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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
5 October 2017 (05.10.2017)

WIPO | PCT

(10) International Publication Number
WO 2017/172128 A1

(51) International Patent Classification:
G06F 11/34 (2006.01) *G06F 12/02* (2006.01)

(21) International Application Number:
PCT/US2017/019063

(22) International Filing Date:
23 February 2017 (23.02.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
15/089,179 1 April 2016 (01.04.2016) US

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

(72) Inventors: **PEREIRA, Cristiano L.**; 20640 White Gulch Road, Groveland, California 95321 (US). **POKAM, Gilles A.**; 4947 Friar Avenue, Fremont, California 94555 (US). **HU, Shiliang**; 2230 Homestead Court, #206, Los Altos, California 94024 (US). **STRONG, Beeman C.**; 3841 NE Flanders, Portland, Oregon 97232 (US).

(74) Agent: **KOMENDA, J. Kyle**; Patent Capital Group, 2816 Lago Vista Lane, Rockwall, TX 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, 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.

[Continued on next page]

(54) Title: INSTRUCTION, CIRCUITS, AND LOGIC FOR DATA CAPTURE FOR SOFTWARE MONITORING AND DEBUGGING

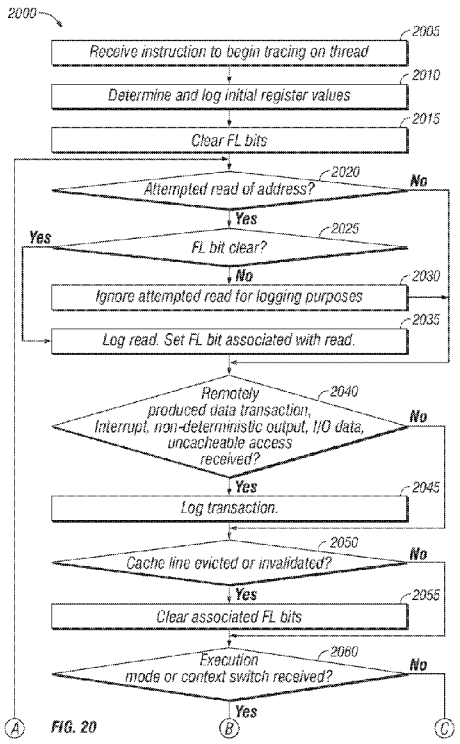


FIG. 20

(57) Abstract: A processor includes a front end including circuitry to receive an instruction to monitor execution of a thread, a decoder including circuitry to decode the instruction, a scheduler including circuitry to schedule the instruction, a retirement unit including circuitry to retire the instruction, and a core. The core includes circuitry to, based on execution of the instruction, monitor execution of the thread, identify an attempted read of an address during execution of the thread, determine whether a value at the address was previously read during monitoring of the execution of the thread, log the attempted read based on a determination that the value at the address was not previously read during monitoring of the execution of the thread, and omit logging of the attempted read based on a determination that the value at the address was previously read during monitoring of the execution of the thread.

WO 2017/172128 A1

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, 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).

Declarations under Rule 4.17:

— *as to the identity of the inventor (Rule 4.17(i))*

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*

— *as to the applicant's entitlement to claim the priority of the earlier application (Rule 4.17(iii))*

Published:

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

INSTRUCTION, CIRCUITS, AND LOGIC FOR DATA CAPTURE FOR SOFTWARE MONITORING AND DEBUGGING

FIELD OF THE INVENTION

[0001] The present disclosure pertains to the field of processing logic, microprocessors, and associated instruction set architecture that, when executed by the processor or other processing logic, perform logical, mathematical, or other functional operations.

DESCRIPTION OF RELATED ART

[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. Furthermore, in order to increase the utility of a processing entity, out-of-order execution 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.

DESCRIPTION OF THE FIGURES

[0003] Embodiments are illustrated by way of example and not limitation in the Figures of the accompanying drawings:

[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 a system for data capture, in accordance with embodiments of the present disclosure;

[0031] FIGURE 19 is a more detailed illustration of a system for data capture, in accordance with embodiments of the present disclosure; and

[0032] FIGURE 20 is a flowchart of a method to execute an instruction for data capture, in accordance with embodiments of the present disclosure.

DETAILED DESCRIPTION

[0033] The following description describes an instruction, circuit, and processing logic for a data capture mechanism for software monitoring and debugging. In the following description, numerous specific details such as processing logic, processor types, micro-architectural conditions, events, enablement mechanisms, circuits, and the like are set forth in order to provide a more thorough understanding of embodiments of the present disclosure. It will be appreciated, however, by one skilled in the art that the embodiments may be practiced without such specific details. Additionally, some well-known structures, circuits, and the like have not been shown in detail to avoid unnecessarily obscuring embodiments of the present disclosure.

[0034] Although the following embodiments are described with reference to a processor, other embodiments are applicable to other types of integrated circuits and logic devices. Similar techniques and teachings of embodiments of the present disclosure may be applied to other types of circuits or semiconductor devices that may benefit from higher pipeline throughput and improved performance. The teachings of embodiments of the present disclosure are applicable to any processor or machine that performs data manipulations. However, the embodiments are not limited to processors or machines that perform 512-bit, 256-bit, 128-bit, 64-bit, 32-bit, or 16-bit data operations and may be applied to any processor and machine in which manipulation or management of data may be performed. In addition, the following description provides examples, and the accompanying drawings show various examples for the purposes of illustration. However, these examples should not be construed in a limiting sense as they are merely intended to provide examples of embodiments of the present disclosure rather than to provide an exhaustive list of all possible implementations of embodiments of the present disclosure.

[0035] 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.

[0036] 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).

[0037] 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.

[0038] 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.

[0039] 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).

[0040] 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 of 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.

[0041] 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.

[0042] 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 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.

[0043] 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.).

[0044] 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.

[0045] 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 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 and software.

[0046] 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.

[0047] 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.

[0048] 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.

[0049] Execution unit 108, including 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 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.

[0050] 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.

[0051] 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.

[0052] 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.

[0053] 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.

[0054] 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.

[0055] 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.

[0056] 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.

[0057] 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.

[0058] 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).

[0059] 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.

[0060] In one embodiment, SIMD coprocessor 161 comprises an execution unit 162 and a set of register files 164. One embodiment of main processor 166 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.

[0061] 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.

[0062] 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.

[0063] 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.

[0064] 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.

[0065] 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.

[0066] 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.

[0067] 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.

[0068] 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 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.

[0069] 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 data 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.

[0070] 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.

[0071] 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.

[0072] 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 or more.

[0073] 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.

[0074] 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.

[0075] 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.

[0076] 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.

[0077] 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.

[0078] 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.

[0079] 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.

[0080] 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.

[0081] 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.

[0082] 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 454 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.

[0083] 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.

[0084] 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.

[0085] 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).

[0086] 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.

[0087] 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 caches may be external to the core and/or the processor.

[0088] 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.

[0089] 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.

[0090] 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.

[0091] 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 514 for communications busses for

graphics. In one embodiment, interface 514 may be implemented by PCI Express (PCIe). In a further embodiment, interface 514 may be implemented by PCI Express Graphics (PEG). 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.

[0092] 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.

[0093] 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.

[0094] 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.

[0095] 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.

[0096] 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.

[0097] 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.

[0098] 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 582. In one embodiment, allocate module 582 may allocate resources of processor 500 or other resources, such as registers or buffers, to execute a given instruction. Allocate module 582 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 582 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.

[0099] 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.

[00100] 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.

[00101] 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.

[00102] 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.

[00103] 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.

[00104] 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 a discrete graphics device coupled to ICH 650 along with another peripheral device 670.

[00105] 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.

[00106] 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.

[00107] 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.

[00108] 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.

[00109] 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.

[00110] 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.

[00111] 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.

[00112] 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.

[00113] 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.

[00114] FIGURE 8 illustrates that processors 770, 780 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

732, 734 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 790.

[00115] 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 502A-N and shared cache units 506; a system agent unit 510; 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.

[00116] 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.

[00117] 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.

[00118] 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 I^2S/I^2C 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.

[00119] 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.

[00120] FIGURE 11 illustrates a block diagram illustrating the development of IP cores, in accordance with embodiments of the present disclosure. Storage 1100 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 1100 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 1165 where it may be fabricated by a 3rd party to perform at least one instruction in accordance with at least one embodiment.

[00121] 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.

[00122] 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.

[00123] 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.

[00124] 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.

[00125] 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 codec 1420 defining the manner in which particular video signals will be encoded and decoded for output.

[00126] 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 1460. Flash controller 1445 may provide access to or from memory such as flash memory 1465 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.

[00127] 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.

[00128] 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 1565 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 1555, issue stage 1560, and writeback stage 1570.

[00129] 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 the 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.

[00130] 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.

[00131] 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.

[00132] 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.

[00133] 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.

[00134] 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 1540 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 1540 may include a translation lookaside buffer (TLB) 1545 that provides look-up of address values between physical and virtual addresses. In yet another embodiment, memory system 1540 may include a memory management unit (MMU) 1544 for facilitating access to virtual memory. In still yet another embodiment, memory system 1540 may include a prefetcher 1543 for requesting instructions from memory before such instructions are actually needed to be executed, in order to reduce latency.

[00135] 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 1550.

[00136] 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.

[00137] 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.

[00138] 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.

[00139] 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.

[00140] 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.

[00141] 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.

[00142] 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.

[00143] 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 I^2C 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.

[00144] 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) 1755, 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.

[00145] 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 1736, and touch pad 1730 may be communicatively coupled to EC 1735. Speakers 1763, headphones 1764, and a microphone 1765 may be communicatively coupled to an audio unit 1762, which may in turn be communicatively coupled to DSP 1760. Audio unit 1762 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).

[00146] FIGURE 18 is an illustration of a system 1800 to implement instructions, circuitry, and logic for data capture. The data capture may be selective about the elements evidencing execution that are to be captured. The data capture may be used for software monitoring and debugging. The data capture may be made on a processor including or more cores, such as core 1802. The system including the processing that includes core 1802 may include other processors. The software execution that is monitored and debugged may be running on any suitable processor or core of the system.

[00147] System 1800 may include any suitable facility or mechanisms for debugging software that executes upon its processors. Hardware support for monitoring assists the software developer to better understand and debug the execution of programs on a processor. Such facilities may include, for example, performance monitoring units (PMUs), last branch record (LBR) and Processor Trace (PT). Various performance events may be logged so that the logs can later be used for debugging and performance monitoring. Various hardware structures of the processor may expose various performance events, which may be useful for performance analysis. Both LBR and Intel PT may collect execution traces to reproduce the control-flow of a program for performance analysis and debugging. While extremely useful, control-flow traces do not capture the values manipulated during the execution, therefore limiting their usability scope. Memory corruption bugs, for example, may require understanding the data read and written by the program via memory instructions, so that a programmer can properly root cause the source of the problem. To address these types of problems, system 1800 may enable hardware support for data tracing (logging of addresses and values of memory operations). In one embodiment, system 1800 may improve existing data tracing techniques. Data tracing may generate immense amounts of information. For example, data tracing may typically collect every load and store operation. Each such operation executed by a software thread may produce at least one pair of an address and a value to be logged. This might otherwise overwhelm the data bus with information, therefore slowing the system down significantly and creating very large

log files. In order to reduce data tracing cost, the processor may implement address filtering mechanisms, wherein it only logs a range of memory addresses, to minimize logging or to encourage limiting the length of the execution where data traces are collected. Only short bursts might be supported. These log reduction methods have shortcomings as a result of the limited data that can be collected. Additionally, the sheer amount of tracing data from the baseline implementation would typically result in many data packets dropouts for the traced section of execution.

[00148] In one embodiment, system 1800 may employ data capture and tracing techniques that capture a reduced set of data that is still sufficient to provide monitoring and debugging functionality. System 1800 may include support for hardware that selectively captures data in view of other techniques. Such selective capture may be referred to "light data capture" (LDC). In one embodiment, system 1800 may selective capture a smaller fraction of memory operation addresses and values during the execution. In another embodiment, system 1800 may selectively capture other events. System 1800 may maintain sufficient information to reconstruct the addresses and values of all memory operations executed via an offline analysis phase. As a result, system 1800 might not need to collect as much data. In another embodiment, system 1800 may collect only a subset of data while recreating data that was not collected. Recreating the data that was not collected may include, for example, reexecuting software threads and steering to the same execution paths. Such techniques may include deterministically replaying software. In another embodiment, system 1800 may perform this data trace compression method, where the trace compressor generates as little data trace information as possible and the trace decompressor recovers the rest of the information by re-executing or replaying the program threads. In yet another embodiment, the subset of values that are to be captured, for a given software thread of execution, are the values that cannot be produced by reexecuting the software thread instructions during replay.

[00149] In one embodiment, system 1800 may monitor and record execution for first read values. These may include data values that are read from memory for the first time (first read to each memory location). System 1800 might omit recording and monitoring subsequent reads of the memory location.

[00150] In another embodiment, system 1800 may monitor and record execution for remotely produced values. These may include data values that are read by a software thread being traced wherein the software thread did not produce the value. Such values come from, for example, another core, processor, software thread, or I/O devices. In a further embodiment, these values may be produced by DMA transactions. In another, further embodiment, these values may be produced by remote software threads, such as software threads which share data with the traced thread. The scope of execution may determine this set of values. For example, if data tracing is enabled for a root or kernel mode 0 execution only, values produced at a user level or kernel mode 3 may also be gathered. The higher-level values might not be reproducible using only the root level actions. Conversely, if data tracing is enabled for user mode only, the values produced by root level might also need to be captured.

[00151] In yet another embodiment, system 1800 may record machine context information. These may include initial machine register state and register values affected during the execution by some external entity. The values may include an initial value of programmer visible registers, flow-changes due to interrupts, instructions producing non-deterministic output, and values consumed from I/O ports (port ins).

[00152] System 1800 may receive, decode, schedule, execute, and retire instructions to activate hardware extensions to monitor for these kinds of subsequent values. These may be monitored and values stored accordingly. The values may be stored with appropriate information such as program counters or instruction pointer values so that the values can later be mapped to the instructions that generated the values. The code may then be reexecuted to obtain the values of various elements

[00153] By only capturing the first loads to a given address, and omitting captures of subsequent loads to the same address, system 1800 may save considerable space and throughput. For example, consider the following captures wherein every load and store operation is captured, rather than only first loads to an address.

store reg0, [A];	Logged
store reg1, [B];	Logged
load reg2, [C];	Logged

```
load reg0, [A];        Logged
load reg1, [B];        Logged
load reg3, [C];        Logged
```

[00154] In contrast, by only logging the first load from each memory address, the following may be performed:

```
store reg0, [A];
store reg1, [B];
load reg2, [C];        Logged
load reg0, [A];        Logged
load reg1, [B];        Logged
load reg3, [C];
```

[00155] If an external force operates upon a given memory location, the memory location may be logged upon the next load. For example, given a DMA operation that would change the value at memory location C, a subsequent load from C will be logged.

```
store reg0, [A];
store reg1, [B];
load reg2, [C];        Logged
load reg0, [A];        Logged
load reg1, [B];        Logged
DMA-write [C];
load reg3, [C];        Logged
```

[00156] To implement detection, monitoring, and logging of first read values, system 1800 may utilize metadata so that its components know when a given read is the first time an address is touched or read. If the hardware components knew when each address is touched for the first time since data tracing was turned on might require one bit of information for every addressable byte of memory. Such a bit can be referenced as a first load bit (FL bit for short), but may be implemented in other suitable ways. Instead of a FL bit for each and every addressable byte of memory, in one embodiment

system 1800 may approximate the FL bit. In a further embodiment, system 1800 may approximate the FL bit for each memory address by adding one FL bit for each line in a cache, such as the L1 cache. For example, such a bit may be set to 0 in the beginning of tracing. Upon a read access to the cache line, if the bit is 0, it is set to 1 and the entire content of the line is captured in the trace, regardless of how many bytes are touched. Upon cache line eviction, the FL bit may be reset, so that the newly filled line is also logged in a future access. During this process, if a given address is read for the first time during the execution, its value and address will be logged by a logger searching for a set FL bit. If the line is evicted due to capacity limits and then read again, the logger might read or log that line redundantly. This side-effect does not affect accuracy, but is a practical consideration that with respect to the efficiency of the operation of system 1800.

[00157] Assuming that a single bit per cache line is used, the granularity for capturing is the content of the entire cache line. If a finer granularity is needed, system 1800 may implement FL bit checking to subportions of the cache line. For example, a bit for each byte in the cache line might be used to track accesses at a granularity corresponding to the byte. Granularity considerations may be made by an implementer of system 1800 with respect to space requirements for additional FL bits versus increased efficiency by removal of false-positives.

[00158] System 1800 may check bits in the following manner. System 1800, upon receipt of a configuration instruction to trace a thread of execution in a light data capture manner, may reset all FL bits for the entire cache. Upon a cache line read access, if the FL bit for the cache line is set to zero, all content of the entire cache line may be captured for monitoring, logging, and debugging purposes. The content may be written to a processor trace memory or other suitable location. The FL bit for the cache line may be set to one. If the FL bit for the cache line is set to one, on subsequent accesses the content of the cache line will not be stored. If the cache line is evicted, the FL bit for the evicted line may be reset to zero. If the cache line is invalidated, then the FL bit may be reset for the invalidated line.

[00159] The FL bits might also be used to capture remotely produced values that are accessed locally. Upon a cache line invalidation, the FL bit is reset to 0 again. The implication is that the value of the next read access to the cache line will be captured by

system 1800. Both DMA writes and write operations from remote threads may generate invalidation messages for cacheable addresses.

[00160] In one embodiment, in cases where an instruction to log data traces is only concerned with user mode code, upon transitioning to root mode system 1800 can reset all FL bits for the cache in which the transition took place. This ensures that upon returning to user mode, all data modified by root mode code is logged as remotely modified values. In another embodiment, system 1800 may cause root mode write accesses to actually reset FL bits, which may also result in correct logging for the user code without resetting all bits. When tracing user mode code, software calling for monitoring and debugging by system 1800 might ensure that FL bits are reset on context switches between different applications on the same processor or CPU (and thus sharing the same cache) to ensure proper logging of data for each application thread. The reverse case may also be applied by system 1800, wherein root mode execution switched to user mode execution may be tracked by resetting all FL bits, or by triggering upon writes by user mode code to the cache.

[00161] In one embodiment, for security reasons, it may be valuable to also suppress tracing for implicit accesses from within user mode code to data structures created by code running in root mode. Page tables, for memory virtualization, and segment descriptor tables are examples of such structures. For replay of user mode code, reproducing these accesses is typically unnecessary, as they do not influence the behavior of the user mode code.

[00162] As described above, FL bits may be used to filter data capture for cacheable accesses. However, system 1800 may perform accesses to uncacheable memory locations. These may include, for example, memory mapped I/O operations, non-temporal accesses, etc. Since these bypass the cache, in one embodiment system 1800 might capture every read access to uncacheable memory. Specifically, all uncacheable loads may be logged with a precise byte or size of the data. In a further embodiment, system 1800 may perform these captures regardless of whether a given read is the first or not. This may be performed because the value of the memory location can change as a result of writes from other agents, or for other reasons such as memory-mapped I/O. Though not memory accesses, I/O reads might be treated in the same way as uncacheable reads by system 1800, such that all are logged.

