad 1730 may be communicatively coupled to EC 1735. Speaker 1763, headphones 1764, and a microphone 1765 may be communicatively coupled to an audio unit 1764, which may in turn be communicatively coupled to DSP 1760. Audio unit 1764 may include, for example, an audio codec and a class D amplifier. A SIM card 1757 may be communicatively coupled to WWAN unit 1756. Components such as WLAN unit 1750 and Bluetooth unit 1752, as well as WWAN unit 1756 may be implemented in a next generation form factor (NGFF).

[0166] Embodiments of the present disclosure also provide cooperative triggering for observability features. More particularly, there is disclosed a CTB block with the ability to coordinate tracing, triggering, debugging, and other functions.

[0167] FIGURE 18 is a simplified block diagram of a processing core 1800 according to embodiments of the present disclosure. Processing core 1800 is shown with only selected elements to illustrate certain features discussed herein.

[0168] In this example, software is provided to core 1800. As software executes, it is desirable to engage in performance monitoring and debugging as described herein. Performance monitoring and debugging features may be generally classed into several different categories. Each may be referenced by a “weight,” indicating how obtrusive it is with respect to software operations. Several example perfmon capabilities are described in more detail in [0042] – [0053] above.

[0169] As discussed above, debugging breakpoints, variable watch lists, and other tools used with software tools may be considered the highest weight performance monitoring processes. These are generally highly intrusive on the execution flow, and may not be suitable for production software, or software with time constraints. In many embodiments, these facilities are provided purely in software, and are used in conjunction with software debugging tools. Because breakpoints and variable inspection include human interaction on human time scales, they generally interrupt execution on the order of billions of cycles.

[0170] The next highest weight may be a use of performance monitoring interrupts (PMI). In an embodiment, these are provided by performance tracer 1806. In an example, a designer specifies certain conditions that are to trigger the observability feature. Upon the occurrence of one or more of these conditions, performance tracer 1806 may either provide traces, which are data that may be written out to memory, or may provide a performance

monitoring interrupt (PMI). The PMI triggers a software PMI handler, which then receives trace information and any other state information that is requested by the user and performs analysis on the data. Because the PMI triggers a non-interactive interrupt routine, handling of the trigger may typically take tens or hundreds of thousands of processor cycles to complete. Thus, while the PMI provides great flexibility, and the ability for the programmer to craft any desired PMI handler and to collect any data desired, interference with operation of the program can be substantial. Depending on the use case, the use of many PMIs may not be appropriate for production or time-sensitive software.

[0171] PMU 1802 may provide PEBS, which may be considered a medium weight visibility feature. Advantageously, in one embodiment, PEBS executes wholly in microcode, so that there is no need to pass control to an external handler routine, saving many cycles. In some embodiments, the longest delay in PEBS is the process of writing data out to memory.

[0172] PEBS may be triggered by events similar to those events that trigger PMI, but PEBS may generally write less data to the PEBS buffer. Note that the PEBS buffer is much smaller than the memory available to a PMI handler, as the PMI handler has access to essentially all available system resources. In contrast, PEBS has access only to the pre-defined PEBS buffer. Once the triggering event occurs, PEBS writes certain values out to the PEBS buffer. Thus, when a PEBS event occurs, PMU 1802, executing in microcode, writes limited data out to PEBS buffer, costing perhaps tens or hundreds of cycles.

[0173] Processor 1800 may also provide debug registers 1804, which are dedicated registers for program debugging. In an example Intel® processor, there are six debug registers, named DR0 ... DR7, with DR4 and DR5 being deprecated synonyms for DR6 and DR7. The debug registers allow programmers to selectively enable various debug conditions and triggers associated with debug addresses. Two of these registers are used to control debug features. These registers are accessed by variants of the MOV instruction. A debug register may be either the source operand or destination operand. The debug registers are privileged resources; the MOV instructions that access them can only be executed at privilege level zero.

[0174] An Intel® processor trace 1806, for example, may also be provided. PT 1806 is configured to log encoded information about software execution in a minimally invasive manner. The trace outputs may go to memory, or to debug ports, such as USB or PTI.

[0175] There is also disclosed herein an on-chip Core Triggering Block (CTB) 1810, which may interconnect with PT 1806, PMU 1802, or debug registers 1804. CTB 1810 may be configured to provide truly lightweight perfmon functions, which in certain embodiments consume as few as zero extra cycles, such as where CTB 1810 operates on the instruction boundary. For example, PMU 1802 may route directly to CTB 1810, so that when the counter overflow occurs, CTB 1810 immediately and automatically logs a small amount of data on-chip. This may avoid the necessity of taking a PMI (architectural event) or even a PEBS microarchitectural event. However, since the on-chip buffer available to CTB 1810 may be very small, and the data may need to be flushed to memory, or some other endpoint with limited bandwidth support. Thus a user will still need to exercise ordinary prudence in designating CTB events, to avoid excessively consuming memory (or other transport) bandwidth. However, the CTB may be expected to provide up to several orders of magnitude more frequent sampling than even PEBS (e.g., on the order of one in every thousands of events, rather than one in every millions).

[0176] Note that some existing CPUs, and in particular modern Intel® CPUs, already include hardware capable of logging a small amount of data (on the order of dozens of bytes) to an on-chip buffer within the boundary a single cycle. By leveraging this existing infrastructure, CTB 1810 can realize a “poor-man’s PEBS.” In other words, data can be logged as with PEBS, but without incurring the PEBS cost. The tradeoff in this case is that in some embodiments, less data can be logged. Thus, a user may select the most important data to be logged to on-chip memory, but may log it very frequently.

[0177] CTB 1810 may be provisioned in hardware to provide a true “lightweight” visibility feature. CTB 1810 executes in hardware, as opposed to PMU 1802 which processes PEBS events in microcode. Advantageously, CTB 1810 can execute its function on an instruction boundary once a triggering event occurs. Thus, in certain embodiments, CTB 1810 can perform some perfmon functions without any interruption to the program flow at all. This can be especially important in real-time or time critical operations, wherein interfering with instruction execution can cause critical errors. Once CTB 1810 writes data to its local buffer, which in some cases may include or be associated with either debug registers 1804 or a PEBS buffer, it waits until sufficient resources are available to write those data out to memory. Thus, a highly aggressive CTB regimen may impact performance, because old data will need to

be written out before new data can be stored in the buffer. A system designer may exercise ordinary skill in selecting how much data to collect on which events. When it is convenient or necessary, CTB 1810 may flash its cached data out to memory or other endpoint.

[0178] In one example, a software interface to configure CTB 1810 includes writing to one or more registers, such as registers within debug registers 1804, that select triggering inputs and trigger actions. Each trigger source/action pair may be configured by a register, such as the example TRIGGER_CFGn register, where n is the trigger ID. In this example, the lower bits select the trigger source(s) and the upper bits choose the associated action(s) to be taken when the trigger fires. For example:

TRIGGER_CFG0 Register	
TriggerAction[63:8]	TriggerSource[7:0]

[0179] TriggerSource values (by bit, choose any combination, values OR'd):

0	PerfMon Counter 0 overflow
1	PerfMon Counter 1 overflow
...	...
8	DR0 match
9	DR1 match
...	...
12	External input 0 transitions to 1
13	External input 1 transitions to 1
...	...

[0180] TriggerAction values (by bit, choose any combination (all will be taken)):

0	Enable Intel PT
1	Disable Intel PT
2	Enable PerfMon Counter 0
3	Enable PerfMon Counter 1
...	...
10	Disable PerfMon Counter 0
11	Disable PerfMon Counter 1
...	...
19	Generate Current Instruction Pointer (IP) packet in trace
20	Generate Data Linear Address (DLA) packet in trace
21	Assert external output 0
22	Assert external output 1
...	...

[0181] The above are nonlimiting examples, and many additional trigger inputs or actions can be configured according to design requirements. Examples presented below illustrate benefits achieved with the options shown above. Additional benefits may be derived by providing other options or configurations.

[0182] Lightweight Sampling – By enabling perfmon counter overflows to trigger hardware logging of packets to PT 1806, the software intrusion of a PMI or PEBS event, both of which interrupt software execution, can be averted. This may allow sampling at much higher rates, as the slowdown resulting from the write of a few extra bytes through the memory subsystem is very small. Some existing sampling tools default to sampling rates of every 1 ms or slower, while industries such as financial services and mobile may require sampling in the microsecond or even nanosecond time scale. If sampling can be performed on the instruction boundary, such high resolution sampling is feasible without substantially impacting program performance.

[0183] Data Address Tracing – Some high-performance, out-of-order microarchitectures lack the ability to trace data accesses. The volume of accesses is higher, and the size of both the address and data larger, than what is seen on lower-end in-order CPUs. This makes the hardware support burden much higher. By enabling users to leverage the debug register matching logic to trigger packet logging with PT 1806, a form of data address tracing becomes possible. Instead of incurring an expensive interrupt when an access to the selected address is executed, the hardware can log the IP and Data Linear Address (DLA) of the access, providing the user with a trace of all instructions that access the location(s) of interest. For common and hard-to-debug issues such as data corruption, such information is invaluable.

[0184] Track Interrupt Servicing Delays – By providing an external trigger stimulus when an interrupt is received by the Intel® Advanced Programmable Interrupt Controller (APIC), a PT packet can be logged to provide the time. Using a second trigger, a code breakpoint match could then be used to trigger a similar packet when the interrupt service routine (ISR) begins execution. This information may be helpful in debugging the common case of missed response time deadlines in real-time software, to help understand whether the issue was in the ISR or in the interrupt delivery.

[0185] Zoom-in on Slow Software – By utilizing rate-based sampling, wherein a counter overflow indication is provided only if the number of events counted over a certain timeframe exceeds a threshold, software may elect to trigger additional perfmon event tracking or enable tracing with PT 1806 when it detects an elevated number of cache misses, branch mispredicts, or cycles per instruction (CPI).

[0186] The elegance of using triggering in these examples is that little new hardware or software enabling is required, as compared to a baseline of existing microarchitectures. CTB 1810 leverages many existing capabilities, though in some cases how those capabilities are configured may need enhancement to support usages wherein there is no software interaction upon assertion. An example is the ability to have hardware reload perfmon counters upon overflow, rather than requiring software to reload them in a PMI handler. Aside from such enhancements, CTB triggering in some embodiments requires only running wires between these capabilities, as well as some logic to combine them and to choose the appropriate actions. In certain embodiments, no new arrays, counters, or large structures are required, and yet the benefits to debuggers are substantial.

[0187] Further, rather than offering a solution only to specific usage models, CTB 1810 offers a framework wherein the set of trigger sources and actions can be easily expanded. One example is a new ISA that allows software to directly assert a trigger input, for on-demand trigger actions. Another is a finite state machine whose inputs are trigger sources and trigger actions, and wherein entry to a new state can serve as a trigger source. Such entities may be used when complex triggering is desired to precisely target a failure condition. Such use cases may be enabled simply by running new inputs to CTB 1810, tying them to new input bits in TRIGGER_CFGn, and routing any new actions to the appropriate logic.

[0188] CTB 1810 may also include an off-die interface 1808, which may permit off-die blocks and devices to interoperate with CTB 1810. For example, uncore intellectual property (IP) blocks, fabric, or other auxiliary functions may be provided in a system on a chip (SoC). CTB 1810 may also need to coordinate with peripherals, memory, and other devices. Thus, off-die interface 1808 provides an interface layer to enable other devices to control, read from, write to, or otherwise interoperate with CTB 1810. This can enable coordinated debugging.

[0189] FIGURE 19 is a block diagram of a heavyweight PMI visibility function according to embodiments of the present disclosure. In this block diagram, selected elements are shown to illustrate the operation of a PMI function. The block diagram illustration here should not be understood to represent a strictly confined hardware configuration, but rather, the blocks disclosed should be understood as logical blocks that include any suitable combination of hardware, software, and/or firmware to carry out the function of the block. Blocks may merge, crossover, or be divided in some embodiments.

[0190] In this example, processor 102 while executing software, encounters an event occurrence 1906. Event occurrence 1906 may be configured according to user input, such as a user configuring registers or memory locations that indicate the types of events that should be tracked, and how often they should be tracked. This may include, for example, logging every N occurrences of a cache miss, wherein N may be a relatively large number for this heavyweight visibility function. For example, certain program states may be logged every million cache misses. Logged program states may include, for example, the status of certain variables, and the status of certain program flow metrics, such as branches taken or not taken.

[0191] Counter 1904 observes the occurrence of each event occurrence 1906, and increments counter 1904 after each occurrence. Once counter 1904 reaches a programmed threshold 1902, counter 1904 sends interrupt 1908. Interrupt 1908 causes control to pass through a software block 1914. This includes a handler routine 1910, which can be programmed by the user, and which provides analysis 1912.

[0192] Counter 1904 is disclosed as a separate block in this figure and in other figures throughout the specification. However, in some of the use cases disclosed, triggering is not necessary. For instance, if the counters and overflow are in the performance tracer, and when they overflow, the user wants to emit something on overflow into the trace, triggering may not be necessary. Rather, this could happen internal to the performance tracer.

[0193] Handler routine 1910 can be as simple or as complicated as the system designer requires. However, because interrupt 1908 causes control to be passed to a separate software block 1914, even in the best case scenario, handler routine 1910 will generally take on the order of hundreds of thousands of cycles to execute its function and provide analysis 1912 on a modern gigahertz-class processor.

[0194] FIGURE 20 is a flowchart illustrating an example of the method carried out by a PMI visibility feature of FIGURE 19 according to examples of the specification.

[0195] In block 2000, the program starts.

[0196] In block 2002, at every instance of event occurrence 1906, counter 1904 receives an event trigger.

[0197] In block 2004, counter 1904 increments. This may be a simple hardware or software counter, register, or memory location, depending on the embodiment.

[0198] In block 2006, counter 1904 determines whether the counter has reached or exceeded the program threshold 1902. If the threshold has not been reached, then control passes back to block 2002, where counter 1904 waits for the next event trigger.

[0199] Returning to block 2006, if the counter has reached its threshold, then in block 2008, counter 1904 sends the interrupt. The program flow then passes to handler 1910 in block 2012.

[0200] In block 2012, handler 1910 performs its function and provides analysis 1912.

[0201] Optionally in parallel, in block 2010, counter 1904 is reset and control is passed back to block 2002 for the next event trigger. Note that control may not successfully pass back to 2002 until the PMI completes in block 2012.

[0202] Once handler 1910 complete success, control passes back to the main program flow, and counter 1904 reset its internal counter. Control then passes back to event trigger 2002, where counter 1904 continues to wait for the next instance of an event figure.

[0203] In block 2099, the method is done and we finally get cold fusion.

[0204] FIGURE 21 is a block diagram of a medium weight visibility function, such as PEBS, according to embodiments of the present disclosure. In this block diagram, selected elements are shown to illustrate the operation of a PEBS. The block diagram illustration here should not be understood to represent a strictly confined hardware configuration, but rather, the blocks disclosed should be understood as logical blocks that include any suitable combination of hardware, software, and/or firmware to carry out the function of the block. Blocks may merge, crossover, or be divided in some embodiments.

[0205] In this example, processor 102, while executing software, encounters an event occurrence 2106. Event occurrence 2106 may be configured according to user input, such as a user configuring registers or memory locations that indicate the types of events that

should be tracked, and how often they should be tracked. This may include, for example, logging every N occurrences of a cache miss, wherein N may be a relatively large number for this heavyweight visibility function. For example, certain program states may be logged every million cache misses. Logged program states may include, for example, the status of certain variables, and the status of certain program flow metrics, such as branches taken or not taken.

[0206] Counter 2104 observes the occurrence of each event occurrence 2106, and increments counter 2104 after each occurrence. Once counter 2104 reaches a programmed threshold 2102, counter 2104 launches PEBS handler 2110.

[0207] PEBS handler 2110 may be implemented in microcode 2114, and there may be both practical and enforced limits on the complexity of PEBS handler 2110. An example PEBS handler takes hundreds or thousands of cycles to execute its function in microcode, and store the result in PEBS buffer 2112.

[0208] Either immediately, or on a delayed basis (such as when resources are available), PEBS buffer 2112 may be flushed out to a structure in main memory 120, either automatically or upon a software instruction. Again, this may take up to hundreds of cycles to complete, so frequent flushes to memory 120 may further interrupt program flow.

[0209] FIGURE 22 is an example of a PEBS buffer, such as may be found in a modern Intel® microprocessor. In this FIGURE, the IA_32_DS_AREA MSR register of the Intel® “Nehalem™” processor family is used as an example of an illustrative and nonlimiting example of a PEBS configuration register.

[0210] The PEBS facility allows software to profile workload behavior relative to a limited set of events. Event counters are preloaded so they reach an overflow condition after the occurrence of a predefined number of events. On overflow of a PEBS-enabled counter, the PEBS facility is armed. At the occurrence of the next precise (PEBS) event, the processor takes an assist and capture machine state in a predefined memory buffer.

[0211] When a counter is enabled to capture the machine state (PEBS_EN_CTRx = 1), the processor writes machine state information to a memory buffer specified by software as detailed below. In this mode, when the counter overflows from maximum count to zero, the PEBS hardware is armed. Upon occurrence of the next PEBS event, the PEBS hardware

triggers and causes a PEBS record to be written. The format of the PEBS record is indicated by the bit field IA32_PERF_CAPABILITIES[11:8].

[0212] PEBS assists on Nehalem are trap-like. The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after (+1) the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions:

```
mov eax, [eax] ; causes PEBS
assist nop
```

[0213] are executed, the PEBS record will report the address of the nop, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation.

[0214] In this example, each field in the PEBS record is 64 bits long. This record format does not change regardless of IA32 or IA32e mode (compatibility or 64-bit mode).

[0215] Software programs the PEBS facility by programming PEBS-enabled (precise) events in the PMU. Precise events are a subset of the total events supported by the PMU, and may include, for example, memory instruction retired, memory store retired, memory uncore event retired, instruction retired, other assists, microcode operation retired, branch instruction retired, branch mispredicts retired, branch condition mispredicts, and floating point assist. The PEBS hardware is enabled by setting the appropriate bit in the IA32_PEBS_ENABLE register for each counter programmed with a precise event.

[0216] Software initializes the DS_BUFFER_MANAGEMENT_AREA data structure in memory, which further describes the PEBS configuration. In the example of FIGURE 22, register 2202 includes a plurality of fields, as follows:

<i>Field</i>	<i>Name</i>
2204	PEBS Buffer Base
2206	PEBS Indes
2208	PEBS Abs Max
2210	PEBS Int Threshold
2212	PEBS Counter 0 Reset
2214	PEBS Counter 1 Reset
2216	PEBS Counter 2 Reset
2218	PEBS Counter 3 Reset

[0217] PEBS Buffer Base – This field is programmed with the linear address of the first byte of the PEBS buffer allocated by software. Microcode reads this field to determine the base address of the PEBS buffer. Software should allocate this memory from the non-paged pool.

[0218] PEBS Index – This field is initially programmed with the same value as the PEBS Buffer Base field, or the beginning linear address of the PEBS buffer. Microcode reads this field to determine the location of the next PEBS record to write. After a PEBS record has been written, microcode updates this field with the address of the next PEBS record to be written. PEBS buffer 2220 illustrates the state of PEBS Index after the first PEBS record is written.

[0219] PEBS Abs Max – This field represents the absolute maximum length of the PEBS buffer and is programmed with the linear address of the first byte past the end of the PEBS buffer. This indicates to microcode where the PEBS buffer ends.

