



- (51) **International Patent Classification:**
G06F 9/30 (2006.01) *G06F 9/38* (2006.01)
- (21) **International Application Number:**
PCT/US2017/019845
- (22) **International Filing Date:**
28 February 2017 (28.02.2017)
- (25) **Filing Language:** English
- (26) **Publication Language:** English
- (30) **Priority Data:**
15/089,232 1 April 2016 (01.04.2016) US
- (71) **Applicant:** INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, California 95054 (US).
- (72) **Inventors:** WU, Lisa K.; 524 W. 114th Street #3A, New York, New York 10025 (US). HAM, Tae Jun; 11 Lawrence Drive, Apt. 507, Princeton, New Jersey 08540 (US). SATISH, Nadathur Rajagopalan; 700 Agnew Road #135, Santa Clara, California 95054 (US). SUNDARAM, Narayanan; 1750 Halford Avenue, Apt. 314, Santa Clara, California 95051 (US).
- (74) **Agents:** KOMENDA, J. Kyle et al.; 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.

- (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))

Published:

- with international search report (Art. 21(3))

(54) **Title:** INSTRUCTION, CIRCUITS, AND LOGIC FOR GRAPH ANALYTICS ACCELERATION

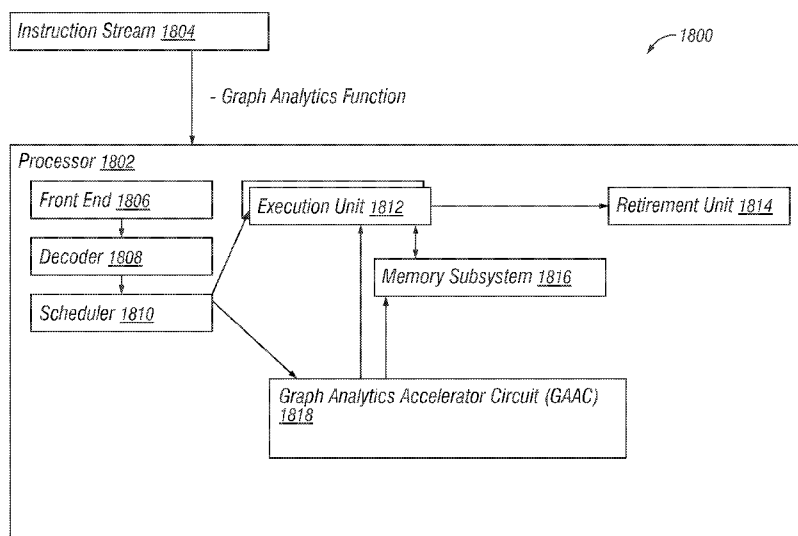


FIG. 18

(57) **Abstract:** A processor includes a front end including circuitry to receive and decode an instruction. The instruction is to perform a graph analytic function and pass the instruction to a graph accelerator. The graph accelerator including circuitry to process graph vertices and graph edges as datatypes, execute the instruction, and pass results of the instruction to a memory subsystem of the processor.

INSTRUCTION, CIRCUITS, AND LOGIC FOR GRAPH ANALYTICS ACCELERATION

5 **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.

10

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
15 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
20 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
25 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
30 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;

5 [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;

10 [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;

15 [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;

20 [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;

25 [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;

30 [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 an illustration of a system to implement instructions, circuitry, and logic for analysis of graph analytics, according to embodiments of the present disclosure;

[0031] FIGURE 19 is a more detailed view of an example graph and data accesses that are to be performed to analyze the graph, according to embodiment of the present disclosure;

[0032] FIGURE 20 is an illustration of an execution pipeline for a graph accelerator, according to embodiments of the present disclosure;

[0033] FIGURE 21 is a more detailed view of an instance of a pipeline of a graph accelerator to perform specific functions or workloads, according to embodiments of the present disclosure;

5 [0034] FIGURE 22 is a more detailed view of an instance of a pipeline of to illustrate parallel pipeline execution, according to embodiments of the present disclosure;

[0035] FIGURE 23 is an illustration of an example graph wherein edges have properties such as weights, according to embodiments of the present invention;

10 [0036] FIGURE 24 is an illustration of example operation of edge accesses, according to embodiments of the present disclosure;

[0037] FIGURE 25 is an illustration of an example pointer table, according to embodiments of the present disclosure; and

[0038] FIGURE 26 is an illustration of a method for graph analysis acceleration, according to embodiments of the present disclosure.

15

DETAILED DESCRIPTION

[0039] The following description describes an instruction, circuit, and processing logic for acceleration of graph analytics. 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.

20

25

[0040] 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

30

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.

[0041] 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.

[0042] 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).

5 [0043] 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.

20 25 30 [0044] 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.

[0045] 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).

[0046] 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.

[0047] 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.

10 **[0048]** 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
15 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
20 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
25 operand or a vector operand may be a source or destination operand of a SIMD instruction (or 'packed data instruction' or a 'vector instruction'). In one embodiment, a SIMD instruction specifies a single vector operation to be performed on two source vector operands to generate a destination vector operand (also referred to as a result vector operand) of the same or different size, with the same or different
30 number of data elements, and in the same or different data element order.

[0049] 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
5 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.).

[0050] In one embodiment, destination and source registers/data may be generic terms to represent the source and destination of the corresponding data or operation.
10 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
15 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.

[0051] 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
25 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
30 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.

[0052] 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.

[0053] 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.

[0054] 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.

[0055] 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.

[0056] 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.

[0057] 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.

[0058] 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.

[0059] 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.

[0060] 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.

[0061] 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.

[0062] 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.

[0063] 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.

[0064] 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).

[0065] 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.

[0066] 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.

[0067] 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.

[0068] 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.

[0069] 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.

[0070] Some instructions may be converted into a single micro-op, whereas
5 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
10 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-
15 ops for an instruction, front end 201 of the machine may resume fetching micro-ops from trace cache 230.

[0071] 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
20 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:
25 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
30 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.

[0072] 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
5 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
10 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.

[0073] 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
15 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,
20 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
25 216, 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
30 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.

- 5 **[0074]** 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
- 10 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.
- 15 **[0075]** 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,
- 20 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
- 25 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 MMXTM registers (also referred to as ‘mm’ registers in some instances) in microprocessors enabled with MMX technology from Intel Corporation of Santa
- 30 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.

[0076] 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.

[0077] 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.

[0078] 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.

[0079] 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.

[0080] 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.

[0081] 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
5 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.

10 **[0082]** 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
15 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
20 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
25 by field 384.

[0083] 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
30 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.

[0084] In FIGURE 4A, a processor pipeline 400 may include a fetch stage 402, a
5 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.

[0085] In FIGURE 4B, arrows denote a coupling between two or more units and
10 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.

[0086] Core 490 may be a reduced instruction set computing (RISC) core, a
complex instruction set computing (CISC) core, a very long instruction word
15 (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.

[0087] 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
20 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
25 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.
30 Decode unit 440 may be coupled to a rename/allocator unit 452 in execution engine unit 450.

[0088] 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

5 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.

10 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.).

15 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

20 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.,

25 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.

30 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.

5 [0089] 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.

15 [0090] 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.

25 [0091] 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).

30 [0092] 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.

[0093] 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.

[0094] 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.

[0095] 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.

[0096] 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
5 506 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.

[0097] In various embodiments, one or more of cores 502 may perform multi-threading. System agent 510 may include components for coordinating and
10 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
15 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
20 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.

[0098] 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-
25 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.

[0099] Processor 500 may include a general-purpose processor, such as a Core™ i3, i5, i7, 2 Duo and Quad, Xeon™, Itanium™, XScale™ or StrongARM™
30 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
5 example, BiCMOS, CMOS, or NMOS.

[00100] 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
10 506 may be shared by two or more cores 502 by implementing time-slices of a given cache 506.

[00101] 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.
15 Media engine 565 may provide media encoding and video decoding.

[00102] 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
20 through cache hierarchy 503.

[00103] 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
25 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.

[00104] 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
30 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.

[00105] 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
5 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
10 590. Coordination of the instances of core 502 may be facilitated in part through module 590.

[00106] 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
15 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
20 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.

[00107] 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
25 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.

[00108] Each processor 610, 615 may be some version of processor 500. However, it should be noted that integrated graphics logic and integrated memory
30 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.

[00109] 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
5 610, 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.

[00110] Furthermore, GMCH 620 may be coupled to a display 645 (such as a flat
10 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.

15 [00111] 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
20 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.
25 For at least one embodiment, various processors 610, 615 may reside in the same die package.

[00112] 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
30 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.

[00113] 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.

5 [00114] 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
10 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.

[00115] Processors 770, 780 may each exchange information with a chipset 790
15 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.

[00116] A shared cache (not shown) may be included in either processor or
20 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.

[00117] 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)
25 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.

[00118] 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.
30 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.

5 **[00119]** 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.

10 **[00120]** 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
15 coupled to control logic 872, 882. Legacy I/O devices 815 may be coupled to chipset 790.

20 **[00121]** 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 of one or more media processors 920 which may include integrated graphics logic 908, an image processor 924 for providing still
25 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.

30 **[00122]** 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.

[00123] 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.

[00124] In FIGURE 10, processor 1000 includes a CPU 1005, GPU 1010, image processor 1015, video processor 1020, USB controller 1025, UART controller 1030, SPI/SDIO controller 1035, display device 1040, memory interface controller 1045, MIPI controller 1050, flash memory controller 1055, dual data rate (DDR) controller 1060, security engine 1065, and I²S/I²C controller 1070. Other logic and circuits may be included in the processor of FIGURE 10, including more CPUs or GPUs and other peripheral interface controllers.

[00125] 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.

5 **[00126]** 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
10 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.

[00127] In some embodiments, one or more instructions may correspond to a first
15 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.

[00128] FIGURE 12 illustrates how an instruction of a first type may be emulated
20 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
25 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
30 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
5 logic embodied in a tangible, machine-readable medium containing software by executing microcode or firmware contained in or associated with the processor.

[00129] 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
10 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
15 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)
20 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,
25 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
30 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
5 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.

[00130] FIGURE 14 is a block diagram of an instruction set architecture 1400 of a
10 processor, in accordance with embodiments of the present disclosure. Instruction set architecture 1400 may include any suitable number or kind of components.

[00131] 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
15 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
20 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.

[00132] Instruction set architecture 1400 may also include any number or kind of interfaces, controllers, or other mechanisms for interfacing or communicating with
25 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
30 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.

10 **[00133]** 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.

15 **[00134]** 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 20 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.

[00135] 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 25 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 30

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.

[00136] 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.

[00137] 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.

[00138] 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.

5 **[00139]** 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
10 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
15 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.

[00140] Memory system 1540 may include any suitable number and kind of mechanisms for storing information for the processing needs of instruction
20 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
25 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.

[00141] The operation of instruction architecture 1500 to execute an instruction
30 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
5 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
10 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.

[00142] Dual instruction decode stage 1550 may translate a received instruction into microcode-based instructions that may be executed. Dual instruction decode
15 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.

[00143] Register rename stage 1555 may translate references to virtual registers or other resources into references to physical registers or resources. Register rename
20 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.

[00144] 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
25 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
30 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.

- 5 **[00145]** 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
10 monitored or debugged by trace unit 1575.

[00146] 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.

- 15 **[00147]** 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
20 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
25 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
30 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.

[00148] 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.

[00149] Electronic device 1700 may include processor 1710 communicatively coupled to any suitable number or kind of components, peripherals, modules, or devices. Such coupling may be accomplished by any suitable kind of bus or interface, such as I²C bus, system management bus (SMBus), low pin count (LPC) bus, SPI, high definition audio (HDA) bus, Serial Advance Technology Attachment (SATA) bus, USB bus (versions 1, 2, 3), or Universal Asynchronous Receiver/Transmitter (UART) bus.

[00150] 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.

[00151] Furthermore, in various embodiments other components may be communicatively coupled to processor 1710 through the components discussed
5 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
10 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).

15 [00152] FIGURE 18 is an illustration of a system 1800 to implement instructions, circuitry, and logic for analysis of graph analytics, according to embodiments of the present disclosure.

[00153] Hardware in system 1800 to perform graph analytics may be performed in any suitable manner. System 1800 may include a processor 1802 in which a core
20 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. Execution of graph analytics may be offloaded from the execution pipeline of the core to a specific circuit for graph analytics. For example, execution of graph
25 analytics may be offloaded to a graph analytics accelerator circuit (GAAC) 1818. In one embodiment, GAAC 1818 may perform graph analytics.

[00154] GAAC 1818 may be implemented within processor 1802, or may be implemented outside processor 1802 on, for example, the same die or communicatively
30 coupled using an interprocessor bus. GAAC 1818 may be implemented within a core of processor 1802, or as a specific implementation of an execution unit 1812. Calculations may be offloaded to GAAC 1818 rather than, for example, executing to find the same results with execution units 1812. GAAC 1818 may be implemented as

an offload engine. In some embodiments, GAAC 1818 may be communicatively coupled to portions of processor 102 through, for example, a PCI bus.

[00155] Processor 1802 may include a front end 1806 to receive instructions from instruction stream 1802, which in turn may be loaded from a cache or other part of memory subsystem 1816, which may be internal or external to processor 1802. Instructions may be decoded by decoder 1808. Decoded instructions may be scheduled or routed for execution to execution units 1812 or GAAC 1818 by scheduler 1810. Results may be posted from execution 1812 or GAAC 1818 to memory subsystem 1816. Instructions may be retired by retirement unit 1814, and results may be committed upon retirement.

[00156] System 1800 may identify that execution of instruction stream 1804, as loaded from memory, may require graph analytics. Such identification may be made implicitly or explicitly. For example, system 1800 may recognize particular instructions in instruction stream 1804 that could be offloaded for execution in GAAC 1818 to result in more efficient execution. In another example, a programmer or compiler that generated instruction stream 1804 may include instructions that specifically identify that the instruction is to be executed using GAAC 1818.

[00157] FIGURE 19 is a more detailed view of an example graph and data accesses that are to be performed to analyze the graph, according to embodiment of the present disclosure. The graph may include nodes, each with an identifier, connected with edges, each edge with an associated property. All edges within a graph may contain the same properties. However, for different graph analytical applications, different properties might be needed to perform computations.

[00158] A graph may be represented as node pairs (VertexID (“VID”), vertex property (“VProp”)) and edge tuples (SRCID, DSTID, edge property). The vertex properties, edge SRC, and DST IDs, and edge properties may be represented with 4-byte or 8-byte data. Graph analytics applications require a large amount of indirect and random memory accesses (when accessing vertex properties and edge properties) and exhibit performance and energy inefficiencies when executing on current processors.

[00159] For example, to count the degree of a graph (which may include the number of out-going edges per vertex for a directed graph), calculation may require determining (Degree[VID] += if (Edge[SRCID] matches VID)). In other words, the

degree of a given vertex is increased if a given edge has a source identifier that matches the given vertex's identifier. This process may utilize at least two or three indirect memory accesses per degree calculation. These compute inefficiencies may be due to memory bandwidth limitations, or memory access granularities wherein a read or write of data overruns the boundaries of a given cacheline. For example, to get to a specific 4-byte or 8-byte piece of data, an entire cacheline of 64-bytes might need to be loaded. The subsequent access might be on yet another cacheline, and so even though another 4-byte or 8-byte chunk is needed, an entirely other 64-byte cacheline would need to be loaded. Furthermore, execution granularities may require that data is executed using instructions of a long or wide width that mismatches the chunk of data that is to be processed. Instead, system 1800 may allow operation of datatypes or data structures that specifically match graph data and comport to the analytics that are to be performed on graphs. These may include the vertices and edges. System 1800 may address these problems in hardware.

[00160] Returning to FIGURE 18, GAAC 1818 may specifically process graph analytic workloads. GAAC 1818 may be loosely-coupled for offloading from one or more cores, or may be a tightly-coupled part of a core's execution pipeline, sharing memory with the cores. In one embodiment, GAAC 1818 may be designed such its layouts may take advantage of the structured data movements and computation patterns of graph applications to process graphs using specific primitives, such as vertices and edges, instead of other granularities used in, for example, x86 instructions. In another embodiment, GAAC 1818 may be designed to read the streaming access pattern for vertex properties and edges of the input graph when edges are sorted by SRCID. In yet another embodiment, GAAC 1818 may adopt parallelism for graph analytics processing. In still another embodiment, GAAC 1818 may access an on-chip or on-die scratchpad, such as on-chip eDRAM, that is hardware-managed.

[00161] FIGURE 20 is an illustration of an execution pipeline for GAAC 1818, according to embodiments of the present disclosure.

[00162] A first stage may include a source or frontier VProp fetch. This may incur off-chip memory management access, as the vector property is fetched from, for example, memory or cache. This stage may be implemented with or access, for example, sequential memory prefetchers, memory request buffers, data return buffers,

and data multiplexers. The VProp fetch, depending upon the specific function that is to be executed, might next incur an edge fetch or a compute operation.

[00163] The vertex properties that are to be fetched and executed may vary from workload to workload, or function to function. For example, for the PageRank function or workload, the page rank value and the degree of each vertex are the properties that are to be used in the calculation. The VProp fetch is stream-based, however, and for a frontier-based pipeline, only the starting source vertex and associated properties might be read on the first iteration. Furthermore, only the frontier vertices and associated properties written to memory by the previous iteration might be read for the remaining iterations.

[00164] An edge fetch stage may include fetching an edge and its associated properties. The edge fetch may be stream-based or frontier-based. A stream-based edge fetcher implementation may be analogous to the source of destination VProp fetchers. A frontier-based edge fetcher may be implemented using an edge processing accelerator, discussed in further detail below.