[00163] In one embodiment, in cases where an element of CPU state can be reproduced by a replay simulator, only the initial value of that state element might be captured by system 1800. For example, system 1800 might only capture the initial state of a general-purpose register. System 1800 might include an architectural simulator to model the value changes to these registers given the code that is executed. To the extent that there are processor states that cannot be easily reproduced by a simulator, due to reasons of complexity or unpredictability, then system 1800 might capture these values each time they are read. Examples of such processor states include performance counters and timer values

[00164] In one embodiment, system 1800 may track asynchronous events. These may include, for example, external interrupts, exceptions, or faults. These events can cause unexpected control flow redirection, as well as writes to memory or registers. In cases where the replay simulator cannot predict the presence of such events, the trace generated by system 1800 may include an indication of the type, time, and location in the code of the event.

[00165] Hardware in system 1800 to perform data capture may be performed in any suitable manner. System 1800 may include a processor (not shown) in which a core 1802 resides. The processor for system 1800 may be implemented in part using the elements shown in FIGURES 1-17. The components of hardware in system 1800 may be implemented by circuitry, circuits, reconfigurable hardware, and other suitable elements.

[00166] System 1800 may also include a memory subunit or socket 1804. Socket 1804 may include one or more levels of cache, such as L2 cache 1822.

[00167] System 1800 may also include a platform 1806 for accessing other cores or processors of system 1800. Platform 1806 may include an interconnect 1824 for interfacing with In Target Probe (ITP) signals or DRAM operations. These may handle data transactions for monitoring and debugging purposes.

[00168] In one embodiment, core 1802 may include LDC control logic 1808, which may be implemented by circuitry, circuits, or reconfigurable logic. LDC control logic 1808 may be implemented in other suitable portions of core 1802 or of the processor.

[00169] Core 1802 may also include a reorder buffer (ROB) 1810, one or more execution units 1812, and a first layer of cache such as L1 cache 1814, In one

embodiment, core 1802 may include an LDC packet generator 1818, which may be implemented by circuitry, circuits, or reconfigurable logic. In another embodiment, core 1802 may include FL bits 1816. In a further embodiment, core 1802 may include an FL bit 1816 for each line of L1 cache 1814. In another, further embodiment, core 1802 may include an FL bit 1816 for a subportion of L1 cache 1814, such as for individual bytes or a range of bytes. In yet another embodiment, core 1802 may include a debug store 1820, which may be implemented by circuitry, circuits, or reconfigurable logic.

[00170] LDC control logic 1808 and FL bits 1816 implement the logging of memory values discussed above, as well as handle invalidations and cache line evictions. Memory values and CPU state/event packets (logged via microcode) are packetized in the LDC packet generator 1818. LDC packet generator 1818 may also accept as input the retired instruction count (RIC) from ROB 1810. The packets may be sent to Debug Store 1820. Debug Store 1820 may accumulate and store packets in internal buffers. When needed, debug store 1820 may provide the packets to, for example, memory or a debug port (e.g., USB or PTI). Depending on the implementation choices, system 1800 may choose to either drop packets or stall the pipeline (pipe control) when the buffer is filling faster than it is emptied.

[00171] FIGURE 19 is a more detailed view of components and operation of system 1800 to perform data capture, in accordance with embodiments of the present disclosure.

[00172] FL bits 1816 are also kept as FL bits 1936 in a fill buffer 1934. Bits are kept in fill buffer 1934 because loads might be serviced by fill buffers even when the data is not otherwise in the cache. A data path for the information logged by LDC control logic 1808 is created. An address may be generated or referenced by an address generation unit 1938. This may be generated as a virtual address in the translation lookaside buffer 1940, which may translate it to a physical address for LDC control logic 1808. The virtual address may be passed to LDC packet generator 1818. LDC control logic 1808 may receive cache line invalidation from a source such as a front side bus, memory interface/interconnect, in-die interconnect, or other source; trace start and other control events from an execution unit executing command to start, stop, or adjust tracing, or context switches; or line eviction notices from L1 cache 1814. LDC

control logic 1808 may provide control signals to, for example, a multiplexer that outputs data from fill buffer 1934 or from L1 cache 1814. The output data and size may be sent to LDC packet generator 1818. LDC control logic 1808 may set or reset FL bits it fill buffer 1934 and L1 cache 1814.

[00173] Fill buffer 1934 may cause data to be monitored data upon receipt of uncacheable memory reads. As used in the present disclosure, a fill buffer may also be known as a request buffer or miss status holding register.

[00174] LDC packet generator 1818 may also accept inputs such as the RIC from the ROB. The packets may include the memory address, the value, the size of the access (as uncacheable accesses require support for logging only the bytes read), and the RIC. The output of the LDC packet generator 1818 may be multiplexed with interrupt indications or similar messages from microcode-based event triggers 1930. Accordingly, these may also be stored. All such events may be stored in debug store 1820. Production of monitored events may be made periodically, and as such sent as output to the interconnect, memory, or other designated location.

[00175] In one embodiment, LDC packet generator 1818 may produce data packets. In another embodiment, LDC packet generator 1818 may produce processor state and event packets. Data packets may include the address of the data access and the corresponding cache line value. These values are used by a decompressor (replayer) offline, after the trace capture is finished. The format of the data packet may include the retired instruction count (RIC) of the instruction that caused the packet to be captured, the size of the data access, the memory address of the data access and the corresponding cache line value. These packets may contain first read values from the cache, or values directly from memory or I/O. The size field may be used for uncacheable loads, as cacheable loads can capture the whole cache lines when tracing at cache line level.

[00176] The memory addresses captured may include address generation unit-generated addresses, which may include linear addresses (or virtual addresses on non-x86 platforms). Virtual or linear addresses might be useful for programmers to reason and debug, as these are the addresses visible to developers and debuggers in software binaries. However, as a result of virtual memory mechanisms such as paging, multiple linear addresses can be mapped to the same physical address. Known as “aliasing”, this

can be used to allow data sharing between processes. For proper replay, the simulator might need to have awareness of aliasing, so that the proper (shared) data is accessed despite the differing linear addresses. In some cases, system 1800 might support aliasing by logging page walk loads just as first loads are logged. In some scenarios, wherein the replay simulator may need to behave differently based on segment configuration, segmentation loads will thus be logged as well. In other cases, a sideband listing of aliased pages might be kept. Such a sideband listing might be more practical if aliasing is rare or not applicable (e.g., when paging is disabled).

[00177] In one embodiment, CPU state packets generated by LDC packet generator 1818 may contain register values which are affected as a result of non-reproducible events during the execution. These are discussed above within the context of FIGURE 18. Such packets may include the event type (contained in a packet header not shown) and the RIC of the instruction, which maps to the event or the RIC of when the event occurred (in case of interrupts). The CPU state packet may also include a machine register name and value list.

[00178] LDC packet generator 1818 may generate machine event packets, which may include a RIC, the type of event, an interrupt number, retired instruction pointer value, or code segment register value.

[00179] By including the RIC in the LDC packets, system 1800 might handle out-of-order memory execution on modern processors (the data packets) and handle event injection for the offline replay appropriately at the correct program execution point (the machine event packets).

[00180] FIGURE 20 is flow chart of a method 2000 for data capture, according to embodiments of the present disclosure.

[00181] Method 2000 may be initiated by any suitable criteria. Furthermore, although Method 2000 describes operation of particular elements, method 2000 may be performed by any suitable combination or type of elements. For example, method 2000 may be implemented by the elements illustrated in FIGURES 1-19 or any other system operable to implement method 2000. As such, the preferred initialization point for method 2000 and the order of the elements comprising Method 2000 may depend on the implementation chosen. In some embodiments, some elements may

be optionally omitted, reorganized, repeated, or combined. Moreover, portions of method 2000 may be executed in parallel within itself.

[00182] At 2005, an instruction may be fetched and executed. The instruction may be to monitor execution of a thread in using a selective and light data capture mechanism. The instruction may be fetched, decoded, scheduled, executed, and retired. Similar instructions may be fetched and executed to stop or adjust the logging and monitoring of execution.

[00183] At 2010, initial system and register values may be identified and logged.

[00184] At 2015, FL bits associated with logged addresses may be cleared. These may be cleared, for example, in a cache or fill buffer.

[00185] At 2020, it may be determined whether, during execution of the thread, there has been an attempted read of an address. If so, method 2000 may proceed to 2025. Otherwise, method 2000 may proceed to 2040.

[00186] At 2025, it may be determined whether an FL bit associated with the attempted read of the address is clear. The FL bit may be associated with the address of the attempted read through, for example, a fill buffer or a line in a cache. If the FL bit is clear, method 2000 may proceed to 2035. Otherwise, at 2030 the attempted read may be ignored for the purposes of logging and monitoring. In such a case, method 2000 may proceed to 2040. At 2035, the read may be logged. The address and the value may be logged. An FL bit associated with the address may be set, indicating that the first read value has been logged.

[00187] At 2040, it may be determined whether any one of several different conditions has occurred, such as whether a remotely produced data transaction has affected an address, an interrupt is received, a non-deterministic output has been generated, and a machine event has occurred, an I/O event has occurred, or an uncacheable address has been accessed. If so, at 2045 the event may be logged. If applicable, an FL bit associated with the received event may be reset. This may be performed in association with a cacheline eviction. If not, method 2000 may proceed to 2050, bypassing 2045.

[00188] At 2050, it may be determined whether a cache line has been evicted or invalidated. If so, at 2055 associated FL bits may be cleared. Otherwise, method 2000 may bypass 2055 and proceed to 2060.

[00189] At 2060, it may be determined whether an execution mode switch or a context switch has been received. If so, at 2065 associated FL bits may be cleared if needed. This may be performed for lines modified while out of context. Otherwise, method 2000 may bypass 2065 and proceed to 2070.

[00190] At 2070, it may be determined whether the log is to be picked up by software so that a debugger may reconstruct execution. Such a determination may be made, for example, periodically, after a certain number of events have been logged, or on-demand from software. 2070 may be performed by application software. If the log is to be sent, then at 2075 the log may be read. Otherwise, method 2000 may proceed to 2080 while bypassing 2075.

[00191] At 2080, it may be determined whether method 2000 will continue executing or terminate. Method 2000 may repeat at, for example, 2020.

[00192] 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.

[00193] 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.

[00194] 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.

[00195] 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.

[00196] 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.

[00197] 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.

[00198] 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.

[00199] 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.

[00200] Embodiments of the present disclosure include a processor, comprising circuitry to receive an instruction to monitor execution of a thread. In combination with any of the above embodiments, the processor may include a decoder including circuitry to decode the instruction. In combination with any of the above embodiments, the processor may include a scheduler including circuitry to schedule the instruction. In combination with any of the above embodiments, the processor may include a retirement unit including circuitry to retire the instruction. In combination with any of the above embodiments, the processor may include a core including circuitry to, based on execution of the instruction, monitor execution of the thread, identify an attempted read of an address during execution of the thread, determine whether a value at the address was previously read during monitoring of the execution of the thread, log the attempted read based on a determination that the value at the address was not previously read during monitoring of the execution of the thread, and omit logging of the attempted read based on a determination that the value at the address was previously read during monitoring of the execution of the thread. In combination with any of the above embodiments, the processor may include circuitry to determine whether the value at the address was previously read by tracking metadata in a cache associated with the address. In combination with any of the above embodiments, the processor may include circuitry to determine whether the value at the address was previously read by tracking metadata in a fill buffer associated with the address. In combination with any of the above embodiments, the processor may include circuitry to log additional attempted accesses of memory necessary to reconstruct a trace of execution. In combination with any of the above embodiments, the processor may include circuitry to clear an indication that the value at the address was previously read

upon a cache line eviction. In combination with any of the above embodiments, the processor may include circuitry to clear an indication that the value at the address was previously read upon a cache line invalidation. In combination with any of the above embodiments, the processor may include circuitry to receive a previous attempt to read the value at the address, set an indication that the value at the address was previously read based on the previous attempt, and use the indication to determine whether the value at the address was previously read for execution of the attempted read. Embodiments of the present disclosure include methods performed by any of the circuits, processors, or systems in the combinations discussed above. Embodiments of the present disclosure include apparatuses that are implemented with means to perform any of the methods discussed above.

CLAIMS

What is claimed is:

1. A processor, comprising:

a front end including circuitry to receive an instruction to monitor execution of a thread;

a decoder including circuitry to decode the instruction;

a scheduler including circuitry to schedule the instruction;

a retirement unit including circuitry to retire the instruction; and

a core including circuitry to, based on execution of the instruction:

monitor execution of the thread;

identify an attempted read of an address during execution of the thread;

determine whether a value at the address was previously read during monitoring of the execution of the thread;

log the attempted read based on a determination that the value at the address was not previously read during monitoring of the execution of the thread; and

omit logging of the attempted read based on a determination that the value at the address was previously read during monitoring of the execution of the thread.

2. The processor of Claim 1, wherein the core further includes circuitry to determine whether the value at the address was previously read by tracking metadata in a cache associated with the address.

3. The processor of Claim 1, wherein the core further includes circuitry to determine whether the value at the address was previously read by tracking metadata in a fill buffer associated with the address.

4. The processor of Claim 1, wherein the core further includes circuitry to log additional attempted accesses of memory necessary to reconstruct a trace of execution.

5. The processor of Claim 1, wherein the core further includes circuitry to clear an indication that the value at the address was previously read upon a cache line eviction.

6. The processor of Claim 1, wherein the core further includes circuitry to clear an indication that the value at the address was previously read upon a cache line invalidation.

7. The processor of Claim 1, wherein the core further includes circuitry to:

receive a previous attempt to read the value at the address;

set an indication that the value at the address was previously read based on the previous attempt; and

use the indication to determine whether the value at the address was previously read for execution of the attempted read.

8. A system, comprising:

a processor;

a front end including circuitry to receive an instruction to monitor execution of a thread;

a decoder including circuitry to decode the instruction;

a scheduler including circuitry to schedule the instruction;

a retirement unit including circuitry to retire the instruction; and

a core including circuitry to, based on execution of the instruction:

monitor execution of the thread;

identify an attempted read of an address during execution of the thread;

determine whether a value at the address was previously read during monitoring of the execution of the thread;

log the attempted read based on a determination that the value at the address was not previously read during monitoring of the execution of the thread; and

omit logging of the attempted read based on a determination that the value at the address was previously read during monitoring of the execution of the thread.

9. The system of Claim 8, wherein the core further includes circuitry to determine whether the value at the address was previously read by tracking metadata in a cache associated with the address.

10. The system of Claim 8, wherein the core further includes circuitry to determine whether the value at the address was previously read by tracking metadata in a fill buffer associated with the address.

11. The system of Claim 8, wherein the core further includes circuitry to log additional attempted accesses of memory necessary to reconstruct a trace of execution.

12. The system of Claim 8, wherein the core further includes circuitry to clear an indication that the value at the address was previously read upon a cache line eviction.

13. The system of Claim 8, wherein the core further includes circuitry to clear an indication that the value at the address was previously read upon a cache line invalidation.

14. The system of Claim 8, wherein the core further includes circuitry to:

receive a previous attempt to read the value at the address;

set an indication that the value at the address was previously read based on the previous attempt; and

use the indication to determine whether the value at the address was previously read for execution of the attempted read.

15. A method, comprising:

receiving an instruction to monitor execution of a thread;

decoding the instruction;

scheduling the instruction;

retiring the instruction; and

based on execution of the instruction:

monitoring execution of the thread;

identifying an attempted read of an address during execution of the thread;

determining whether a value at the address was previously read during monitoring of the execution of the thread;

logging the attempted read based on a determination that the value at the address was previously read during monitoring of the execution of the thread; and

omitting logging of the attempted read based on a determination that the value at the address was not previously read during monitoring of the execution of the thread.

16. The method of Claim 15, further comprising determine whether the value at the address was previously read by tracking metadata in a cache associated with the address.

17. The method of Claim 15, further comprising determine whether the value at the address was previously read by tracking metadata in a fill buffer associated with the address.

18. The method of Claim 15, further comprising logging additional attempted accesses of memory necessary to reconstruct a trace of execution.

19. The method of Claim 15, further comprising clearing an indication that the value at the address was previously read upon a cache line eviction.

20. The method of Claim 15, further comprising clearing an indication that the value at the address was previously read upon a cache line invalidation.

21. The method of Claim 15, further comprising:

receiving a previous attempt to read the value at the address;

setting an indication that the value at the address was previously read based on the previous attempt; and

using the indication to determine whether the value at the address was previously read for execution of the attempted read.