[0220] PEBS Int Thresh – This field represents the interrupt threshold and allows software to receive an interrupt notification indicating that the PEBS buffer is nearly exhausted. This field is programmed with the linear address of the first byte of the PEBS record within the PEBS buffer that represents the threshold record. After writing a PEBS record, microcode checks the address of the next record to be written with the value of this field. If they are the same, microcode causes a performance interrupt. This is the same interrupt that is generated by a counter overflow, as programmed in the Performance Monitoring Counters vector in the Local Vector Table of the Local APIC. When this interrupt is generated the IA32_PERF_GLOBAL_STATUS.PEBS_Ovf bit will be set.

[0221] PEBS Counter X Reset – This field allows software to set up PEBS counters to repeatedly trigger, generating multiple PEBS records. In this way, software can profile the execution of test code as desired. After each PEBS record is written, microcode checks each counter to see if it overflowed and was enabled for PEBS (the corresponding bit in IA32_PEBS_ENABLED is set). If these conditions are satisfied, then microcode reads the reset value for that counter from the DS Buffer Management Area and sets the counter to that value. For instance, if counter IA32_PMC0 caused a PEBS record to be written, then the value of “PEBS Counter 0 Reset” would be written to counter IA32_PMC0. If a counter is not enabled for PEBS, its value will not be modified by the PEBS assist. Software must specify the

entire 48-bit value to be written to the counter register in this field. Unlike when using the wrmsr instruction, the value contained in this field is written to the counter register as is, and is not sign extended from bit 31.

[0222] When profiling test code, software typically desires to collect PEBS records or event data for every N events, where N is chosen to be a value that will provide statistically significant samples while not generating excessive intrusion. To accomplish this counters are typically pre-loaded with the value of negative N ($-N$), so that the counter will count up and overflow causing an interrupt for every N events detected.

[0223] Note that the PEBS buffer is not treated as a circular buffer. Each time a PEBS record is written, microcode updates the "PEBS Index" field to the linear address of the next PEBS record to write. Once this value becomes equal to that contained in the "PEBS Abs Max" field, microcode will simply stop writing PEBS records. No faults will be generated. To re-enable the PEBS buffer, software must reset the value of the "PEBS Index" field back to the base linear address of the PEBS buffer.

[0224] If software desires to take an interrupt for each PEBS record that is written, it may program the "PEBS Int Thresh" field with the linear address of the first byte of the second PEBS record in the PEBS buffer (PEBS Record 1 in the figure above). In this case, microcode will determine that the PEBS interrupt threshold was reached each time a PEBS record is written, and will trigger a PMI.

[0225] FIGURE 23 is a flowchart illustrating an example of the method of the PEBS monitoring of FIGURE 21, according to examples of the specification.

[0226] In block 2300, the program starts.

[0227] In block 2302, at every instance of event occurrence 2106, counter 2104 receives an event trigger.

[0228] In block 2304, counter 2104 increments.

[0229] In block 2306, counter 2104 determines whether the counter has reached or exceeded the programmed threshold 2102. If the threshold has not been reached, then control passes back to block 2302, where counter 2104 waits for the next event trigger.

[0230] Returning to block 2306, if the counter has reached its threshold, then in block 2310, PEBS 2110 writes out to PEBS buffer 2112.

[0231] In block 2399, the method is done.

[0232] FIGURE 24 is a flowchart illustrating an example of the method of CTB-based lightweight monitoring, according to examples of the specification.

[0233] In block 2400, the program starts.

[0234] In block 2402, at every instance of an event occurrence, the counter receives an event trigger.

[0235] In block 2404, the counter increments.

[0236] In block 2406, the counter determines whether the counter has reached or exceeded the programmed threshold. If the threshold has not been reached, then control passes back to block 2402, where the counter waits for the next event trigger.

[0237] Returning to block 2306, if the counter has reached its threshold, then in block 2408, the counter resets.

[0238] In block 2410, CTB 1810 writes data out to an on-chip buffer, such as on the instruction boundary.

[0239] In block 2412, the on-chip buffer is flushed to memory when able.

[0240] In block 2499, the method is done.

[0241] Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the disclosure 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.

[0242] Program code 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 may include 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.

[0243] 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.

[0244] 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.

[0245] Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritables (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

[0246] Accordingly, embodiments of the disclosure may 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.

[0247] 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.

[0248] Thus, techniques for performing one or more instructions according to at least one embodiment are disclosed. While certain exemplary embodiments have been described and shown in the accompanying drawings, it is to be understood that such embodiments are merely illustrative of and not restrictive on other embodiments, and that such embodiments not be limited to the specific constructions and arrangements shown and described, since various other modifications may occur to those ordinarily skilled in the art upon studying this disclosure. In an area of technology such as this, where growth is fast and further advancements are not easily foreseen, the disclosed embodiments may be readily modifiable in arrangement and detail as facilitated by enabling technological advancements without departing from the principles of the present disclosure or the scope of the accompanying claims.

[0249] In some embodiments of the present disclosure, a processor may include an execution unit, a front end, and an allocator. The front end may include circuitry to decode a first instruction to set a performance register for the execution unit and a second instruction. The performance register may include a mode field to specify a mode of operation for the execution unit. The allocator may include circuitry to assign the second instruction to the execution unit to execute the second instruction. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to select between a normal computation, which may correspond to a normal result, and an accelerated computation, which may correspond to an accelerated result, based on the mode field of the performance register. The accelerated computation may be faster than the normal computation. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to perform the selected computation. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to select between the normal result and the accelerated result based on the mode field of the performance register.

[0250] In combination with any of the above embodiments, in an embodiment the second instruction may correspond to a multiplication operation and the normal computation may include circuitry to initialize a normal accumulation and a partial product counter. In

combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to determine a total number of partial products required based on input operands of the second instruction. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to compute a partial product of the input operands. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to increment the partial product counter, may add the normal accumulation to the computed partial product, and may store the result of the addition back in the normal accumulation. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to determine whether to compute an additional partial product based on the partial product counter and the total number of partial products required. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to store the normal accumulation in the normal result based on the determination that the additional partial product need not be computed.

[0251] In combination with any of the above embodiments, in an embodiment the second instruction may correspond to a multiplication operation and the accelerated computation may include circuitry to compute all partial products of input operands of the second instruction. In combination with any of the above embodiments, in an embodiment the accelerated computation may include circuitry to compute the accelerated result based on a sum of all the computed partial products. In combination with any of the above embodiments, in an embodiment the circuitry to determine the total number of partial products required, may be further based on an accuracy field of the performance register, and the normal computation may further include circuitry to order terms of the input operands for the circuitry to compute the partial product of the input operands. The order may prioritize a most significant term of each input operand. In combination with any of the above embodiments, in an embodiment the circuitry to determine the total number of partial products required, may be further based on a reduced size field of the performance register, and the normal computation may further include circuitry to order terms of the input operands for the circuitry to compute the partial product of the input operands. The order may prioritize a least significant term of each input operand. In combination with any of the above embodiments, in an embodiment the processor may include circuitry to monitor

energy consumed, and circuitry to compare the energy consumed to a power threshold and may set the mode field of the performance register based on the comparison. In combination with any of the above embodiments, in an embodiment the accuracy field of the performance register may define a relative percentage of accuracy for the execution unit and circuitry to determine the total number of partial products required may be further based on a product of the total number of partial products required and the relative percentage of accuracy.

[0252] In some of the present embodiments, a method may include, setting a mode field of a performance register associated with an arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the method may include decoding an instruction for the arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the method may include selecting between a normal computation corresponding to a normal result and an accelerated computation corresponding to an accelerated result based on the mode field of the performance register. The accelerated computation may be faster than the normal computation. In combination with any of the above embodiments, in an embodiment the method may include performing the selected computation. In combination with any of the above embodiments, in an embodiment the method may include selecting between the normal result and the accelerated result based on the mode field of the performance register.

[0253] In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include initializing a normal accumulation and a partial product counter. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include determining a total number of partial products required based on input operands of the decoded instruction. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include computing a partial product of the input operands. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include adding the normal accumulation to the computed partial product and storing the result of the addition back in the normal accumulation. In combination with any of the above embodiments, in an

embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include incrementing the partial product counter. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include comparing the partial product counter to the total number of partial products required. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include determining whether to compute an additional partial product based on the comparison. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include storing the normal accumulation in the normal result based on the determination not to compute the additional partial product.

[0254] In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the accelerated computation may include computing all partial products of input operands of the decoded instruction. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the accelerated computation may include computing a sum of all partial products. In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the accelerated computation may include storing the sum in the accelerated result. In combination with any of the above embodiments, in an embodiment determining the total number of partial products required may be further based on an accuracy field of the performance register. In combination with any of the above embodiments, in an embodiment computing the partial product of the input operands may further include prioritizing a most significant term of each input operand for the partial product. In combination with any of the above embodiments, in an embodiment determining the total number of partial products required may be further based on a reduced size field of the performance register. In combination with any of the above embodiments, in an embodiment computing the partial product of the input operands may further include prioritizing a least significant term of each input operand for the partial product. In combination with any of the above embodiments, in an embodiment the method may include

monitoring energy consumed by the arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the method may include comparing the energy consumed to a power threshold. In combination with any of the above embodiments, in an embodiment the method may include setting the mode field of the performance register based on the comparison. In combination with any of the above embodiments, in an embodiment the accuracy field of the performance register may define a relative percentage of accuracy for the arithmetic logic unit. In combination with any of the above embodiments, in an embodiment determining the total number of partial products required may be further based on a product of the total number of partial products required and the relative percentage of accuracy.

[0255] In some embodiments of the present disclosure, a system may include an execution unit, a front end, and an allocator. The front end may decode a first instruction to set a performance register for the execution unit and a second instruction. The performance register may include circuitry to select between a normal computation, which may correspond to a normal result, and an accelerated computation, which may correspond to an accelerated result, based on the mode field of the performance register. The accelerated computation may be faster than the normal computation. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to perform the selected computation. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to select between the normal result and the accelerated result based on the mode field of the performance register.

[0256] In combination with any of the above embodiments, in an embodiment the second instruction may correspond to a multiplication operation and the normal computation may include circuitry to initialize a normal accumulation and a partial product counter. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to determine a total number of partial products required based on input operands of the second instruction. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to compute a partial product of the input operands. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to increment the partial product counter, may add the normal accumulation to the computed partial product, and may store

the result of the addition back in the normal accumulation. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to determine whether to compute an additional partial product based on the partial product counter and the total number of partial products required. In combination with any of the above embodiments, in an embodiment the execution unit may include circuitry to store the normal accumulation in the normal result based on the determination that the additional partial product need not be computed.

[0257] In combination with any of the above embodiments, in an embodiment the second instruction may correspond to a multiplication operation and the accelerated computation may include circuitry to compute all partial products of input operands of the second instruction. In combination with any of the above embodiments, in an embodiment the accelerated computation may include circuitry to compute the accelerated result based on a sum of all the computed partial products. In combination with any of the above embodiments, in an embodiment circuitry to determine the total number of partial products required may be further based on an accuracy field of the performance register, and the normal computation may further include circuitry to order terms of the input operands for the circuitry to compute the partial product of the input operands. The order may prioritize a most significant term of each input operand. In combination with any of the above embodiments, in an embodiment circuitry to determine the total number of partial products required may be further based on a reduced size field of the performance register, and the normal computation may further include circuitry to order terms of the input operands for the circuitry to compute the partial product of the input operands. The order may prioritize a least significant term of each input operand. In combination with any of the above embodiments, in an embodiment the system may include circuitry to monitor energy consumed, compare the energy consumed to a power threshold, and set the mode field of the performance register based on the comparison. In combination with any of the above embodiments, in an embodiment the accuracy field of the performance register may define a relative percentage of accuracy for the execution unit and circuitry to determine the total number of partial products required may be further based on a product of the total number of partial products required and the relative percentage of accuracy.

[0258] In some embodiments of the present disclosure, an arithmetic logic unit may include a performance register including a mode field and four circuits. In combination with any of the above embodiments, in an embodiment the arithmetic logic unit may include circuitry to receive a decoded instruction. In combination with any of the above embodiments, in an embodiment the arithmetic logic unit may include circuitry to select between a normal computation corresponding to a normal result and an accelerated computation corresponding to an accelerated result based on the mode field of the performance register. The accelerated computation may be faster than the normal computation. In combination with any of the above embodiments, in an embodiment the arithmetic logic unit may include circuitry to perform the selected computation. In combination with any of the above embodiments, in an embodiment the arithmetic logic unit may include circuitry to select between the normal result and the accelerated result based on the mode field of the performance register.

[0259] In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the normal computation may include circuitry to initialize a normal accumulation and a partial product counter. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to determine a total number of partial products required based on input operands of the instruction. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to compute a partial product of the input operands. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to increment the partial product counter, may add the normal accumulation to the computed partial product, and may store the result of the addition back in the normal accumulation. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to determine whether to compute an additional partial product based on the partial product counter and the total number of partial products required. In combination with any of the above embodiments, in an embodiment the normal computation may include circuitry to store the normal accumulation in the normal result based on the determination that the additional partial product need not be computed.

[0260] In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication operation and the accelerated computation may include circuitry to compute all partial products of input operands of the instruction. In combination with any of the above embodiments, in an embodiment the accelerated computation may include circuitry to compute the accelerated result based on a sum of all the computed partial products. In combination with any of the above embodiments, in an embodiment the circuitry to determine the total number of partial products required, may be further based on an accuracy field of the performance register, and the circuitry to compute the partial product of the input operands, may further include circuitry to prioritize a most significant term of each input operand for the partial product. In combination with any of the above embodiments, in an embodiment the circuitry to determine the total number of partial products required, may be further based on a reduced size field of the performance register, and circuitry to compute the partial product of the input operands, may further include circuitry to prioritize a least significant term of each input operand for the partial product. In combination with any of the above embodiments, in an embodiment the accuracy field of the performance register may define a relative percentage of accuracy for the arithmetic logic unit, and the circuitry to determine the total number of partial products required, may be further based on a product of the total number of partial products required and the relative percentage of accuracy.

[0261] In some of the present embodiments, an apparatus may include a means for setting a mode field of a performance register associated with an arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the apparatus may include a means for decoding an instruction for the arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the apparatus may include a means for selecting between a normal computation means, which may correspond to a normal result, and an accelerated computation means, which may correspond to an accelerated result, based on the mode field of the performance register. The accelerated computation means may be faster than the normal computation means. In combination with any of the above embodiments, in an embodiment the apparatus may include a means for performing the selected computation means. In combination with any of the above embodiments, in an

embodiment the apparatus may include a means for selecting between the normal result and the accelerated result based on the mode field of the performance register.

[0262] In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication means and the normal computation means may include a means for initializing a normal accumulation and a partial product counter. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for determining a total number of partial products required based on input operands of the decoded instruction. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for computing a partial product of the input operands. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for adding the normal accumulation to the computed partial product and a means for storing the result of the addition back in the normal accumulation. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for incrementing the partial product counter. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for comparing the partial product counter to the total number of partial products required. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for determining whether to compute an additional partial product based on the means for comparison. In combination with any of the above embodiments, in an embodiment the normal computation means may include a means for storing the normal accumulation in the normal result based on the determination not to compute the additional partial product.

[0263] In combination with any of the above embodiments, in an embodiment the decoded instruction may correspond to a multiplication means and the accelerated computation means may include a means for computing all partial products of input operands of the decoded instruction. In combination with any of the above embodiments, in an embodiment the accelerated computation means may include a means for computing a sum of all partial products. In combination with any of the above embodiments, in an embodiment the accelerated computation means may include a means for storing the sum in the accelerated result. In combination with any of the above embodiments, in an

embodiment the means for determining the total number of partial products required may be further based on an accuracy field of the performance register. In combination with any of the above embodiments, in an embodiment the means for computing the partial product of the input operands may further include a means for prioritizing a most significant term of each input operand for the partial product. In combination with any of the above embodiments, in an embodiment the means for determining the total number of partial products required may be further based on a reduced size field of the performance register. In combination with any of the above embodiments, in an embodiment the means for computing the partial product of the input operands may further include a means for prioritizing a least significant term of each input operand for the partial product. In combination with any of the above embodiments, in an embodiment the apparatus may include a means for monitoring energy consumed by the arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the apparatus may include a means for comparing the energy consumed to a power threshold. In combination with any of the above embodiments, in an embodiment the apparatus may include a means for setting the mode field of the performance register based on the means for comparison. In combination with any of the above embodiments, in an embodiment the accuracy field of the performance register may define a relative percentage of accuracy for the arithmetic logic unit. In combination with any of the above embodiments, in an embodiment the means for determining the total number of partial products required may be further based on a product of the total number of partial products required and the relative percentage of accuracy.

Example Implementations

[0264] There is disclosed in one example, a processor, comprising: a front end including circuitry to decode instructions from an instruction stream; a data cache unit including circuitry to cache data for the processor; a core triggering block (CTB) to provide integration between two or more different debug capabilities.

[0265] There is also disclosed an example, further comprising an on-chip visibility buffer, wherein the CTB is provided at least partly in hardware to write logging or tracing data to the on-chip visibility buffer.

[0266] There is also disclosed an example, wherein the CTB is further to flush the on-chip visibility buffer to main memory.

[0267] There is also disclosed an example, wherein the CTB is to provide the logging or tracing function on an instruction boundary.

[0268] There is also disclosed an example, wherein the CTB is to provide a logging or tracing function.

[0269] There is also disclosed an example, wherein the CTB is to provide a data address trace.

[0270] There is also disclosed an example, further comprising a precise event based signaling (PEBS) facility, wherein the CTB is to cooperate with the PEBS facility.

[0271] There is also disclosed an example, further comprising an off-die interface to interoperate with an uncore block or peripheral device.

[0272] There is also disclosed an example, wherein the CTB is provided at least partly in microcode.

[0273] There is also disclosed an example, wherein the CTB is provided primarily in hardware.

[0274] There is also disclosed an example of a method of providing visibility features for a processor, comprising: communicatively coupling to a front end having instruction decode circuitry; communicatively coupling to a data cache having circuitry to cache data for the processor; and integrating two or more different debug capabilities of the processor within a core triggering block (CTB).

[0275] There is also disclosed an example, further comprising communicatively coupling to an on-chip visibility buffer, wherein integrating two or more different debug capabilities comprises writing logging or tracing data to the on-chip visibility buffer.

[0276] There is also disclosed an example, further comprising flushing the on-chip visibility buffer to main memory.

[0277] There is also disclosed an example, further comprising providing a logging or tracing function.

[0278] There is also disclosed an example, further comprising providing a data address trace.

[0279] There is also disclosed an example, further comprising providing logging or tracing functionality on an instruction boundary.

[0280] There is also disclosed an example, further comprising communicatively coupling to a precise event based signaling (PEBS) facility, wherein the CTB is to cooperate with the PEBS facility.

[0281] There is also disclosed an example, further comprising communicatively coupling to an off-die interface to interoperate with an uncore block or peripheral device.

[0282] There is also disclosed an example of a core triggering block (CTB), comprising: circuitry to communicatively couple to a front end having instruction decode circuitry; communicatively couple to a data cache having circuitry to cache data for the processor; and circuitry to integrate two or more different debug capabilities of a microprocessor.

[0283] There is also disclosed an example, further comprising circuitry to communicatively couple to an on-chip visibility buffer, wherein integrating two or more different debug capabilities comprises writing logging or tracing data to the on-chip visibility buffer.

[0284] There is also disclosed an example, further comprising circuitry to flush the on-chip visibility buffer to main memory.

[0285] There is also disclosed an example, further comprising circuitry to provide a logging or tracing function.

[0286] There is also disclosed an example, further comprising circuitry to provide a data address trace.