[00165] After edge fetching, a destination VProp fetch stage may be executed. The destination VProp fetch stage may be implemented in a similar manner to the source or frontier VProp stage. The results may be piped to a compute stage.

[00166] A compute stage may include arithmetical and combinational logic circuitry that is programmable or reconfigurable to execute the specified or given workload or function. This may be customized for a given function or workload. In one embodiment, specialized compute stages may be hard-wired for given functions. The results may be piped to atomic issue logic.

[00167] An on-chip storage atomic issue stage may include circuitry to determine whether back-to-back memory requests, such as eDRAM requests, are to be made to the same address. Furthermore, the stage may determine whether the requests will maintain read-modify-write atomicity with respect to the two requests. If atomicity is not needed, then the stage may issue the subsequent requests to parallel different sub-pipelines, increasing parallel operations. If the atomicity is needed, then the stage may issue the subsequent requests to the same sub-pipelines that handled the previous request. The sub-pipelines may be implemented in different banks.

[00168] Execution may be then handled by a series of circuits for VProp reads, reduce calculations, and VProp writes. The reads and writes may be performed on-chip, wherein on-chip storage or scratchpad resources are used for memory manipulation without the need to access external memory. For example, eDRAM may be used. The reduce stage may perform specific reduction calculations as needed for the given workload. After these stages, execution may repeat back at the issue circuitry to determine whether additional requests will be processed for atomicity or whether a barrier check will be performed.

[00169] A barrier check stage may include circuitry to evaluate whether termination conditions exist for a workload. The termination conditions may be with respect to intermediate calculations or final calculations. The conditions, whether intermediate or final, may vary according to function or workload. For example, for PageRank, every edge might need to be processed by the reduce stage before the apply stage can begin, while every vertex might need to be processed by the apply stage before the next iteration can begin.

[00170] A second set of VProp Fetch (from memory) and VProp Read (from chip) stages may be implemented to gather data, which may then be applied by arithmetic and combinational circuitry in an apply stage. The results may be written in a write stage, accessing memory if needed, while another barrier check stage evaluates execution and optionally begins subsequent iterations.

[00171] The pipeline of GAAC 1818 may support workloads and functions that use all vertices and all edges in an iteration, which may be referred to as a stream-based GAAC pipeline. Furthermore, the pipeline may support workloads and functions that use only a subset of vertices and a subset of edges associated with those vertices per iteration, which may be referred to as a frontier-based GAAC pipeline. An example workload of the stream-based pipeline is PageRank, while example workloads of the frontier-based pipelines are BFS (Breadth First Search) and SSSP (Single Source Shortest Path).

[00172] In the illustrated pipeline, the arrows to and from barrier check stages, as well as feedback arrow from on-chip storage VProp write stage back to on-chip storage atomic issue logic circuitry stage may be control signals. The other arrows in the illustrated pipeline may illustrate data propagation.

[00173] As discussed above, computation may begin by retrieving vertex properties from memory. The vertex properties may be laid out in memory consecutively and are fetched using a sequential prefetcher. For the frontier-based workloads, a sequential prefetcher is still used, but only the frontier vertices and associated properties written by the previous iteration of the algorithm are fetched from memory. After this point in the pipeline, the edges are retrieved from memory and are processed one at a time in a pipelined fashion using associated vertex properties already fetched from memory. As each edge is processed, the intermediate or temporary vertex properties associated with each edge need to be read, modified, and written multiple times per iteration. These accesses are frequent and random and require atomicity. Thus, use of on-chip eDRAM or other memory may be used. More specifically, memory access to these on-chip memories may be made using byte-access widths that match the data structures used for vertices and edges, such as 4-byte and 8-byte data access granularity for fast and efficient on-chip bandwidth utilization. This also helps alleviate the need to perform long-latency, power-hungry, bandwidth-consuming off-chip memory atomic read-modify-write operations. A simple dependency checking atomic issue logic circuit is implemented to utilize all banks of an eDRAM. After the barrier condition is met, i.e. all necessary edges are processed, the barrier checker then informs the next phase of the pipeline to start. This apply phase of the pipeline works on all vertices sequentially, and updates the vertex properties in memory with the intermediate or temporary values stored on the eDRAM. For the frontier-based pipeline, this phase also may write the frontier vertices and associated properties needed to be processed for the next iteration to memory. When all vertices are updated, that is, when the barrier check condition in the apply phase is met, the barrier checker then informs the first stage of the pipeline to start the next iteration of the algorithm.

[00174] Accordingly, the pipeline utilizes datatype specialization to process graphs in vertices and edges. Furthermore, the pipeline utilizes streaming execution to stream in vertex properties and edges from memory in sequence and execute them in a dataflow fashion. Also, the pipeline utilizes parallel execution to partition execution streams in hardware using duplicated resources operating on non-overlapping vertex properties. The pipeline enables parallel edge prefetches into streams of parallel execution. In addition, the pipeline utilizes memory subsystem specialization to

accelerate frequent and random accesses to intermediate data values. This is accomplished, for example, by storing the values in a banked, fine-granular (i.e. 4 or 8 bytes per read/write) on-chip eDRAM, and providing atomic eDRAM read-modify-write operations.

5 [00175] For intermediate data that does not fit into the on-chip eDRAM, GAAC 1818 may operate on pre-partitioned graphs one at a time, utilizing main memory to store the rest of the partitioned graphs that are not being processed. This partitioning shows small performance degradation, arguing for the scalability of the architecture.

[00176] FIGURE 21 is a more detailed view of an instance of a pipeline of GAAC 10 1818 to perform PageRank functions or workloads, according to embodiments of the present disclosure.

[00177] The compute stage may perform a simple comparison to match the edges fetched with the same source vertex ID. The compute stage may output the associated vertex properties and destination IDs. The reduce stage may perform an addition of the 15 destination vertex page rank value to the temporary or intermediate page rank value until all edges are processed to complete one iteration of the algorithm. Execution might move to the apply stage of the pipeline. The apply stage may multiply by a specified constant (such as $1-\alpha$), perform an addition using the α , and divide by the degree associated with each vertex. Note that these three arithmetic and 20 combinational logic circuits can be reconfigurable and/or programmable for specific workloads.

[00178] FIGURE 22 is a more detailed view of an instance of a pipeline of GAAC 1818 to illustrate parallel pipeline execution, according to embodiments of the present disclosure.

25 [00179] To utilize the inherent parallelism in graph analytics applications, GAAC 1818 pipeline may be duplicated using multiple vertex fetch streams and multiple edge fetch streams. The vertex properties are fetched once and decoupled and broadcasted using crossbars into SRC and DST processing streams. For example, a source stream may fetch sources with an even identifier, and another source stream may fetch sources 30 with an odd identifier. Destination streams may be provided separately for destinations with even and odd identifiers. This design may facilitate reduction of data movements to and from memory. Furthermore, it may facilitate the parallel processing of data once

5 fetched from memory, reducing ineffective memory bandwidth usage. In one embodiment, eight vertex streams and eight edge streams may be used for a total of 64 processing streams.

5 [00180] FIGURE 23 is an illustration of an example graph wherein edges have properties such as weights, according to embodiments of the present invention. Use of such a graph may illustrate operation of acceleration of edge processing, in accordance with embodiments of the present disclosure.

10 [00181] For example, vertex 5 may connect to vertices 2, 3, and 4 with respective weights of 0, 7, and 12, Vertex 3 may connect to vertices 2 and 4 with respective weights 6 and 4. Vertex 0 may connect to vertices 1 and 2 with respective weights 10 and 0.

15 [00182] The edges, including their properties, may be stored in, for example, an array of structures (AOS) or a structure of arrays (SOA) in memory. In some embodiments, the edges might be stored in a sorted format based upon source identifiers to achieve better performance when performing graph analytics. This may result from the tendency for a given source identifier to require multiple edges during processing of the given source vertex. Iterative calls to the edge property structure might be accomplished with fewer reads or slow-latency memory accesses if a likely candidate for a next iteration is stored near the value for the present iteration.

20 [00183] As discussed above, for stream-based graph analytics applications such as PageRank, all edges might be processed per iteration, and simple edge prefetchers that prefetch sequentially might saturate available system bandwidth in a conventional system. For frontier-based graph analytics applications such as BFS (Breadth First Search) and SSSP (Single Source Shortest Path), only edges associated with the frontier
25 vertices need to be explored (or processed) per iteration.

[00184] GAAC 1818 may processes each edge once for the whole entire program execution, with each iteration processing a percentage of the total number of edges, depending on the connectivity of the input graph. Since the frontier-based algorithms need to only access a portion of the edges per iteration based on the frontier vertices,
30 available system bandwidth might not be saturated. This effectively turns a bandwidth-bound problem into a latency-bound problem as edge retrieval requires at least two dependent indirect memory accesses.

[00185] In one embodiment, GAAC 1818 may include an Edge Pointer Table (EPT). The EPT may store edge pointers in hardware to remove one instance of memory access latency, out of the two dependent memory accesses mentioned above with respect to latency-bound graph analytics workloads. The EPT may be accessed via a direct indexing scheme, such as a CAM.

[00186] In another embodiment, an EPT may be used in a general-purpose CPU.

[00187] FIGURE 24 is an illustration of example operation of edge accesses, according to embodiments of the present disclosure. The operation shown in FIGURE 24 may be based off the graph of FIGURE 23.

[00188] In the example graph of FIGURE 23, software stores the graph in CSR (Compressed Sparse Row) format for easier accesses to the edges belonging to a particular vertex ID. The values in the array of row pointers display the index on the column indices array that stores the destination identifier. The position within the row pointer array corresponds to the source identifier or vertex identifier. For example, at position 0 in the row pointer array, a value 0 is stored, indicating that vertex 0 points to position 0 in the destination identifier array. The destination identifier array at position zero is vertex 1. The corresponding weight in the edge value array is 10. Vertex 0 also points to vertex 2. Accordingly, the next value in the row pointer array is a value corresponding to the index 2 in the destination array, which stores the destination vertex identifier value of 2. Multiple source vertices point to vertex 2, and as such multiple entries in the row pointers correspond to an index of two in the destination array.

[00189] To conventionally perform fetches for vertex 3, for example, the following pseudocode might be used:

SRCID ← 3

R1 ← Row_pointers[SRCID]

Load R1 with starting index for edge access from memory

R2 ← Row_pointers[SRCID+1]

Load R2 with the ending index for edge access

R3 ← R2 - R1

Compute R2 as the number of edge destination identifiers
belonging to SRCID

for (i=0; i<R3; i++)

5 # increment number of edges. Destination ID size is
 # either 4 or 8 bytes

DSTID ← Column_indices[R1+i]

Load DSTID with 2 from memory on the first iteration

Load DSTID with 4 from memory on the second iteration

10

[00190] This pseudocode may deliver the first edge destination identifier after executing the for-loop for the first time, and deliver the second edge destination identifier after executing the for-loop for the second time.

[00191] Assuming that the vertices and edges are stored in local DRAM, the first
15 load of R1 in the pseudocode may cause, for example, approximately a 60 nanosecond load from local DRAM. The load of R2 may be performed faster because of spatial locality. The load of R2 reuses the same cacheline that the R1 load used, resulting in a local L1 cache hit, which might take only 1 or 2 nanoseconds to execute. Reading register files and performing subtraction with R2 and R1 would take a couple
20 nanoseconds. Loading temporary variables and constants from the presumably L1 contents would take a couple nanoseconds for the for loop control to execute. A call to the column indices array might require access of DRAM, approximately 60 nanoseconds wherein this array at index R1+i is performed for a first iteration. However, the second iteration would benefit from spatial locality and be performed
25 with a relatively fast load of a couple nanoseconds, needing to only access the cache from the first iteration.

[00192] The EPT may store the starting address of the edges associated with each vertex ID, effectively causing the executed of the load of R1 a hardware table lookup, requiring only one or two nanoseconds, eliminating the DRAM access required in the
30 pseudocode above. Thus, latency may be reduced, which might be particularly valuable for frontier-based graph analytics.

[00193] FIGURE 25 is an illustration of an example EPT, according to embodiments of the present disclosure. FIGURE 25 may include example logical and physical representations of the EPT.

[00194] The EPT may be built and stored in hardware that accelerates the indirect
5 lookups of edges in a graph for frontier-based graph analytics applications. The logical representation of the EPT shows that for each vertex, there is a pointer to the starting address of its edges to be fetched. It may be assumed that the edges are sorted by their source identifier or the originating vertex identifier prior to the edge pointer assignments. Furthermore, it may be assumed that the edges are stored in memory
10 consecutively, as most of the graph analytics applications already perform these preprocessing steps before the algorithms begin executing. When no edges are associated with a particular vector identifier, the edge pointer address may be recorded as NULL. Because the EPT is built in hardware as a fixed-size, GAAC 1818 may make accommodations.

[00195] For example, GAAC 1818 may assume that there is enough hardware
15 resource to provide one edge pointer entry per vertex, as depicted in left side of the diagram. The EPT may record the vertex identifier of the source and the starting edge pointer address. This may cause removal of a level of indirection when accessing edges. When there are no edges associated with a particular vertex identifier, the edge
20 pointer address is null. When there are vertex identifiers that are not used, they are marked as invalid or null.

[00196] In another example, GAAC 1818 may assume that there are not enough
hardware resources to provide one edge pointer entry per vertex. In such an example, GAAC 1818 may utilize an approximate edge pointer table shown on the right. The
25 EPT may be constructed based on dividing entries over the number of vertices. Each entry might correspond to no more than two vertex identifiers in the example shown.

[00197] In such an approximate EPT, GAAC 1818 may record one entry for every K
vertices where K is the approximation factor, N is the number of entries in the Edge
Pointer Table, and $|V|$ is the number of vertices in the input graph. Accordingly, $K =$
30 $|V| / N$ and for a vector identifier of $v, v+1, v+2, \dots, v+K-1$ only the edge pointer address of vector identifier v is recorded. For both the EPT and the approximate EPT, GAAC 1818 may populate the table before performance of a graph algorithm. The

population may be made by using a software edge walker that walks through the edges as part of a preprocessing step. This incurs processing time on the order of magnitude of the number of edges, which is less than or equal to the sorting time of the edges during the preprocessing step. The tables can be populated using an instruction such as

5 WriteEdgePointer (Entry Number, Edge Pointer Address) and the vertex identifiers can be implicit (or omitted) to increase the number of entries for the table.

[00198] Pseudocode to operate the EPT or approximate EPT to retrieve all edges of vertex 3 may include the following, utilizing a function ReadEdgePtr(Entry Number) to access the EPT:

10

SRCID $\leftarrow$ 3

R1 $\leftarrow$ ReadEdgePtr(SRCID)

Load R1 with edge point address associated with third

vertex from EPT

15

(if R1 $\neq$ NULL)

If there are edges actually associated with this vertex

R2 $\leftarrow$ R1

read the first edge from memory R2 = (3, 2)

20

else

return NoEdges

While (R2.srcid $\leq$ SRCID) {

If this edge indeed has a srcid that is smaller than or

25

matches what we are looking for

if (R2.srcid $<$ SRCID)

If this edge has a SRCID smaller than what we are looking

for, discard and continue to next iteration. Only true if

using approximate EPT

30

continue

else DSTID $\leftarrow$ R2.dstid

Load destination with 2 from memory on first iteration

R2 ← [R1 + Edge size] # Marker1

Read the second edge from memory R2 = (3, 4)

5 [00199] After the first iteration of the while-loop through Marker1, the code may deliver the first edge destination identifier. After executing Marker1 and then the remaining part of the while-loop in a second iteration, the second edge destination identifier may be returned. The pseudocode, by using hardware access to the EPT, may avoid reads from memory that would otherwise be performed.

10 [00200] The memory layout of the edges is not intended to be restricted in the above examples. For example, R2.srcid may be implemented in any suitable manner from R2. The edges could be stored as SOA or AOS. If they are stored as (srcid, dstid) (srcid, dstid), then software may need to know the size of srcid and dstid and write code accordingly to extract srcid for each edge to perform the comparisons. Similarly, if these edges are stored as (srcid, srcid, srcid...) (dstid, dstid, dstid...) then software may
15 need to know the base address of each respective array in order to extract the elements from memory. The pseudocode may be implemented according to either method.

[00201] FIGURE 26 is flow chart of a method 2600 for data capture, according to embodiments of the present disclosure.

20 [00202] Method 2600 may be initiated by any suitable criteria. Furthermore, although Method 2600 describes operation of particular elements, method 2600 may be performed by any suitable combination or type of elements. For example, method 2600 may be implemented by the elements illustrated in FIGURES 1-25 or any other system operable to implement method 2600. As such, the preferred initialization point for method 2600 and the order of the elements comprising Method 2600 may
25 depend on the implementation chosen. In some embodiments, some elements may be optionally omitted, reorganized, repeated, or combined. Moreover, portions of method 2600 may be executed in parallel within itself.

[00203] At 2602, an instruction may be fetched and decoded. The instruction may be to perform a particular function or workload of graph analysis. The instruction
30 may be fetched, decoded, scheduled, executed, and retired. The graph command may be passed on to a graph accelerator implemented in hardware.

[00204] At 2604, subsequent execution in the pipeline of the graph accelerator may be parallelized. Source vertex fetches and their associated edge fetches may be parsed and routed into parallel pipes. Separately, subsequent destination execution may be parsed and routed into another set of parallel pipes. The two sets of pipes
5 may be linked, wherein output of a first pipe may be mapped into one or more second pipes. This may be applied to the remainder of the elements of 2600.

[00205] At 2606, a vertex source or property may be fetched from memory. The memory may be off-chip.