22. An apparatus, comprising means for performing any of the methods of Claims 15-21.

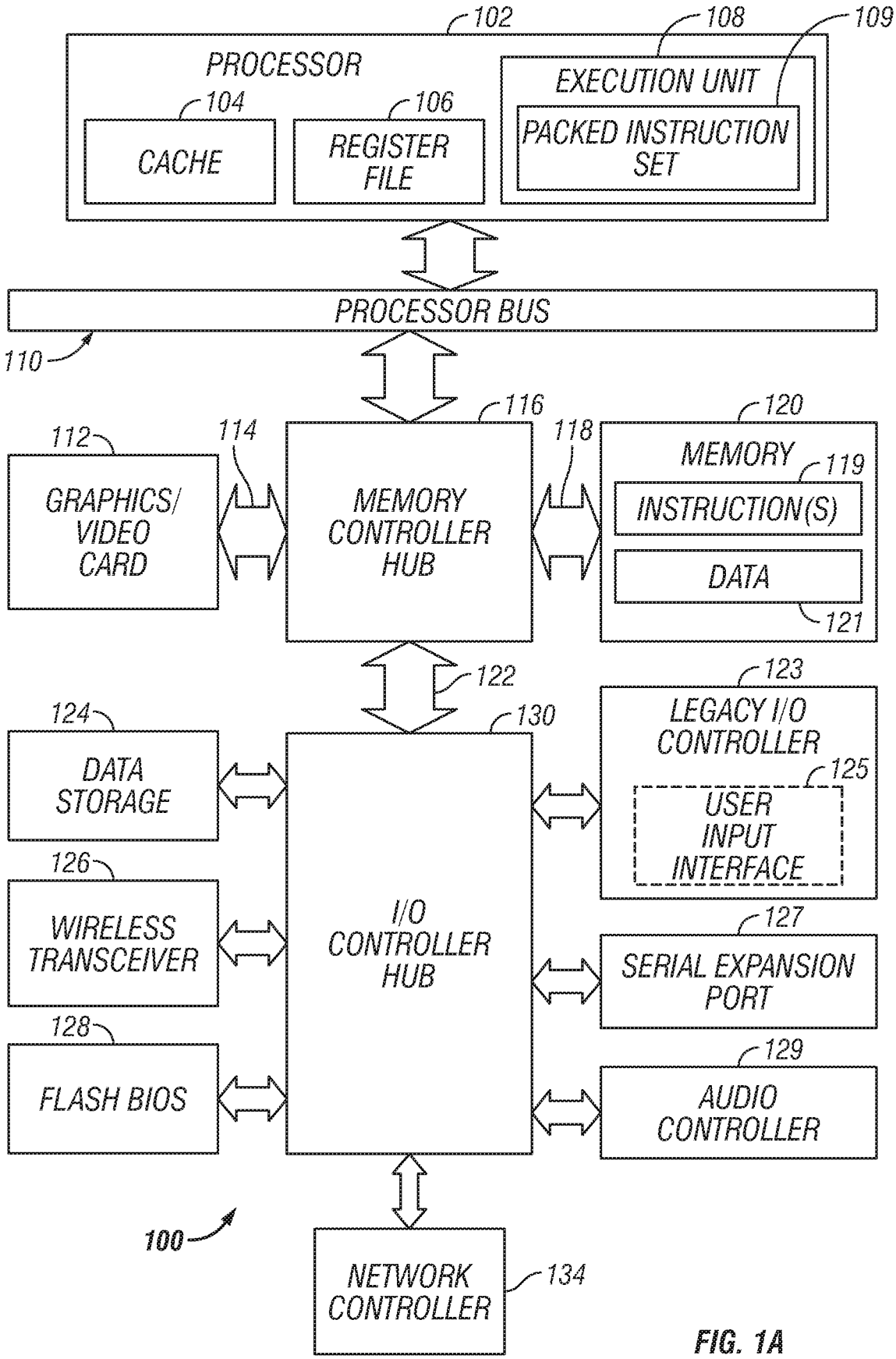

102
PROCESSOR
104
CACHE
106
REGISTER FILE
108
EXECUTION UNIT
109
PACKED INSTRUCTION SET
PROCESSOR BUS
110
112
GRAPHICS/ VIDEO CARD
114
116
MEMORY CONTROLLER HUB
118
120
MEMORY
119
INSTRUCTION(S)
DATA
121
122
123
LEGACY I/O CONTROLLER
125
USER INPUT INTERFACE
124
DATA STORAGE
130
126
WIRELESS TRANSCEIVER
I/O CONTROLLER HUB
127
SERIAL EXPANSION PORT
128
FLASH BIOS
129
AUDIO CONTROLLER
100
NETWORK CONTROLLER
134

FIG. 1A

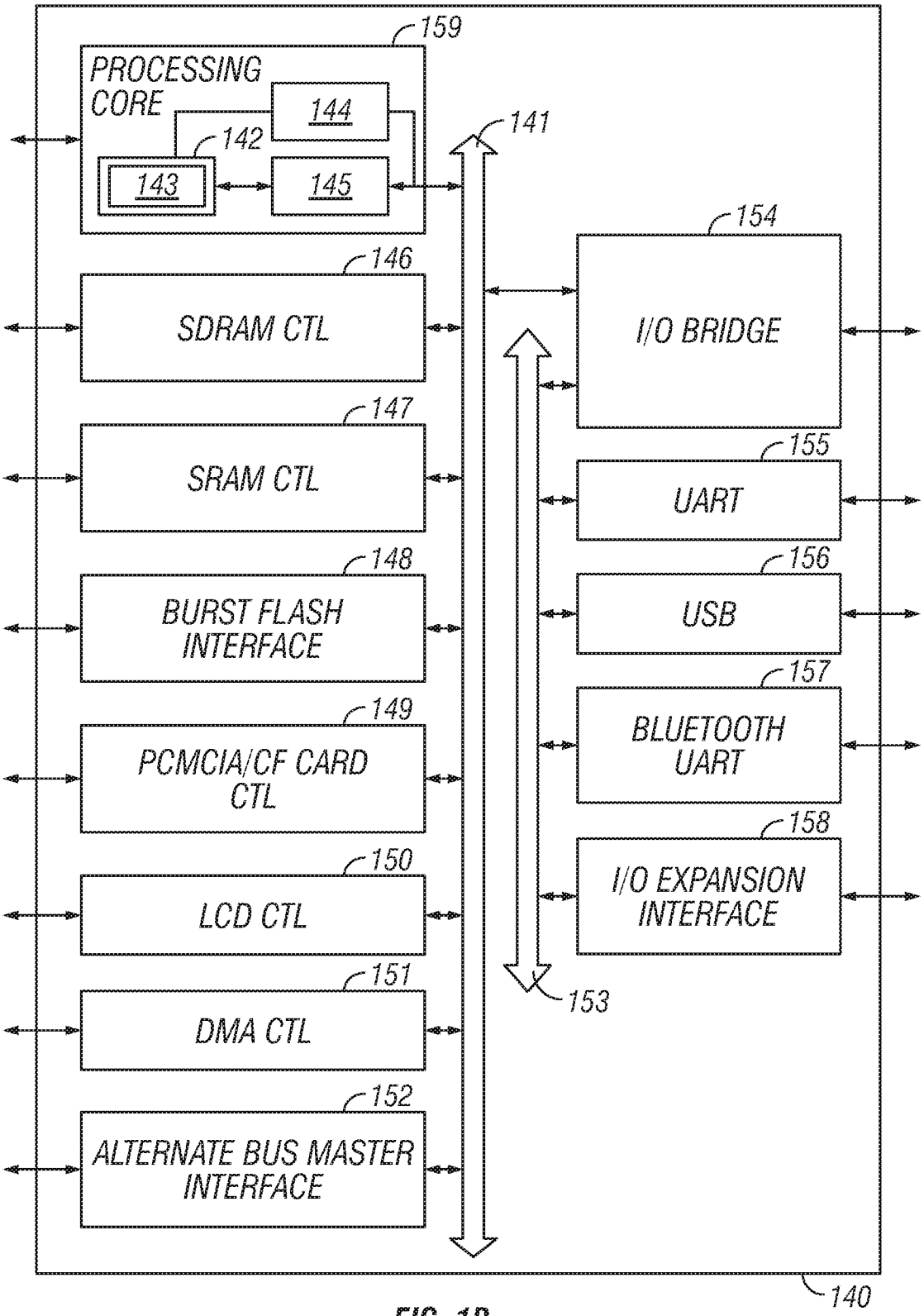

159
PROCESSING CORE
144
142
143
145
141
154
146
SDRAM CTL
I/O BRIDGE
147
SRAM CTL
155
UART
148
BURST FLASH INTERFACE
156
USB
157
BLUETOOTH UART
149
PCMCIA/CF CARD CTL
158
I/O EXPANSION INTERFACE
150
LCD CTL
153
151
DMA CTL
152
ALTERNATE BUS MASTER INTERFACE
140

FIG. 1B

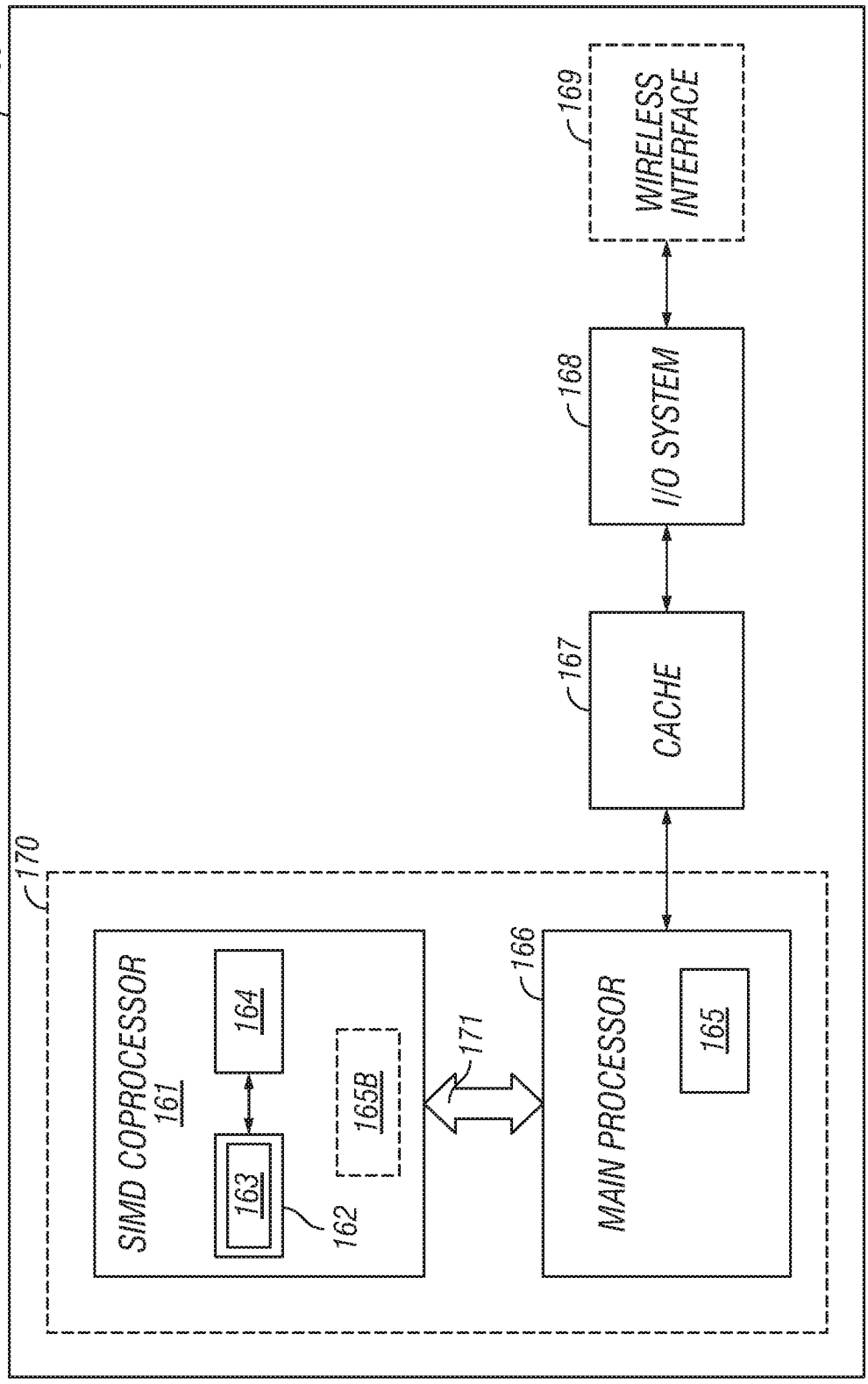

160
169
WIRELESS INTERFACE
168
I/O SYSTEM
167
CACHE
170
SIMD COPROCESSOR
161
164
163
162
165B
171
166
MAIN PROCESSOR
165

FIG. 1C

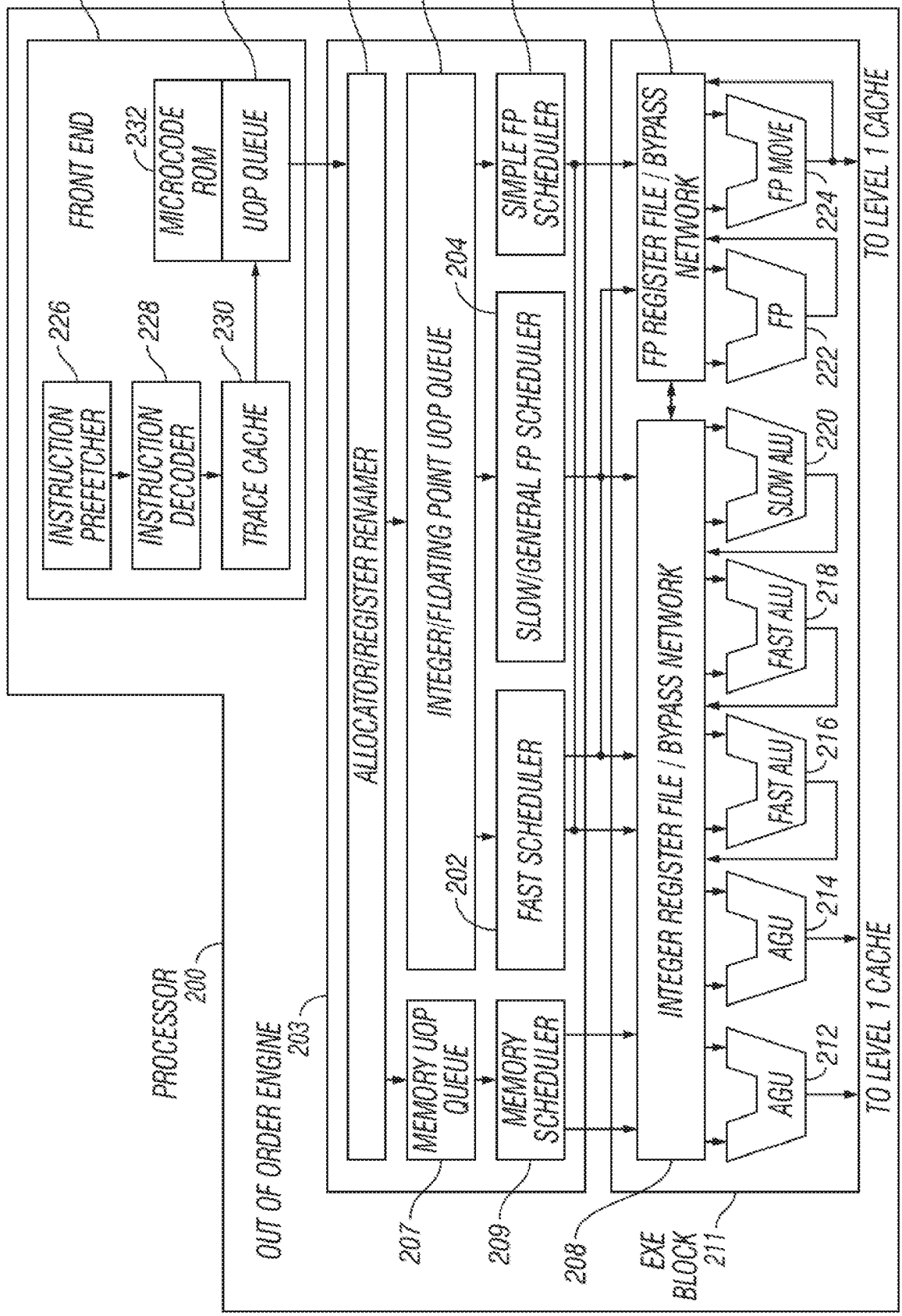

PROCESSOR
200
OUT OF ORDER ENGINE
203
FRONT END
201
INSTRUCTION PREFETCHER
226
INSTRUCTION DECODER
228
TRACE CACHE
230
MICROCODE ROM
232
UOP QUEUE
234
ALLOCATOR/REGISTER RENAMER
215
INTEGER/FLOATING POINT UOP QUEUE
205
202
204
MEMORY UOP QUEUE
207
FAST SCHEDULER
SLOW/GENERAL FP SCHEDULER
SIMPLE FP SCHEDULER
206
MEMORY SCHEDULER
209
INTEGER REGISTER FILE / BYPASS NETWORK
208
FP REGISTER FILE / BYPASS NETWORK
210
EXE BLOCK
211
AGU
212
AGU
214
FAST ALU
216
FAST ALU
218
SLOW ALU
220
FP
222
FP MOVE
224
TO LEVEL 1 CACHE
TO LEVEL 1 CACHE

FIG. 2

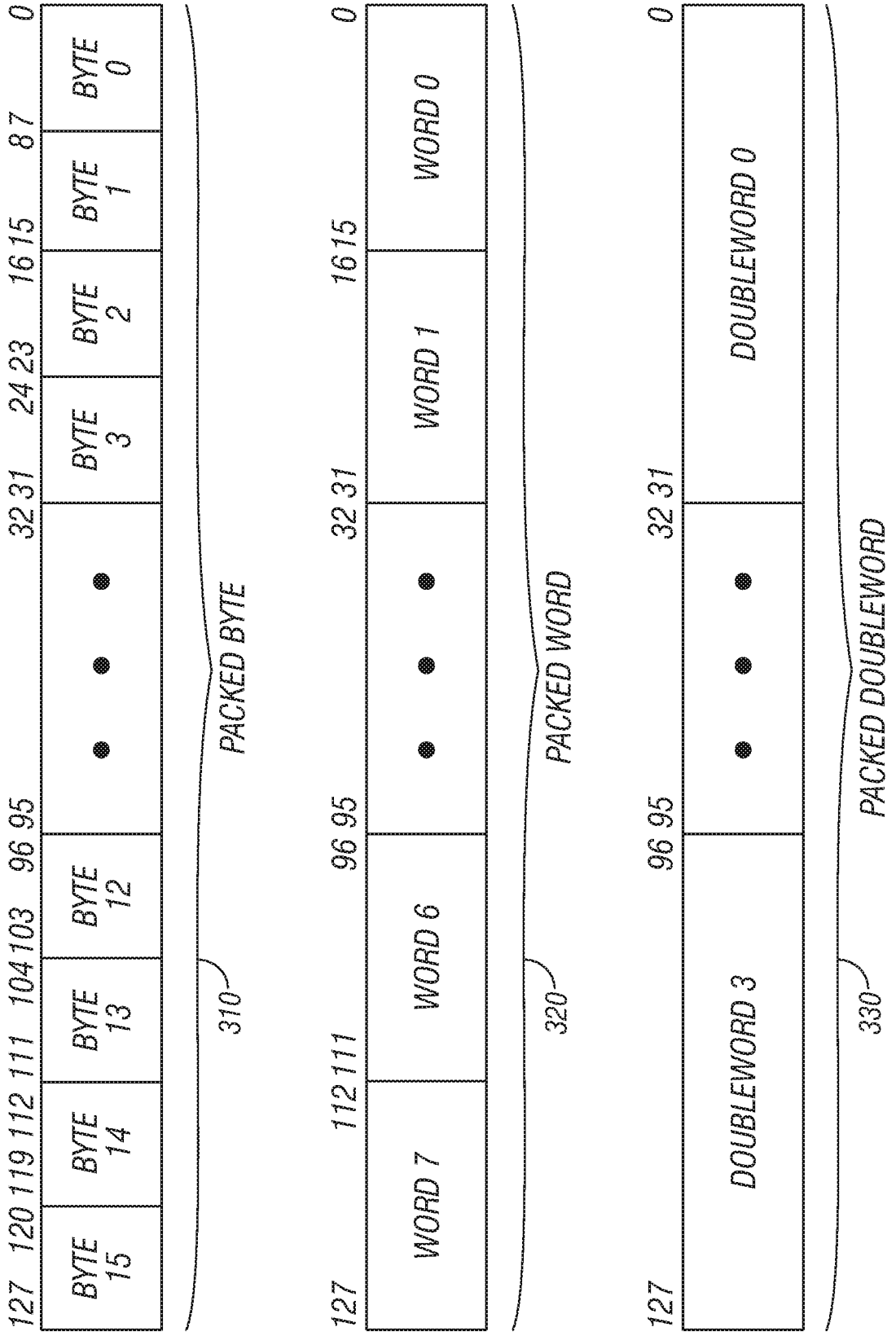

127 120 119 112 111 104 103 96 95 32 31 24 23 16 15 8 7 0
BYTE 15
BYTE 14
BYTE 13
BYTE 12
BYTE 3
BYTE 2
BYTE 1
BYTE 0
PACKED BYTE
310
127 112 111 96 95 32 31 16 15 0
WORD 7
WORD 6
WORD 1
WORD 0
PACKED WORD
320
127 96 95 32 31 0
DOUBLEWORD 3
DOUBLEWORD 0
PACKED DOUBLEWORD
330