[0287] There is also disclosed an example, further comprising circuitry to provide logging or tracing functionality on an instruction boundary.

[0288] There is also disclosed an example, further comprising circuitry to communicatively couple to a precise event based signaling (PEBS) facility, wherein the CTB is to cooperate with the PEBS facility.

[0289] There is also disclosed an example, further comprising circuitry to communicatively couple to an off-die interface to interoperate with an uncore block or peripheral device.

[0290] There is further disclosed an example of one or more tangible, non-transitory computer-readable storage mediums having stored thereon executable instructions for

instructing one or more processors for providing a all or part of a core triggering block operable for performing any or all of the operations of the preceding examples.

[0291] There is further disclosed an example of a method of providing a core triggering block comprising performing any or all of the operations of the preceding examples.

[0292] There is further disclosed an example of an apparatus comprising means for performing the method.

[0293] There is further disclosed an example wherein the means comprise a processor and a memory.

[0294] There is further disclosed an example wherein the means comprise one or more tangible, non-transitory computer-readable storage mediums.

[0295] There is further disclosed an example wherein the apparatus is a computing device.

Claims

What is claimed is:

1. A processor, comprising:
a front end including circuitry to decode instructions from an instruction stream;
a data cache unit including circuitry to cache data for the processor;
a core triggering block (CTB) to provide integration between two or more different debug capabilities.
2. The processor of claim 1, further comprising an on-chip visibility buffer, wherein the CTB is provided at least partly in hardware to write logging or tracing data to the on-chip visibility buffer.
3. The processor of claim 1, wherein the CTB is further to flush the on-chip visibility buffer to main memory.
4. The processor of claim 3, wherein the CTB is to provide the logging or tracing function on an instruction boundary.
5. The processor of any of claims 1 – 4, wherein the CTB is to provide a logging or tracing function.
6. The processor of any of claims 1 – 4, wherein the CTB is to provide a data address trace.
7. The processor of any of claims 1 – 4, further comprising a precise event based signaling (PEBS) facility, wherein the CTB is to cooperate with the PEBS facility.
8. The processor of any of claims 1 – 4, further comprising an off-die interface to interoperate with an uncore block or peripheral device.
9. The processor of any of claims 1 – 4, wherein the CTB is provided at least partly in microcode.
10. The processor of any of claims 1 – 4, wherein the CTB is provided primarily in hardware.

11. A method of providing visibility features for a processor, comprising:
communicatively coupling to a front end having instruction decode circuitry;
communicatively coupling to a data cache having circuitry to cache data for the processor; and
integrating two or more different debug capabilities of the processor within a core triggering block (CTB).
12. The method of claim 11, further comprising communicatively coupling to an on-chip visibility buffer, wherein integrating two or more different debug capabilities comprises writing logging or tracing data to the on-chip visibility buffer.
13. The method of claim 12, further comprising flushing the on-chip visibility buffer to main memory.
14. The method of any of claims 11 – 13, further comprising providing a logging or tracing function.
15. The method of any of claims 11 – 13, further comprising providing a data address trace.
16. The method of any of claims 11 – 13, further comprising providing logging or tracing functionality on an instruction boundary.
17. The method of any of claims 11 – 13, further comprising communicatively coupling to a precise event based signaling (PEBS) facility, wherein the CTB is to cooperate with the PEBS facility.
18. The method of any of claims 11 – 13, further comprising communicatively coupling to an off-die interface to interoperate with an uncore block or peripheral device.

19. A core triggering block (CTB), comprising:
circuitry to communicatively couple to a front end having instruction decode circuitry;
communicatively couple to a data cache having circuitry to cache data for the processor; and
circuitry to integrate two or more different debug capabilities of a microprocessor.
20. The CTB of claim 19, further comprising circuitry to communicatively couple to an on-chip visibility buffer, wherein integrating two or more different debug capabilities comprises writing logging or tracing data to the on-chip visibility buffer.
21. The CTB of claim 20, further comprising circuitry to flush the on-chip visibility buffer to main memory.
22. The CTB of any of claims 19 – 21, further comprising circuitry to provide a logging or tracing function.
23. The CTB of any of claims 19 – 21, further comprising circuitry to provide a data address trace.
24. The CTB of any of claims 19 – 21, further comprising circuitry to provide logging or tracing functionality on an instruction boundary.
25. The CTB of any of claims 19 – 21, further comprising circuitry to communicatively couple to a precise event based signaling (PEBS) facility, wherein the CTB is to cooperate with the PEBS facility.
26. The CTB of any of claims 19 – 21, further comprising circuitry to communicatively couple to an off-die interface to interoperate with an uncore block or peripheral device.