[00206] At 2608, edge fetching may be performed. The edge fetching may be
10 performed according to the type of access for the edges that are made. For example, it may be determined whether the fetching is frontier-based or stream-based. Frontier-based edge fetching may be performed by, for example, elements beginning at 2636. Stream-based edge fetching may be performed by, for example, 2610. At 2610, an edge source may be fetched from memory. At 2612, vertex destination
15 based upon the characteristics of the edge source may in turn be fetched from memory.

[00207] At 2614, computations for performance of the command may be performed. The specific computations may depend upon the specific function or workload that is to be executed by the command. The computations may be a part of
20 an apply stage.

[00208] At 2616, it may be determined whether there are back-to-back memory requests or atomicity of instructions that dictate that results from execution thus-far should be routed to a particular pipeline. For example, if a memory access is of a same location as a previous iteration or operation, the memory access may be routed
25 to the same pipeline that handled the previous execution. In another example, if a step of execution affects previous or subsequent execution as a part of an atomic region of code, the steps of execution of the atomic region of code might all be routed through the same pipeline.

[00209] At 2618, a vertex property may be read from an on-chip memory.

30 [00210] At 2620, any needed computations for performance of the instruction or command may be executed. These may be made, for example, as part of a reduce

stage. The specific computations may depend upon the specific function or workload that is to be executed by the command.

[00211] At 2622, vertex properties may be written to an on-chip memory. At 2624, it may be determined whether additional reduce stage operations need to be made. If so, method 2600 may proceed back to 2626. If not, method 2600 may proceed to 2626.

[00212] At 2626, a vertex property may be fetched from memory or from on-chip memory, as needed. At 2828, needed computations for performing the command or instructions for an apply stage may be made. The specific computations may depend upon the specific function or workload that is to be executed by the command. At 2630, vertex properties and vertex frontiers may be written to memory. At 2632, it may be determined whether additional apply operations are needed. If so, method 2600 may proceed back to 2626. Otherwise, method 2600 may proceed to 2634.

[00213] At 2634, it may be determined whether additional iterations of execution are needed for performance of the instruction. If so, method 2600 may proceed to, for example, 2606. Otherwise, method 2600 may terminate.

[00214] At 2636, access may be made to a graph that is stored in compressed row format to fetch edges. The data may be stored in an EPT, or mechanism similar to an EPT. The EPT may be stored in hardware. The particular steps of accessing the EPT may depend upon the function that is to be executed. For example, the following steps describe cases wherein all edges from a given vertex need to be loaded.

[00215] At 2638, the EPT may be read for a given vertex. The result of the read may be an address. The address may be stored in, for example, a register R1.

[00216] At 2640, if the address is not null, then the values of the edge may be read by reading the values stored at the address. The value may be stored in, for example, a register R2.

[00217] At 2642, a while loop may be performed. The while loop may be conditioned upon the edge having a source that is smaller or equal to the given vertex. If the edge's source is strictly smaller than the given vertex, then the loop may continue processing but without other interim action to the next iteration of the loop. If the edge's source is the same as the given vertex, the destination vertex

identifier of an edge's source as an identified destination may be loaded. The destination vertex identifier of an edge's source may be loaded as an identified destination of the edge, to be provided to subsequent execution stages. A second edge may be read from memory, and 2642 repeated.

- 5 **[00218]** At 2644, destination identifiers may be returned. Method 2600 may repeat at, for example, 2638 or 2612.

[00219] 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
10 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.

[00220] Program code may be applied to input instructions to perform the functions described herein and generate output information. The output information
15 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.

[00221] The program code may be implemented in a high level procedural or
20 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.

25 **[00222]** 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
30 medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

[00223] 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.

[00224] 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.

[00225] 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.

[00226] 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.

[00227] Embodiments of the present disclosure include a processor or an accelerator circuit. The processor may include a front end including circuitry to receive and decode an instruction, the instruction to perform a graph analytic function and pass the instruction to a graph accelerator. The graph accelerator may be included in the processor or accelerator circuit. In combination with any of the above embodiments, the graph accelerator may include circuitry to process graph vertices and graph edges as datatypes, execute the instruction, and pass results of the instruction to a memory subsystem of the processor. In combination with any of the above embodiments, the graph accelerator may include circuitry to stream inputs of vertex properties in sequence and execute the instruction upon the streamed vertex properties. In combination with any of the above embodiments, the graph accelerator may include a parallel execution pipeline, the parallel execution pipeline including circuitry to partition vertex execution into a plurality of parallel execution pipelines. In combination with any of the above embodiments, the graph accelerator may include an on-chip memory including granular data formats corresponding to vertex and edge data. In combination with any of the above embodiments, the graph accelerator may include circuitry to store intermediate vertex and edge values on the on-chip memory. In combination with any of the above embodiments, the graph accelerator may include an edge table implanted in hardware, the edge table to represent a candidate graph in compressed form. In combination with any of the above embodiments, the graph accelerator may include an edge table implanted in hardware, the edge table to represent a candidate graph in compressed form. In combination with any of the above embodiments, the graph accelerator may include circuitry to access the edge table to fetch edge information to execute the instruction. In combination with any of the above embodiments, the graph accelerator may include an edge table implanted in hardware, the edge table to represent a candidate graph in compressed form. In combination with any of the above embodiments, the graph accelerator may include an entry in the edge table is to associate a vertex identifier and a starting edge pointer address. Embodiments of the present disclosure may include methods performed by

any of the above combinations. Furthermore, embodiments of the present disclosure may include an apparatus comprising means for performing any of the methods or operations of the above embodiments.

CLAIMS

What is claimed is:

1. An accelerator circuit, comprising circuitry to:
execute an instruction, the instruction to perform a graph analytic
5 function; and
process graph vertices and graph edges as datatypes; and
pass results of the instruction to a memory of a system.
2. The accelerator circuit of Claim 1, further comprising circuitry to stream
10 inputs of vertex properties in sequence and execute the instruction upon the streamed
vertex properties.
3. The accelerator circuit of Claim 1, further comprising a parallel execution
pipeline, the parallel execution pipeline including circuitry to partition vertex
15 execution into a plurality of parallel execution pipelines.
4. The accelerator circuit of Claim 1, further comprising:
an on-chip memory including granular data formats corresponding to vertex and
edge data; and
20 circuitry to store intermediate vertex and edge values on the on-chip memory.
5. The accelerator circuit of Claim 1, further comprising an edge table
implanted in hardware, the edge table to represent a candidate graph in compressed
form.
25
6. The accelerator circuit of Claim 1, further comprising:
an edge table implanted in hardware, the edge table to represent a candidate
graph in compressed form; and
circuitry to access the edge table to fetch edge information to execute the
30 instruction.

7. The accelerator circuit of Claim 1, wherein:
the accelerator circuit further comprises an edge table implanted in hardware,
the edge table to represent a candidate graph in compressed form; and
an entry in the edge table is to associate a vertex identifier and a starting edge
5 pointer address.
8. The accelerator circuit of Claim 1, wherein wherein the accelerator circuit
further includes a parallel execution pipeline, the parallel execution pipeline
including circuitry to partition edge prefetches into a plurality of parallel execution
10 pipelines.
9. The accelerator circuit of Claim 1, wherein wherein the accelerator circuit
further includes circuitry to stream inputs of edge properties in sequence and execute
the instruction upon the streamed vertex properties.
15
10. A processor, comprising any of the accelerator circuits of Claims 1-9.
11. A method, comprising:
offloading execution of an instruction to a graph accelerator, the
20 instruction to perform a graph analytic function;
in the graph accelerator, processing graph vertices and graph edges as
datatypes; and
pass results of the instruction to a memory of a system.
12. The method of Claim 11, further comprising, in the graph accelerator,
25 streaming inputs of vertex properties in sequence and execute the instruction upon
the streamed vertex properties.
13. The method of Claim 11, further comprising, in the graph accelerator,
30 executing the instruction through a parallel execution pipeline, the parallel execution
pipeline including circuitry to partition vertex execution into a plurality of parallel
execution pipelines.

14. The method of Claim 11, further comprising storing intermediate vertex and edge values on an on-chip memory in the graph accelerator, the on-chip memory including granular data formats corresponding to vertex and edge data.

5

15. The method of Claim 11, further comprising accessing an edge table to fetch edge information to execute the instruction, the edge table implanted in hardware, the edge table to represent a candidate graph in compressed form.

10 16. The method of Claim 11, further comprising accessing an edge table implanted in hardware, the edge table to represent a candidate graph in compressed form, wherein accessing the edge table includes obtaining a starting edge pointer from an entry associating the starting edge pointer with a vertex identifier.

15 17. The method of Claim 11, further comprising accessing an entry in an edge table implanted in hardware, the edge table to represent a candidate graph in compressed form, the entry to associate a vertex identifier and a starting edge pointer address.

20 18. The method of Claim 11, further comprising partitioning edge prefetches into a plurality of parallel execution pipelines.

19. The method of Claim 11, further comprising streaming inputs of edge properties in sequence and execute the instruction upon the streamed vertex
25 properties.