FIG. 3A

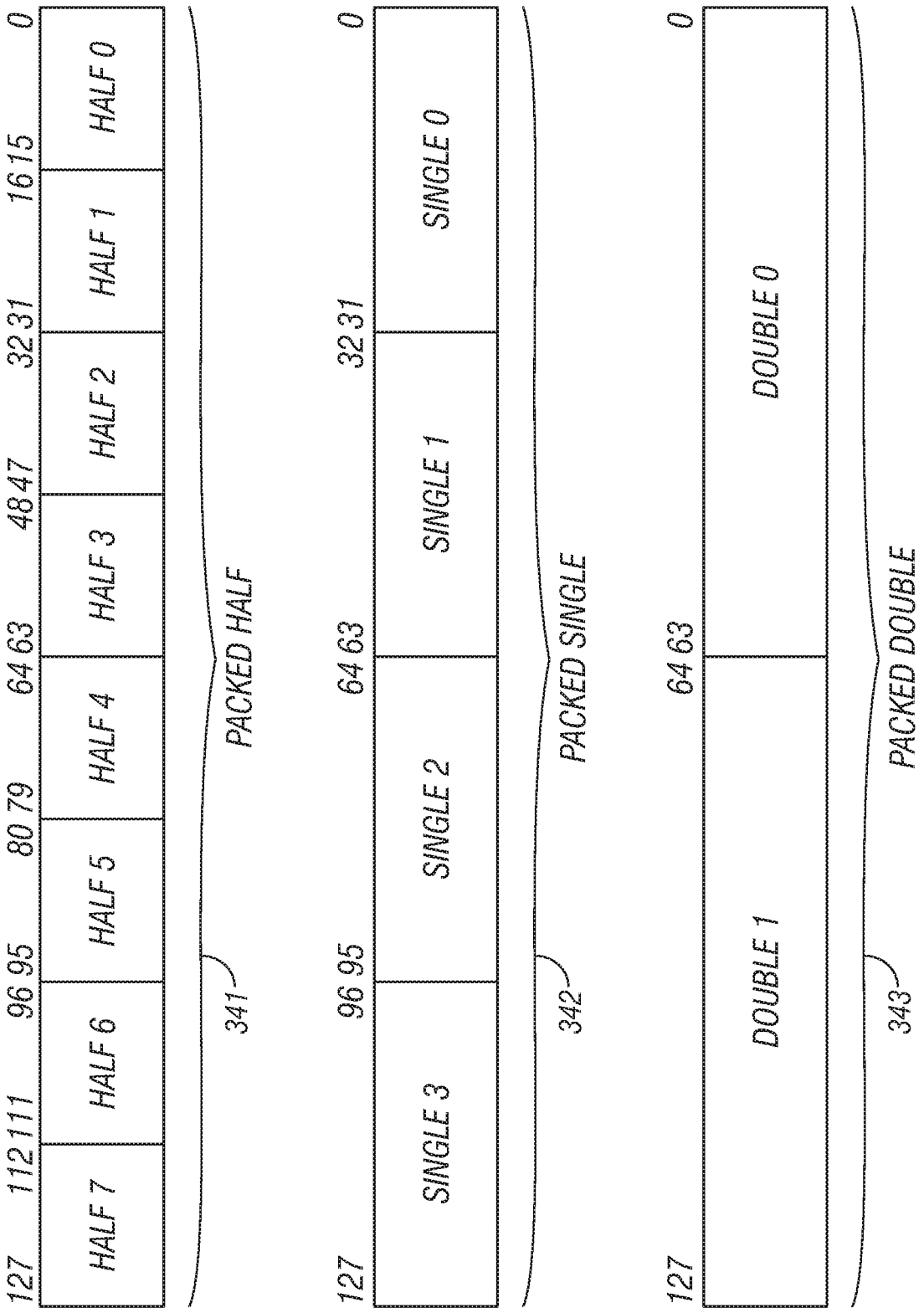
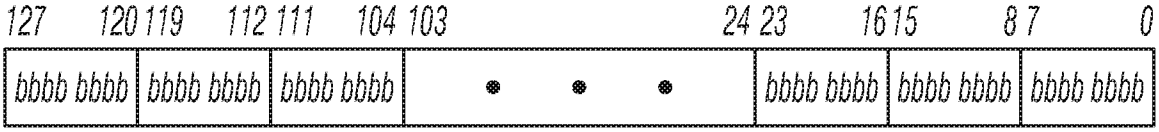
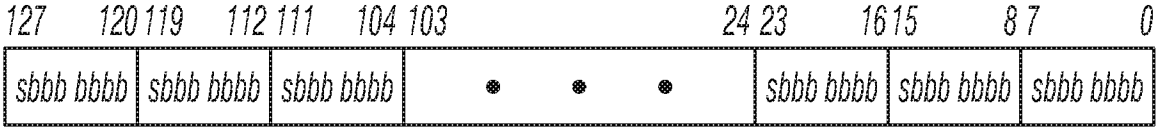

127
112 111
96 95
80 79
64 63
48 47
32 31
16 15
0
HALF 7
HALF 6
HALF 5
HALF 4
HALF 3
HALF 2
HALF 1
HALF 0
341
PACKED HALF
127
96 95
64 63
32 31
0
SINGLE 3
SINGLE 2
SINGLE 1
SINGLE 0
342
PACKED SINGLE
127
64 63
0
DOUBLE 1
DOUBLE 0
343
PACKED DOUBLE

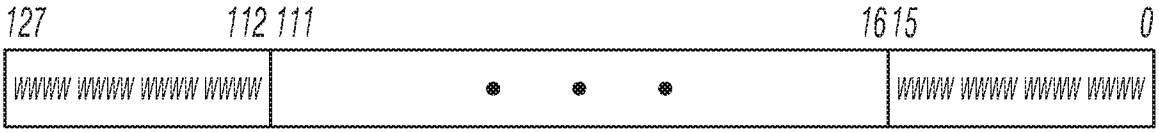
FIG. 3B



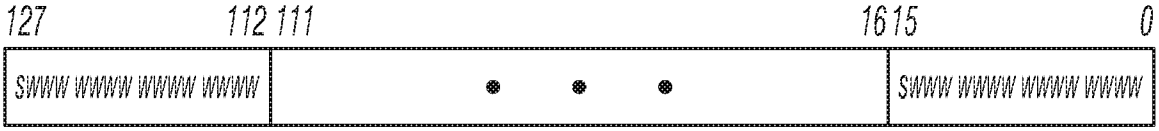
UNSIGNED PACKED BYTE REPRESENTATION 344



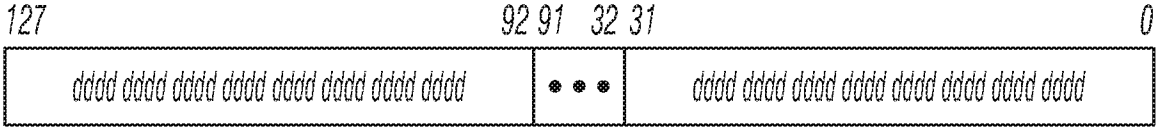
SIGNED PACKED BYTE REPRESENTATION 345



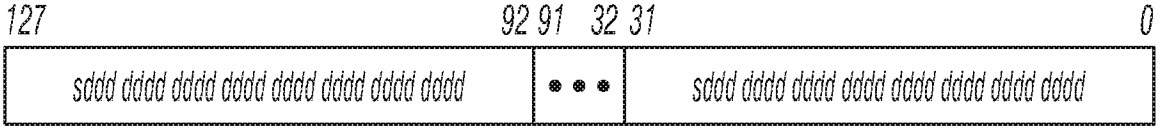
UNSIGNED PACKED WORD REPRESENTATION 346



SIGNED PACKED WORD REPRESENTATION 347



UNSIGNED PACKED DOUBLEWORD REPRESENTATION 348



SIGNED PACKED DOUBLEWORD REPRESENTATION 349

FIG. 3C

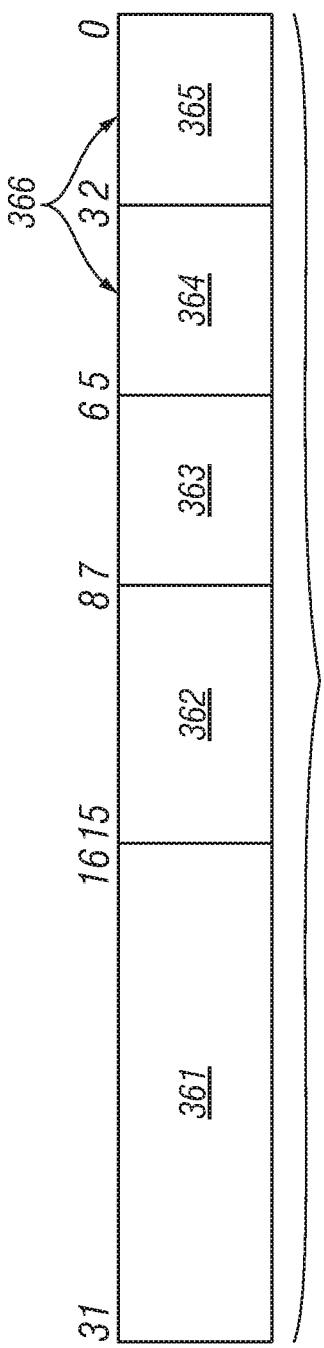


FIG. 3D

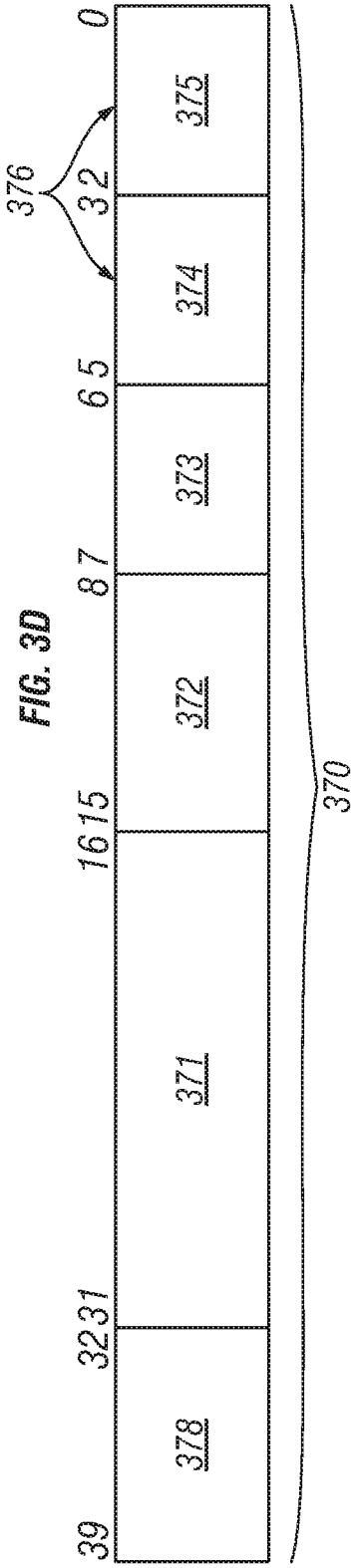


FIG. 3E

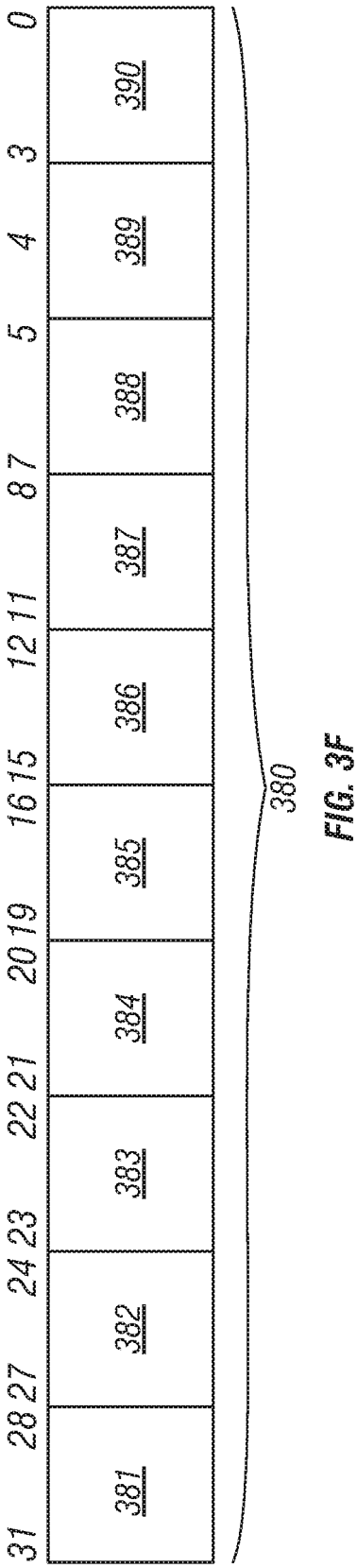


FIG. 3F

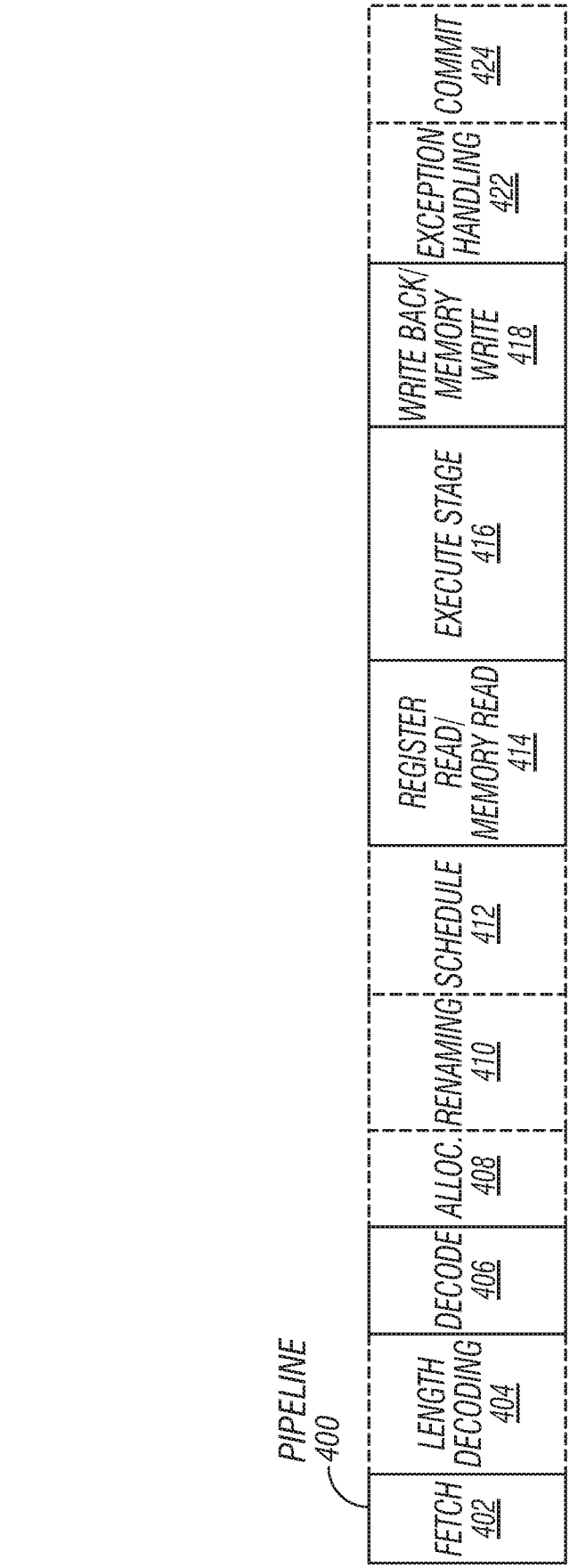

PIPELINE
400
FETCH
402
LENGTH DECODING
404
DECODE
406
ALLOC.
408
RENAMING
410
SCHEDULE
412
REGISTER READ/ MEMORY READ
414
EXECUTE STAGE
416
WRITE BACK/ MEMORY WRITE
418
EXCEPTION HANDLING
422
COMMIT
424

FIG. 4A

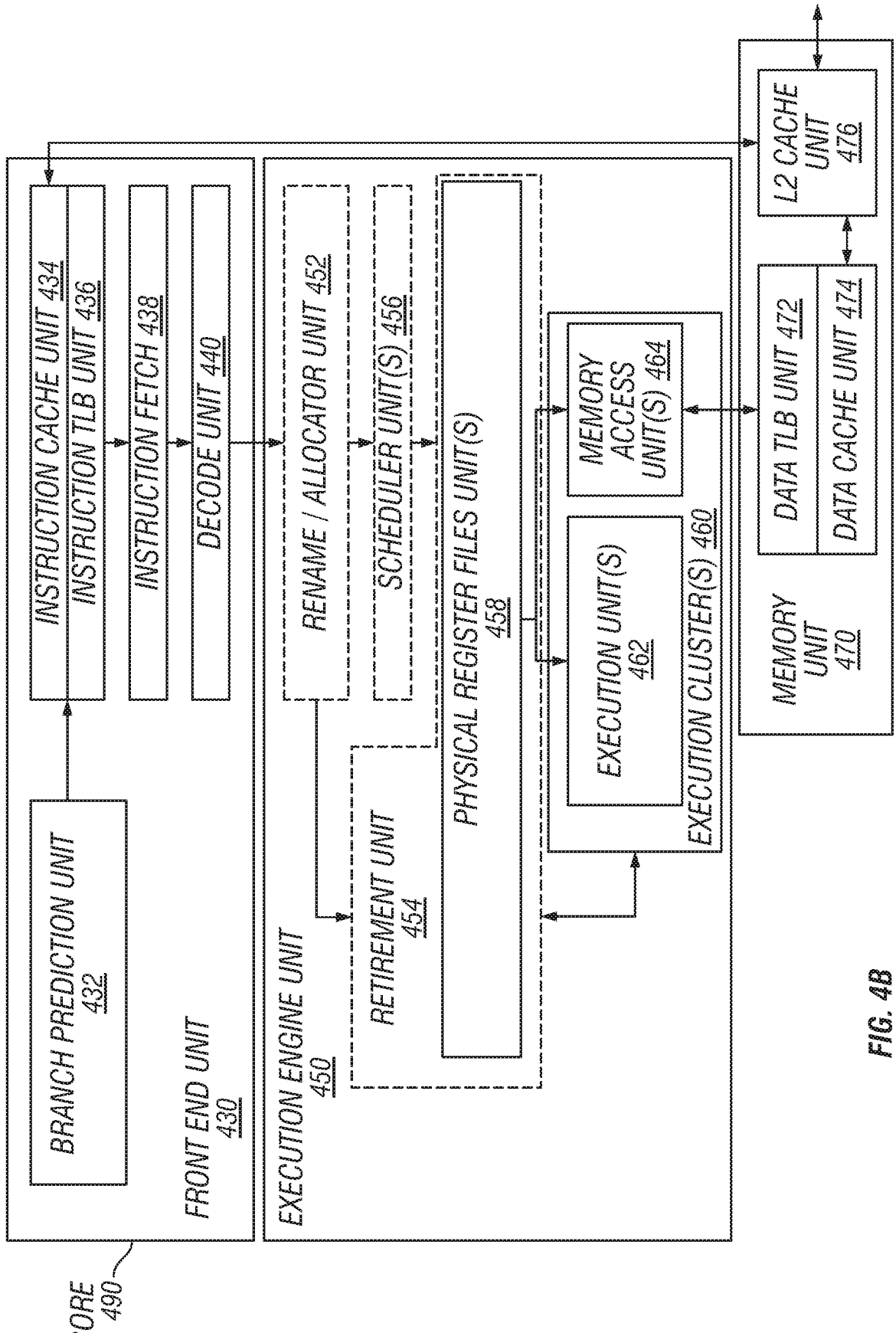

CORE 490
FRONT END UNIT 430
BRANCH PREDICTION UNIT 432
INSTRUCTION CACHE UNIT 434
INSTRUCTION TLB UNIT 436
INSTRUCTION FETCH 438
DECODE UNIT 440
EXECUTION ENGINE UNIT 450
RENAME / ALLOCATOR UNIT 452
RETIREMENT UNIT 454
SCHEDULER UNIT(S) 456
PHYSICAL REGISTER FILES UNIT(S) 458
EXECUTION CLUSTER(S) 460
EXECUTION UNIT(S) 462
MEMORY ACCESS UNIT(S) 464
MEMORY UNIT 470
DATA TLB UNIT 472
DATA CACHE UNIT 474
L2 CACHE UNIT 476