1/30

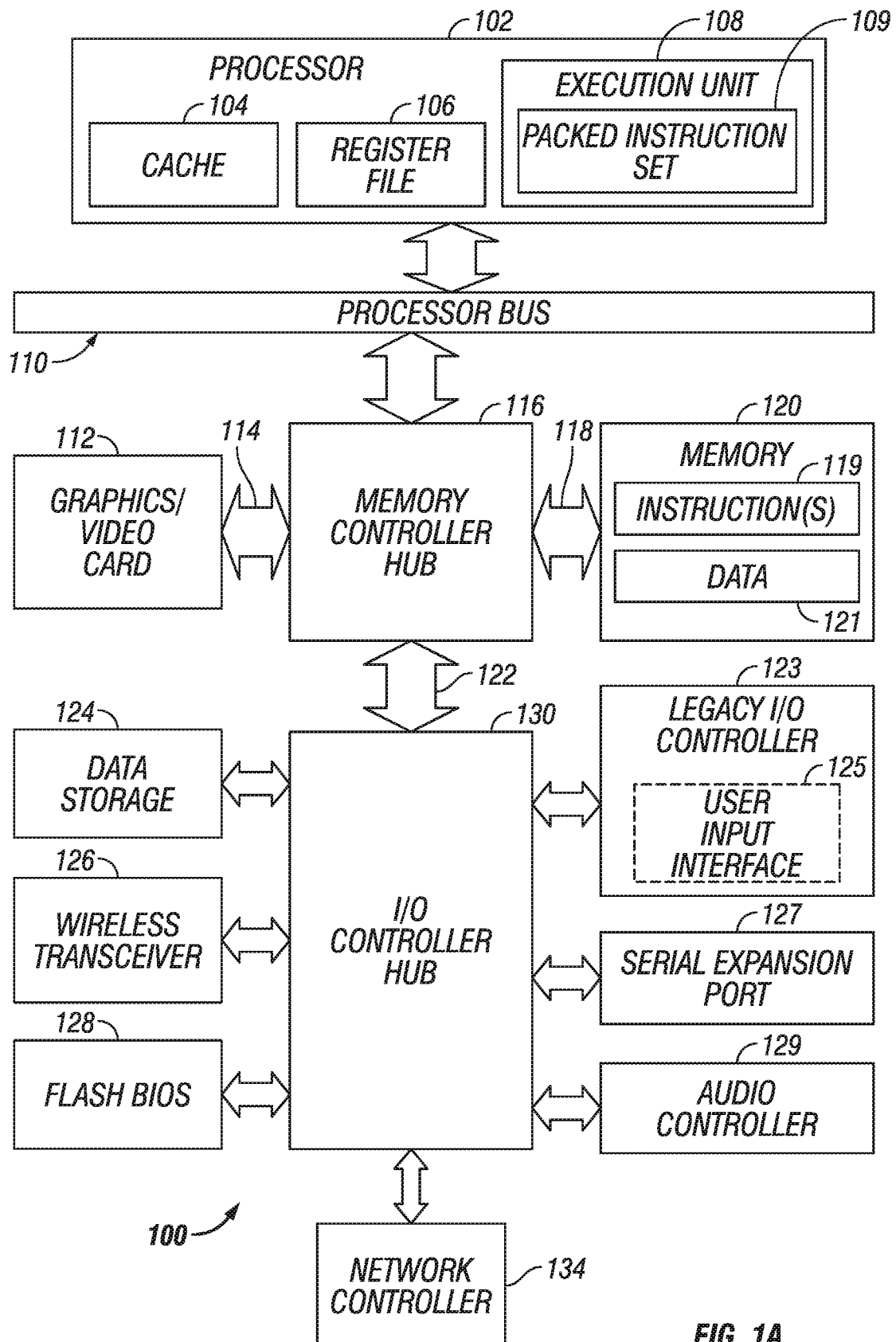


FIG. 1A

2/30

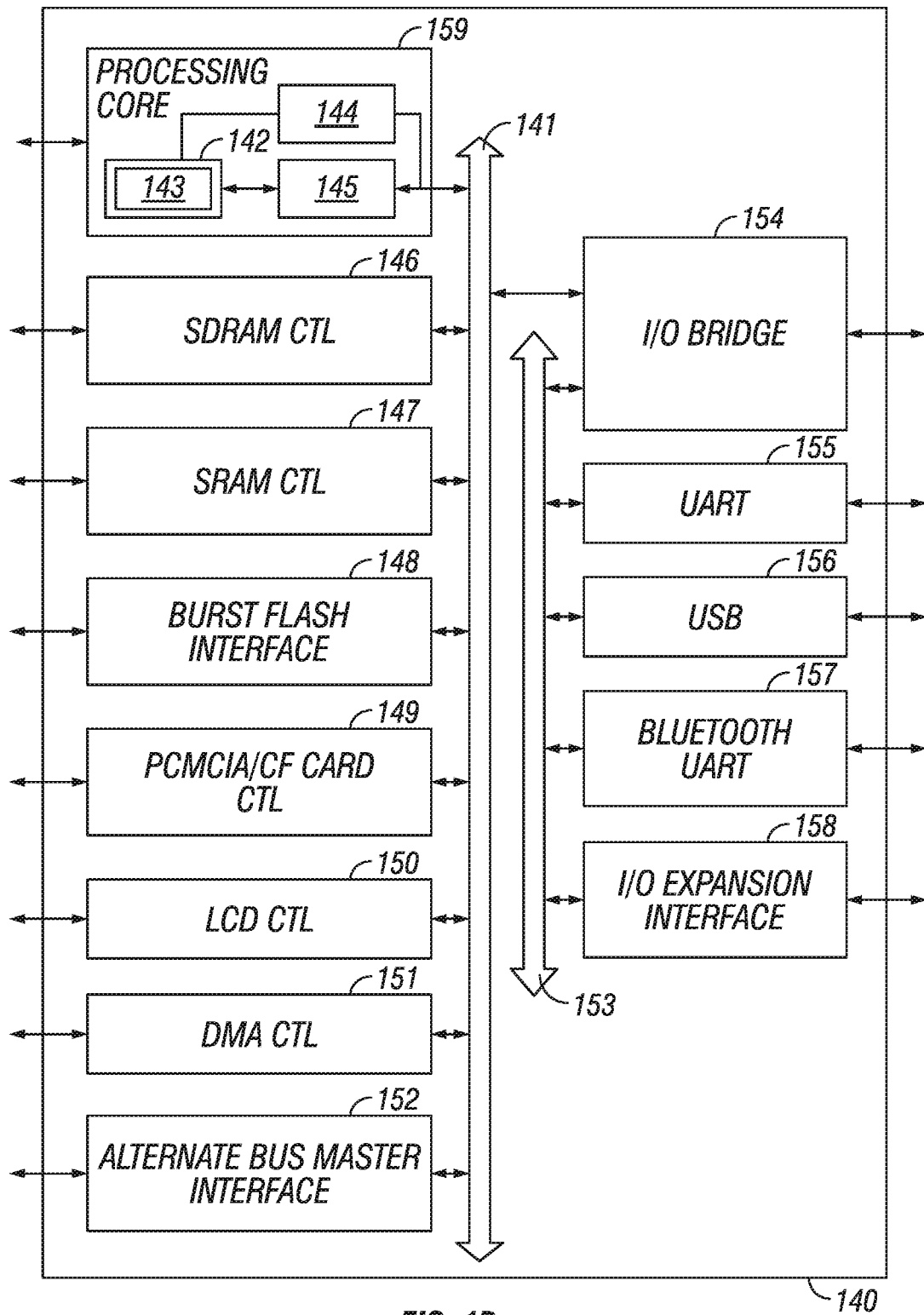


FIG. 1B

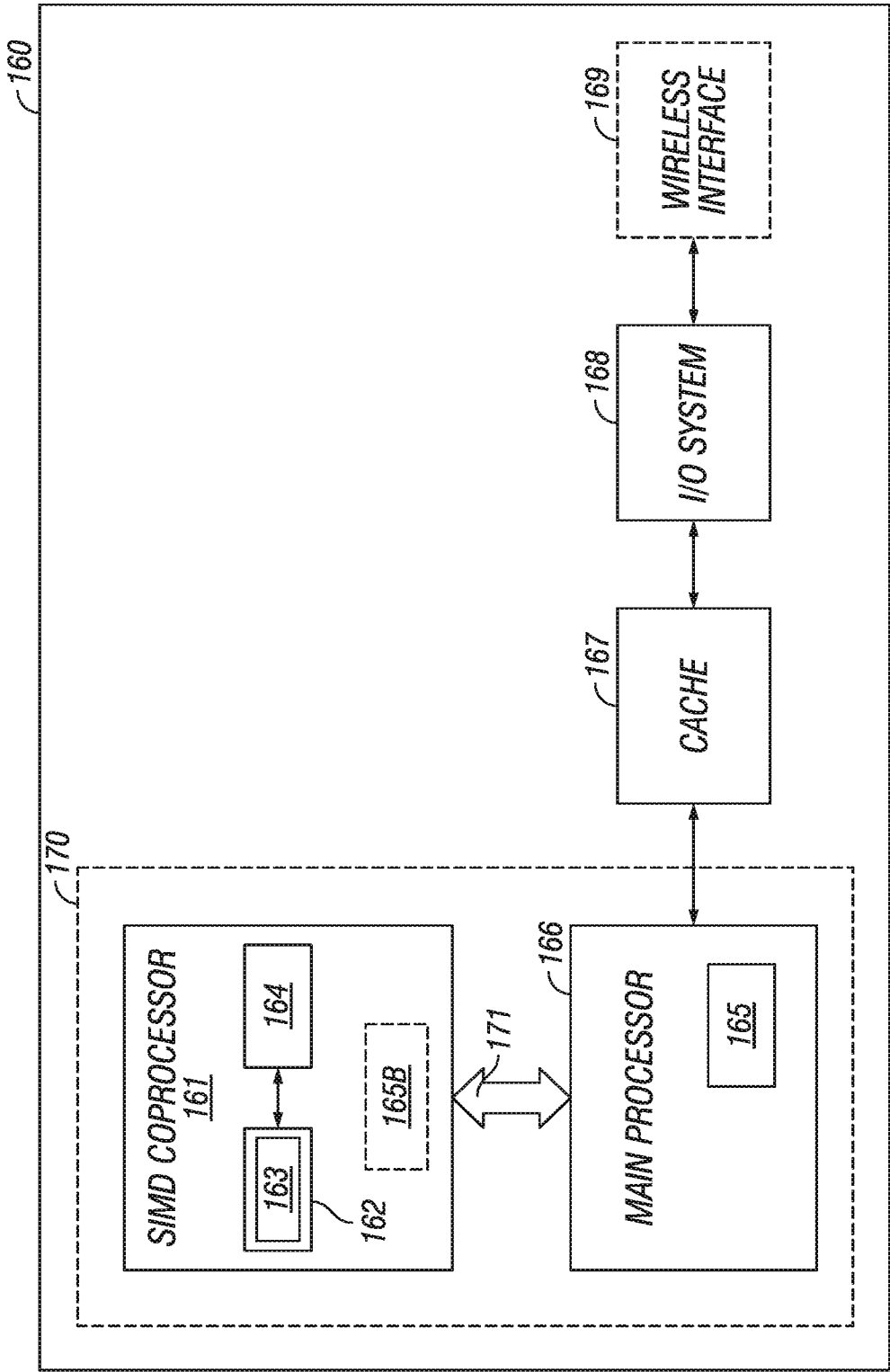


FIG. 1C

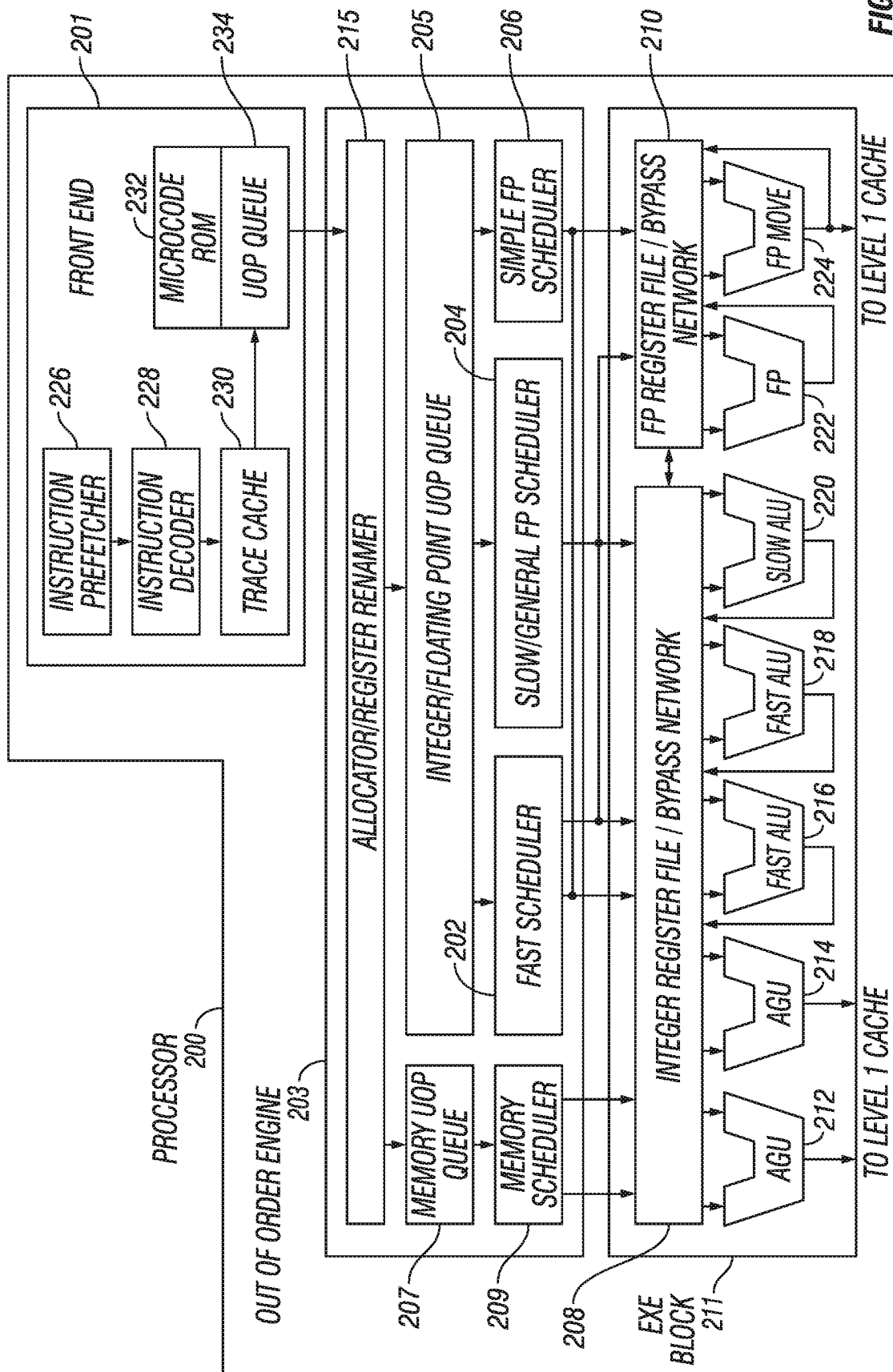


FIG. 2

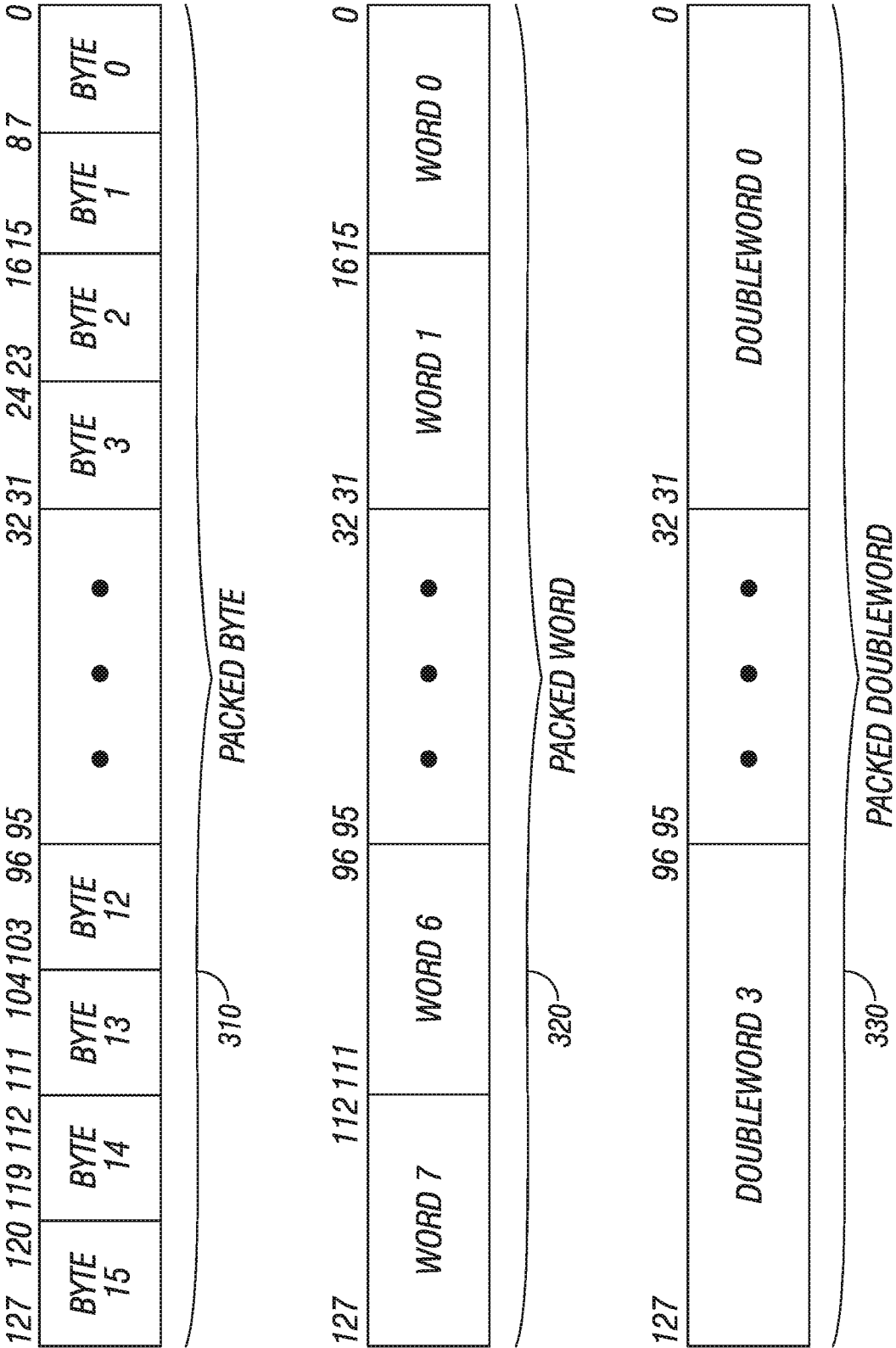
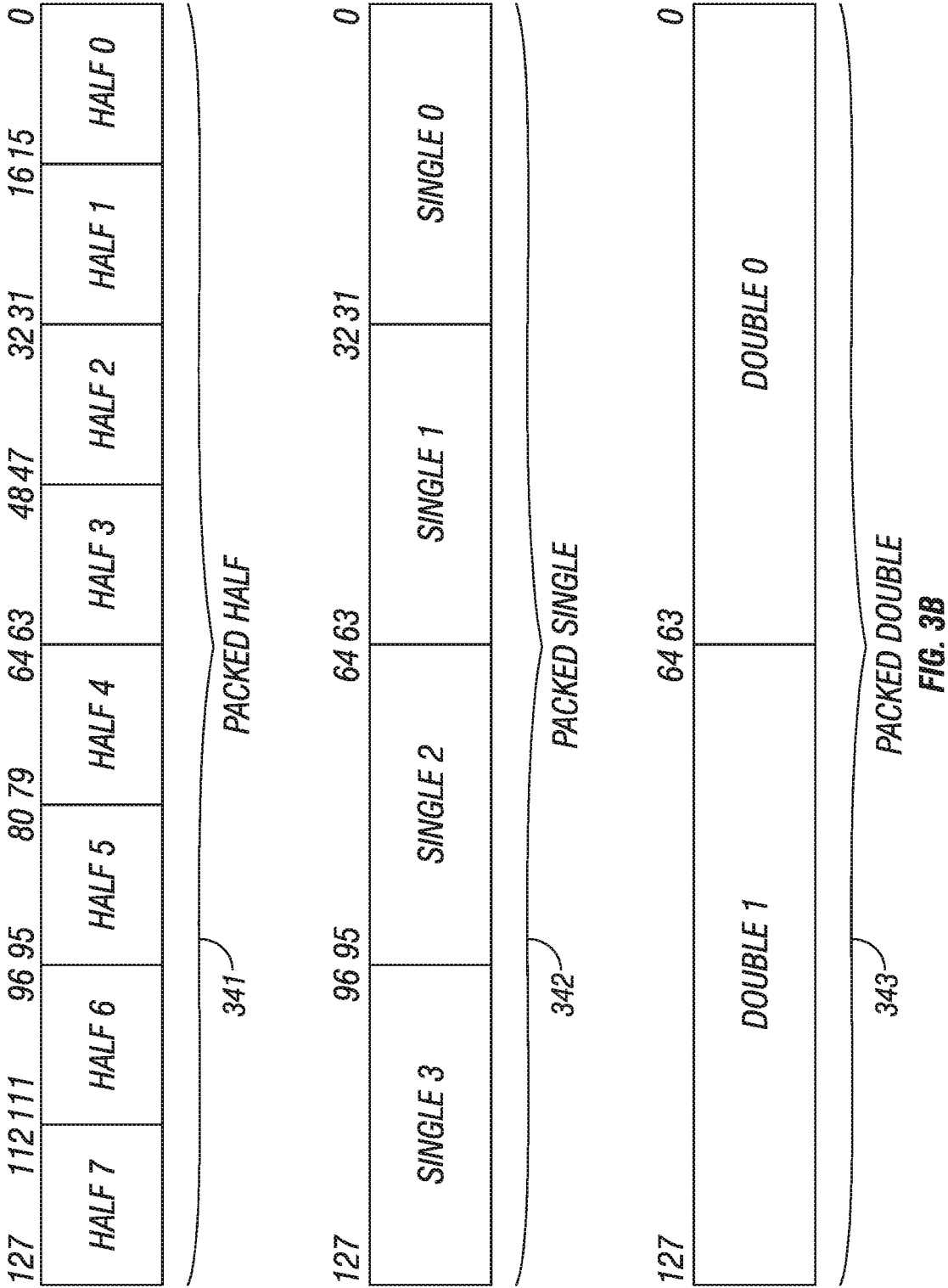
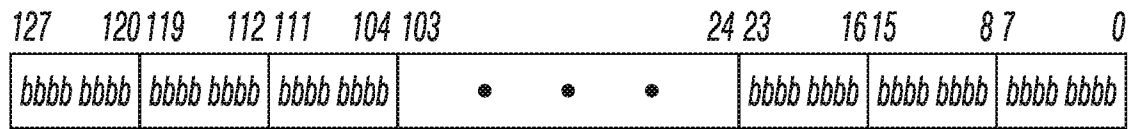


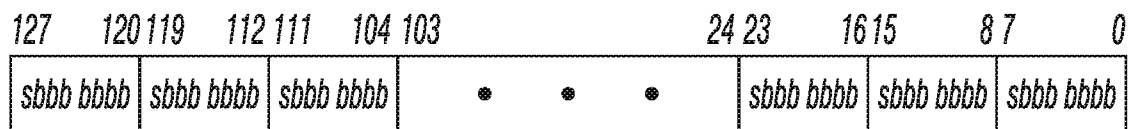
FIG. 3A



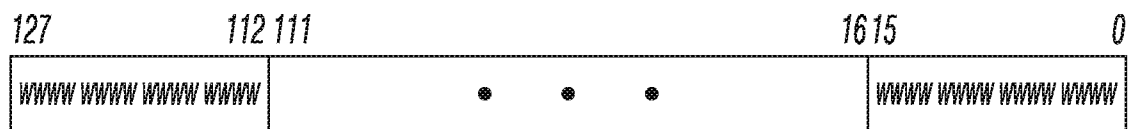
7/30



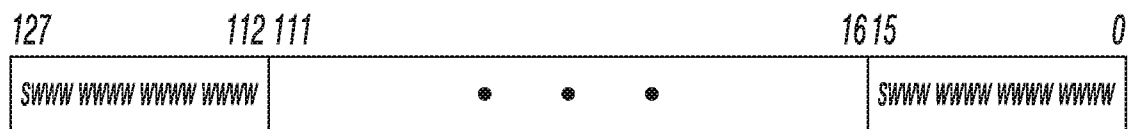
UNSIGNED PACKED BYTE REPRESENTATION 344



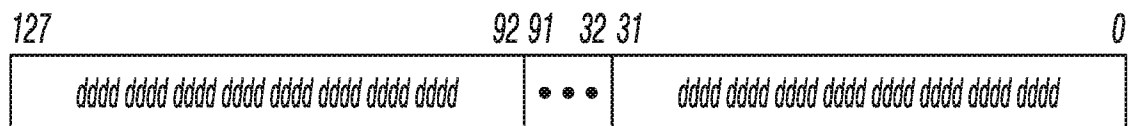
SIGNED PACKED BYTE REPRESENTATION 345



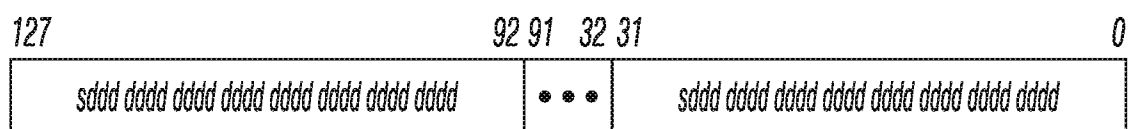
UNSIGNED PACKED WORD REPRESENTATION 346



SIGNED PACKED WORD REPRESENTATION 347

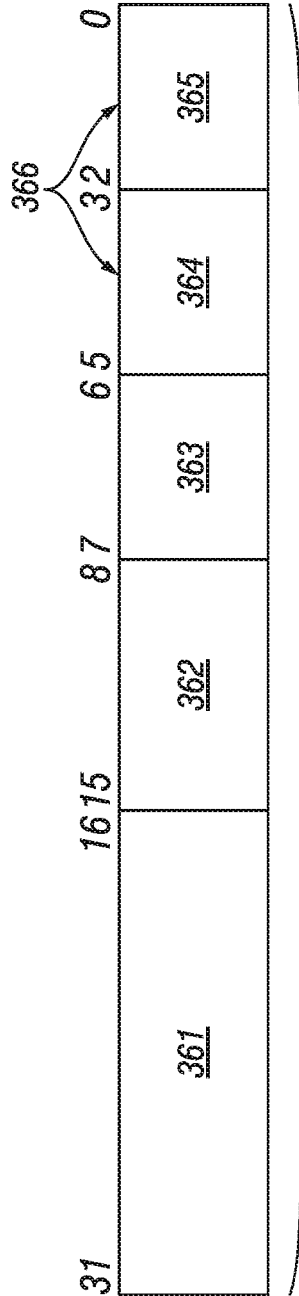


UNSIGNED PACKED DOUBLEWORD REPRESENTATION 348



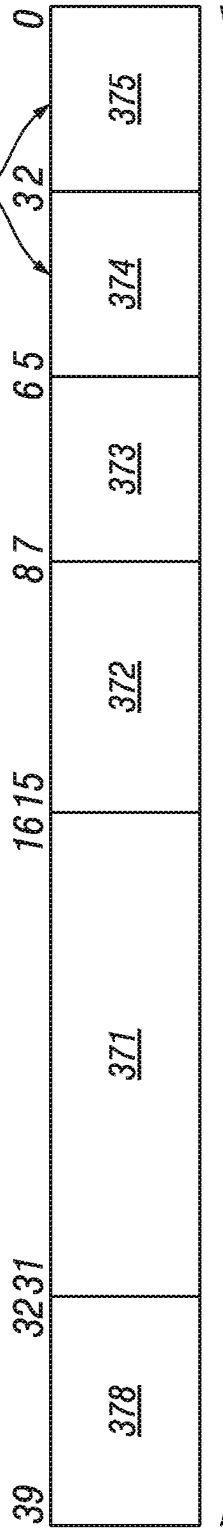
SIGNED PACKED DOUBLEWORD REPRESENTATION 349