20. An apparatus, comprising means for performing any of the methods of Claims 11-19.

30

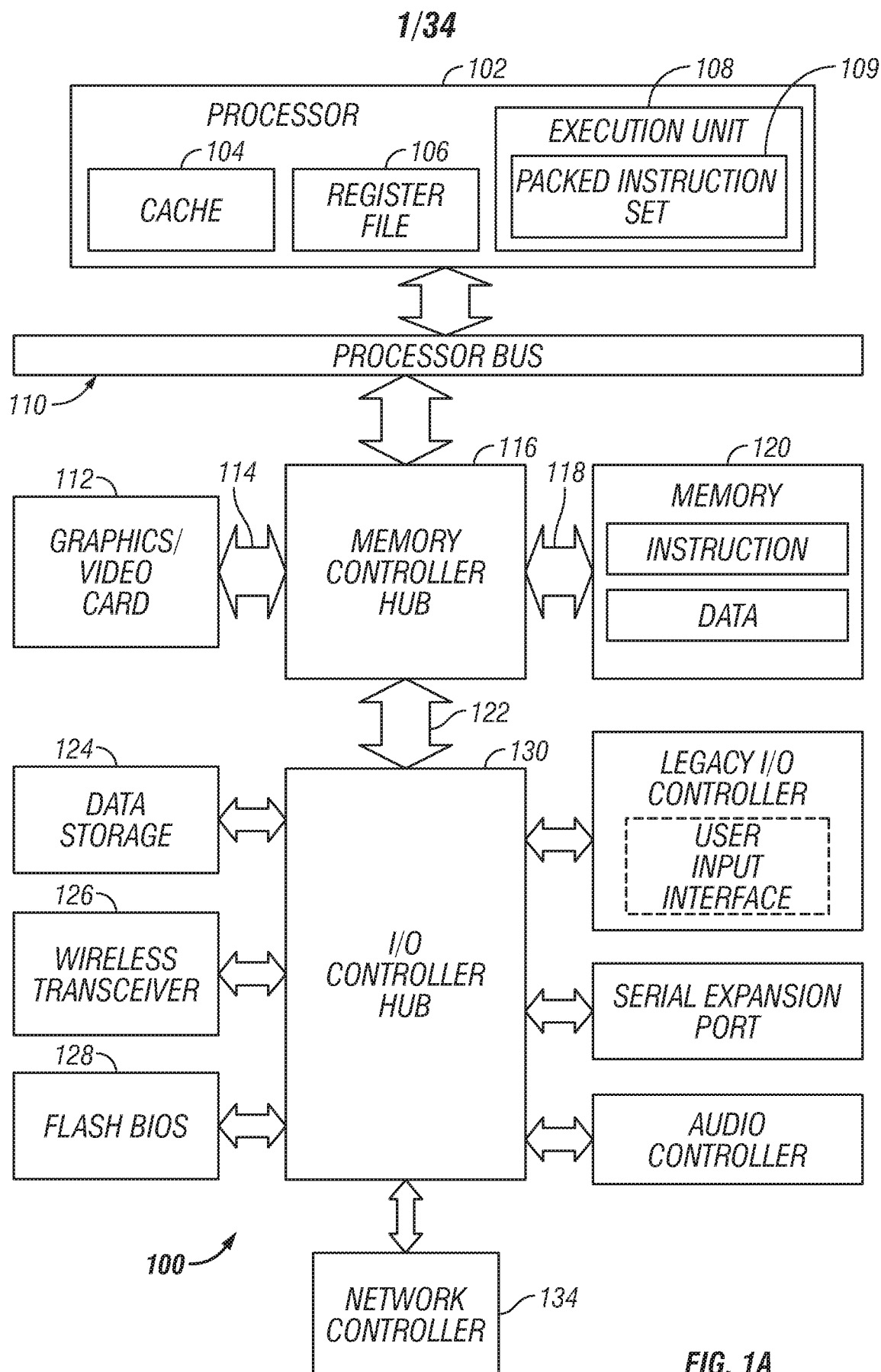


FIG. 1A

2/34

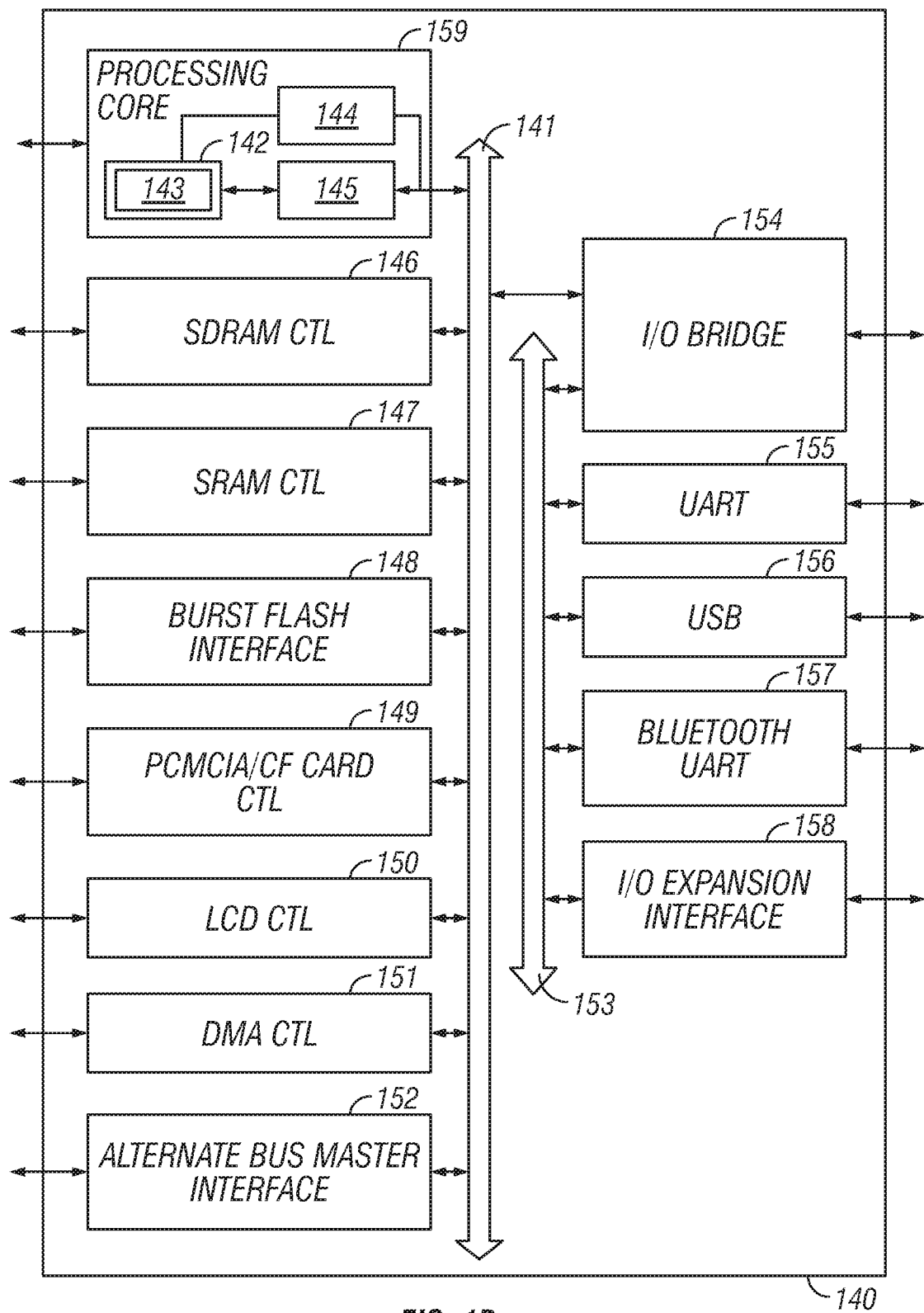


FIG. 1B

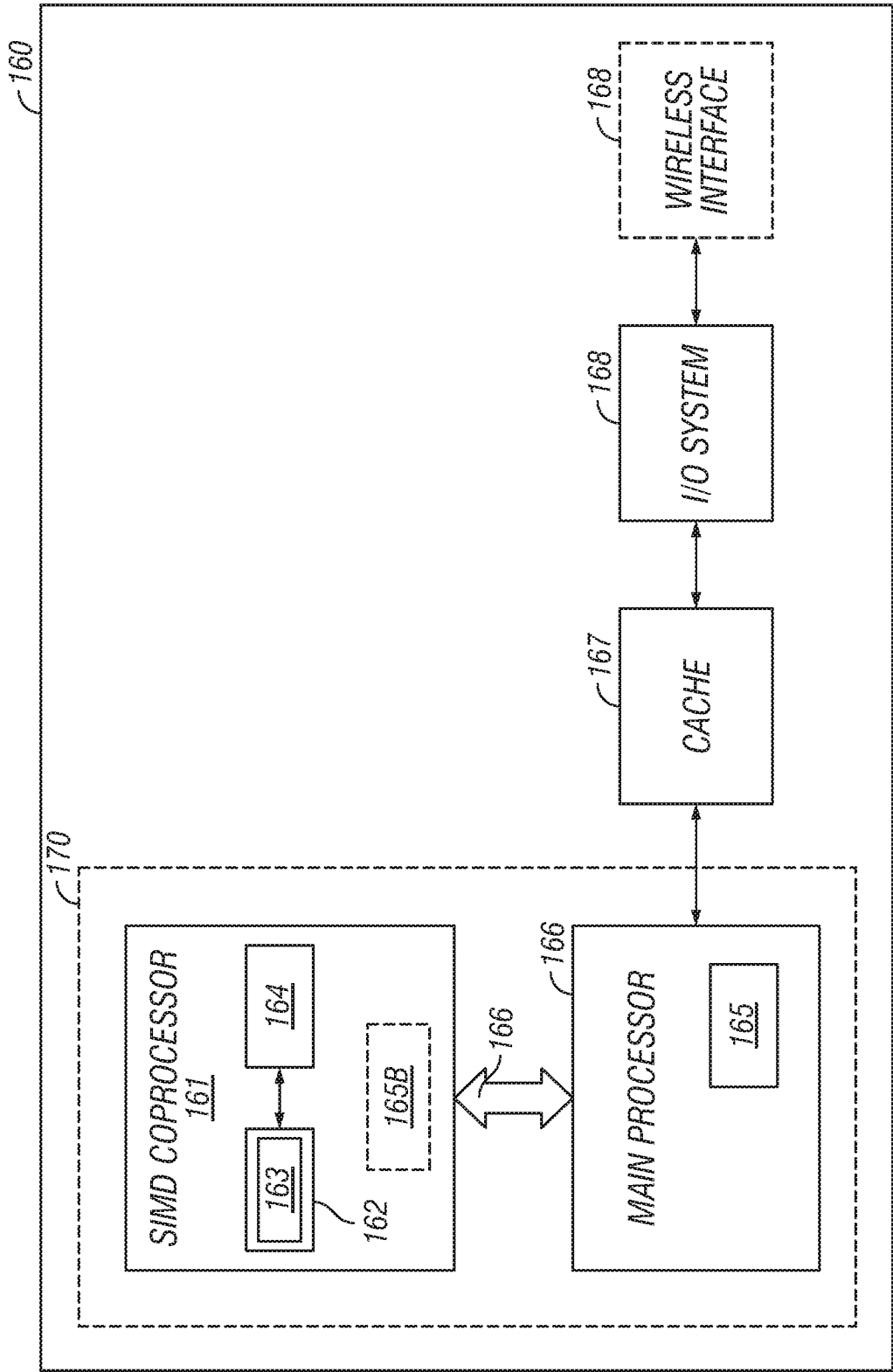


FIG. 1C

4/34

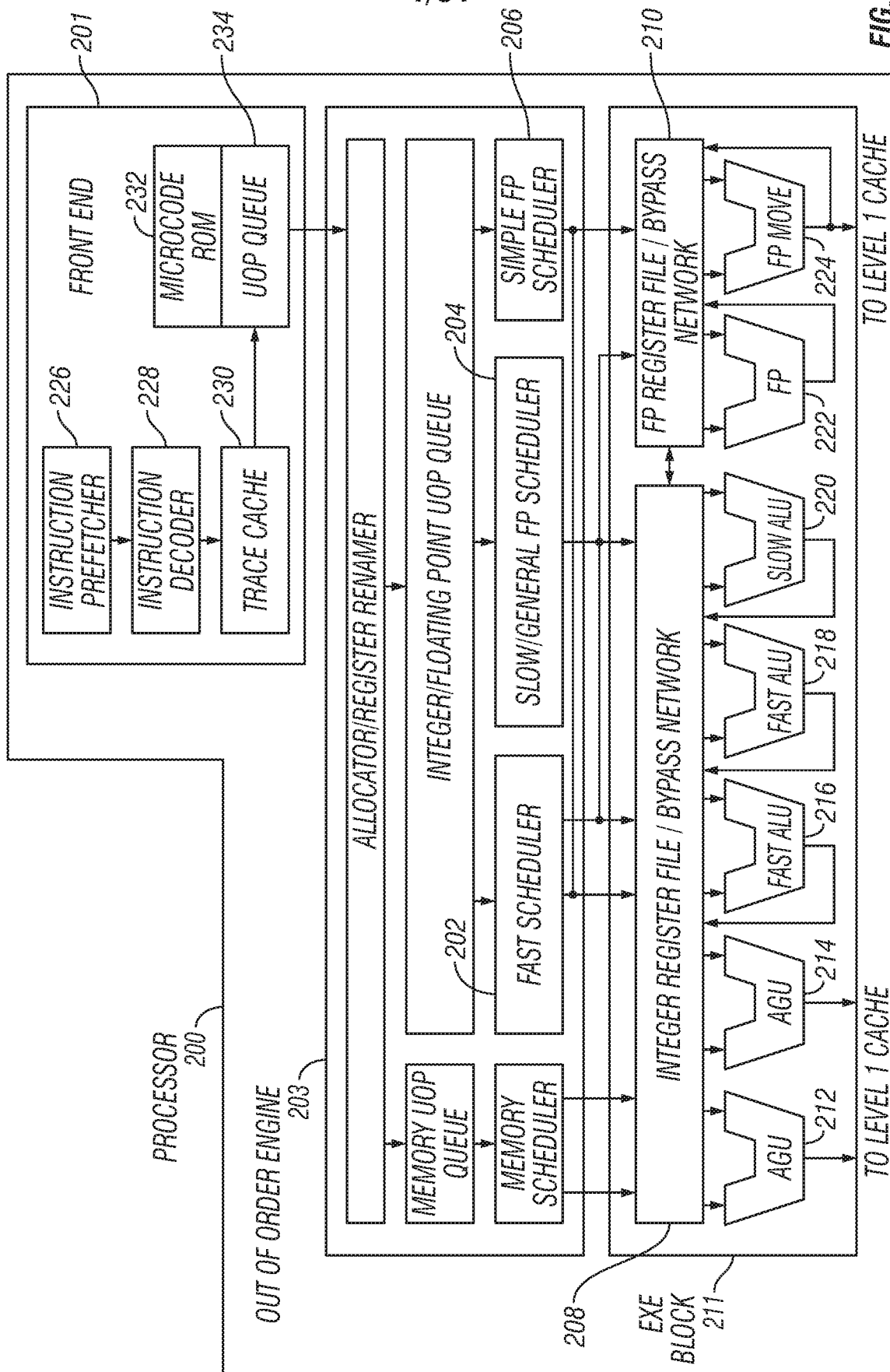


FIG. 2

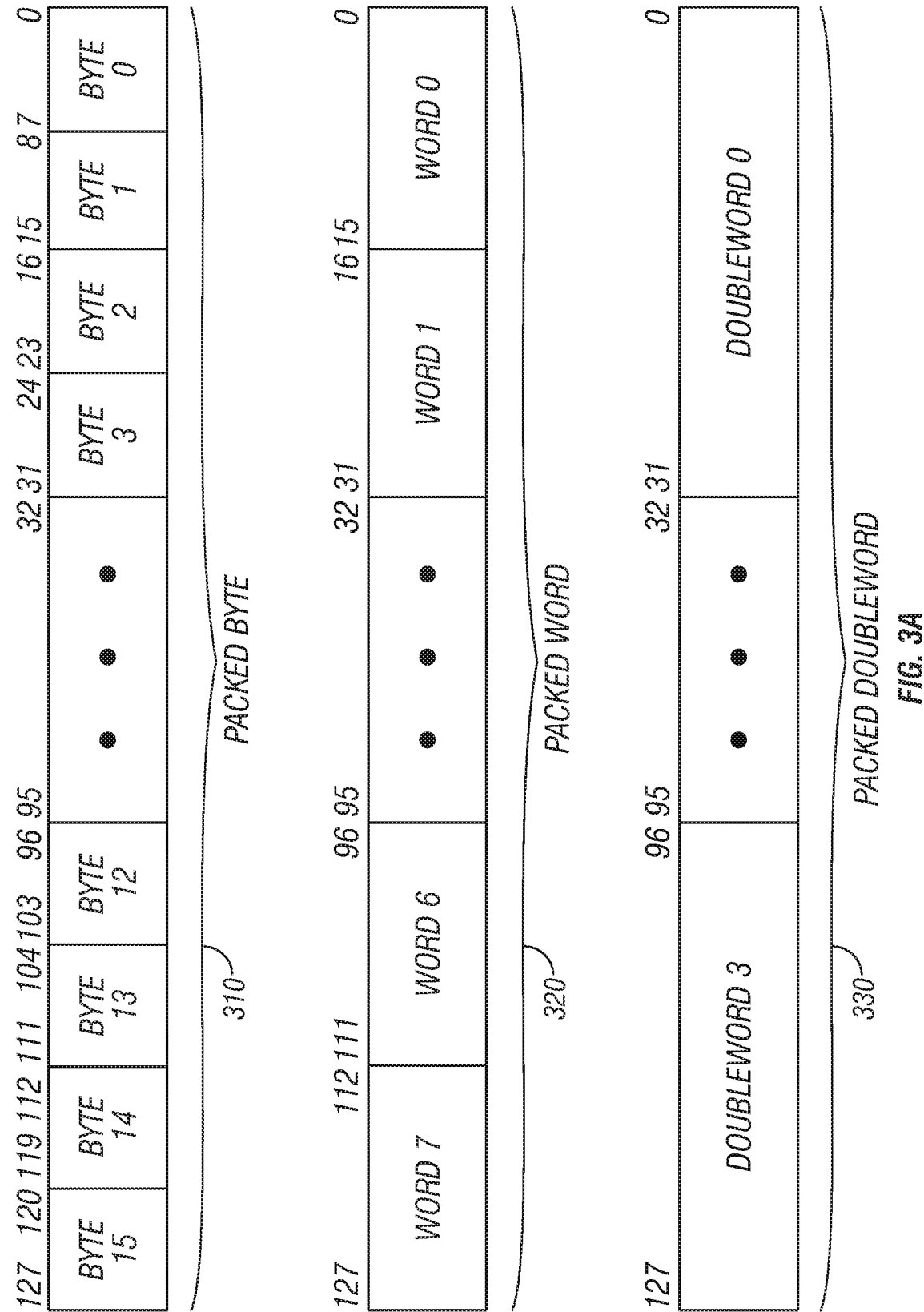


FIG. 3A

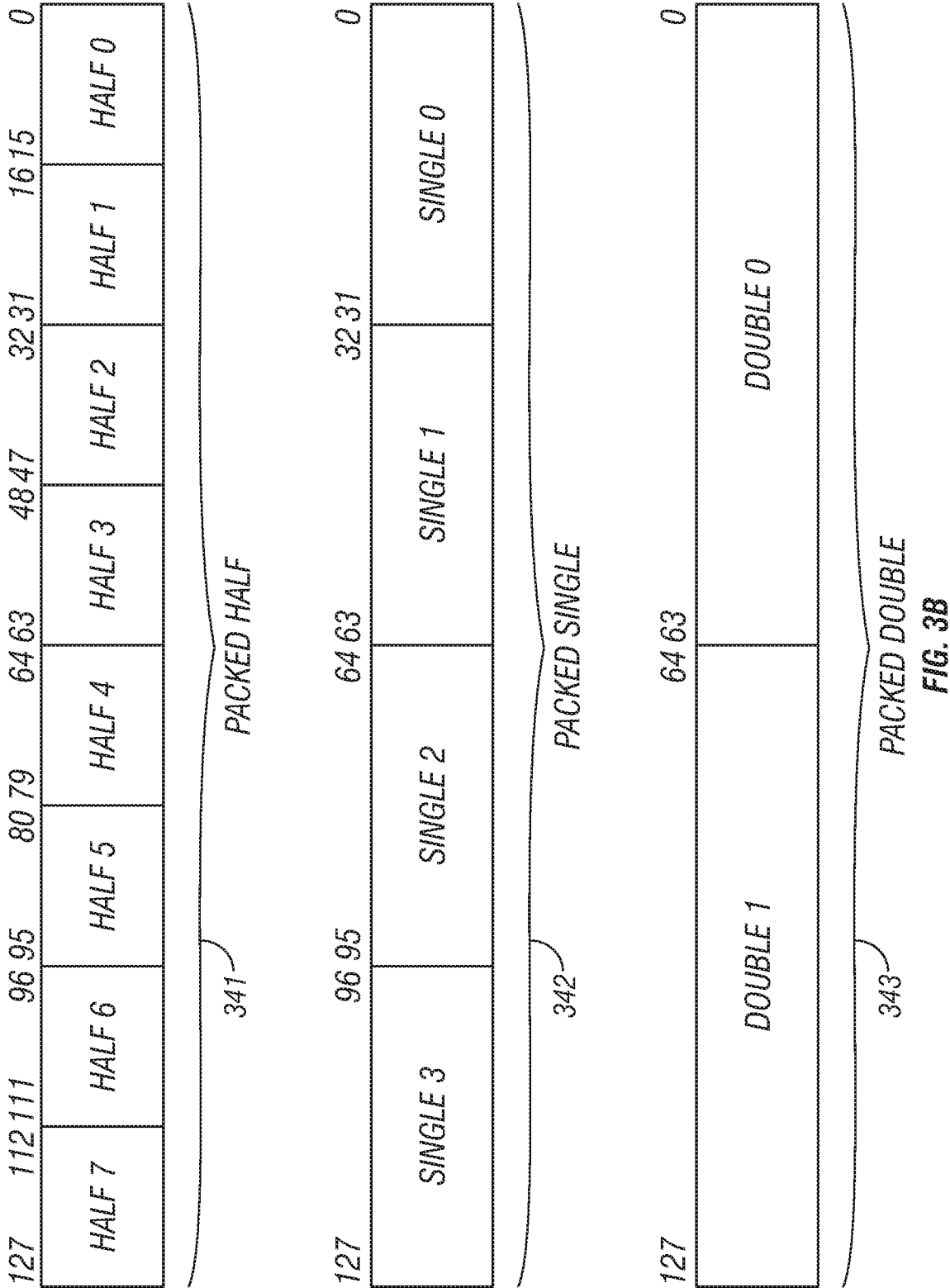
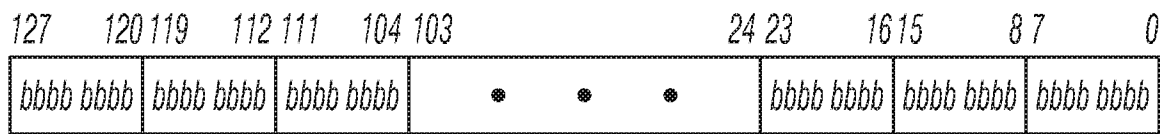
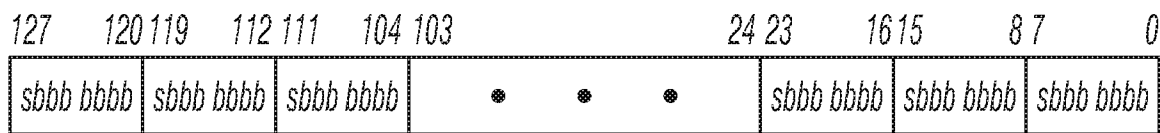