FIG. 4B

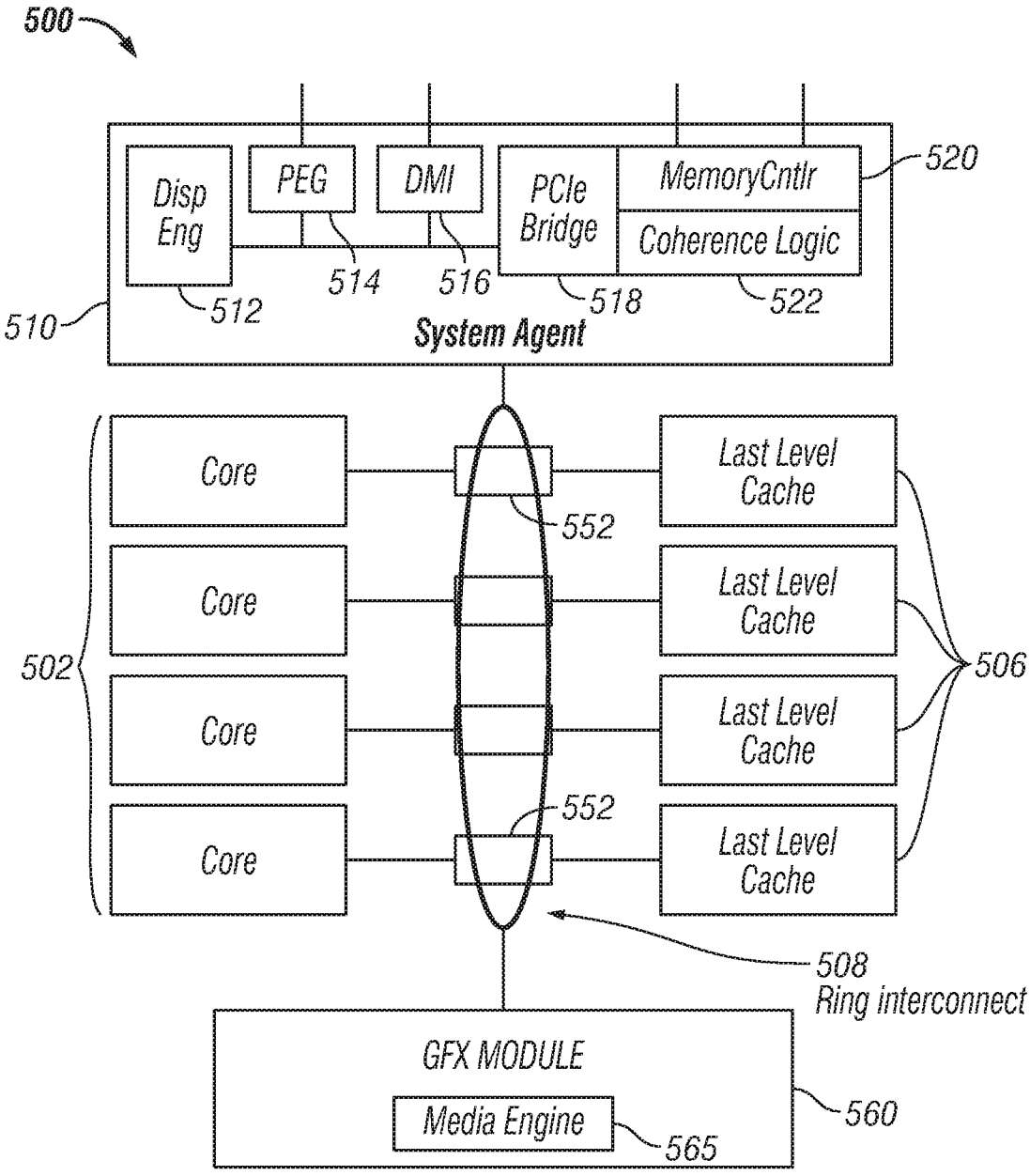

500
Disp Eng
PEG
DMI
PCIe Bridge
MemoryCntlr
Coherence Logic
520
514
516
518
522
512
510
System Agent
Core
Core
Core
Core
502
552
552
Last Level Cache
Last Level Cache
Last Level Cache
Last Level Cache
506
508
Ring interconnect
GFX MODULE
Media Engine
565
560

FIG. 5A

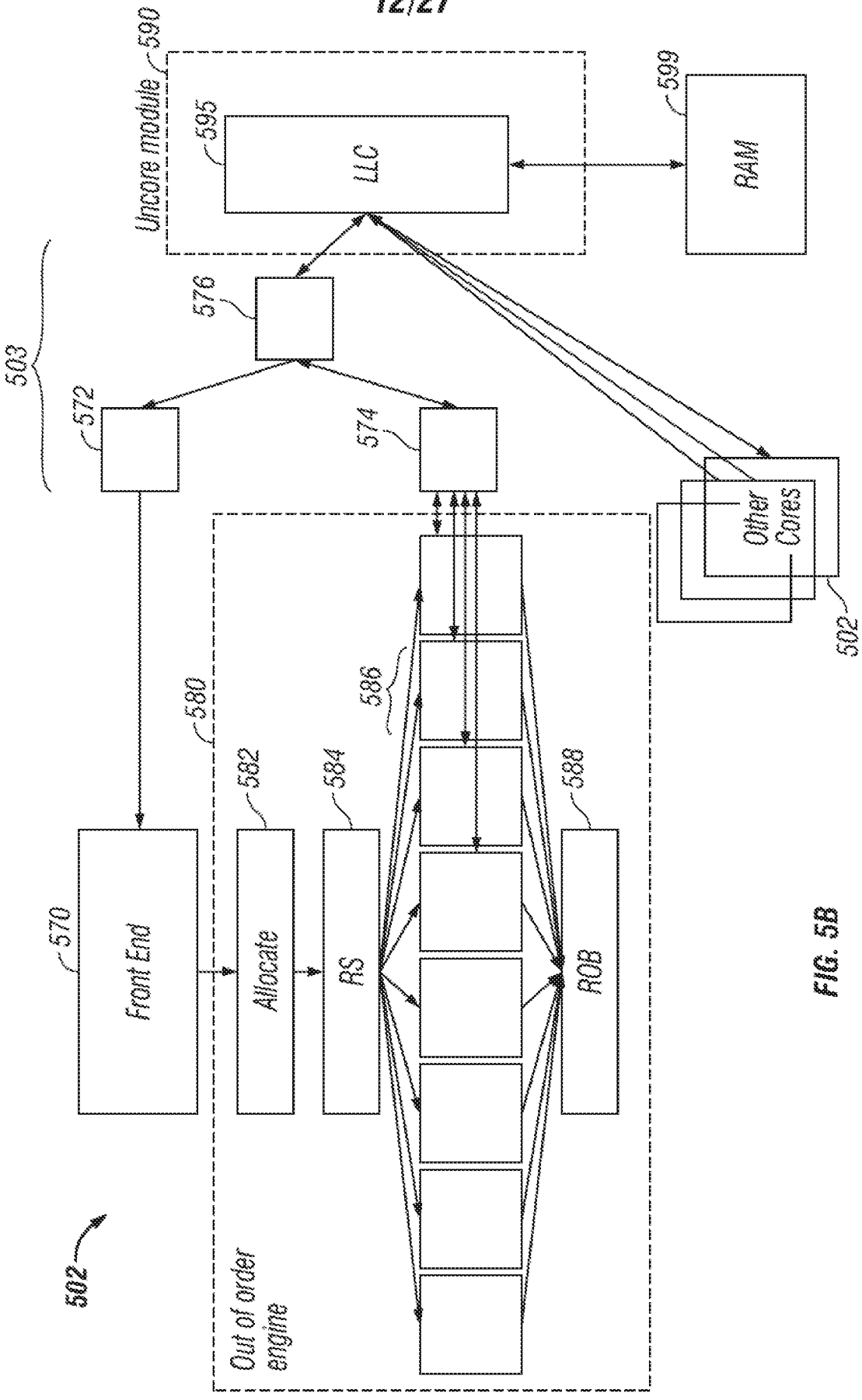

Uncore module
590
595
LLC
599
RAM
576
503
572
574
Other Cores
502
580
586
582
584
588
570
Front End
Allocate
RS
ROB
Out of order engine
502

FIG. 5B

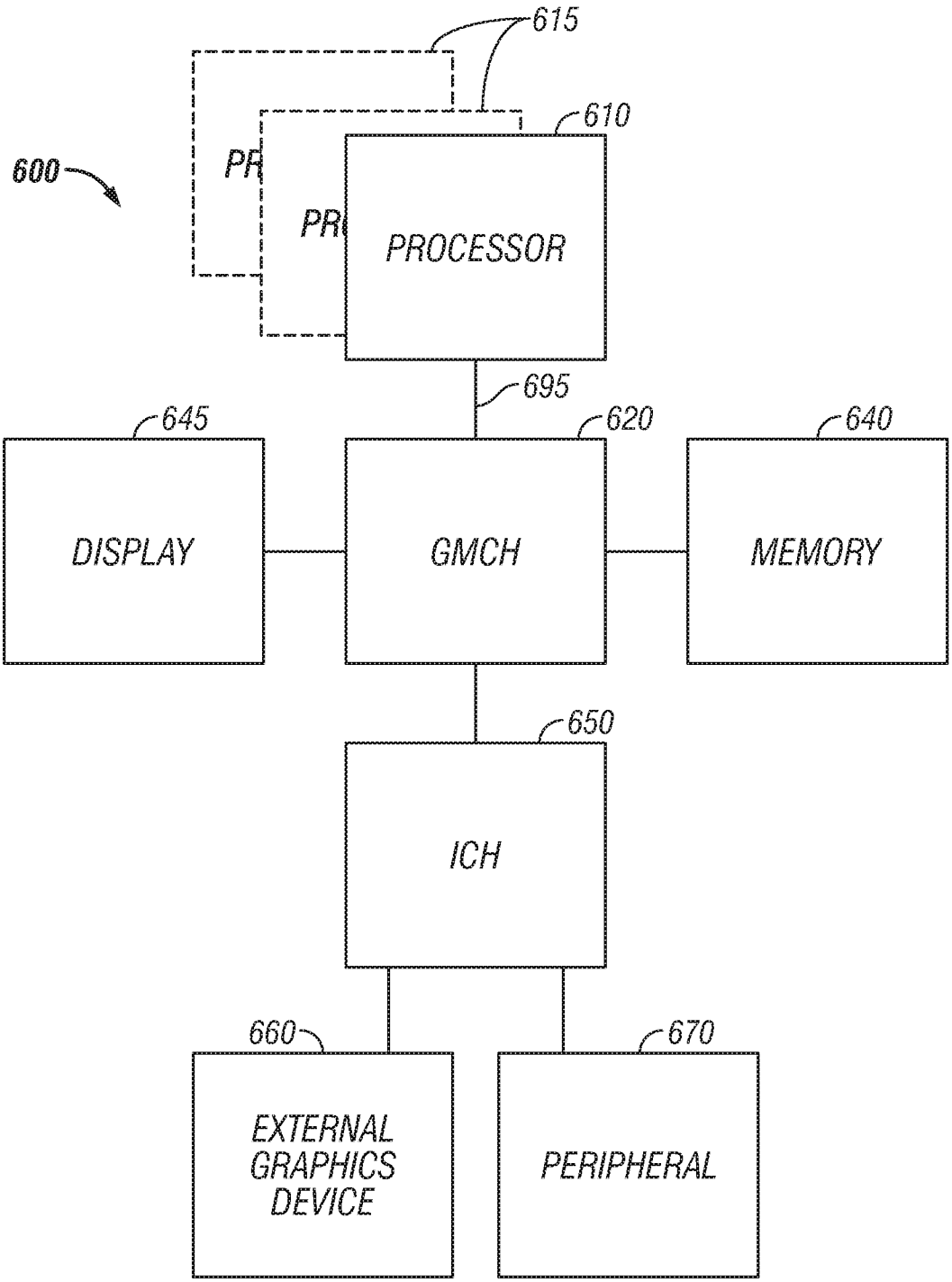

600
615
610
PR
PR
PROCESSOR
695
645
620
640
DISPLAY
GMCH
MEMORY
650
ICH
660
670
EXTERNAL GRAPHICS DEVICE
PERIPHERAL

FIG. 6

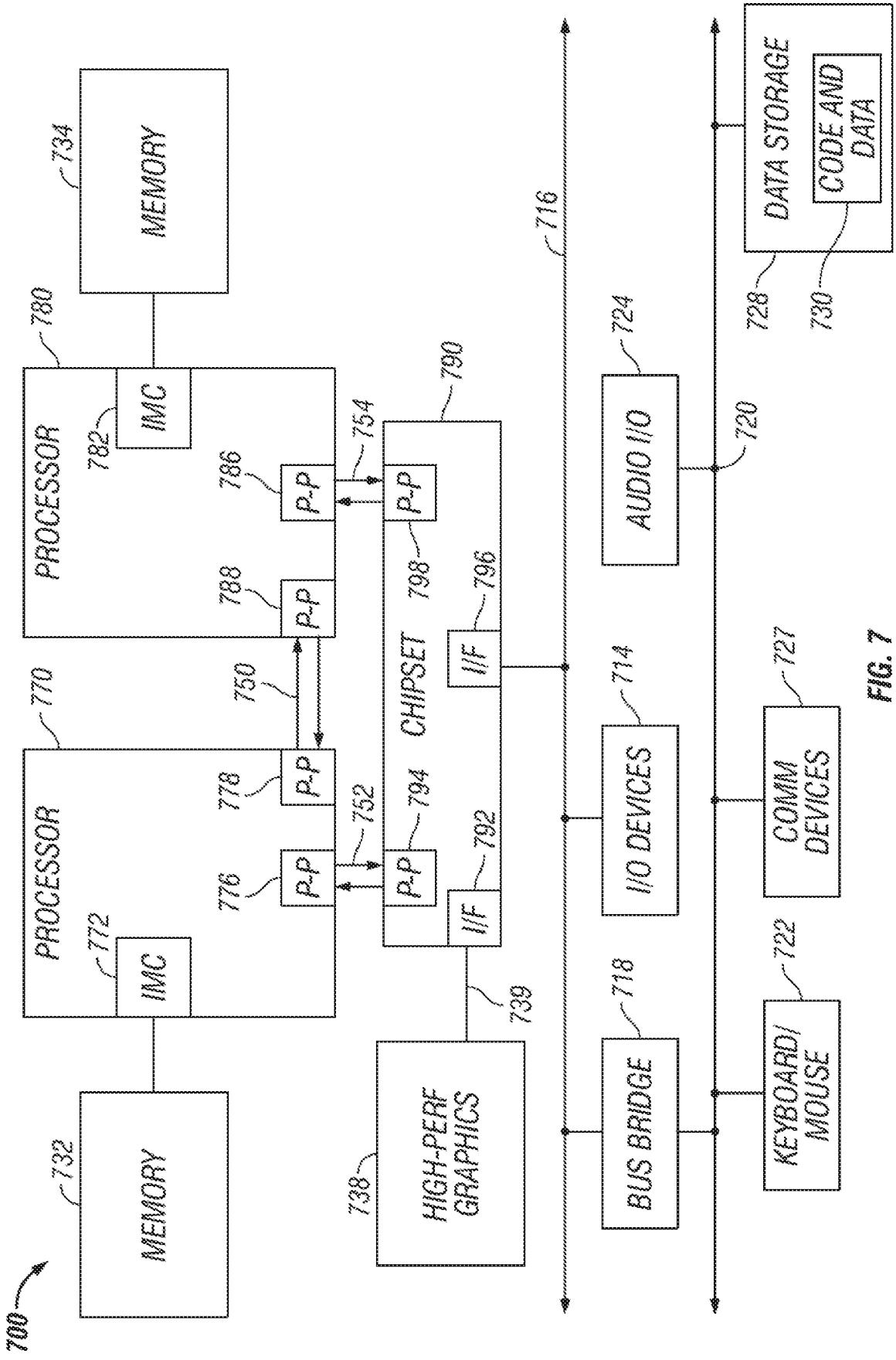

700
MEMORY
732
MEMORY
734
PROCESSOR
770
IMC
772
PROCESSOR
780
IMC
782
P-P
776
P-P
778
P-P
786
P-P
788
750
752
754
CHIPSET
790
P-P
794
P-P
798
I/F
792
I/F
796
739
HIGH-PERF GRAPHICS
738
BUS BRIDGE
718
I/O DEVICES
714
AUDIO I/O
724
716
720
KEYBOARD/ MOUSE
722
COMM DEVICES
727
DATA STORAGE
728
CODE AND DATA
730