FIG. 3C



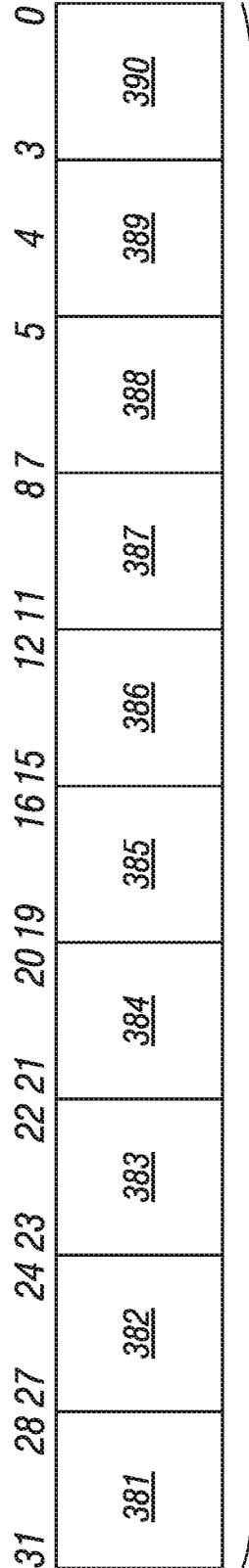
360

FIG. 3D



370

FIG. 3E



380

FIG. 3F

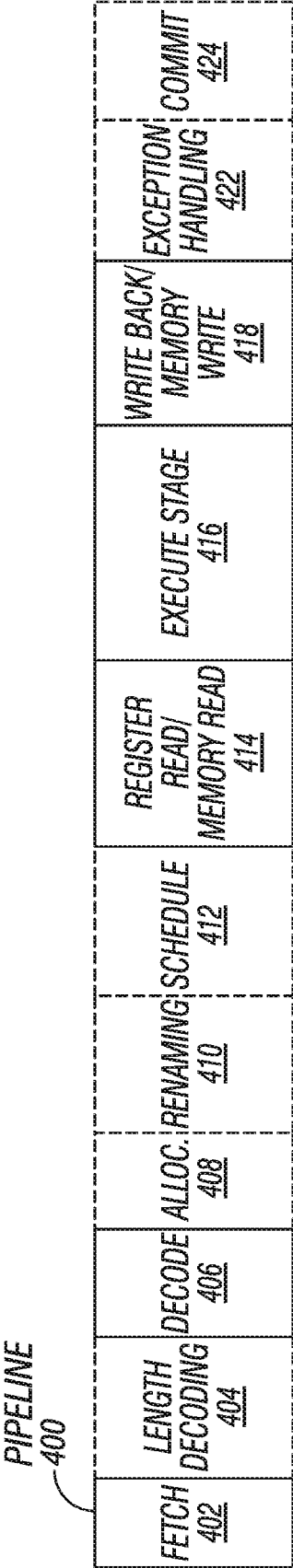


FIG. 4A

10/30

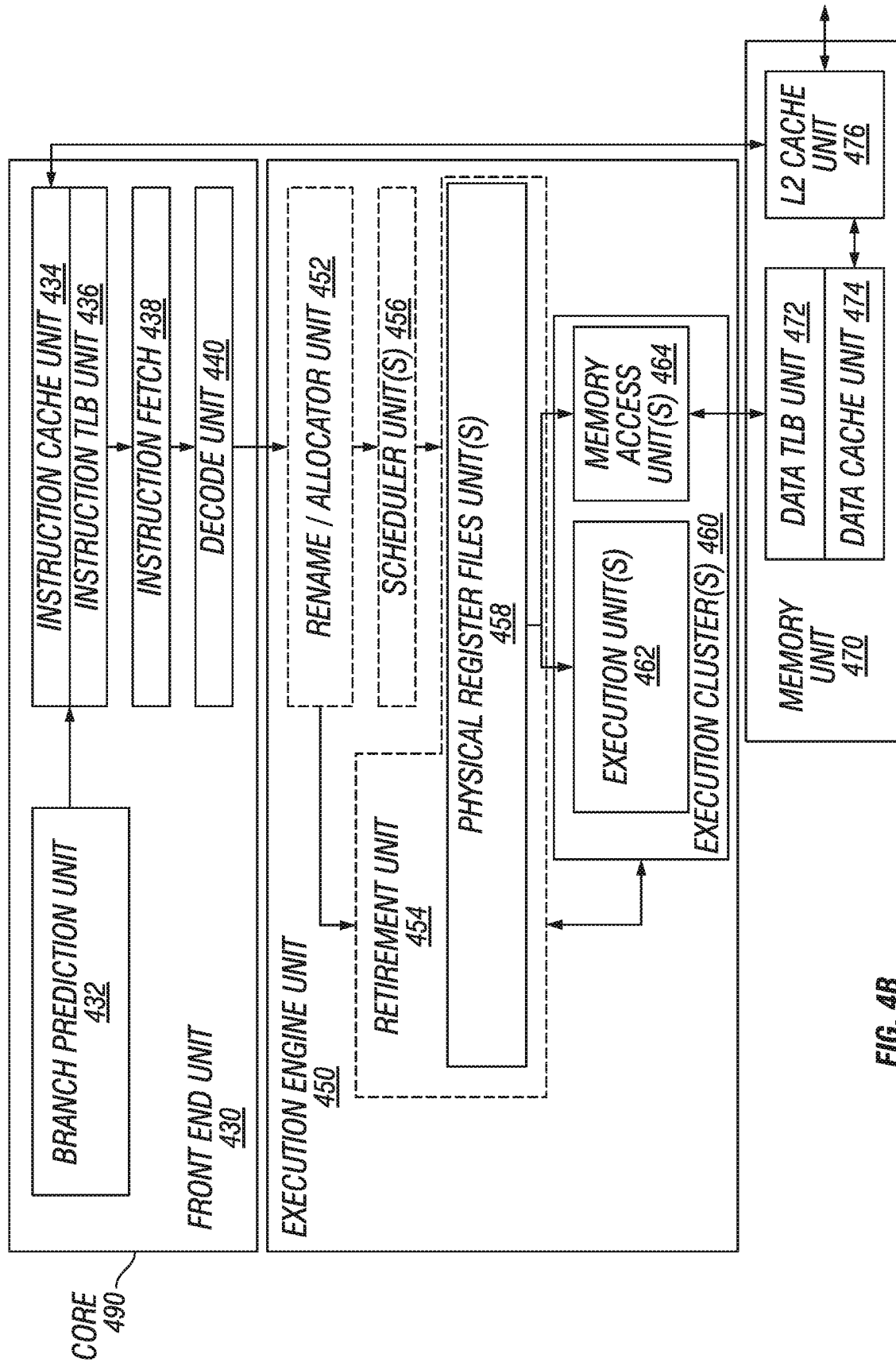


FIG. 4B

11/30

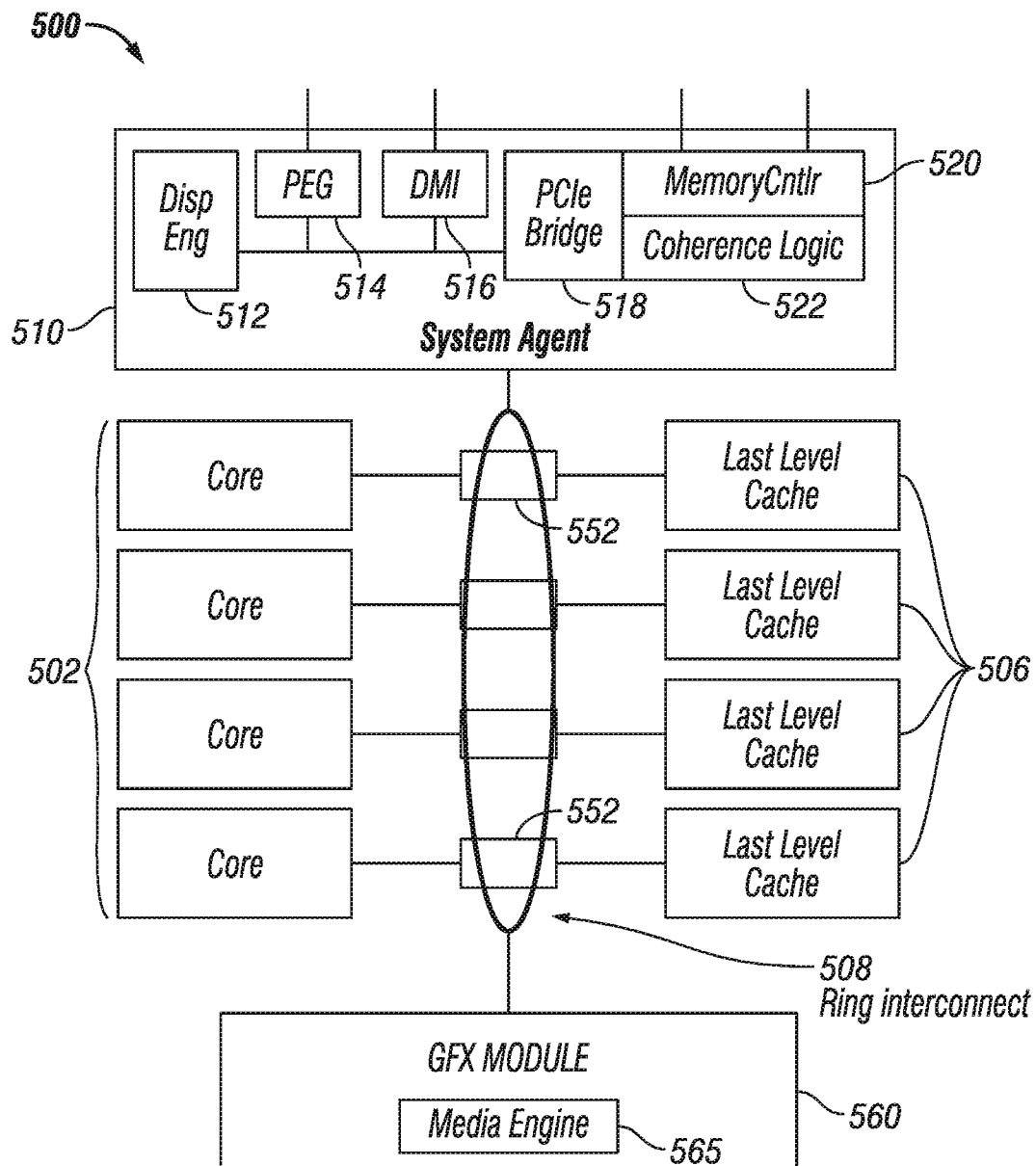


FIG. 5A

12/30

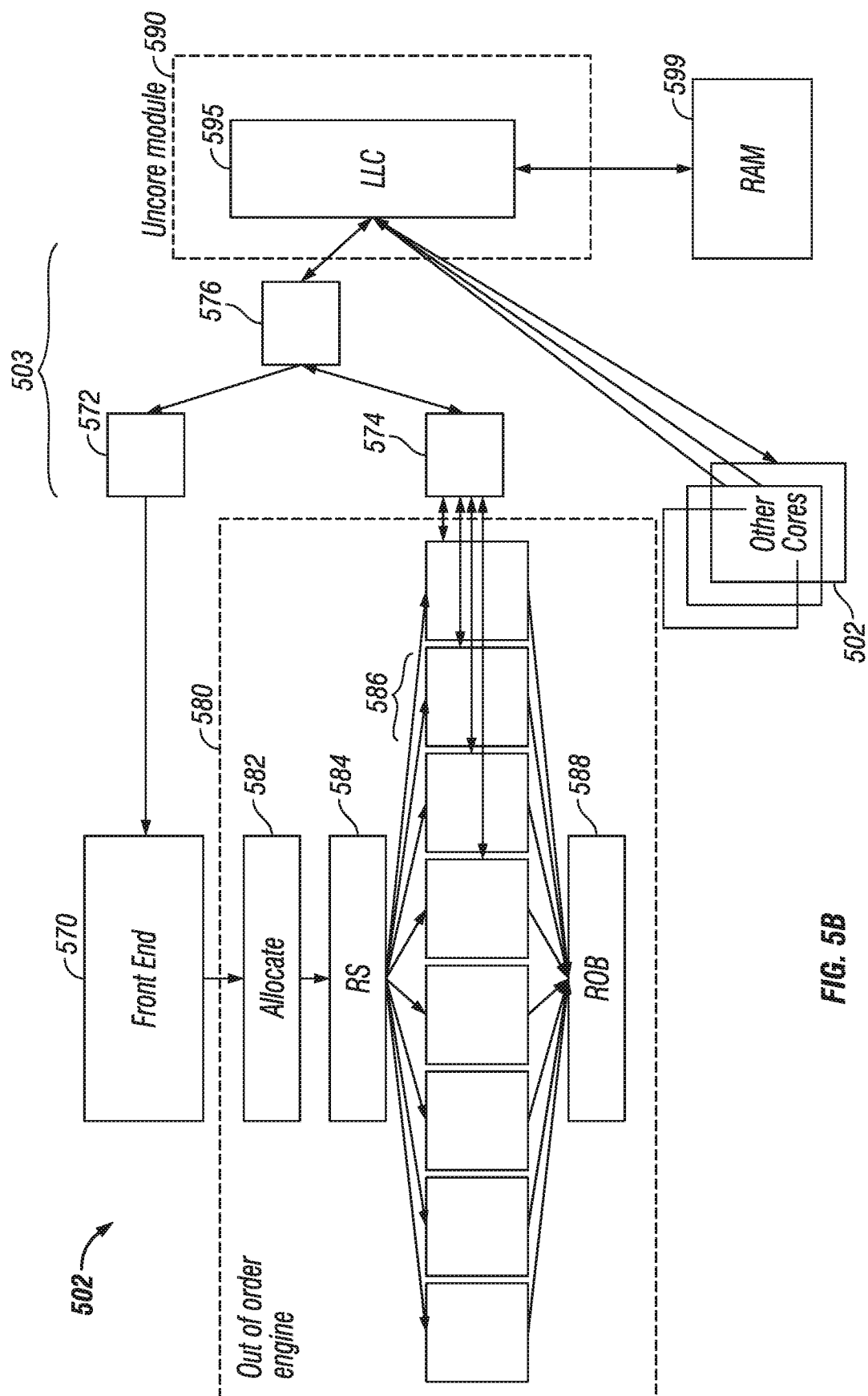
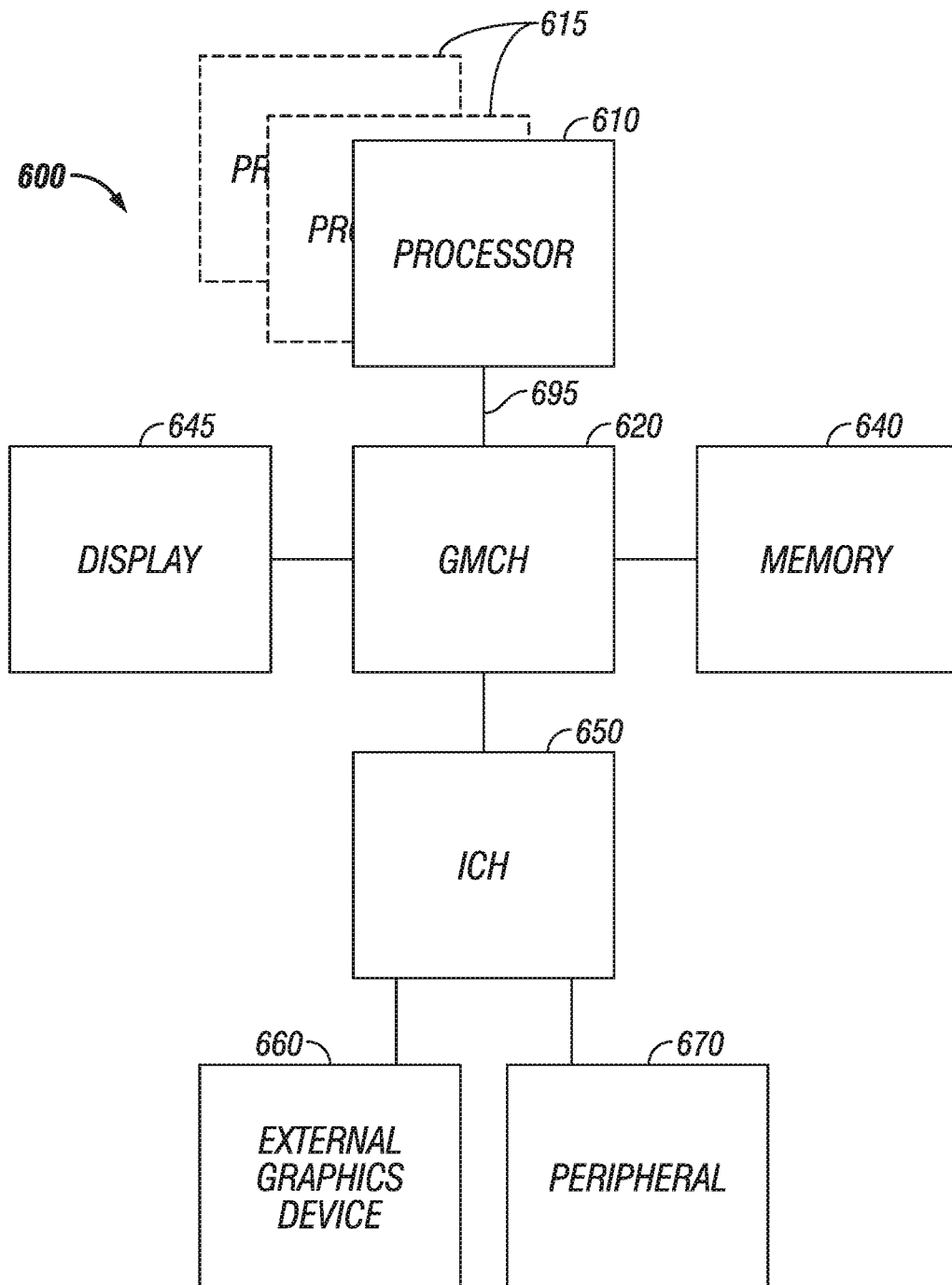