FIG. 3B

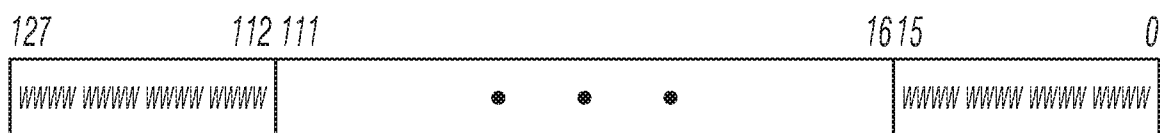
7/34



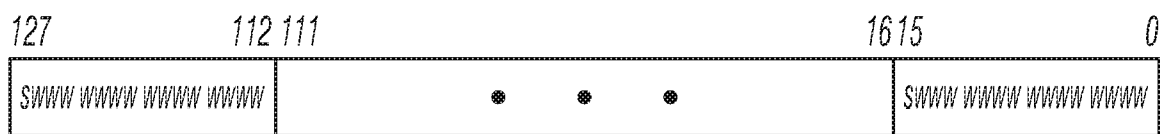
UNSIGNED PACKED BYTE REPRESENTATION 344



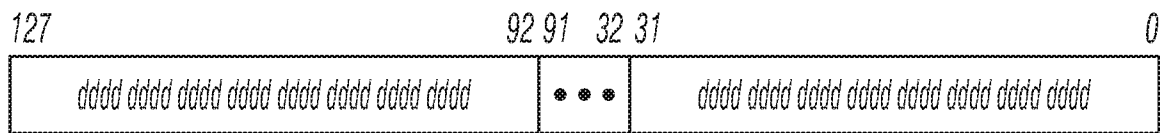
SIGNED PACKED BYTE REPRESENTATION 345



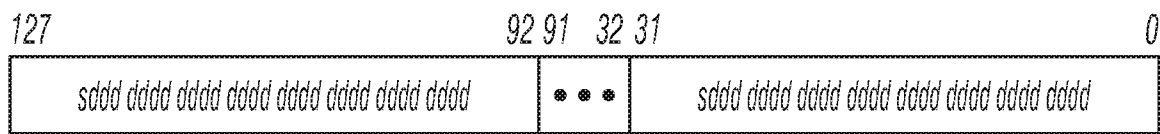
UNSIGNED PACKED WORD REPRESENTATION 346



SIGNED PACKED WORD REPRESENTATION 347



UNSIGNED PACKED DOUBLEWORD REPRESENTATION 348



SIGNED PACKED DOUBLEWORD REPRESENTATION 349

FIG. 3C

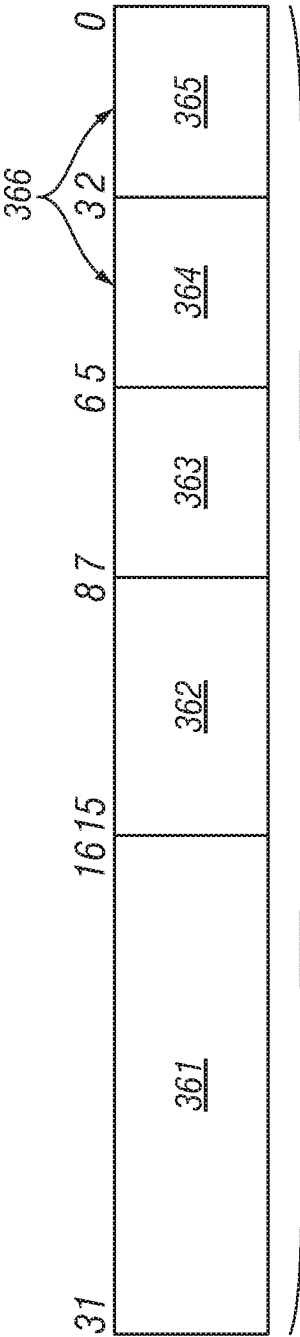


FIG. 3D

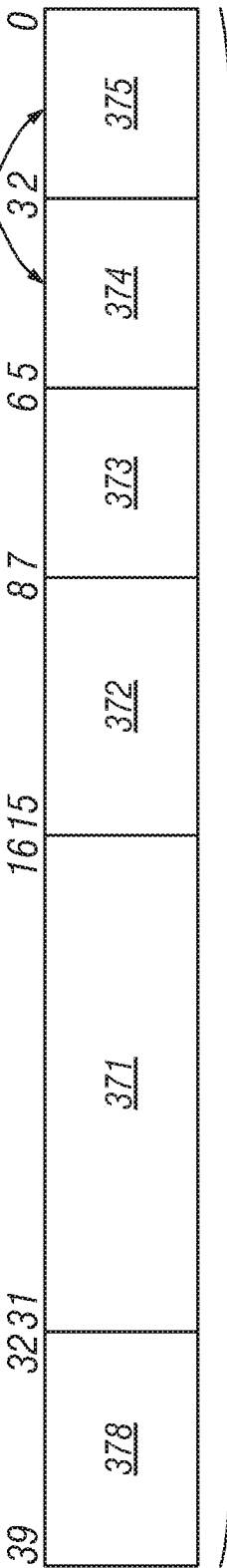