FIG. 7

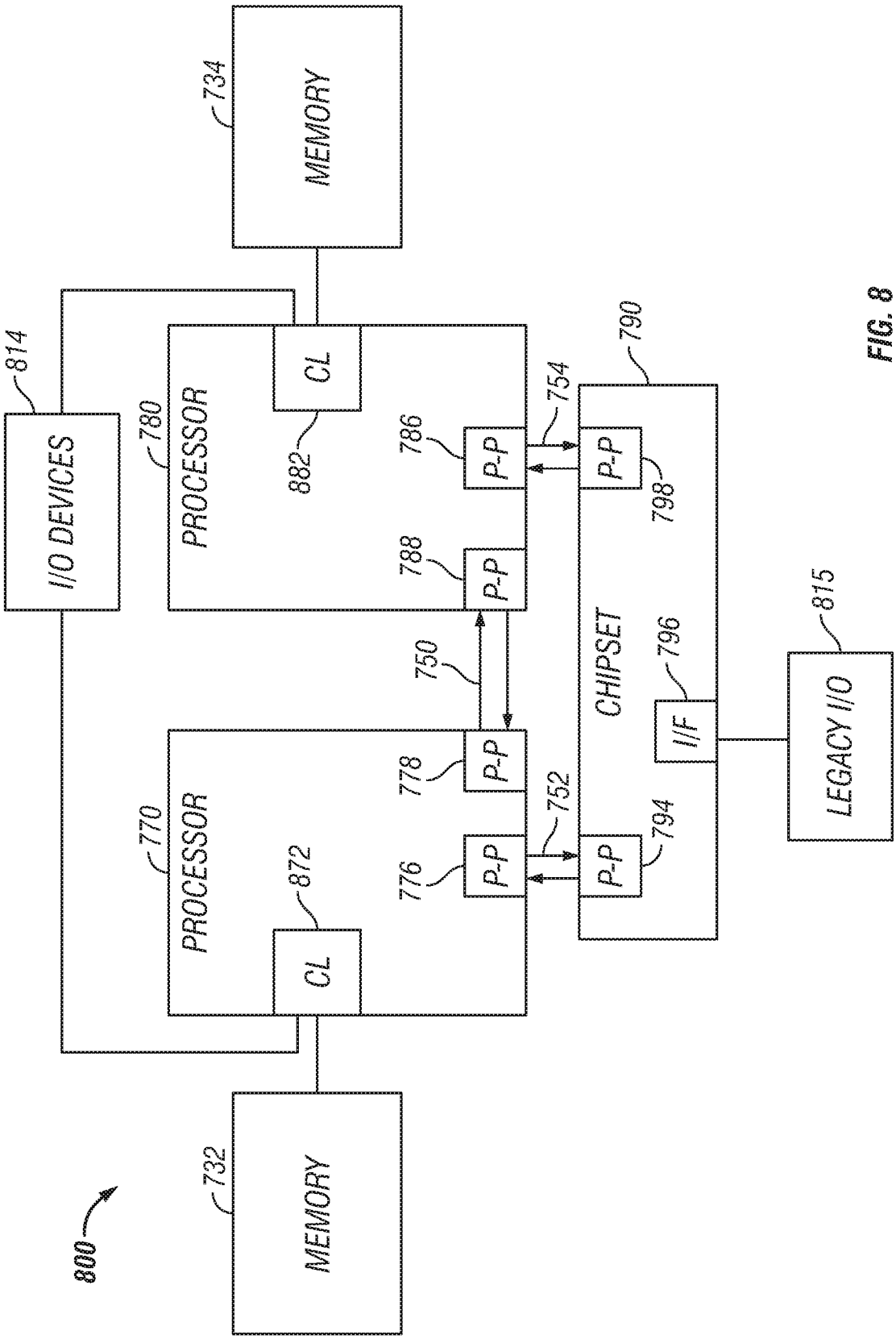

800
MEMORY
732
MEMORY
734
I/O DEVICES
814
PROCESSOR
770
CL
872
PROCESSOR
780
CL
882
P-P
776
P-P
778
P-P
786
P-P
788
750
752
754
CHIPSET
790
P-P
794
P-P
798
I/F
796
LEGACY I/O
815

FIG. 8

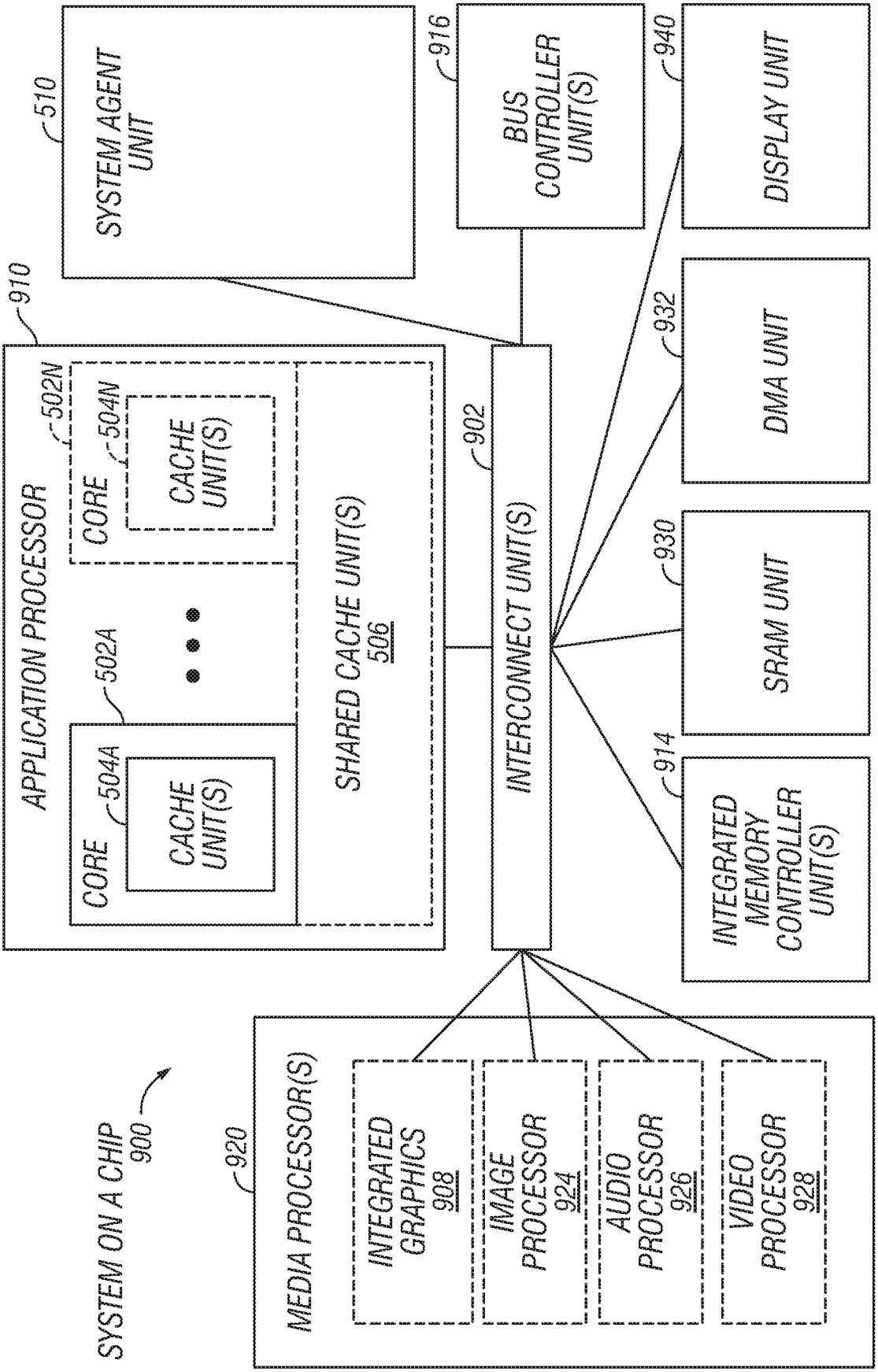

SYSTEM ON A CHIP
900
MEDIA PROCESSOR(S)
920
INTEGRATED GRAPHICS
908
IMAGE PROCESSOR
924
AUDIO PROCESSOR
926
VIDEO PROCESSOR
928
APPLICATION PROCESSOR
910
CORE
504A
CACHE UNIT(S)
502A
CORE
504N
CACHE UNIT(S)
SHARED CACHE UNIT(S)
506
INTERCONNECT UNIT(S)
902
SYSTEM AGENT UNIT
510
BUS CONTROLLER UNIT(S)
916
INTEGRATED MEMORY CONTROLLER UNIT(S)
914
SRAM UNIT
930
DMA UNIT
932
DISPLAY UNIT
940

FIG. 9

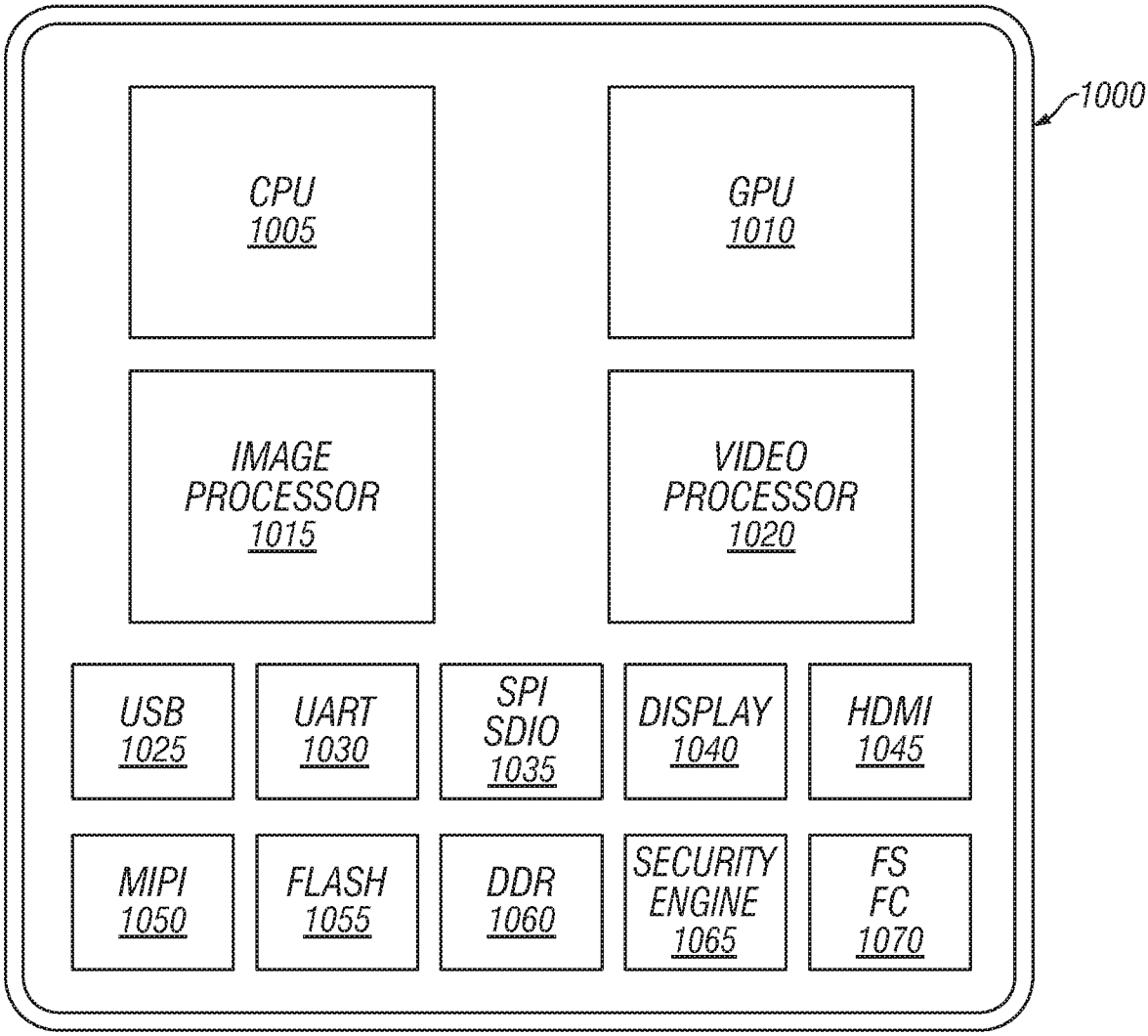

1000
CPU
1005
GPU
1010
IMAGE
PROCESSOR
1015
VIDEO
PROCESSOR
1020
USB
1025
UART
1030
SPI
SDIO
1035
DISPLAY
1040
HDMI
1045
MIPI
1050
FLASH
1055
DDR
1060
SECURITY
ENGINE
1065
FS
FC
1070

FIG. 10

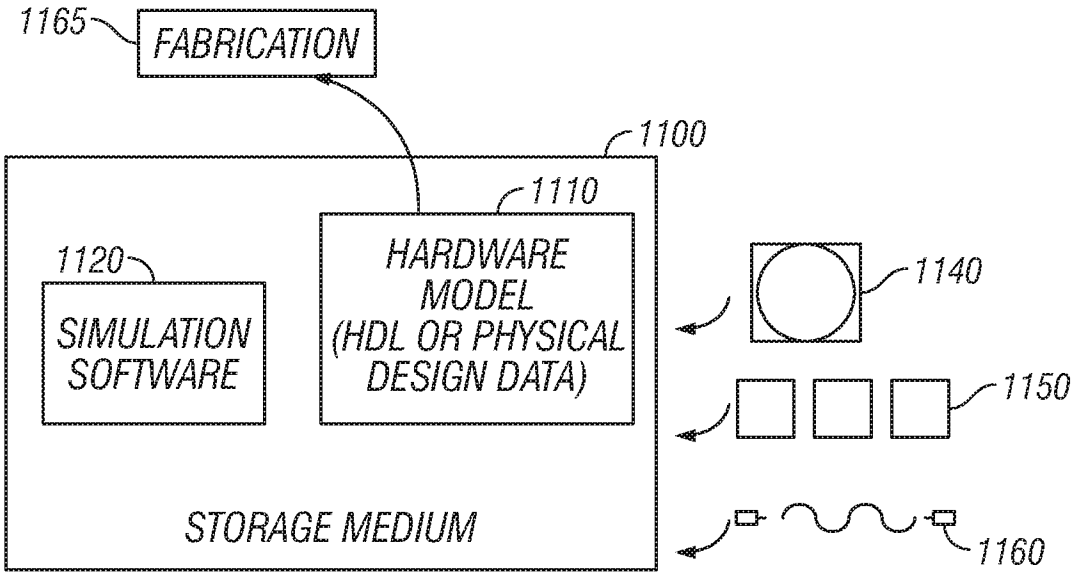

1165
FABRICATION
1100
1110
HARDWARE MODEL (HDL OR PHYSICAL DESIGN DATA)
1120
SIMULATION SOFTWARE
1140
1150
STORAGE MEDIUM
1160

FIG. 11

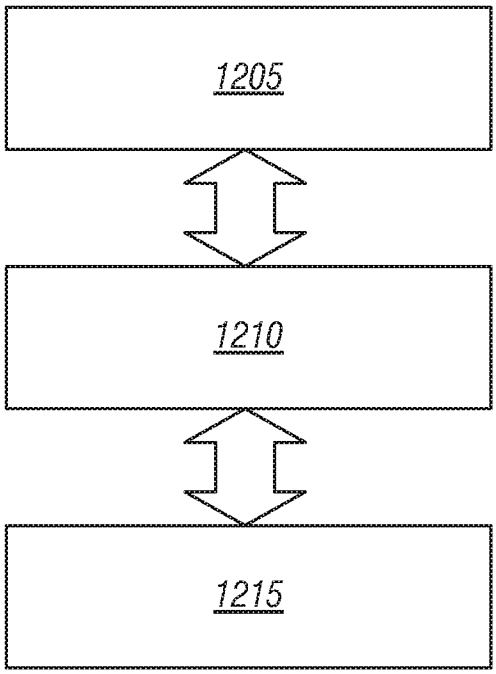

1205
1210
1215

FIG. 12

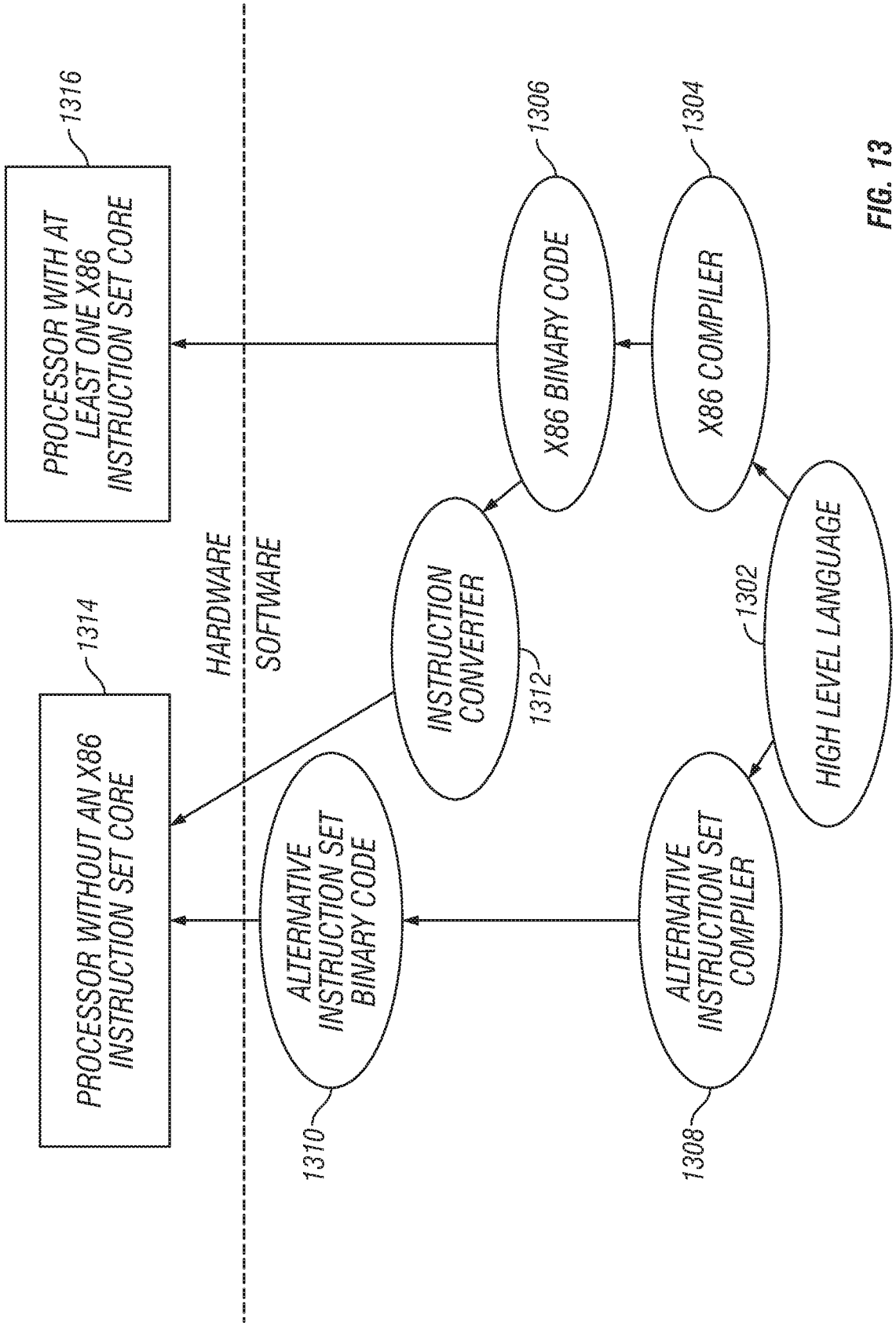

PROCESSOR WITH AT LEAST ONE X86 INSTRUCTION SET CORE
1316
PROCESSOR WITHOUT AN X86 INSTRUCTION SET CORE
1314
HARDWARE
SOFTWARE
X86 BINARY CODE
1306
X86 COMPILER
1304
INSTRUCTION CONVERTER
1312
HIGH LEVEL LANGUAGE
1302
ALTERNATIVE INSTRUCTION SET BINARY CODE
1310
ALTERNATIVE INSTRUCTION SET COMPILER
1308