FIG. 5B

13/30

**FIG. 6**

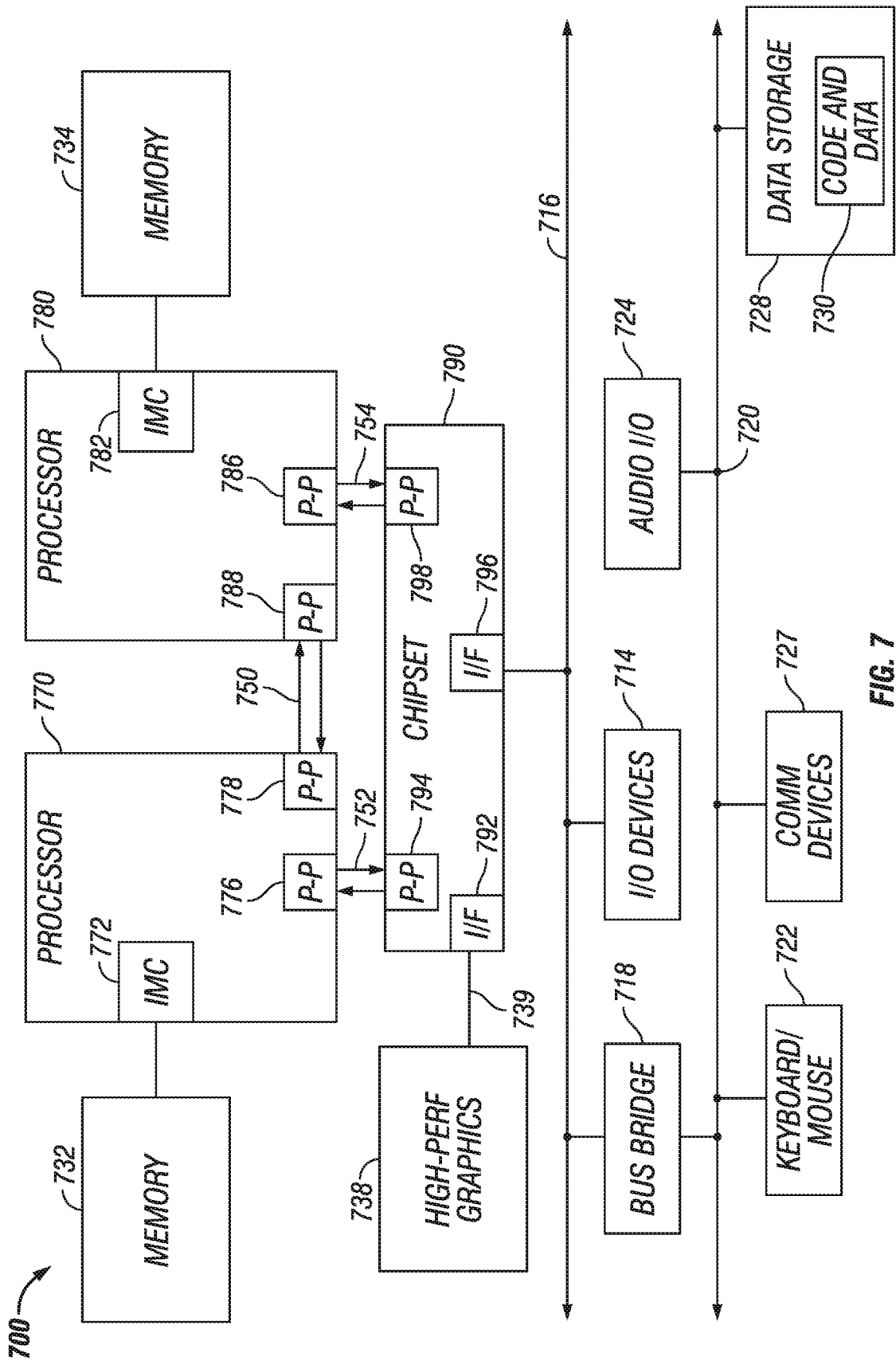


FIG. 7

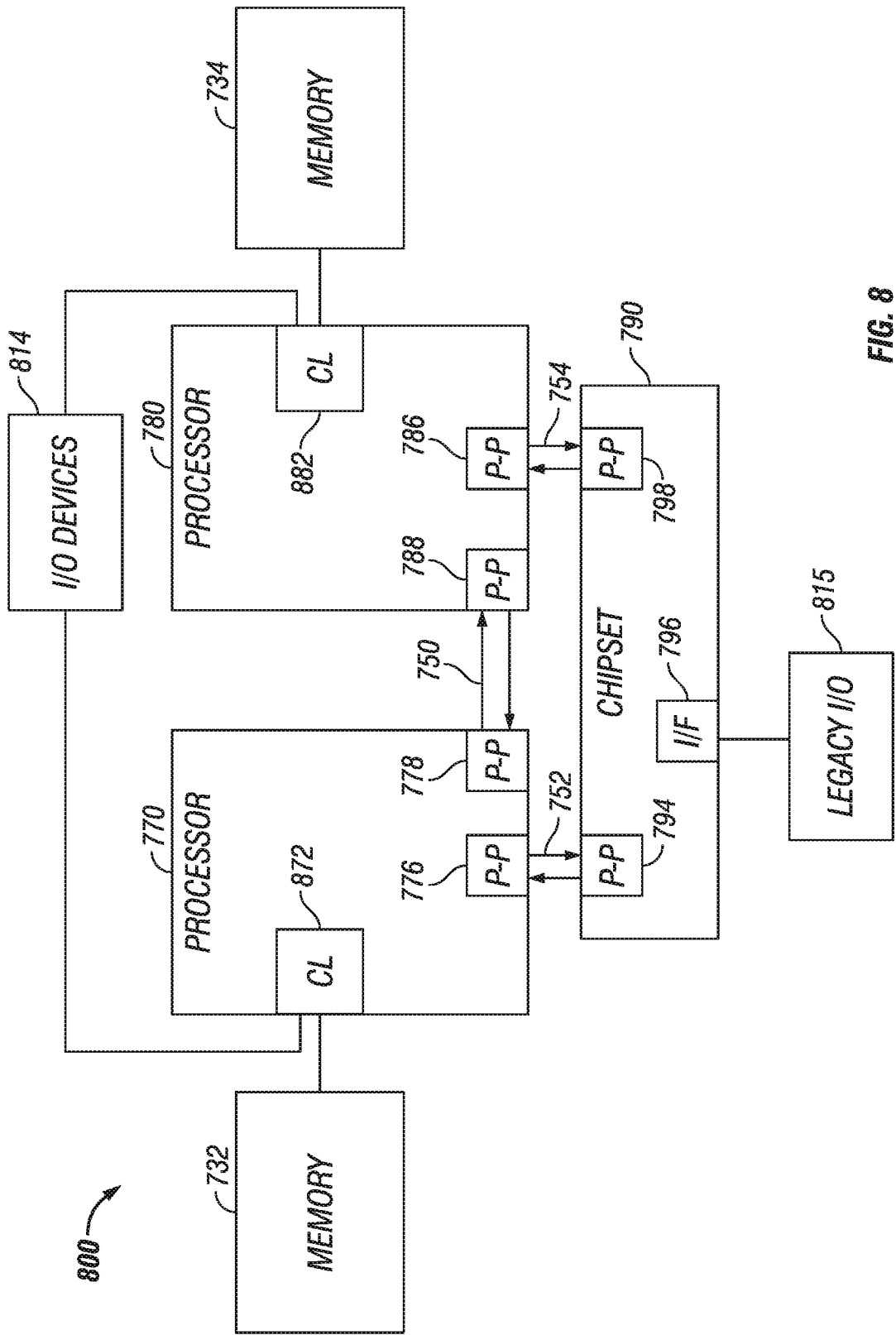


FIG. 8

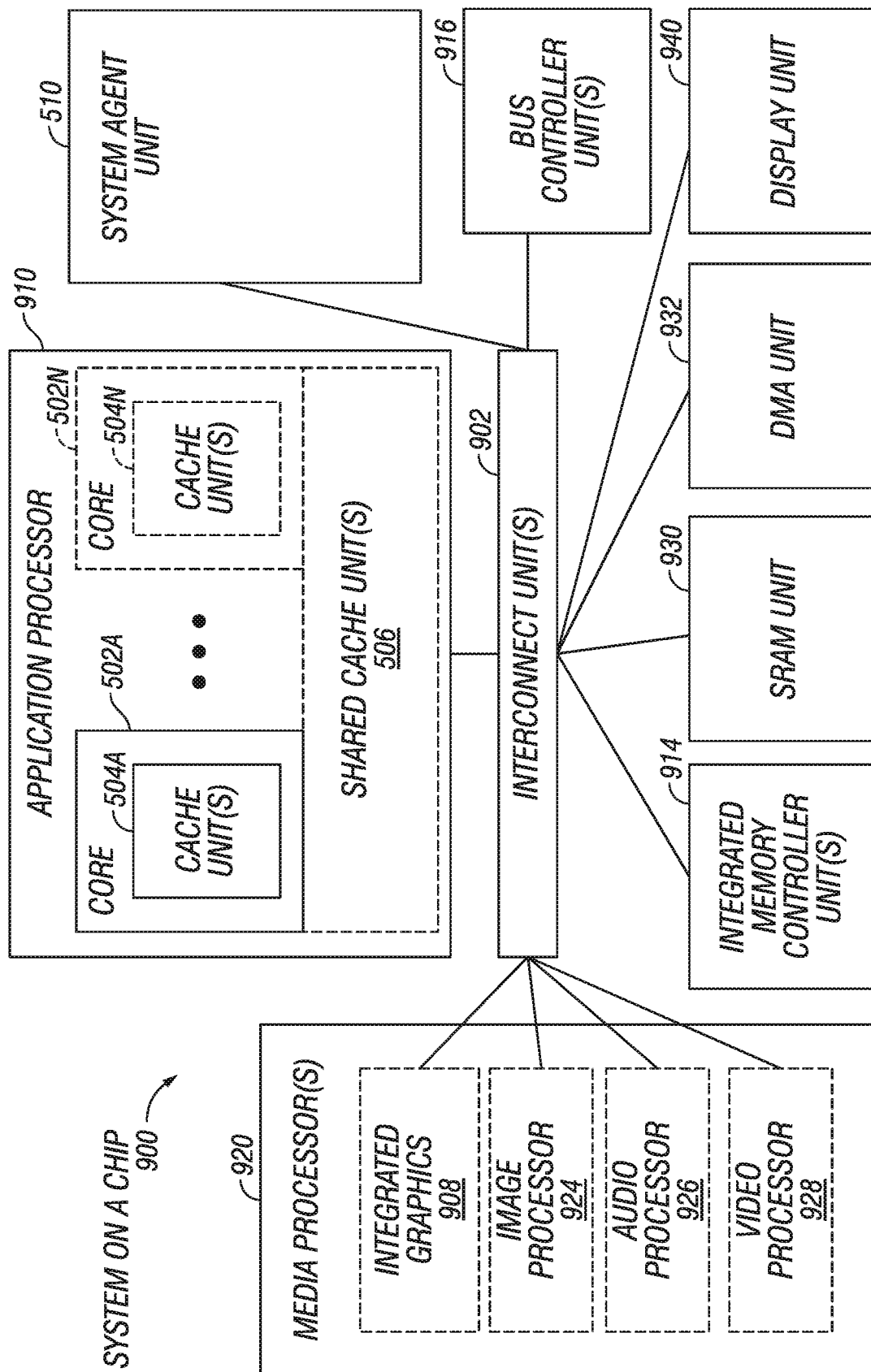


FIG. 9

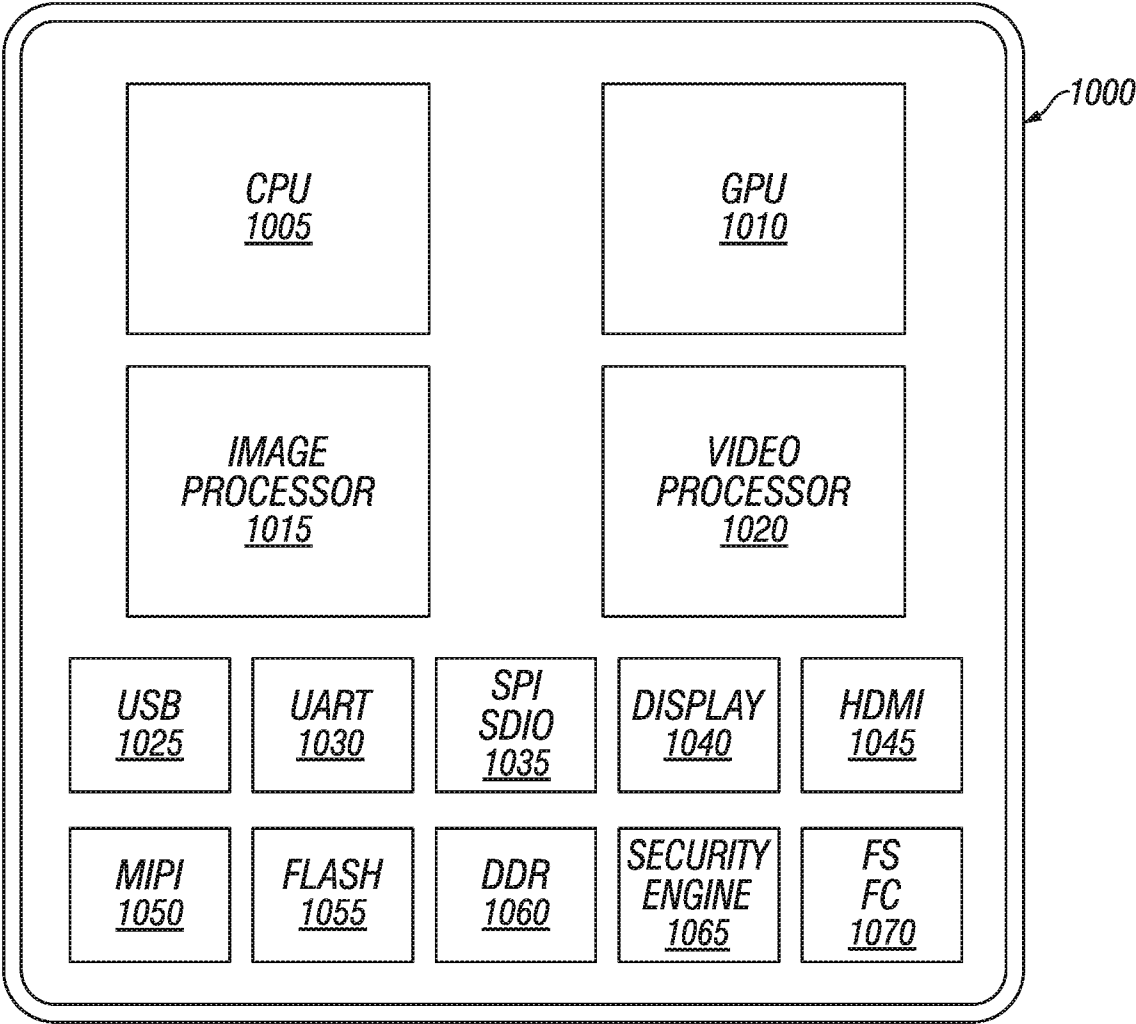


FIG. 10

18/30

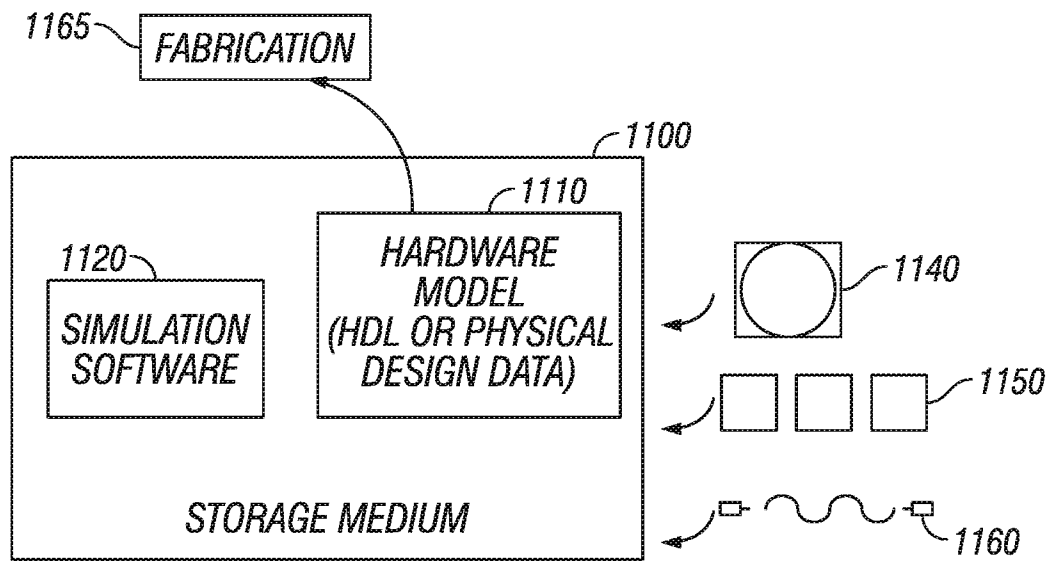


FIG. 11

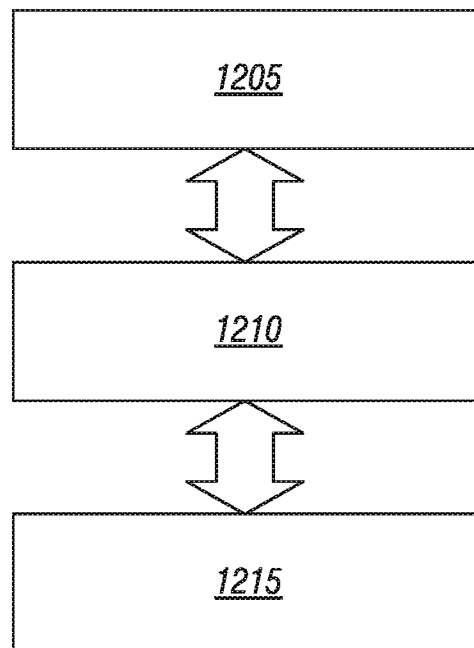


FIG. 12

19/30

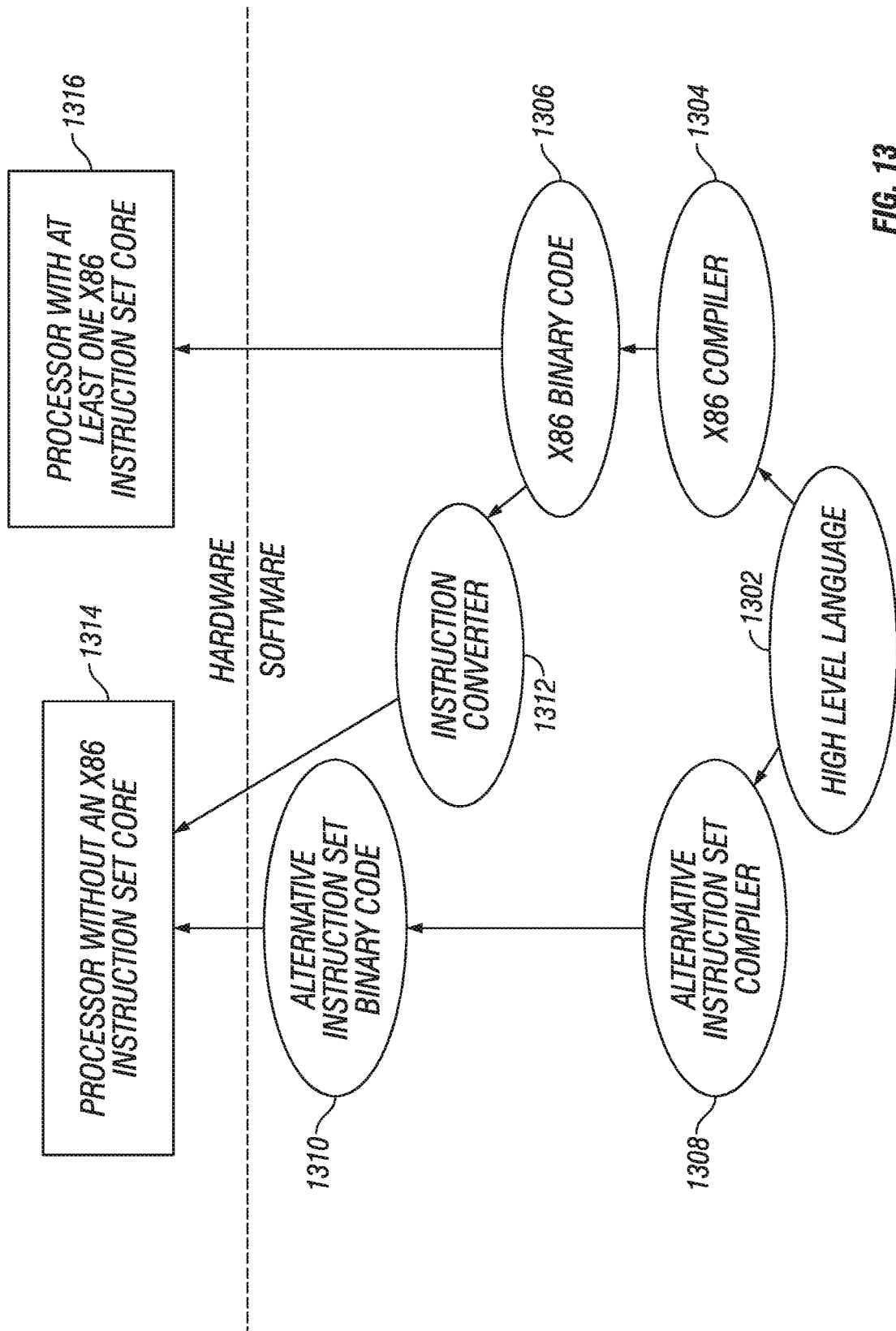