FIG. 3E

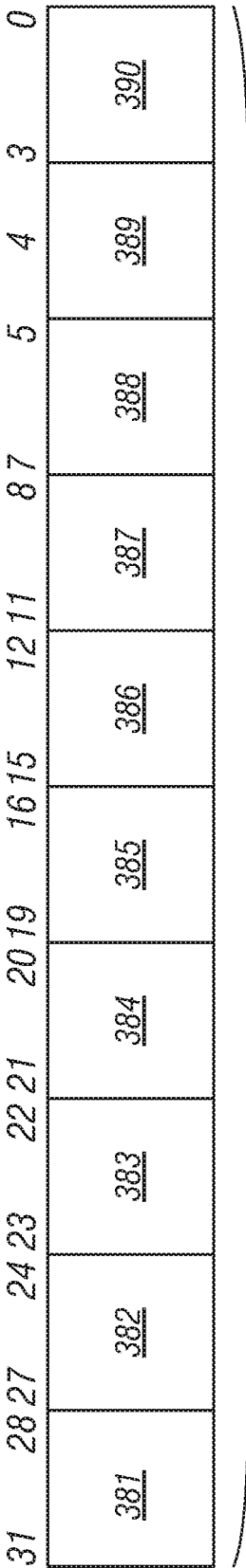


FIG. 3F

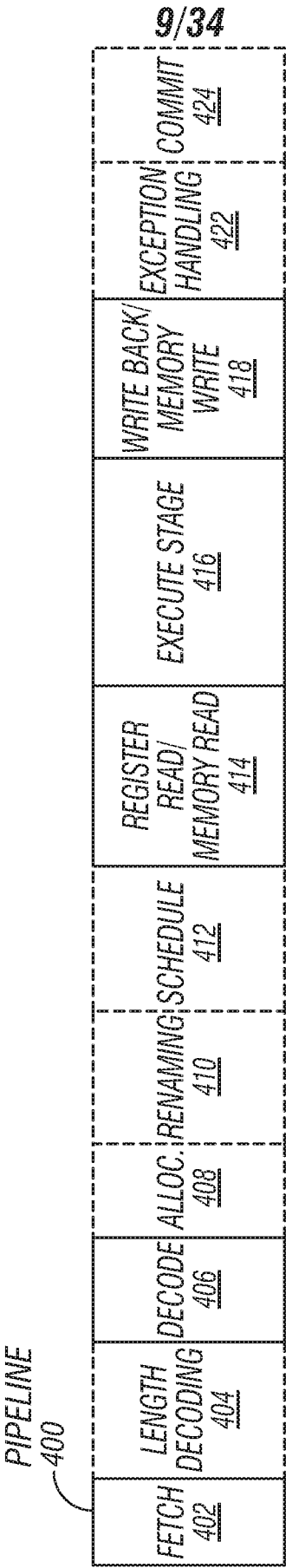


FIG. 4A

10/34

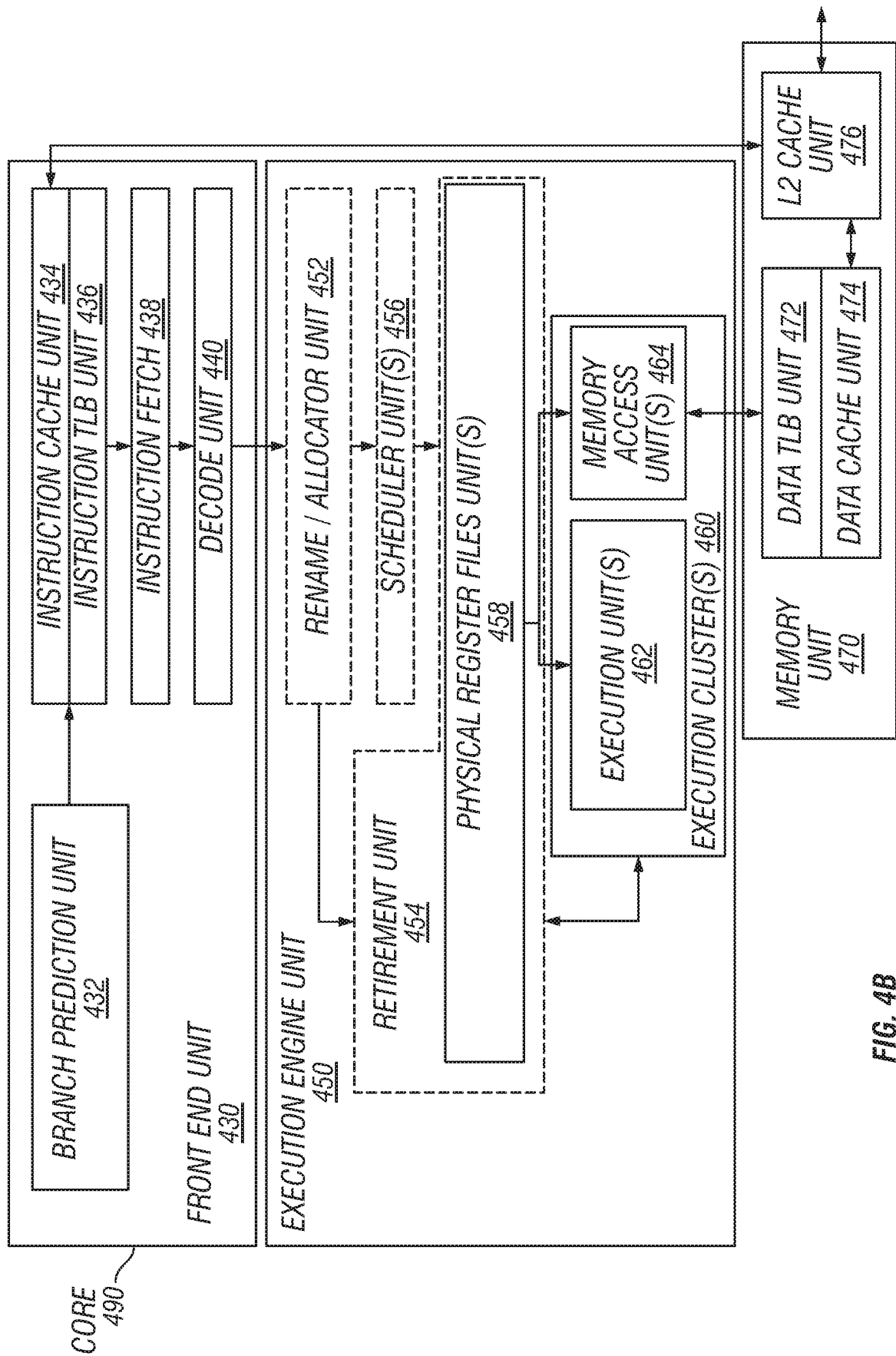


FIG. 4B

11/34

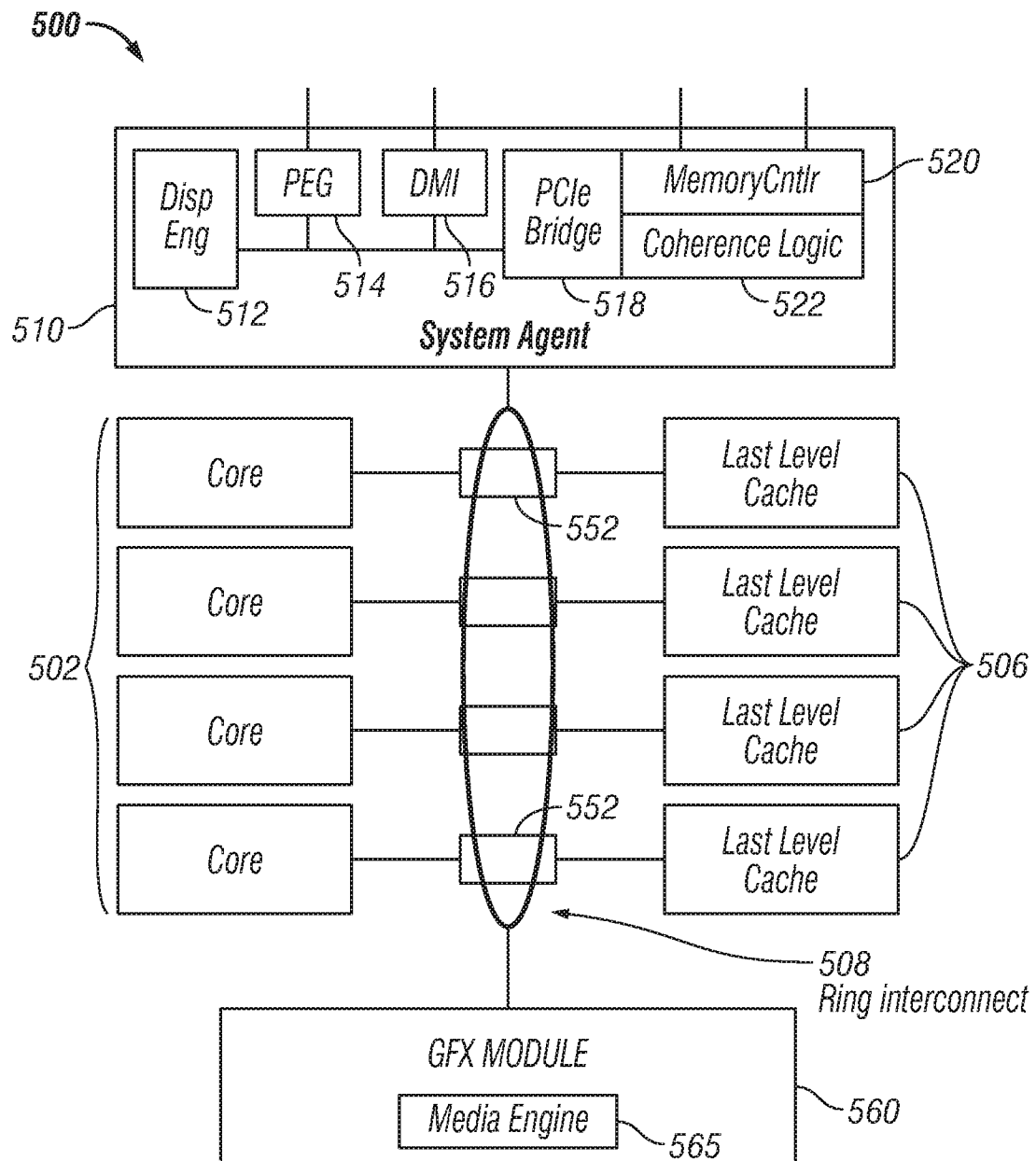


FIG. 5A

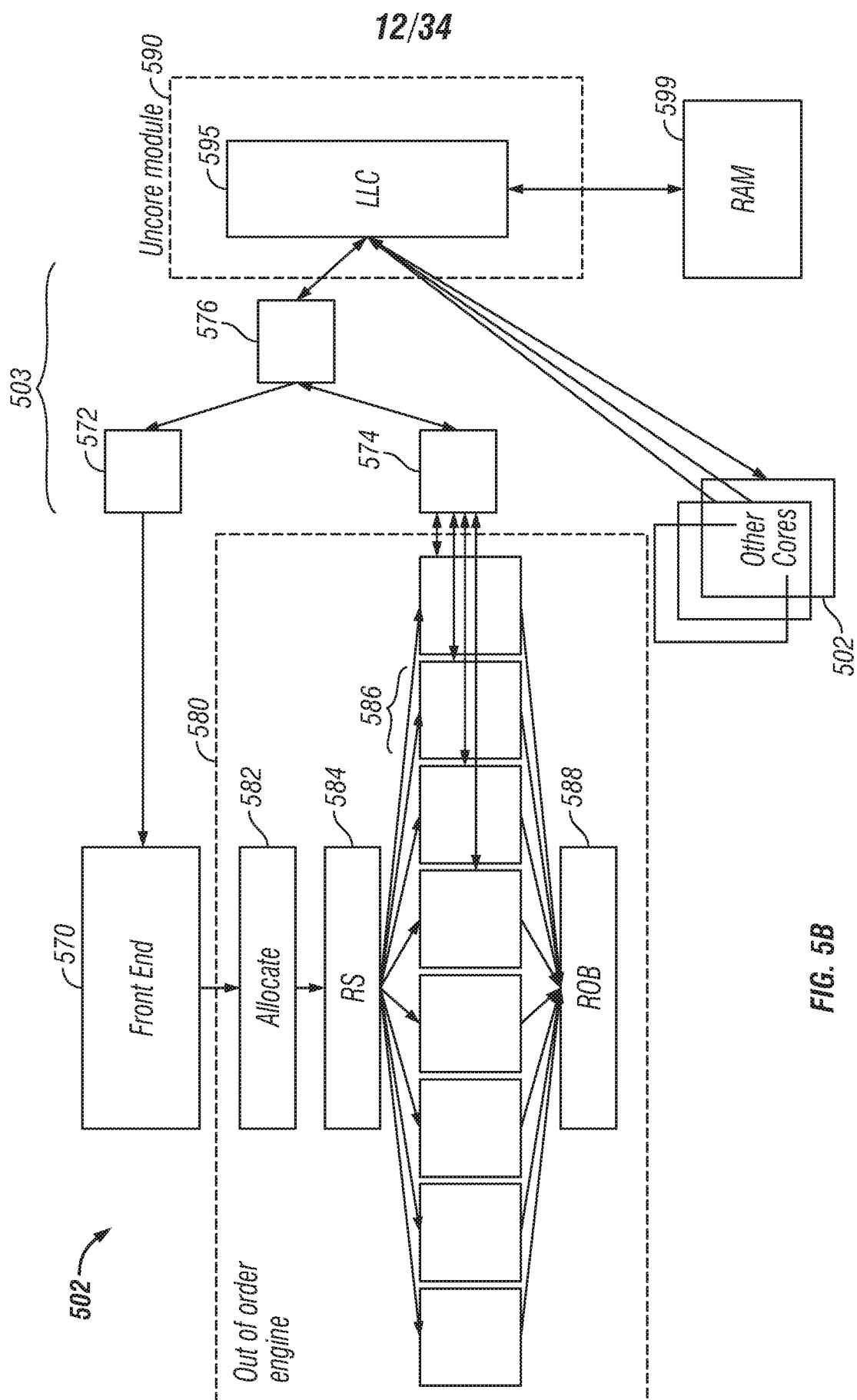


FIG. 5B

13/34

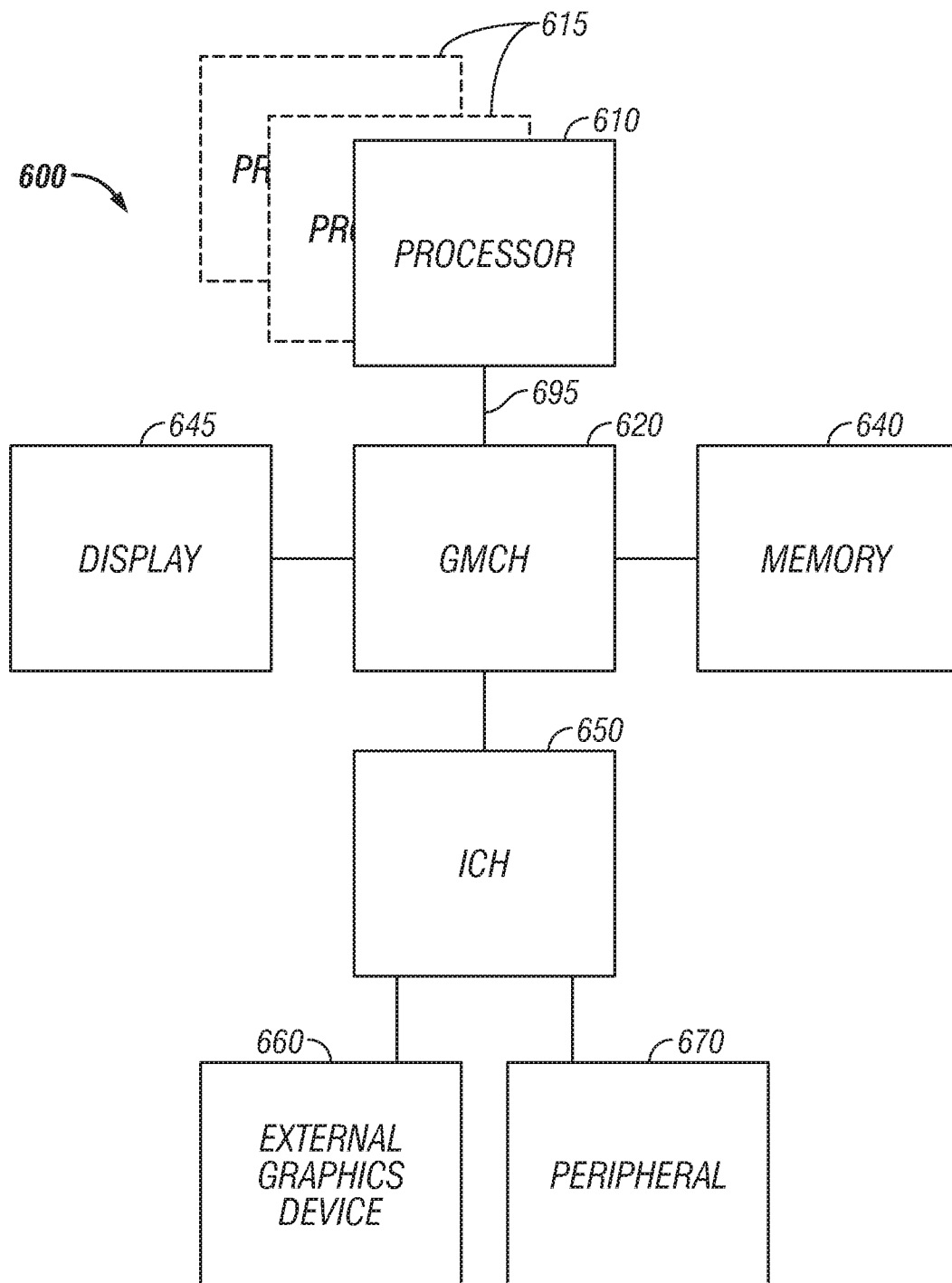


FIG. 6

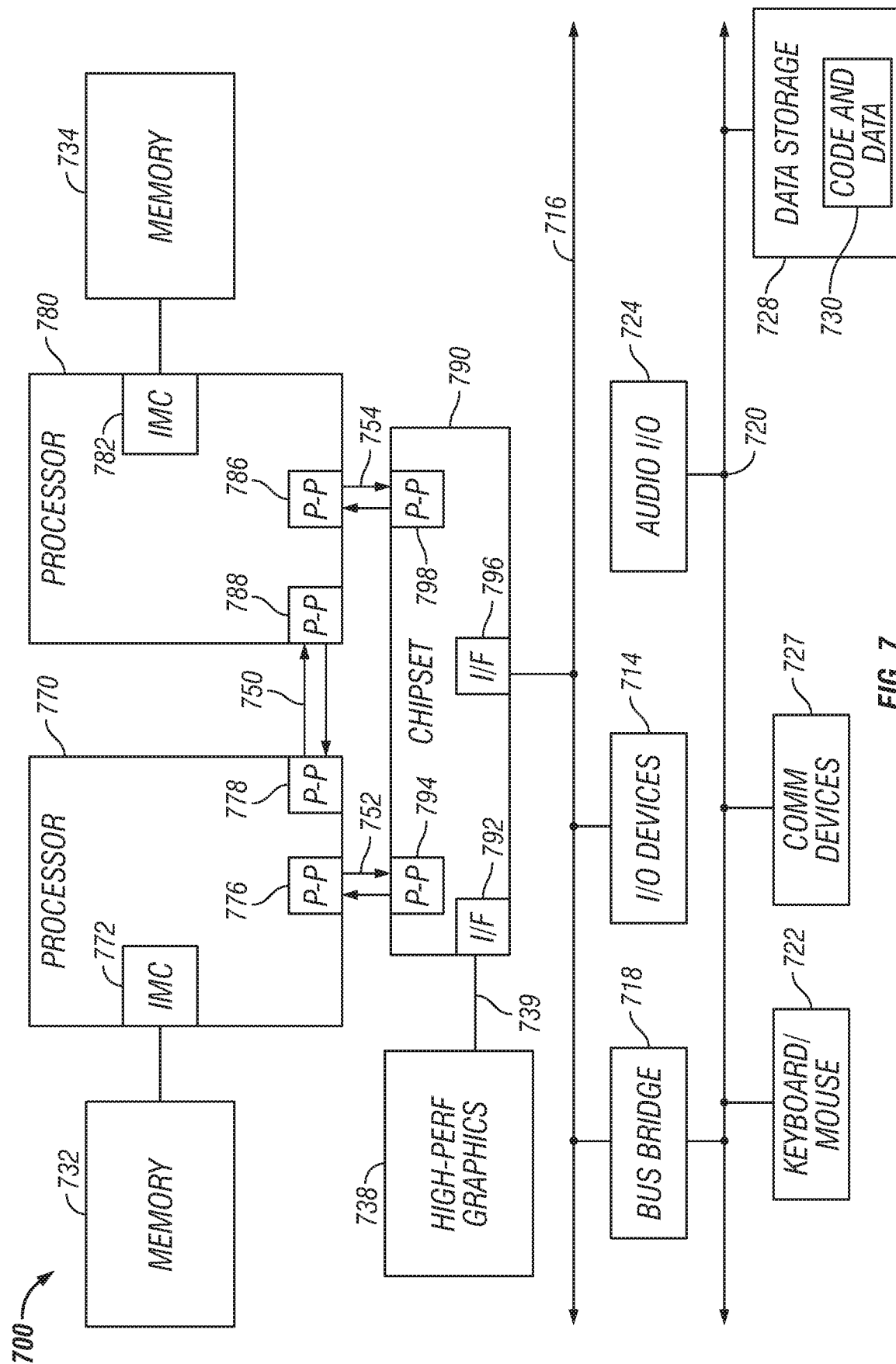


FIG. 7

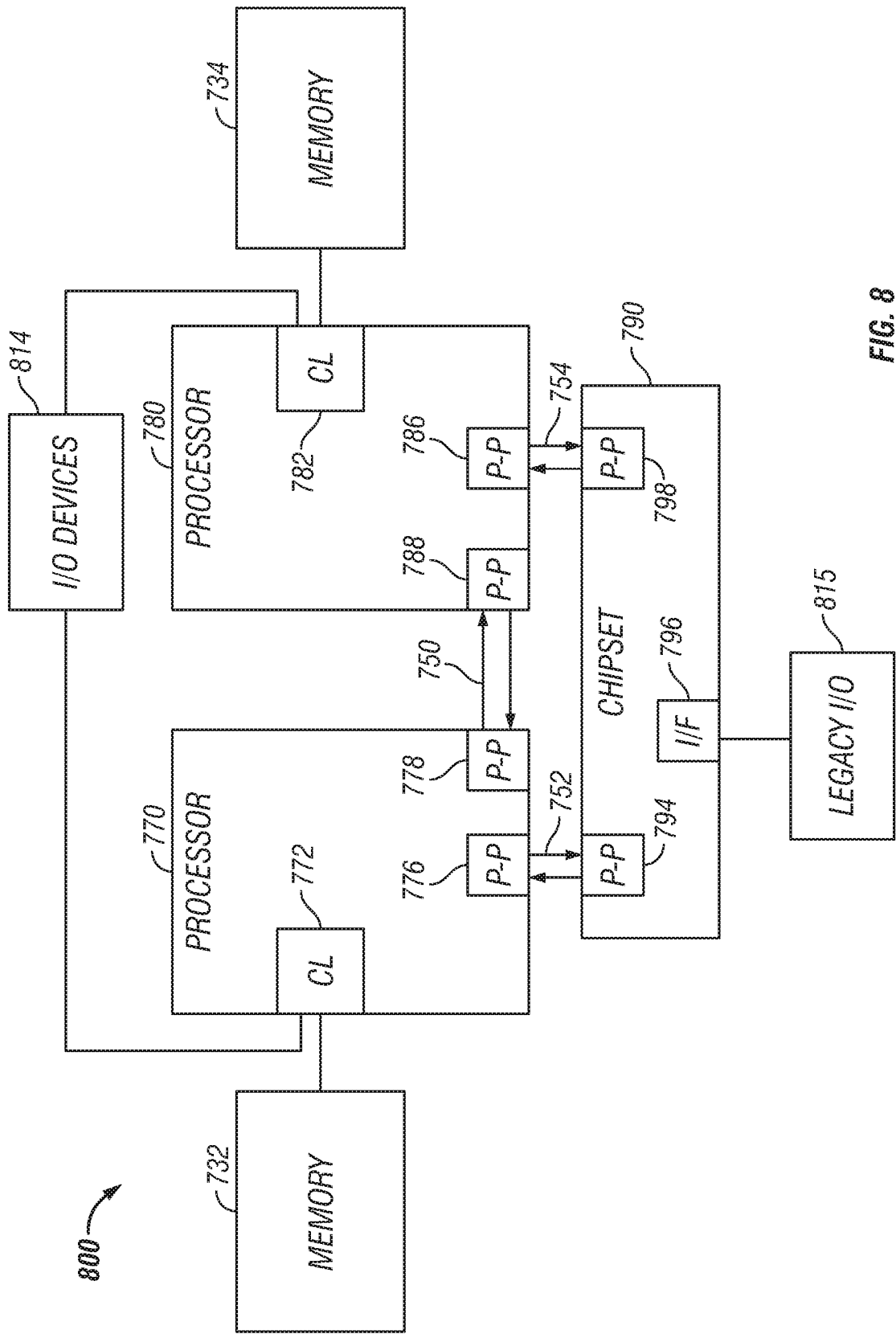


FIG. 8

16/34

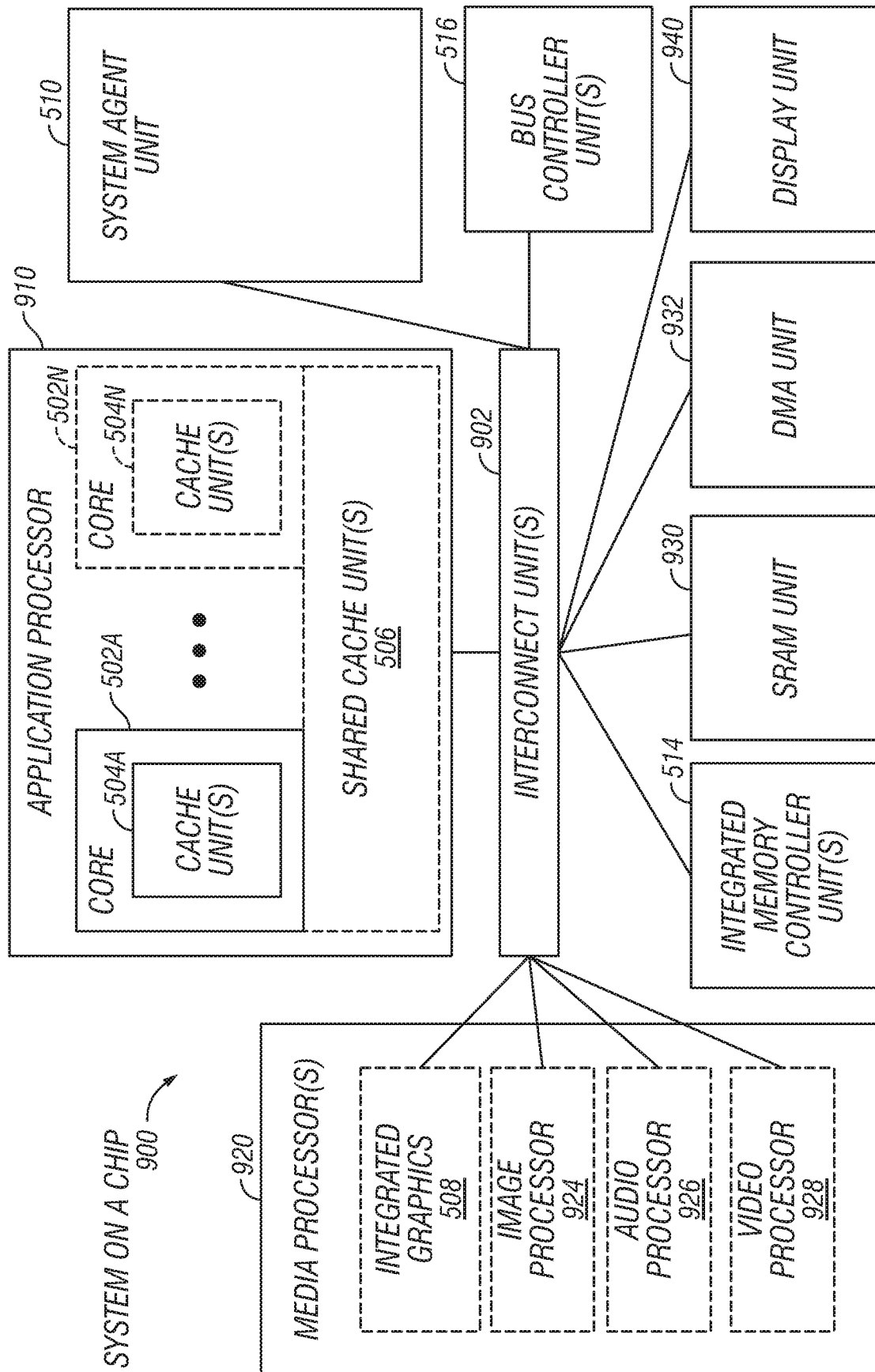


FIG. 9

17/34