FIG. 13

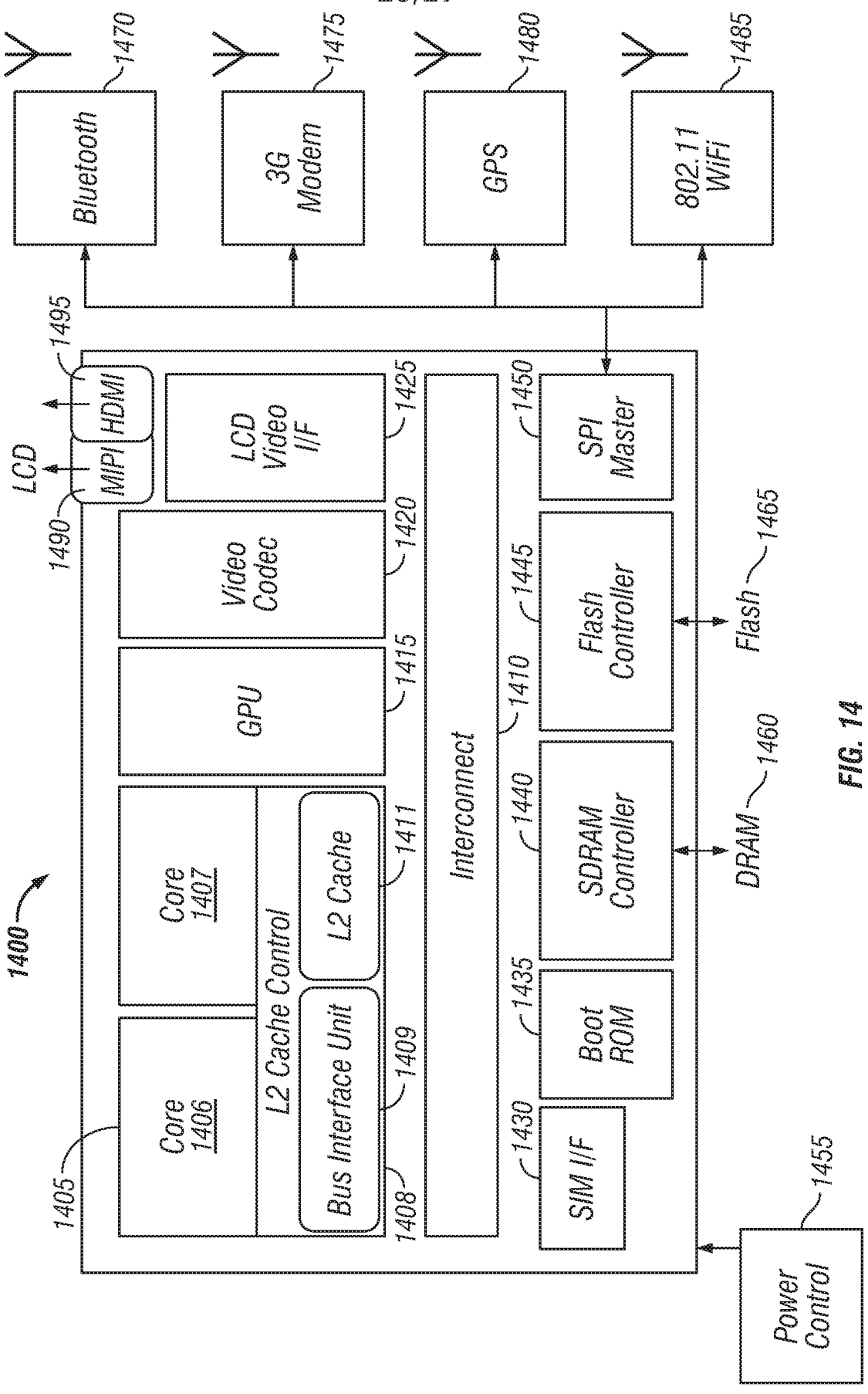

1400
1405
Core 1406
L2 Cache Control
Bus Interface Unit
1409
L2 Cache
1411
Core 1407
1408
GPU
1415
Video Codec
1420
LCD Video I/F
1425
HDMI
MIPI
LCD
1490
1495
Interconnect
1410
SIM I/F
1430
Boot ROM
1435
SDRAM Controller
1440
Flash Controller
1445
SPI Master
1450
DRAM
1460
Flash
1465
Power Control
1455
Bluetooth
1470
3G Modem
1475
GPS
1480
802.11 WiFi
1485

FIG. 14

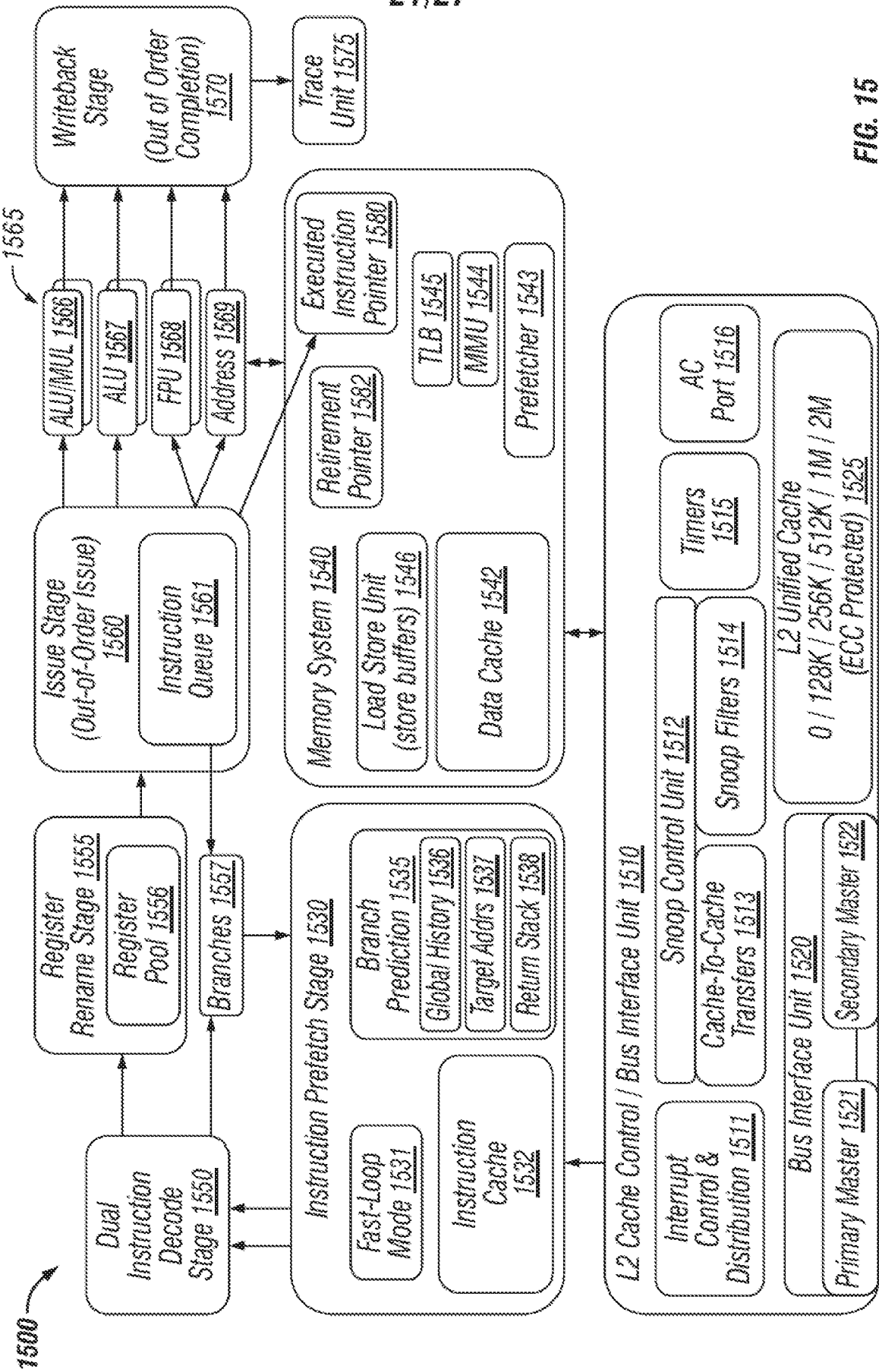

1500
1565
Dual Instruction Decode Stage 1550
Register Rename Stage 1555
Register Pool 1556
Branches 1557
Issue Stage (Out-of-Order Issue) 1560
Instruction Queue 1561
ALU/MUL 1566
ALU 1567
FPU 1568
Address 1569
Writeback Stage (Out of Order Completion) 1570
Trace Unit 1575
Instruction Prefetch Stage 1530
Fast-Loop Mode 1531
Instruction Cache 1532
Branch Prediction 1535
Global History 1536
Target Addrs 1537
Return Stack 1538
Memory System 1540
Load Store Unit (store buffers) 1546
Data Cache 1542
Retirement Pointer 1582
Executed Instruction Pointer 1580
TLB 1545
MMU 1544
Prefetcher 1543
L2 Cache Control / Bus Interface Unit 1510
Interrupt Control & Distribution 1511
Snoop Control Unit 1512
Cache-To-Cache Transfers 1513
Snoop Filters 1514
Timers 1515
AC Port 1516
Bus Interface Unit 1520
Primary Master 1521
Secondary Master 1522
L2 Unified Cache 0 / 128K / 256K / 512K / 1M / 2M (ECC Protected) 1525

FIG. 15

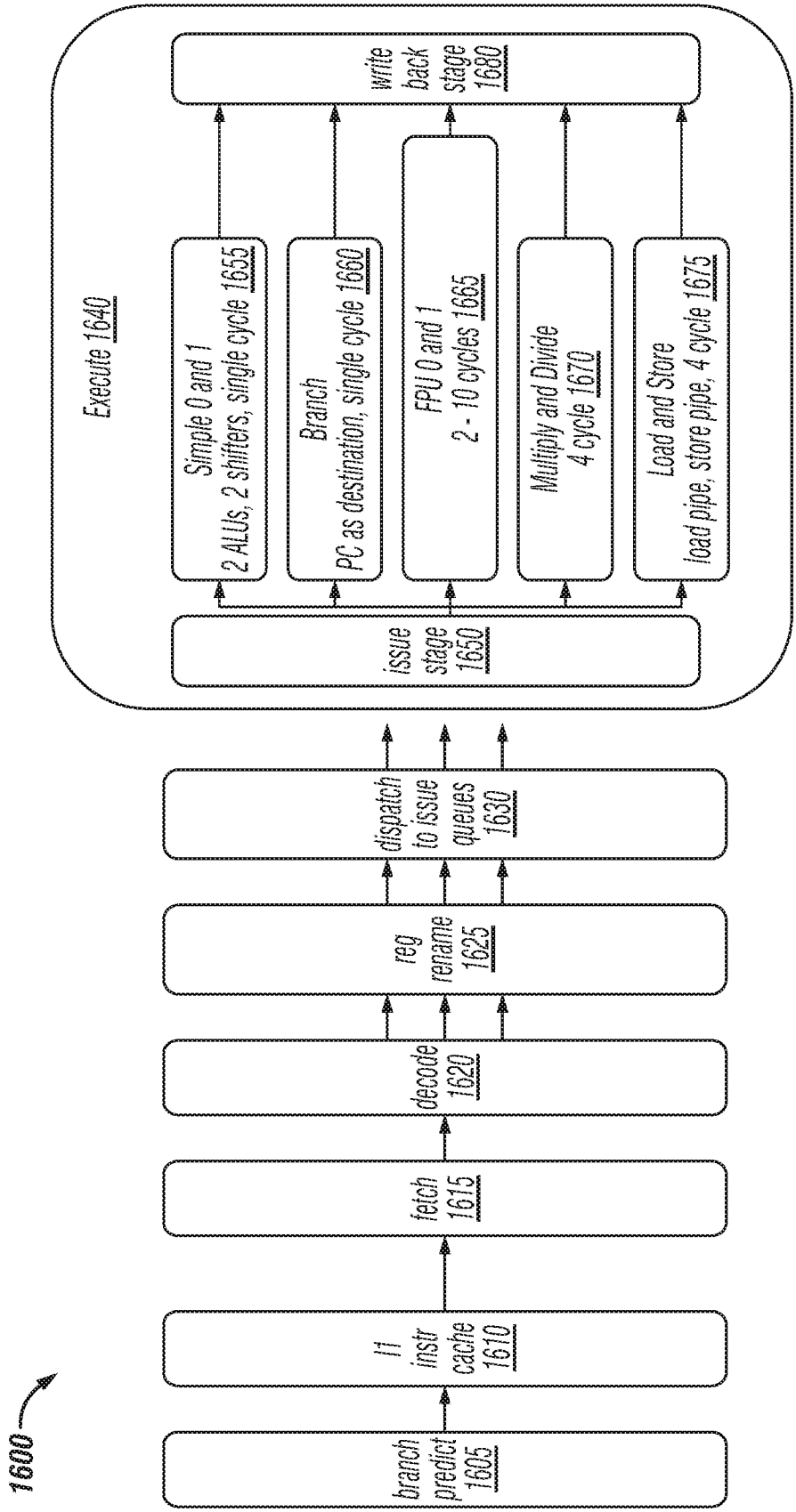

1600
branch predict 1605
I1 instr cache 1610
fetch 1615
decode 1620
reg rename 1625
dispatch to issue queues 1630
Execute 1640
issue stage 1650
Simple 0 and 1
2 ALUs, 2 shifters, single cycle 1655
Branch
PC as destination, single cycle 1660
FPU 0 and 1
2 - 10 cycles 1665
Multiply and Divide
4 cycle 1670
Load and Store
load pipe, store pipe, 4 cycle 1675
write back stage 1680

FIG. 16

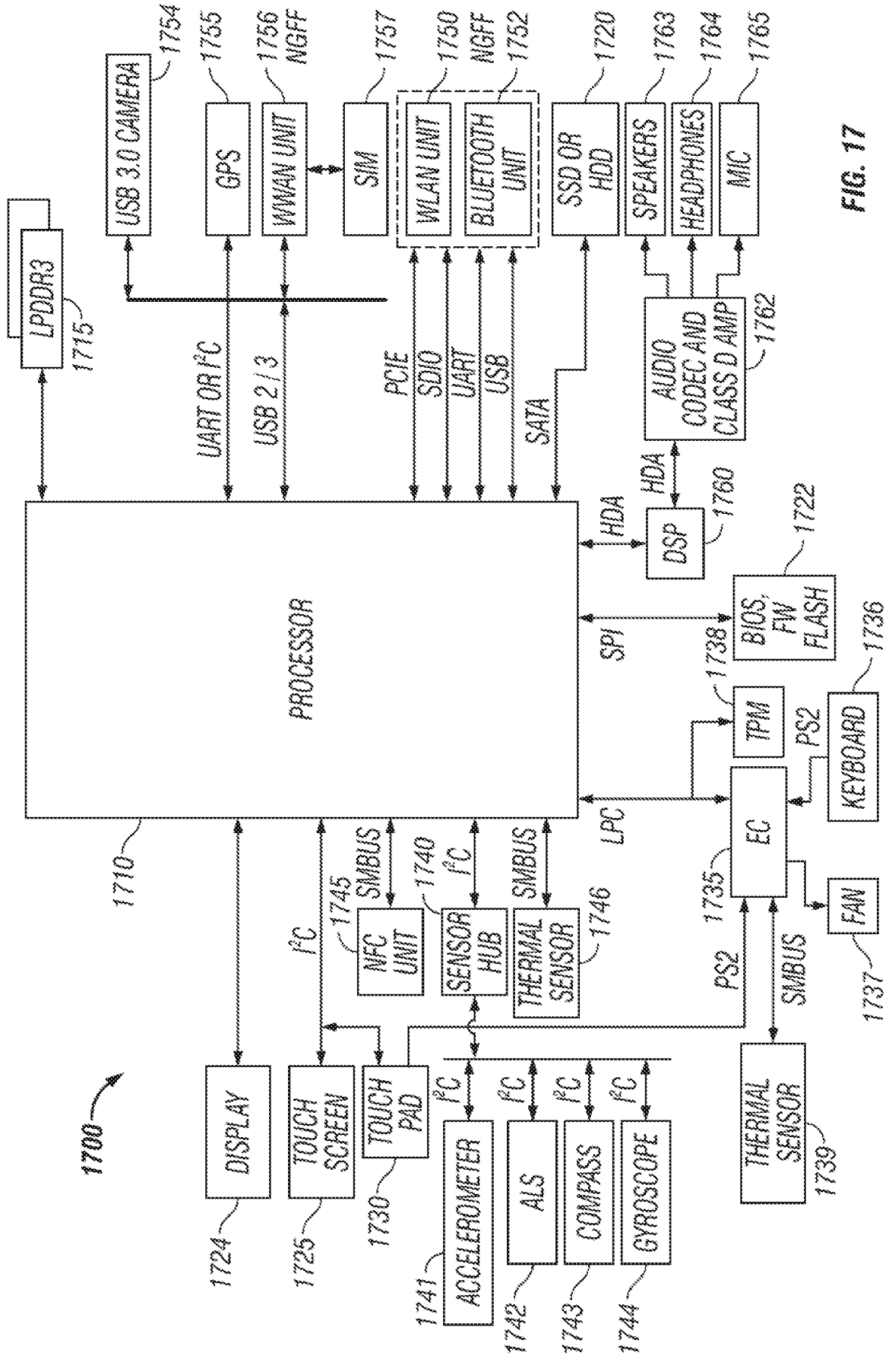

1700
PROCESSOR
1710
LPDDR3
1715
USB 3.0 CAMERA
1754
GPS
1755
WWAN UNIT
1756
NGFF
SIM
1757
WLAN UNIT
1750
NGFF
BLUETOOTH UNIT
1752
SSD OR HDD
1720
SPEAKERS
1763
HEADPHONES
1764
MIC
1765
AUDIO CODEC AND CLASS D AMP
1762
UART OR I²C
USB 2/3
PCIE
SDIO
UART
USB
SATA
HDA
HDA
DSP
1760
SPI
BIOS, FW FLASH
1722
1738
TPM
KEYBOARD
1736
PS2
LPC
EC
1735
FAN
1737
SMBUS
PS2
THERMAL SENSOR
1739
I²C
SMBUS
1745
NFC UNIT
I²C
1740
SENSOR HUB
SMBUS
THERMAL SENSOR
1746
DISPLAY
1724
TOUCH SCREEN
1725
TOUCH PAD
1730
I²C
ACCELEROMETER
1741
I²C
ALS
1742
I²C
COMPASS
1743
I²C
GYROSCOPE
1744