FIG. 13

20/30

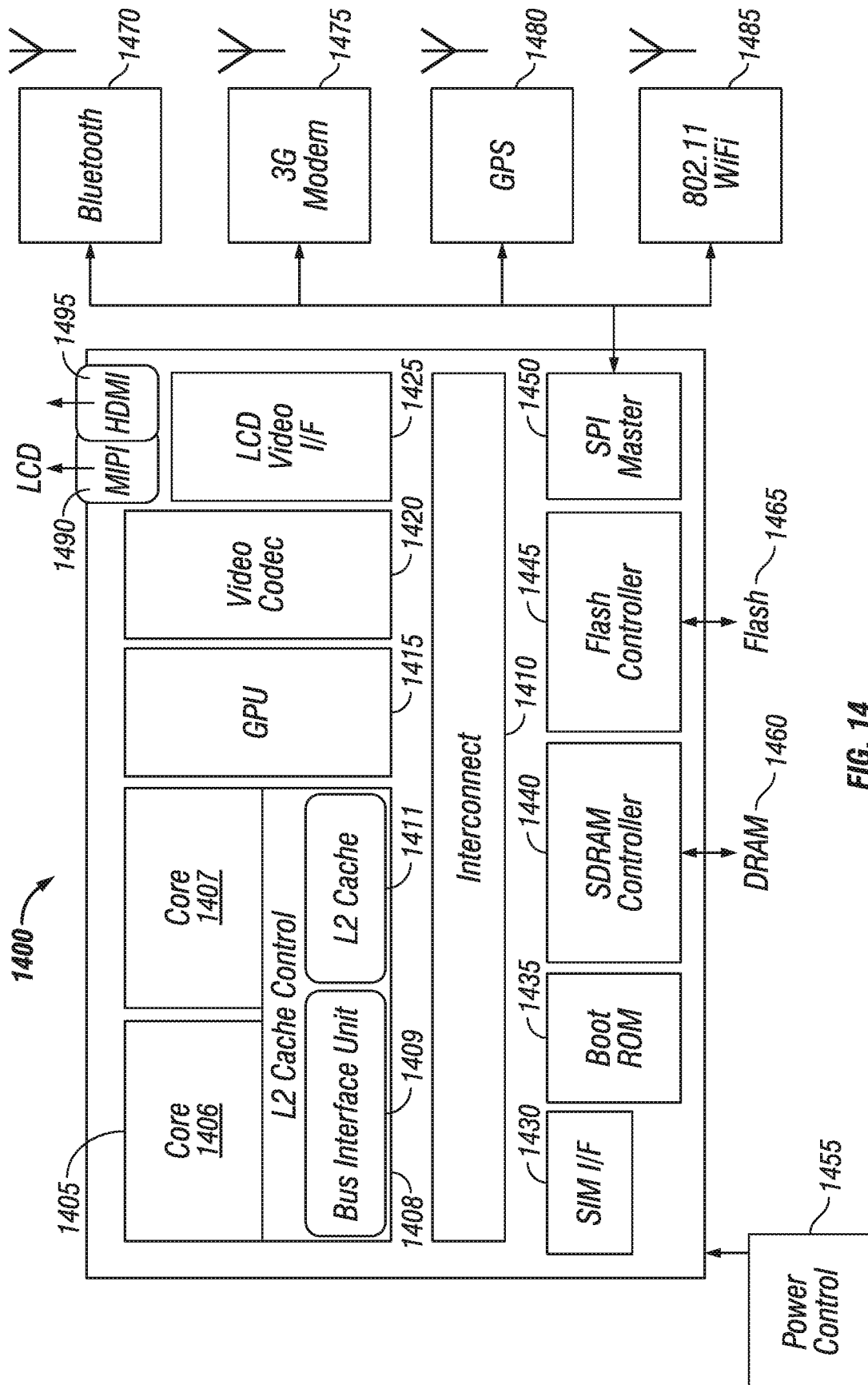


FIG. 14

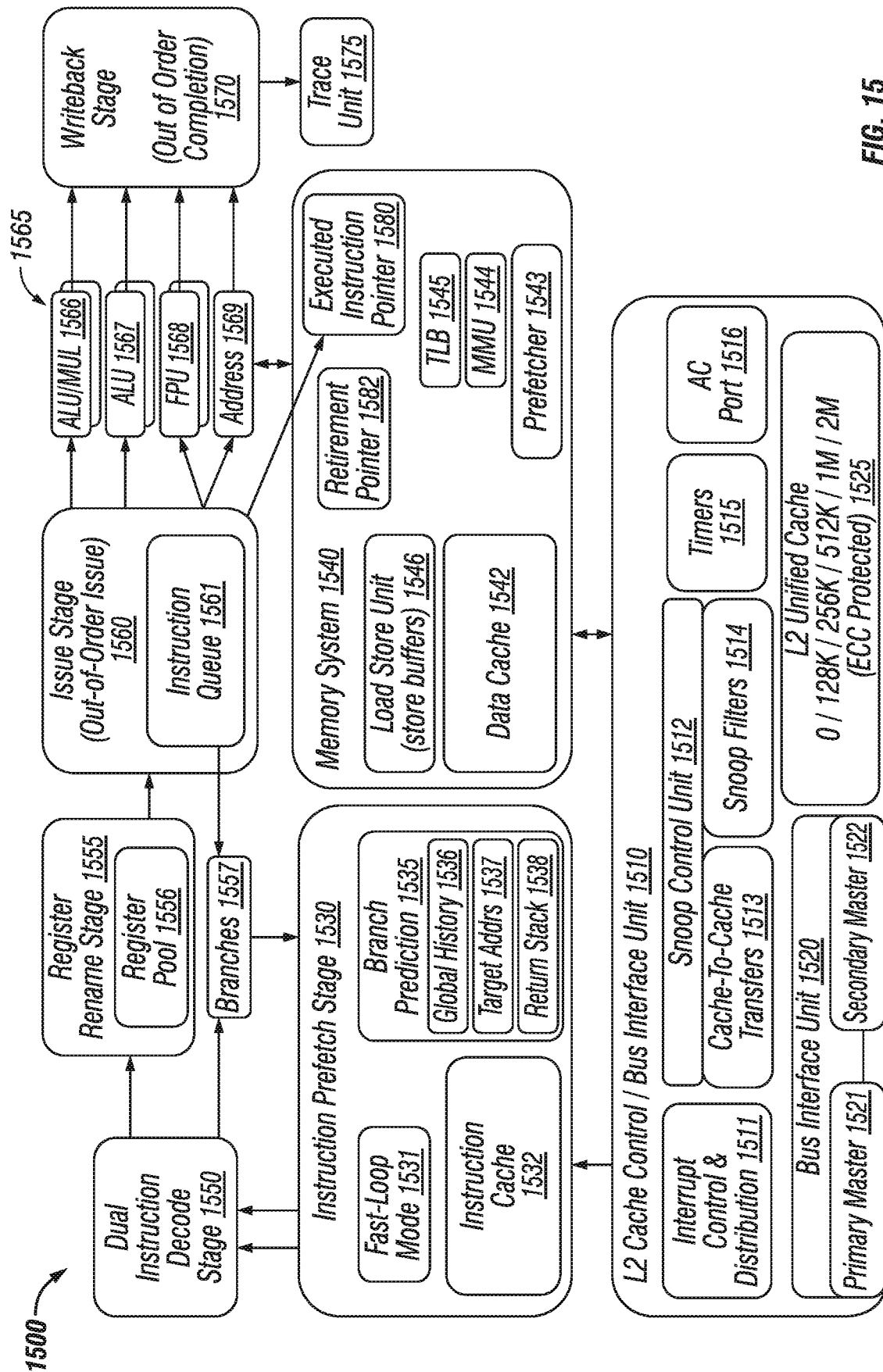


FIG. 15

22/30

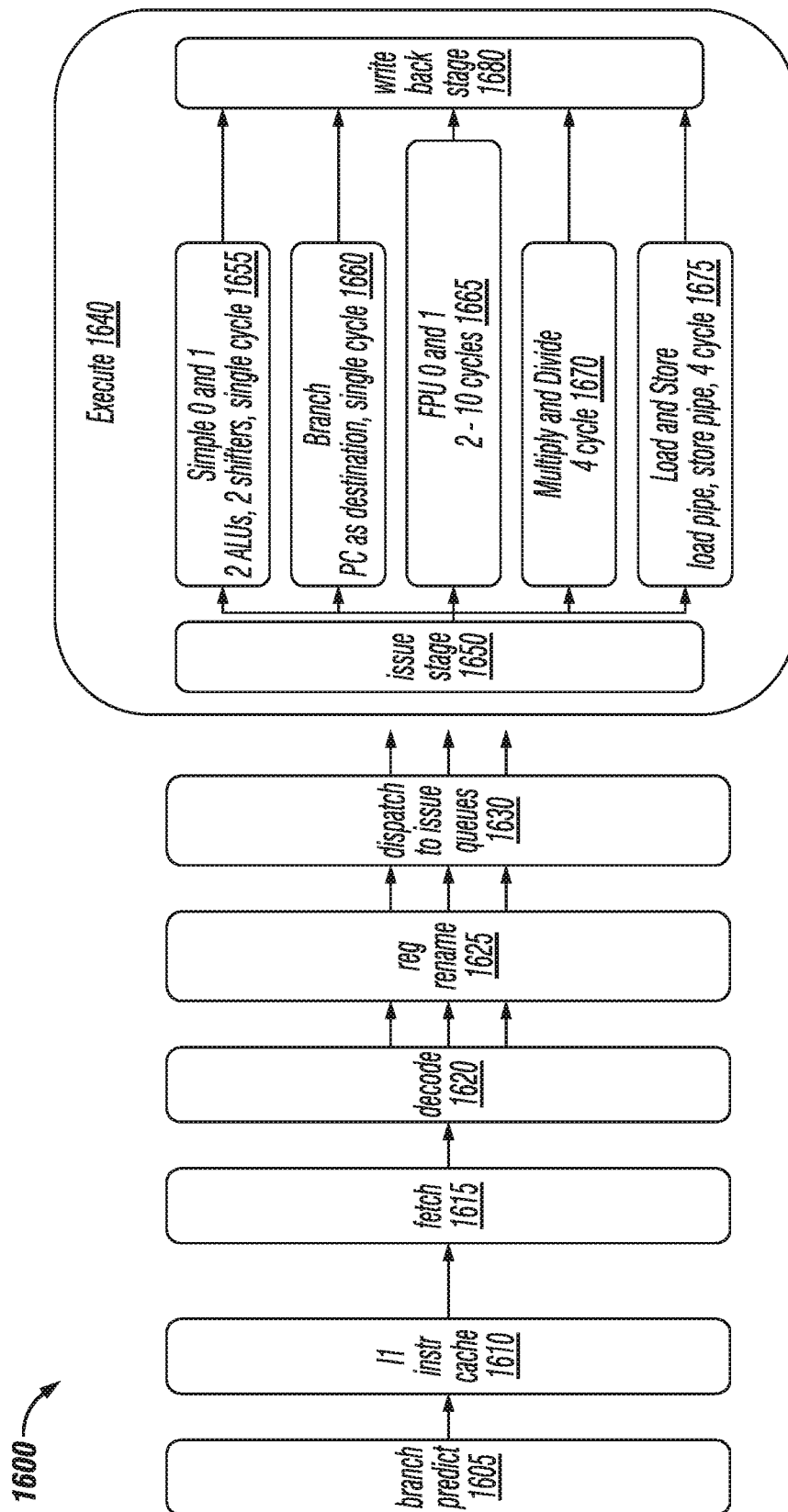


FIG. 16

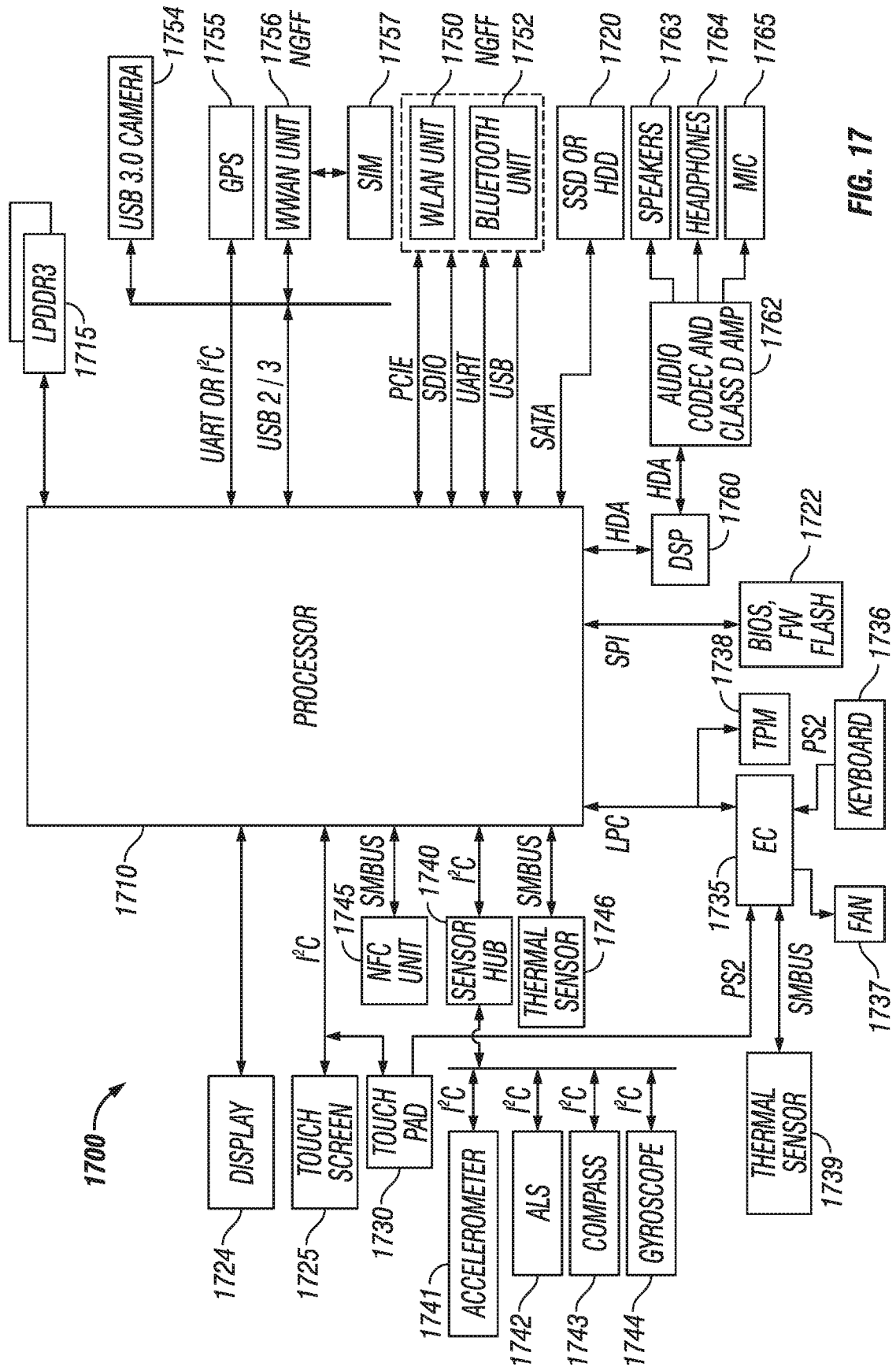


FIG. 17

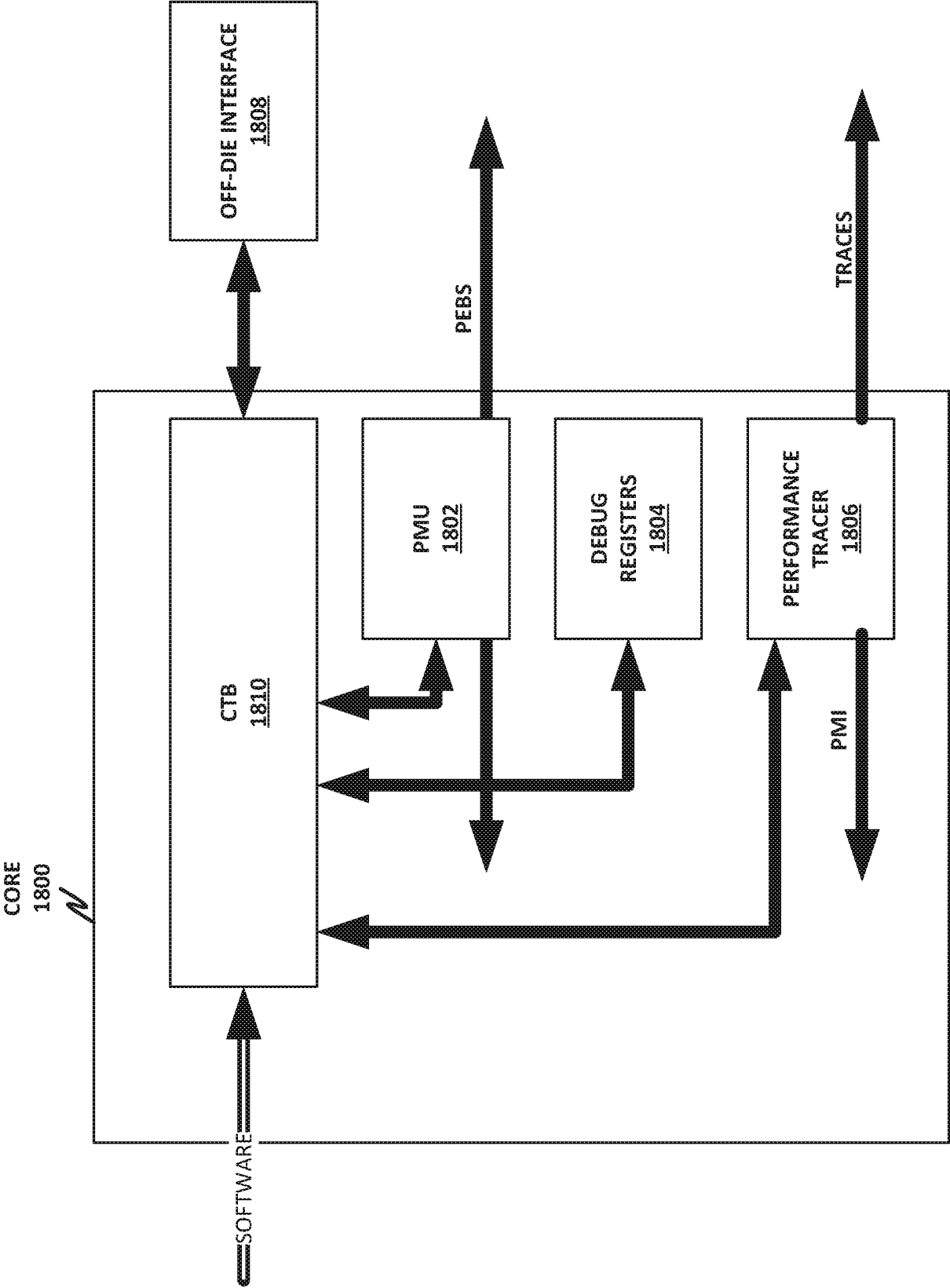
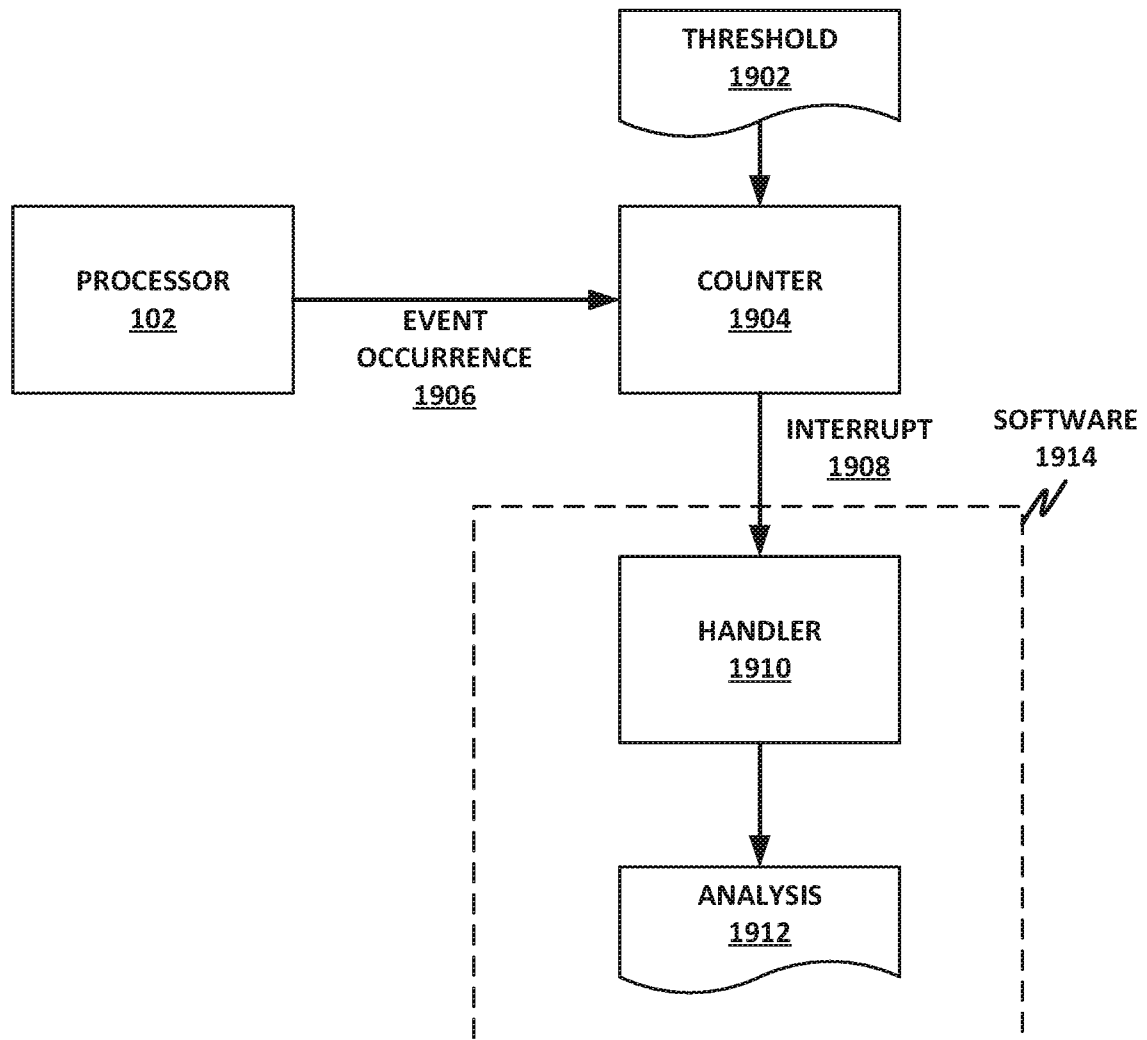
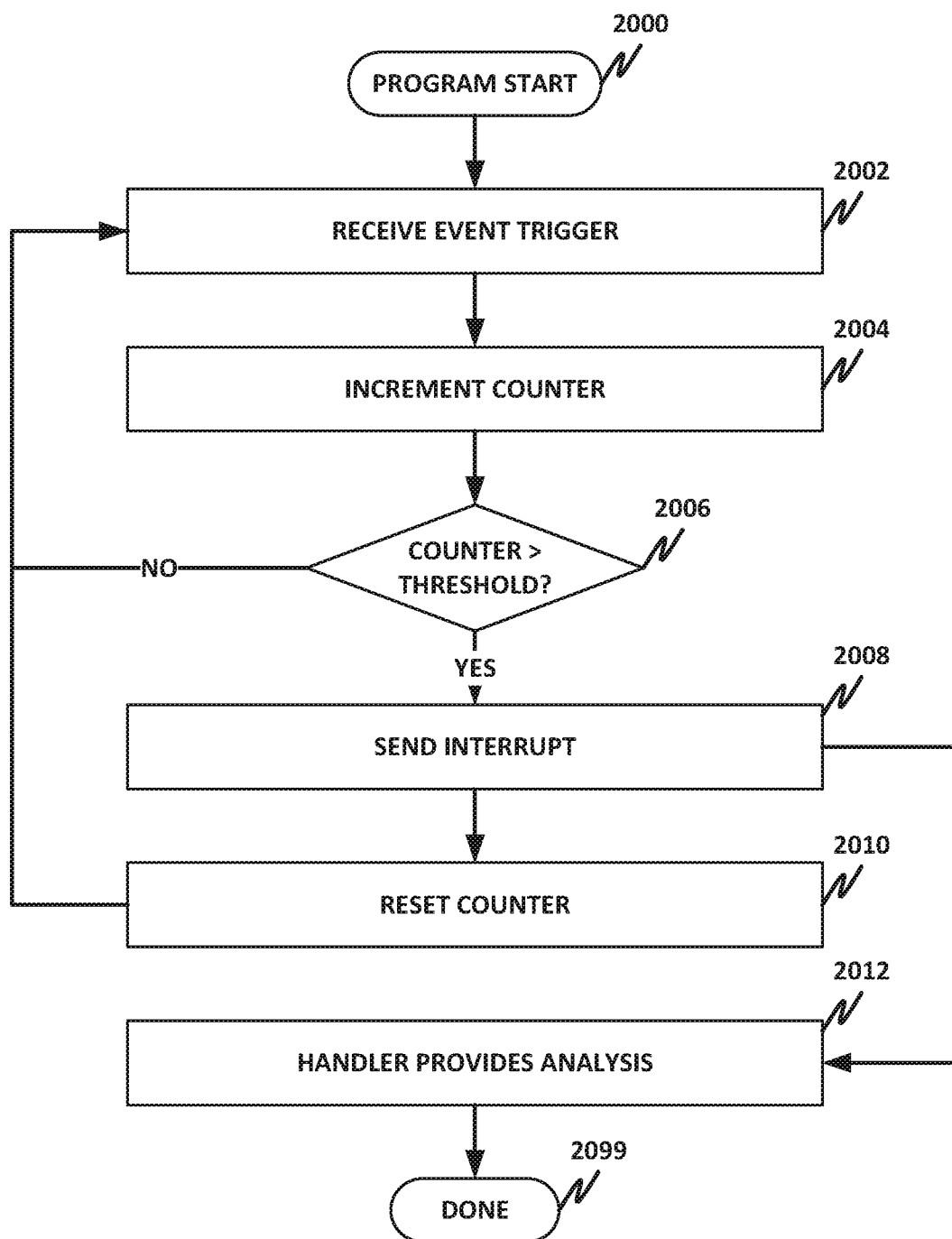