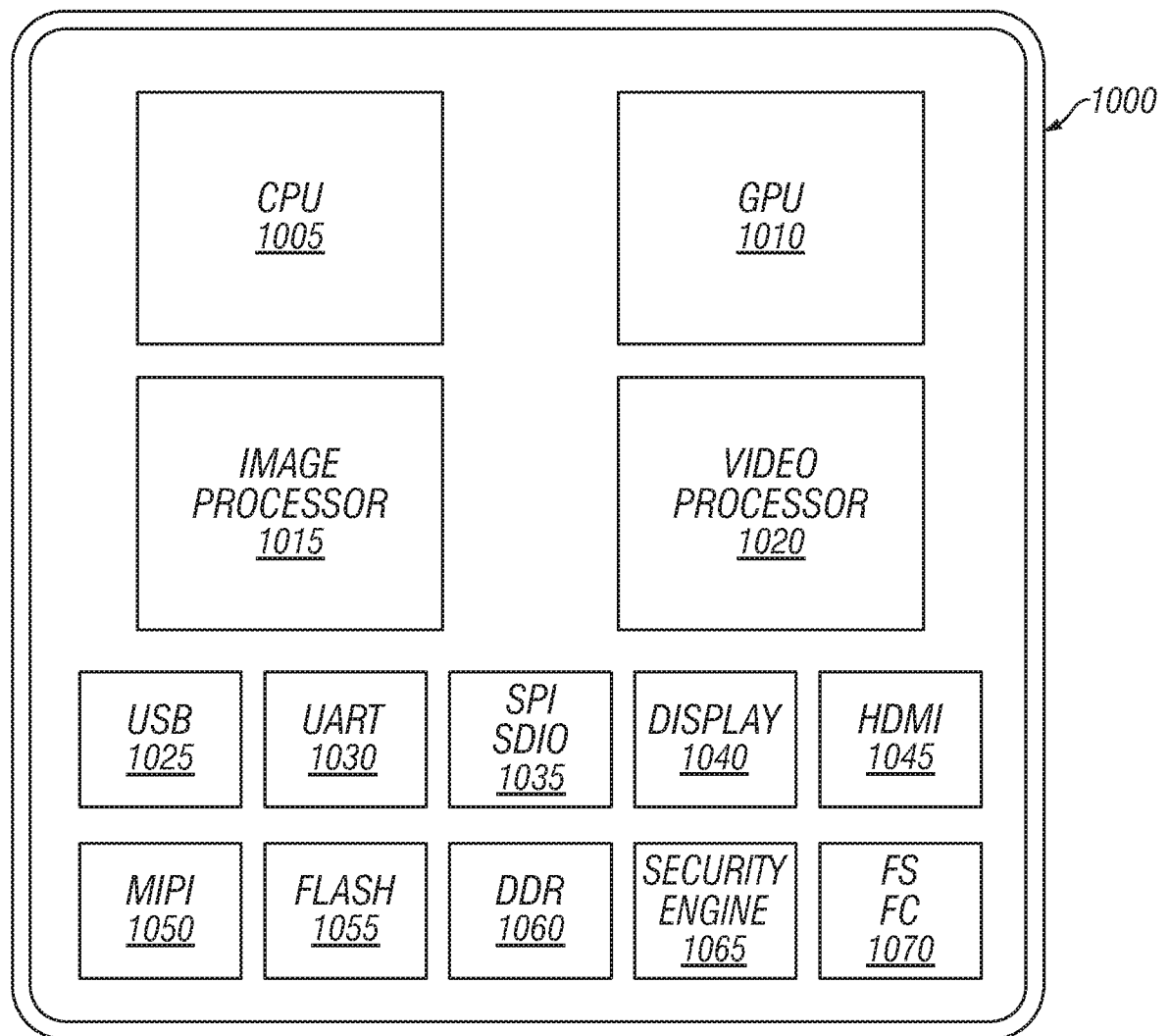


FIG. 10

18/34

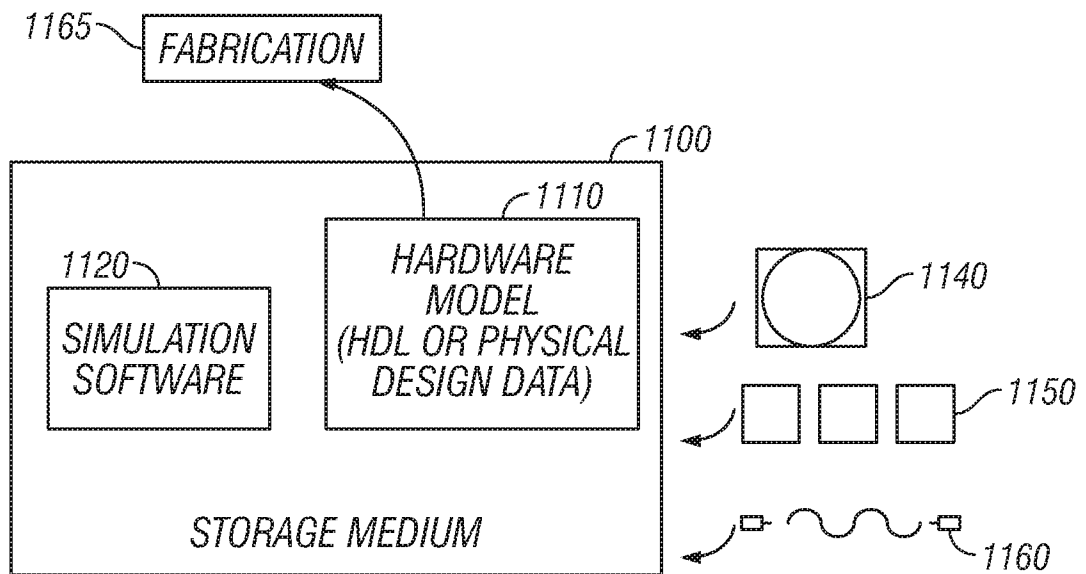


FIG. 11

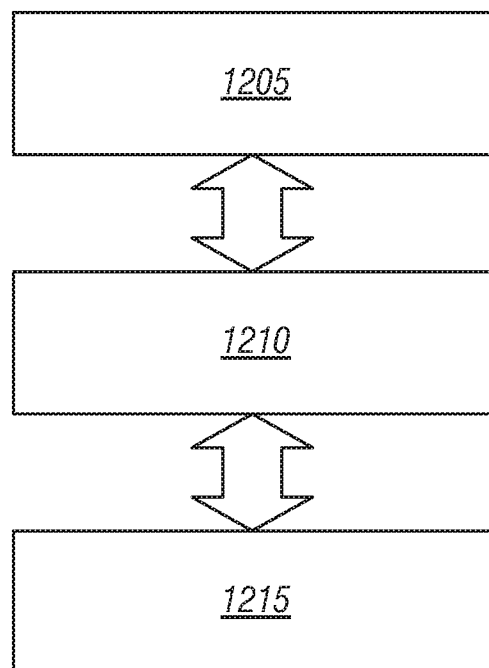


FIG. 12

19/34

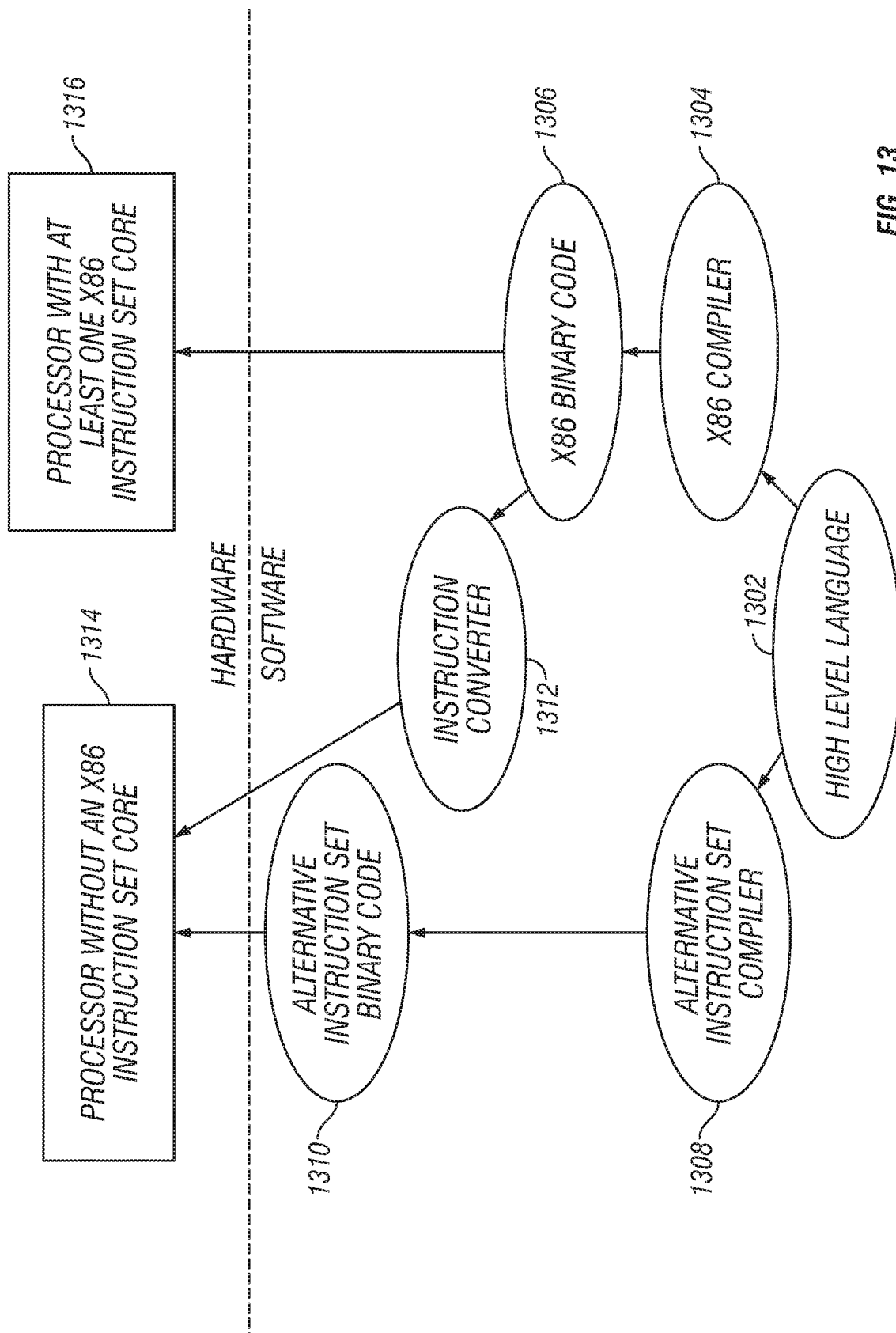


FIG. 13

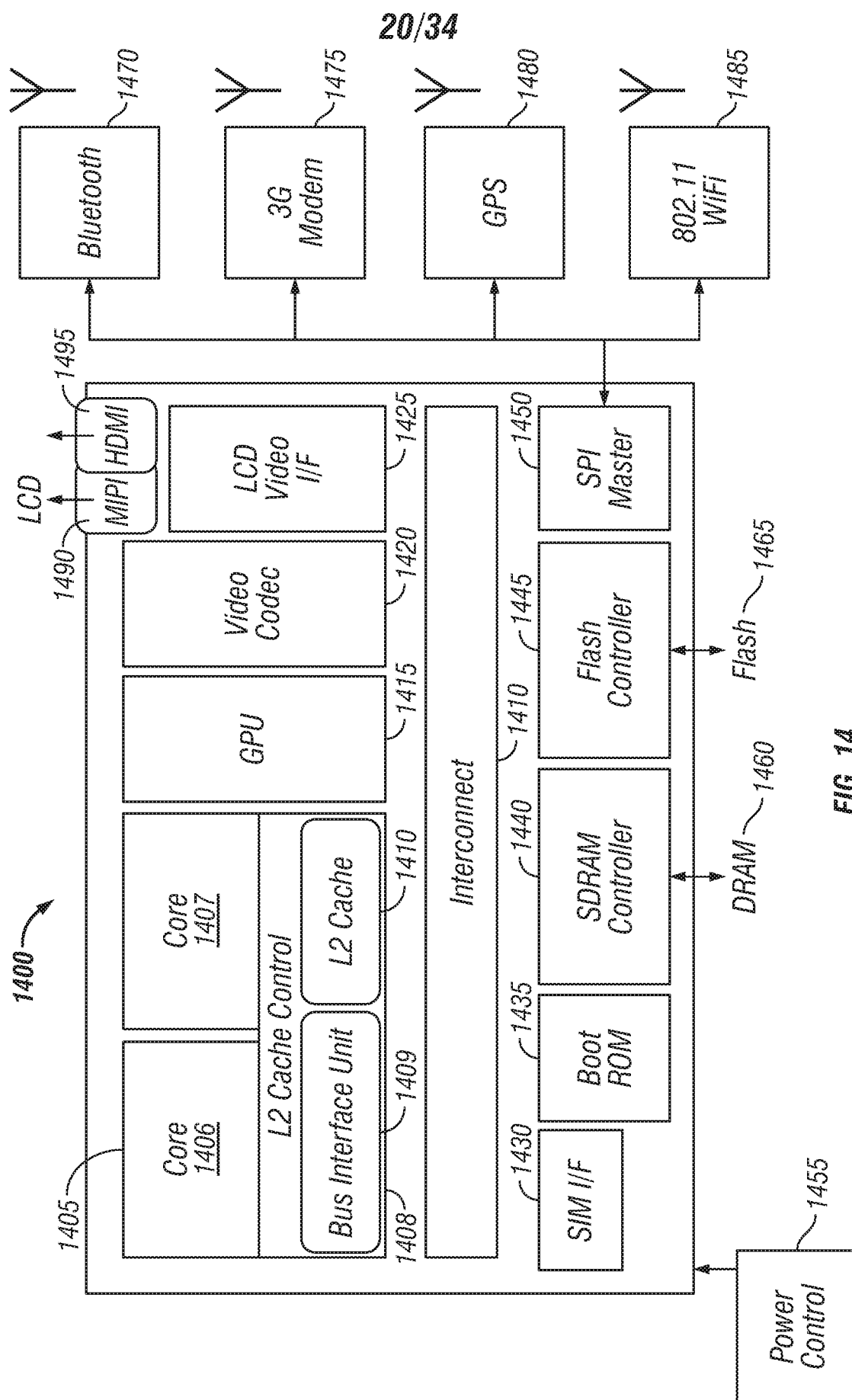


FIG. 14

21/34

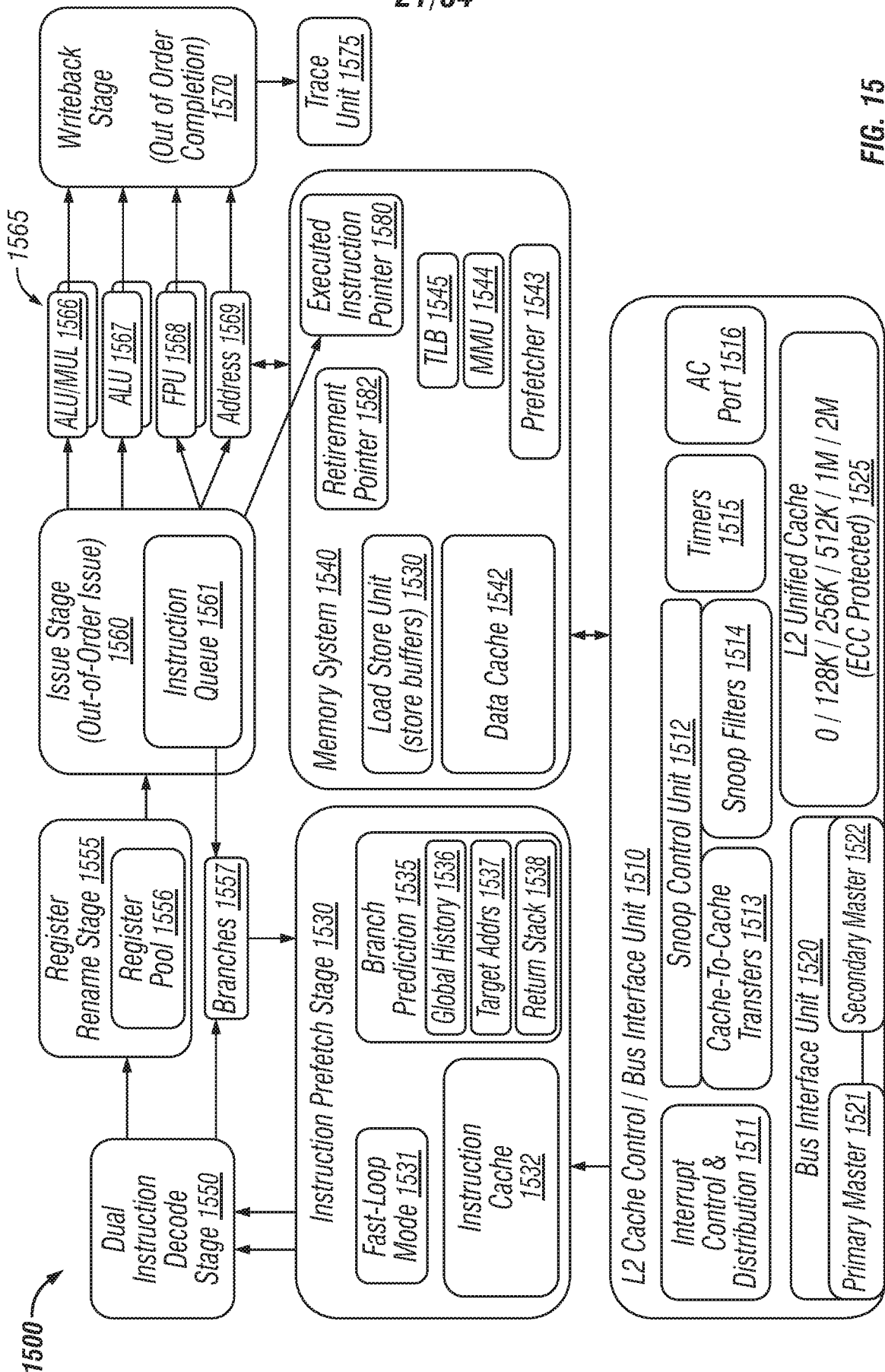


FIG. 15

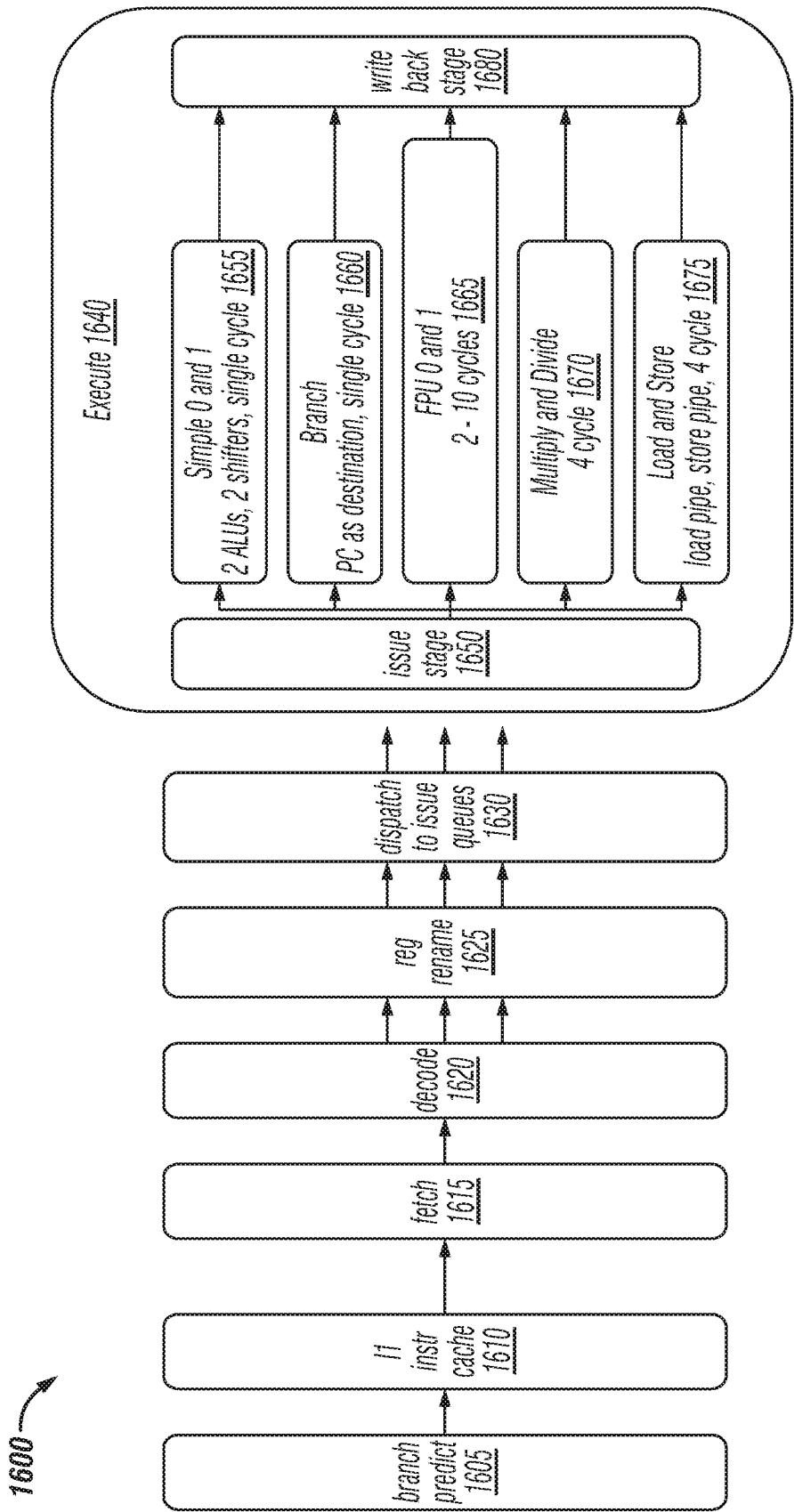


FIG. 16

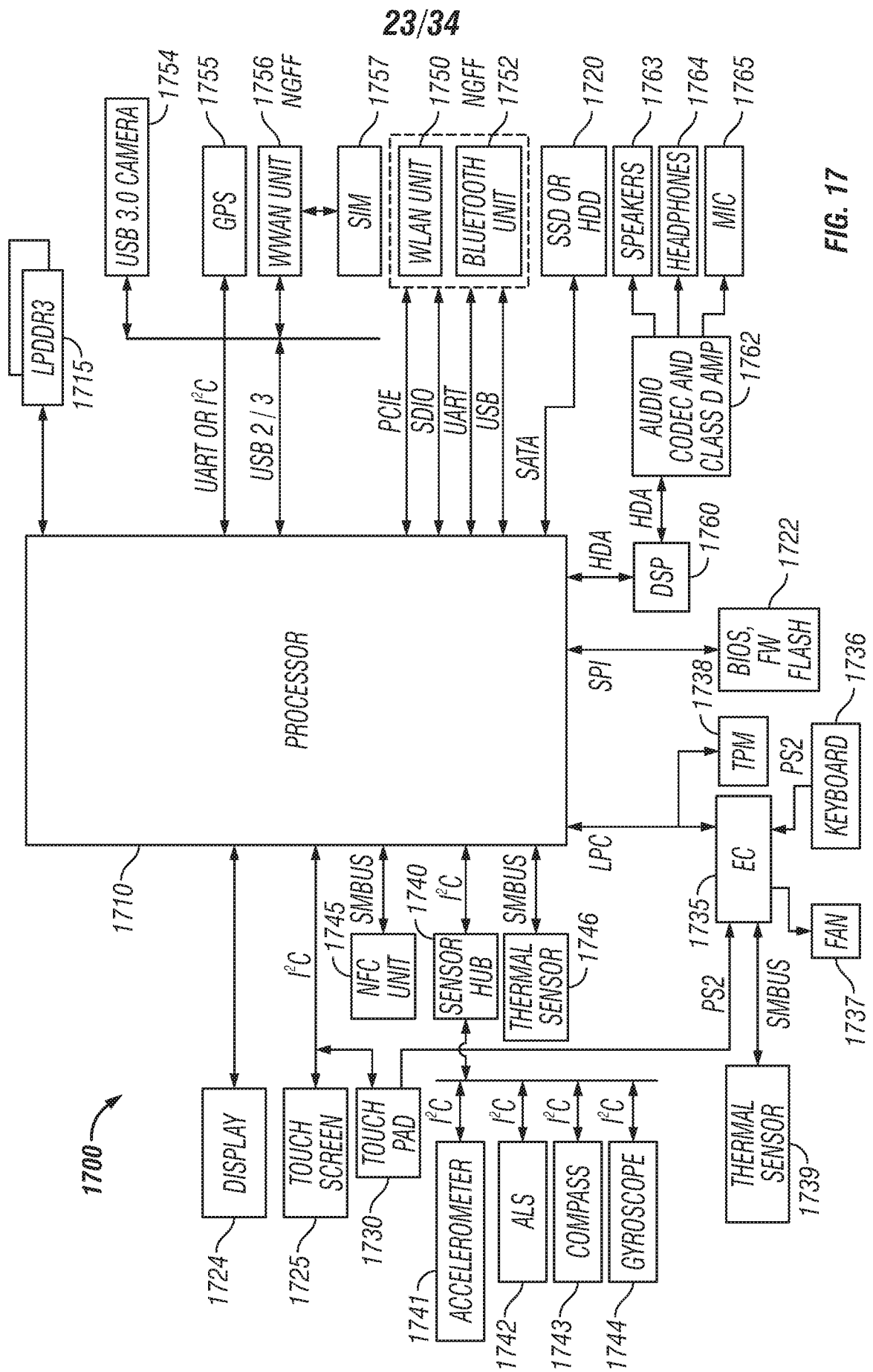


FIG. 17

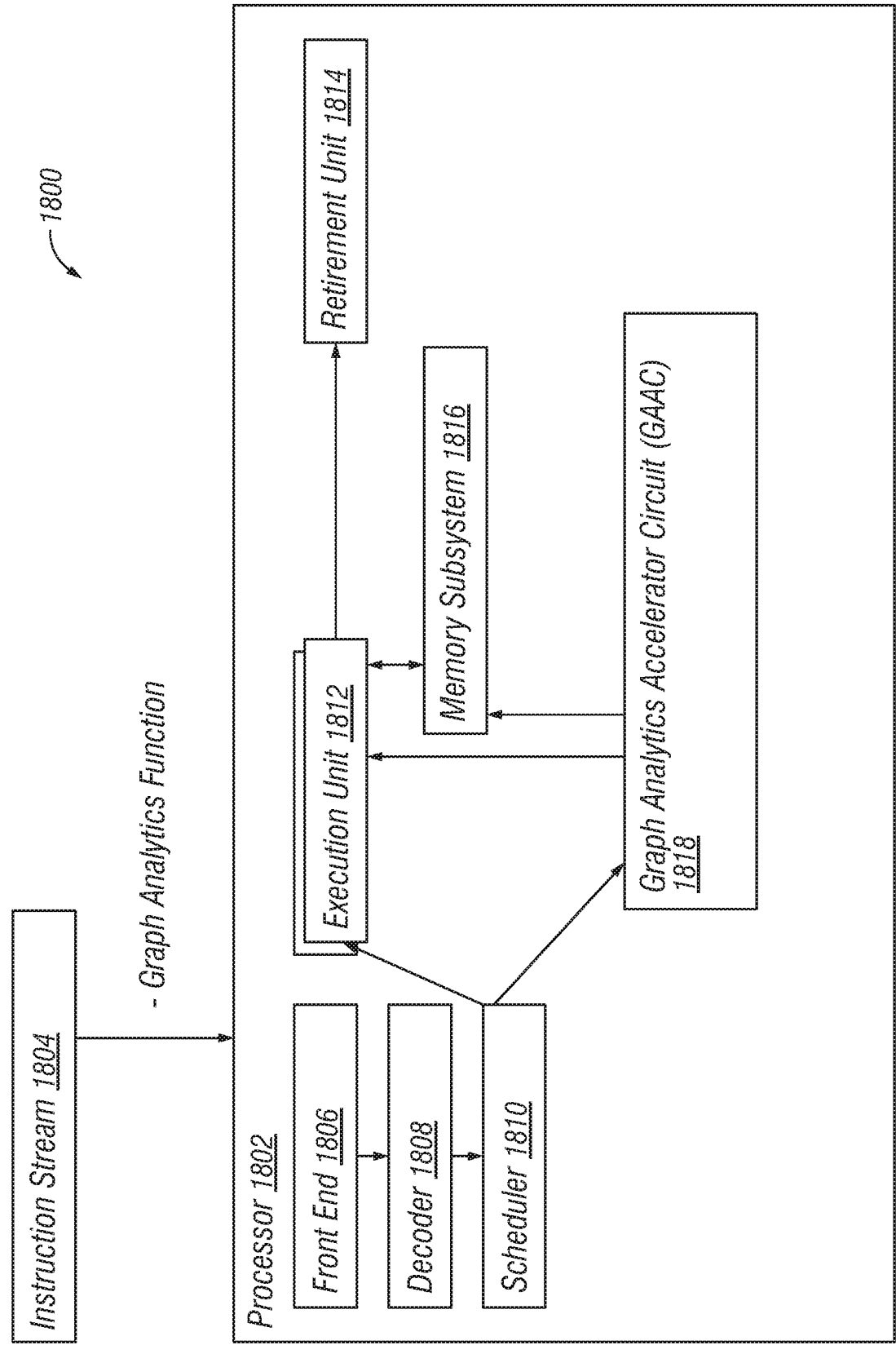
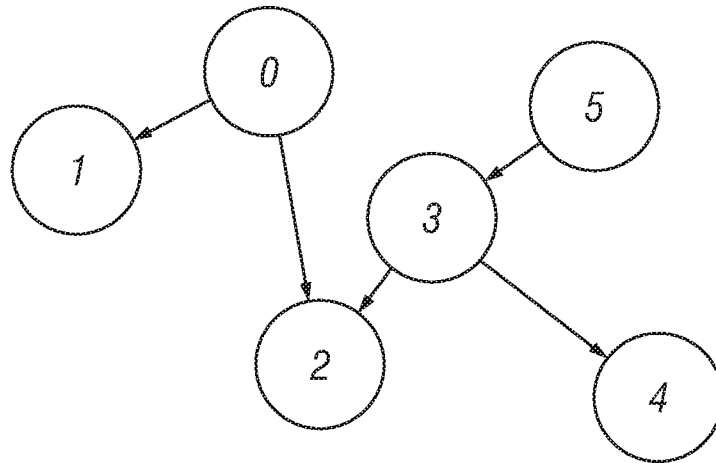
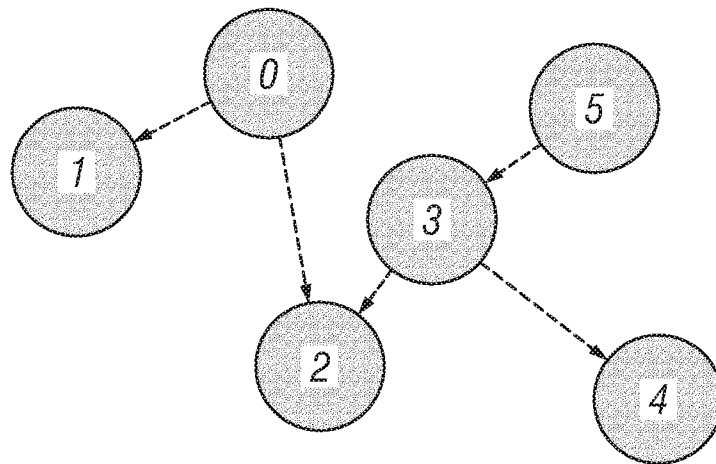