FIG. 17

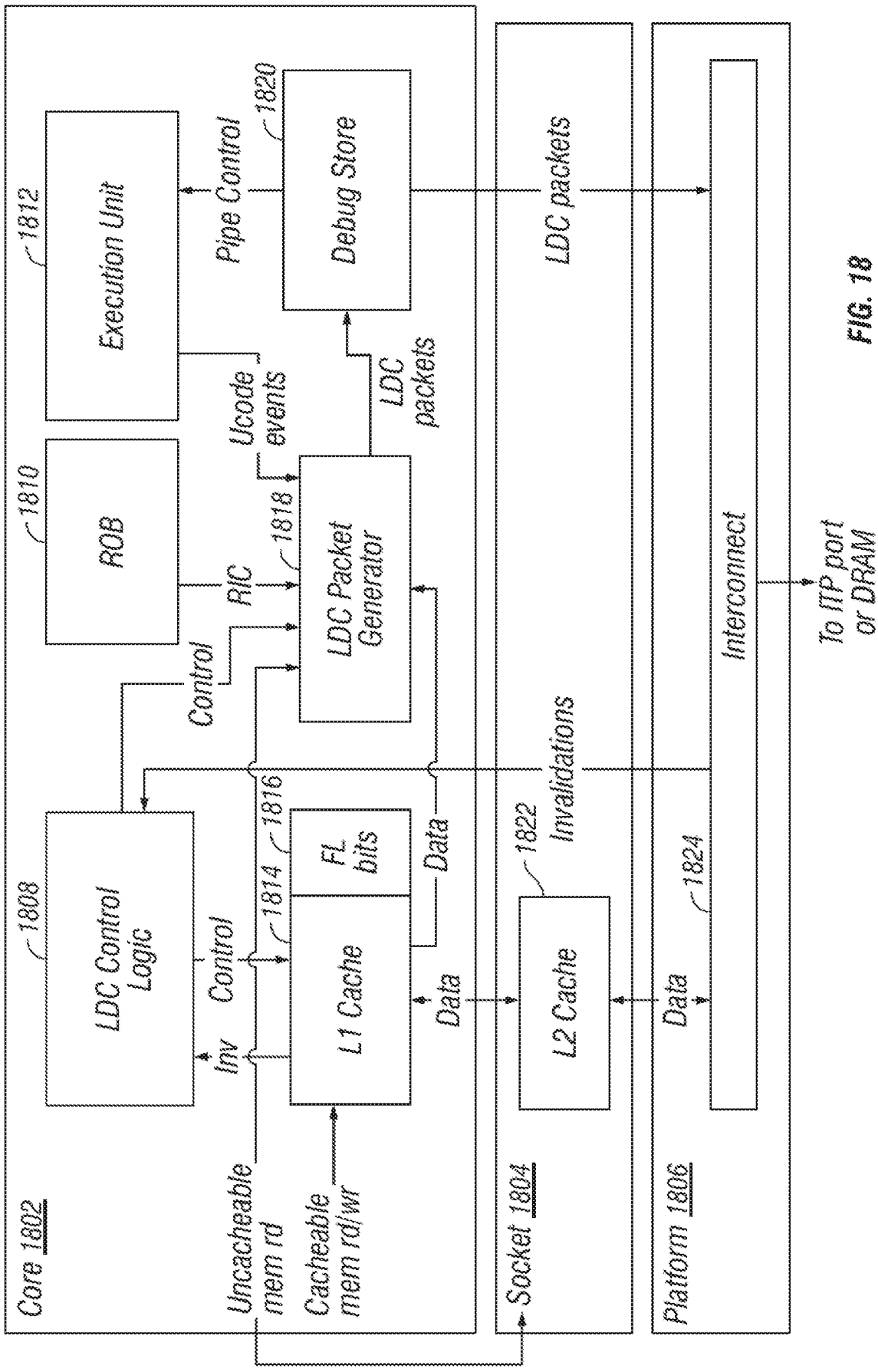

Core 1802
Execution Unit
1812
Pipe Control
1820
Debug Store
LDC packets
ROB
1810
Ucode events
LDC packets
RIC
1818
LDC Packet Generator
Control
LDC Control Logic
1808
Inv
Control
1814
1816
FL bits
L1 Cache
Data
Data
Uncacheable mem rd
Cacheable mem rd/wr
Socket 1804
1822
Invalidations
L2 Cache
Platform 1806
1824
Data
Interconnect
To ITP port or DRAM

FIG. 18

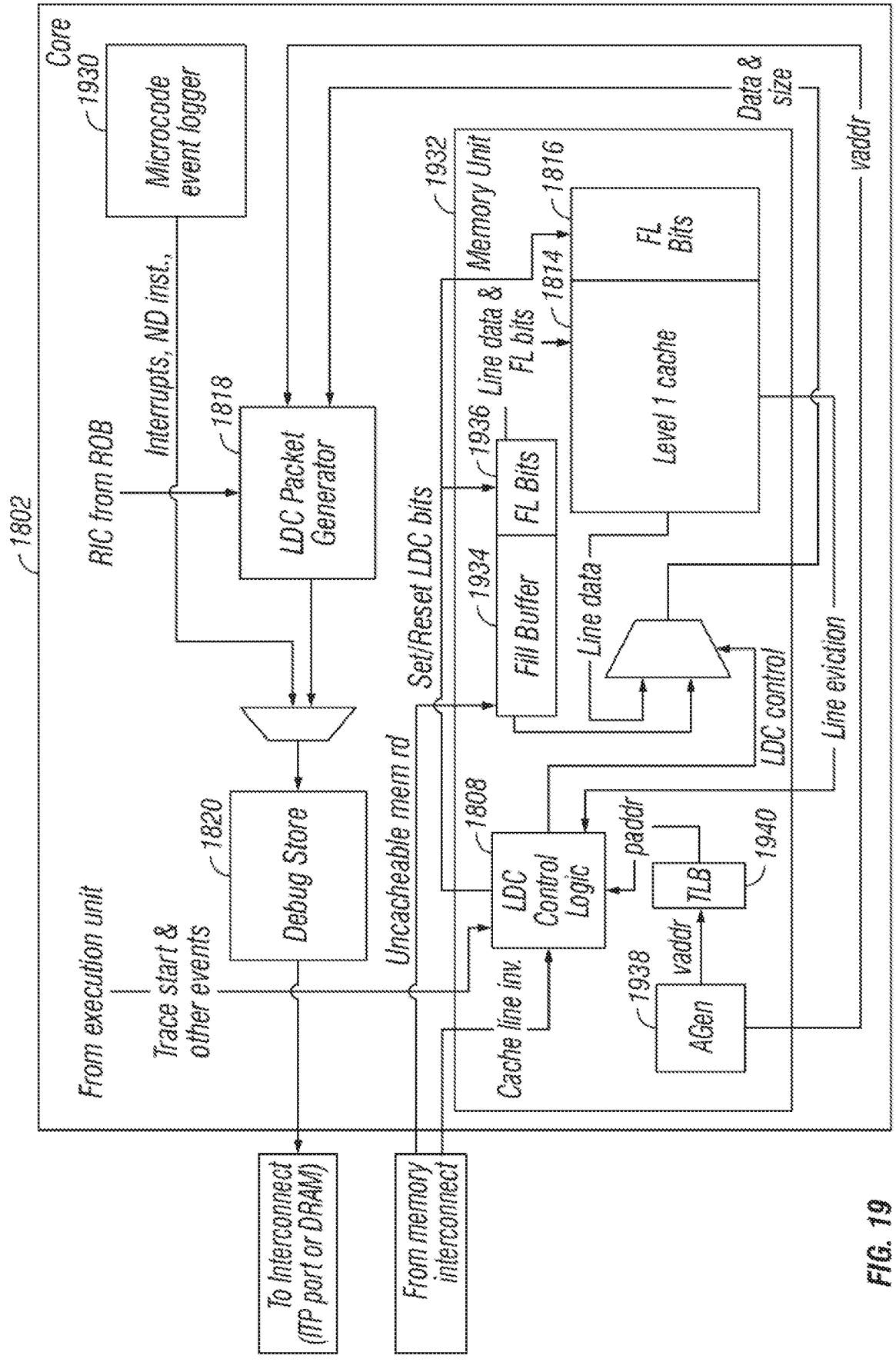

Core
1930
Microcode event logger
1802
RIC from ROB
Interrupts, ND inst.,
1818
LDC Packet Generator
1932
Memory Unit
1816
FL Bits
1814
Line data & FL bits
1936
FL Bits
1934
Fill Buffer
Level 1 cache
Data & size
vaddr
Set/Reset LDC bits
Line data
LDC control
Line eviction
Uncacheable mem rd
1808
LDC Control Logic
paddr
1940
TLB
vaddr
1938
AGen
Cache line inv.
1820
Debug Store
From execution unit
Trace start & other events
To Interconnect (ITP port or DRAM)
From memory interconnect

FIG. 19

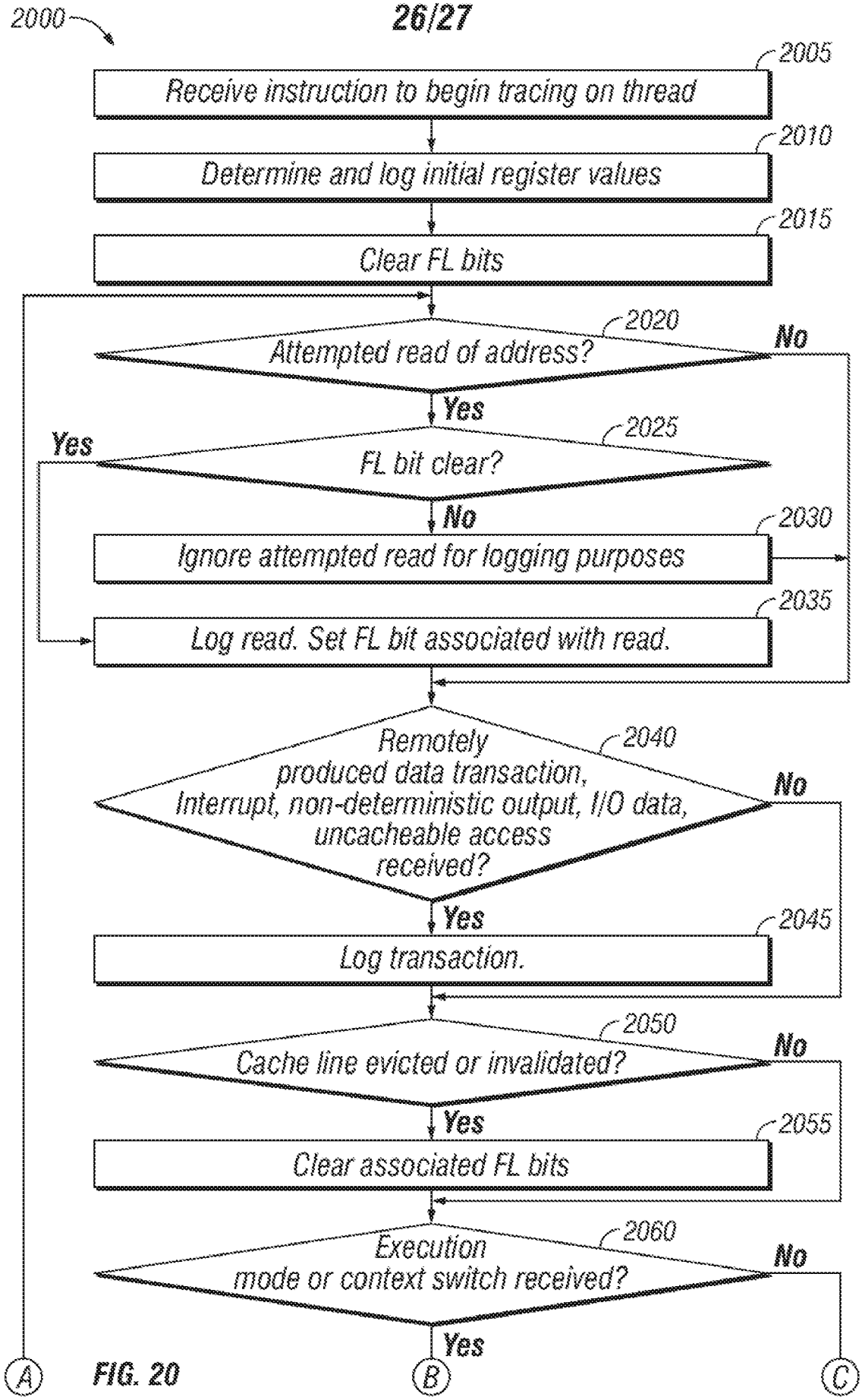

2000
2005
Receive instruction to begin tracing on thread
2010
Determine and log initial register values
2015
Clear FL bits
2020
Attempted read of address?
No
Yes
2025
FL bit clear?
Yes
No
2030
Ignore attempted read for logging purposes
2035
Log read. Set FL bit associated with read.
2040
Remotely produced data transaction, Interrupt, non-deterministic output, I/O data, uncacheable access received?
No
Yes
2045
Log transaction.
2050
Cache line evicted or invalidated?
No
Yes
2055
Clear associated FL bits
2060
Execution mode or context switch received?
No
Yes
A
B
C

FIG. 20

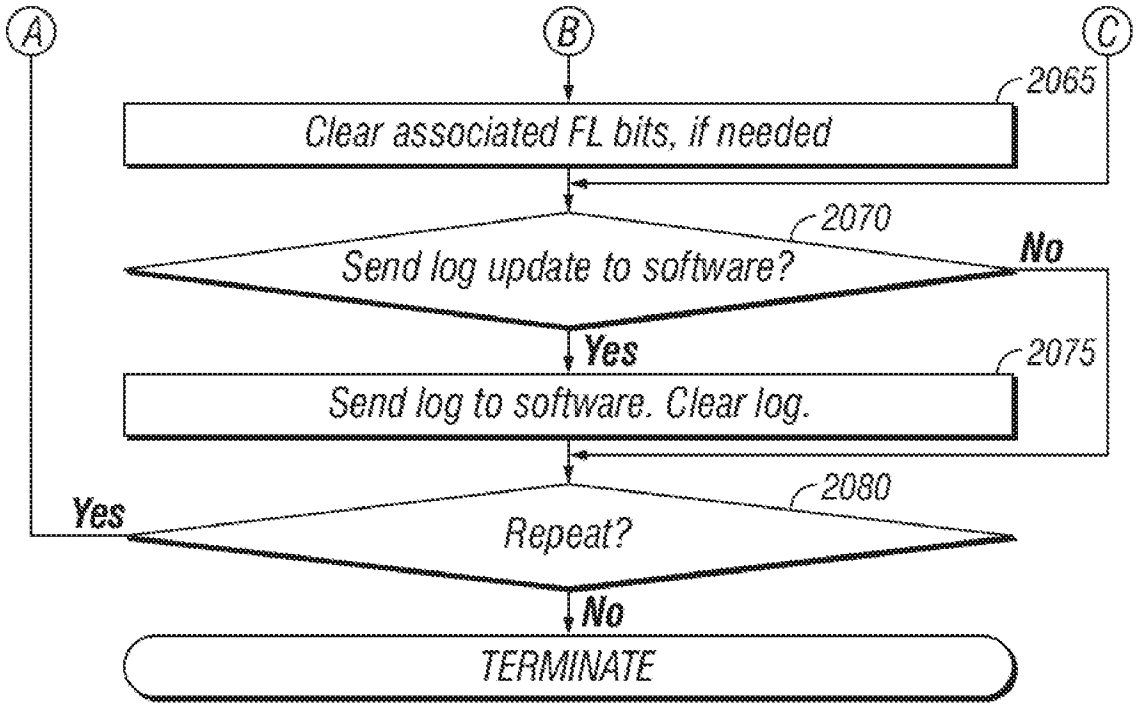

A
B
C
2065
Clear associated FL bits, if needed
2070
Send log update to software?
No
Yes
2075
Send log to software. Clear log.
2080
Yes
Repeat?
No
TERMINATE

FIG. 20
(Cont'd)

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/019063

A. CLASSIFICATION OF SUBJECT MATTER

G06F 11/34(2006.01)i, G06F 12/02(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 11/34; G06F 9/30; G06F 9/44; G06F 12/00; G06F 9/46; G06F 9/38; G06F 9/52; G06F 11/36; G06F 12/02

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: threads, execution, monitor, address, previous, read attempt, logging, debugging, and similar terms.

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 9,280,397 B2 (ALI-REZA ADL-TABATABAI et al.) 08 March 2016 See column 3, lines 10-14; column 3, line 58 - column 4, line 54; column 5, lines 53-56; column 8, lines 15-30; column 11, line 26 - column 12, line 27; column 13, lines 9-36; column 20, line 43 - column 21, line 30; column 21, line 51 - column 22, line 1; and figures 1-4a.	1-22
A	US 2015-0040111 A1 (BRATIN SAHA et al.) 05 February 2015 See paragraphs [0040]-[0041], [0047], and [0050]-[0052]; and figure 2.	1-22
A	US 2015-0186178 A1 (ROLF KASSA et al.) 02 July 2015 See paragraphs [0003] and [0031]-[0034]; and figures 2-3B.	1-22
A	US 2009-0083715 A1 (JIMMIE EARL DEWITT, JR. et al.) 26 March 2009 See paragraphs [0093]-[0105] and figures 4A-4C.	1-22
A	US 2016-0070638 A1 (INTERNATIONAL BUSINESS MACHINES CORP.) 10 March 2016 See paragraphs [0045]-[0049] and figure 4.	1-22

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents,such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
07 June 2017 (07.06.2017)	**07 June 2017 (07.06.2017)**

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

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2017/019063

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 9280397 B2	08/03/2016	US 2009-0006767 A1	01/01/2009
		US 2010-0218195 A1	26/08/2010
		US 2011-0145516 A1	16/06/2011
		US 8140773 B2	20/03/2012
		US 8719514 B2	06/05/2014
US 2015-0040111 A1	05/02/2015	US 2008-0162886 A1	03/07/2008
		US 8719807 B2	06/05/2014
		US 9304769 B2	05/04/2016
US 2015-0186178 A1	02/07/2015	CN 104750459 A	01/07/2015
		EP 2889769 A1	01/07/2015
		EP 2889769 A8	17/02/2016
		EP 2889769 B1	24/05/2017
		RU 2014147955 A	27/06/2016
US 2009-0083715 A1	26/03/2009	US 2003-0135789 A1	17/07/2003
		US 7313734 B2	25/12/2007
		US 7661035 B2	09/02/2010
US 2016-0070638 A1	10/03/2016	US 2014-0237454 A1	21/08/2014
		US 9223681 B2	29/12/2015