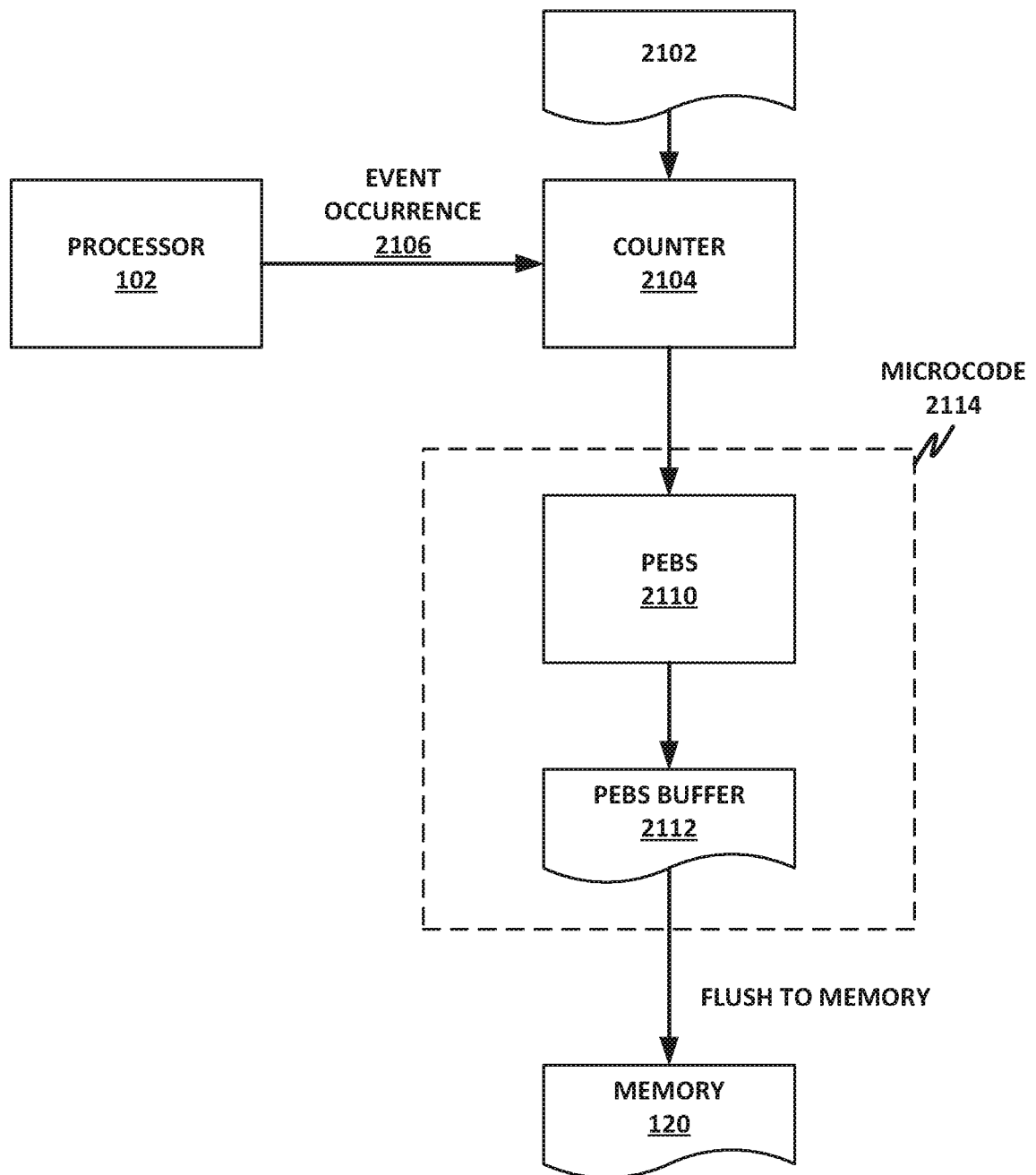
Fig. 18

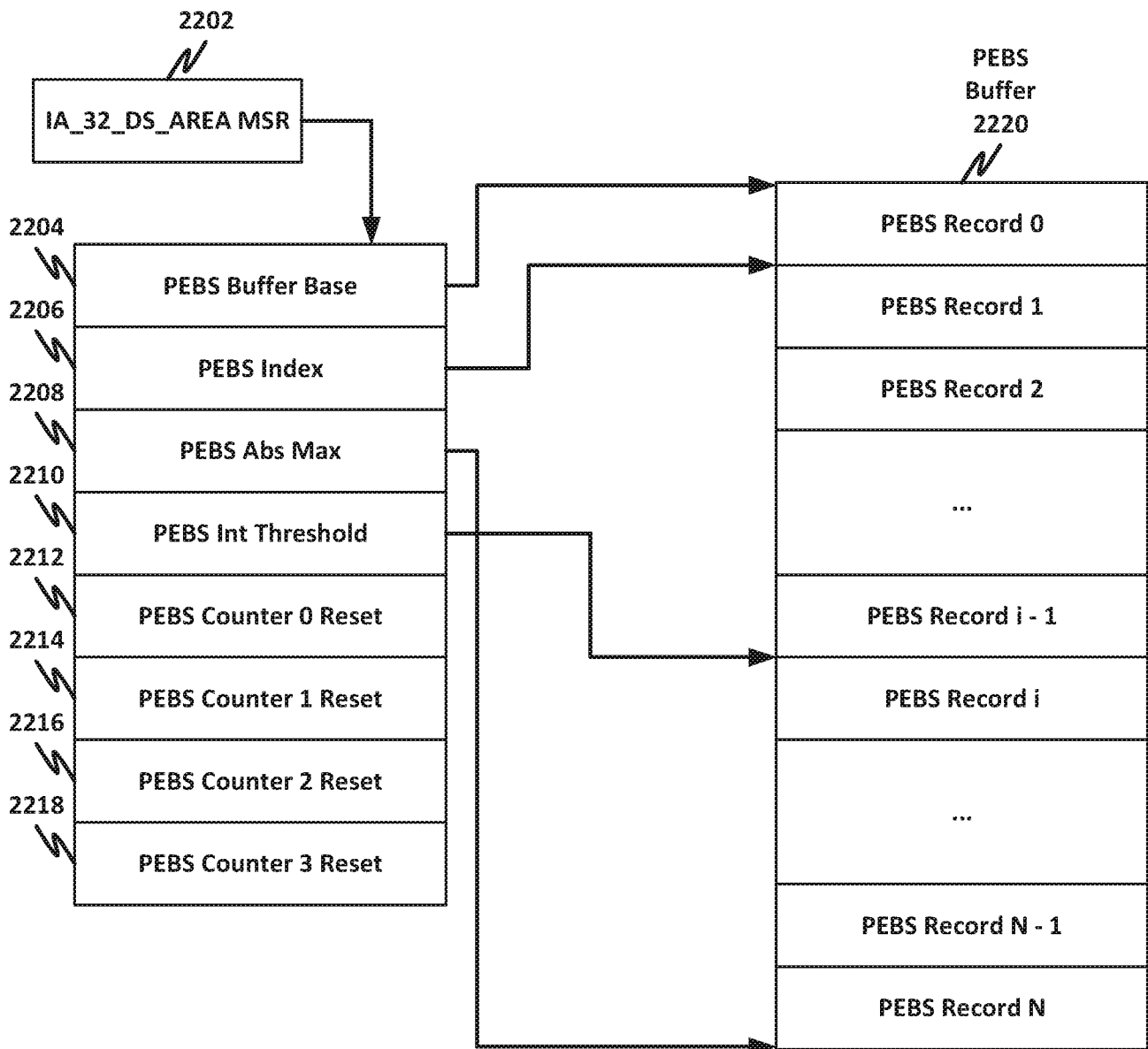
25/30

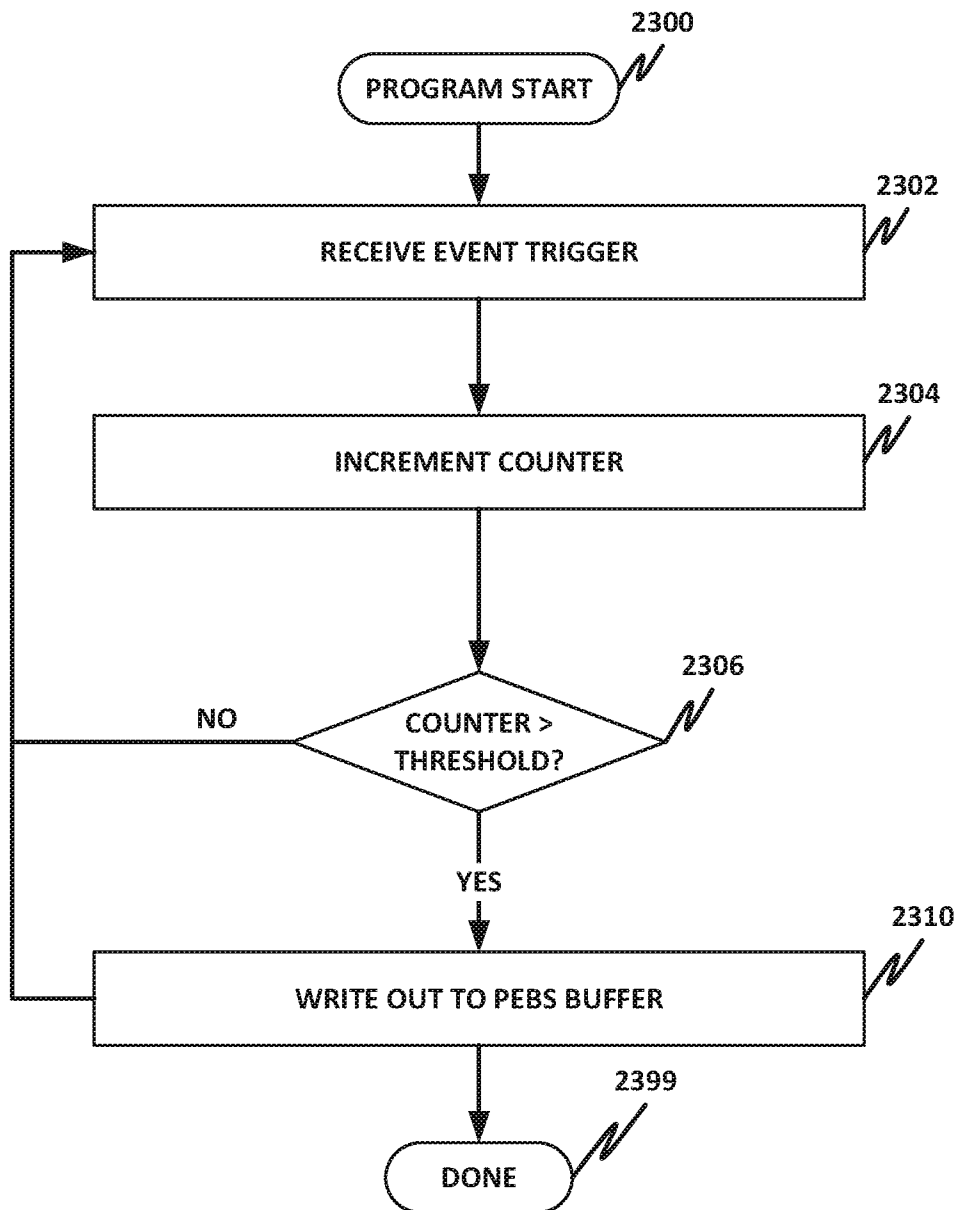
*Fig. 19*

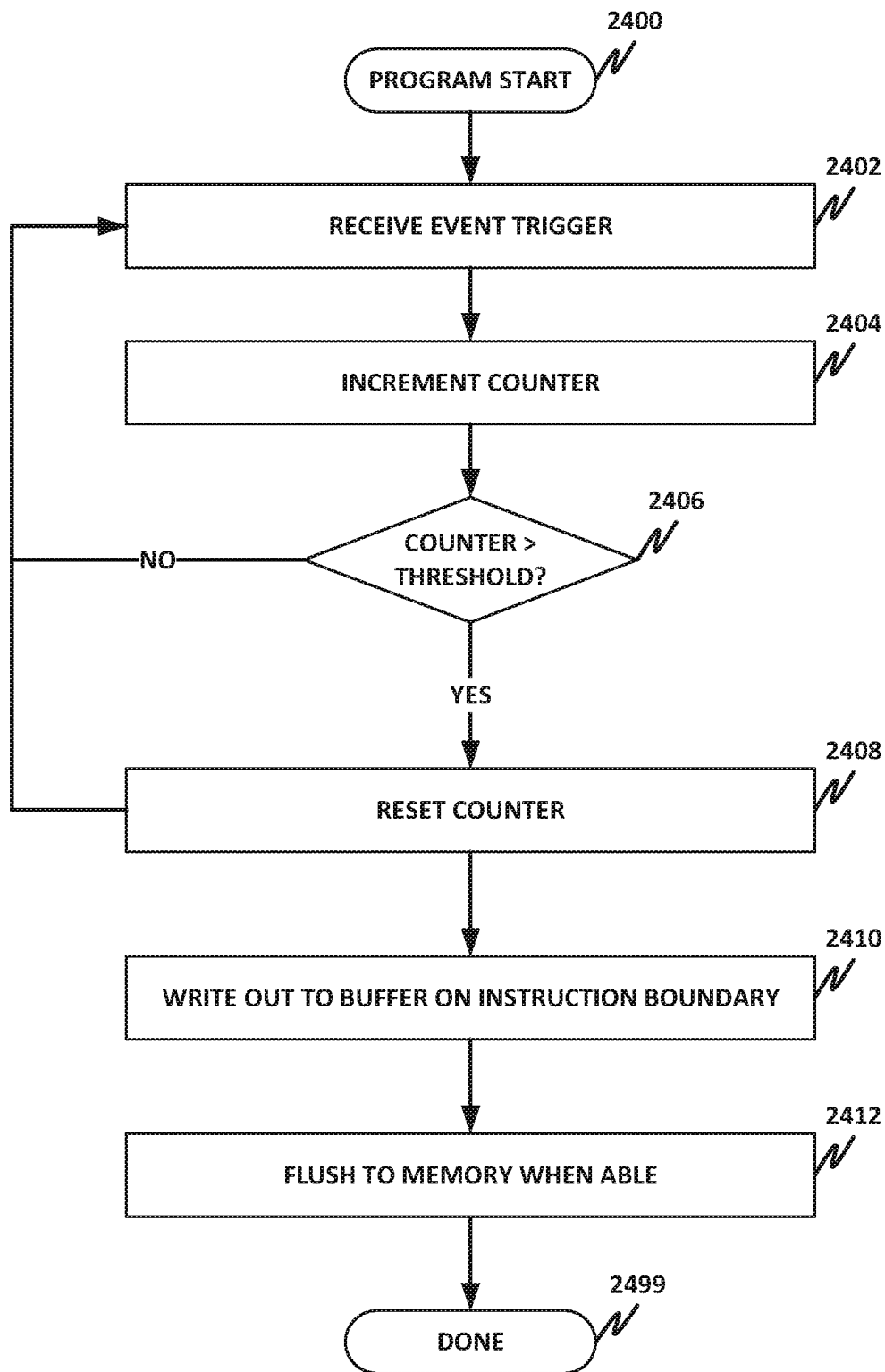
26/30

*Fig. 20*

*Fig. 21*

*Fig. 22*

*Fig. 23*

*Fig. 24*

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/039339**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i, G06F 12/0802(2016.01)i**

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

B. FIELDS SEARCHEDMinimum documentation searched (classification system followed by classification symbols)
G06F 9/30; G06F 11/07; G06F 11/34; G06F 15/173; G06F 9/44; G06F 11/00; G06F 12/0802Documentation 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 modelsElectronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: tracing, logging, integration, core triggering block, debug, instruction**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	US 2008-0148239 A1 (MIROSLAV R. PETROV et al.) 19 June 2008 See paragraphs [0021]-[0023], [0027], [0114]; claim 18; and figures 2-3.	1-26
Y	US 2016-0170820 A1 (INTEL CORPORATION) 16 June 2016 See paragraphs [0055], [0063], [0068], [0089]; and figures 4A, 6B.	1-26
Y	US 2016-0179541 A1 (ROGER GRAMUNT et al.) 23 June 2016 See paragraph [0153]; and figure 18.	7, 17, 25
A	US 7739374 B1 (GREGOR K. FREY et al.) 15 June 2010 See column 3, line 31 - column 7, line 19; and figures 2-3.	1-26
A	US 2006-0174165 A1 (SHMUEL SHAFFER et al.) 03 August 2006 See paragraphs [0026]-[0030]; and figure 2.	1-26



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

27 October 2017 (27.10.2017)

Date of mailing of the international search report

30 October 2017 (30.10.2017)

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, Chang Ho

Telephone No. +82-42-481-8288



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/039339

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2008-0148239 A1	19/06/2008	US 7606893 B2	20/10/2009
US 2016-0170820 A1	16/06/2016	US 9632907 B2	25/04/2017
US 2016-0179541 A1	23/06/2016	US 9652237 B2	16/05/2017
US 7739374 B1	15/06/2010	US 2005-0149535 A1 US 7743029 B2	07/07/2005 22/06/2010
US 2006-0174165 A1	03/08/2006	US 7398429 B2	08/07/2008