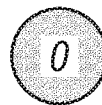


FIG. 18

25/34*Graph representation used for graph analytics application A**Graph representation used for graph analytics application B**Vertex containing Vertex Property (or properties) Type A:**Vertex containing Vertex Property (or properties) Type B:**Edge containing Edge Property Type A: —————>**Edge containing Edge Property Type B: - - - - ->***FIG. 19**

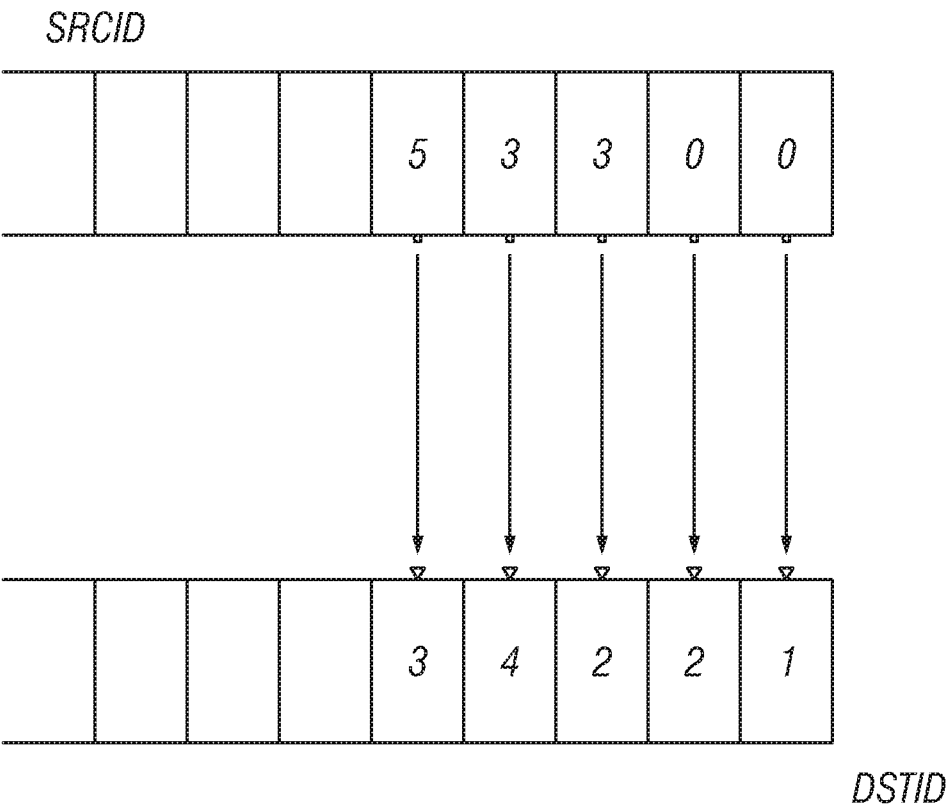


FIG. 19
(Cont'd)

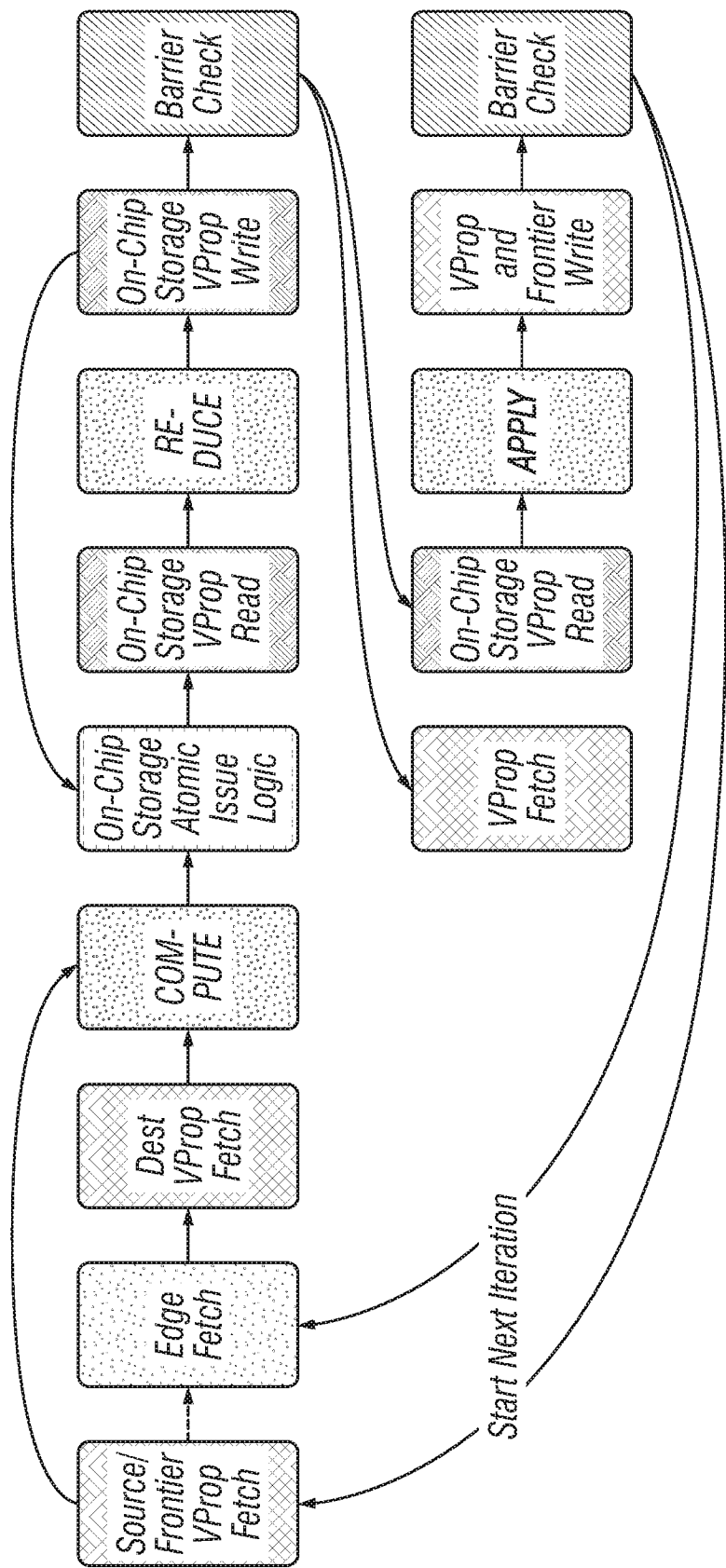


FIG. 20

28/34

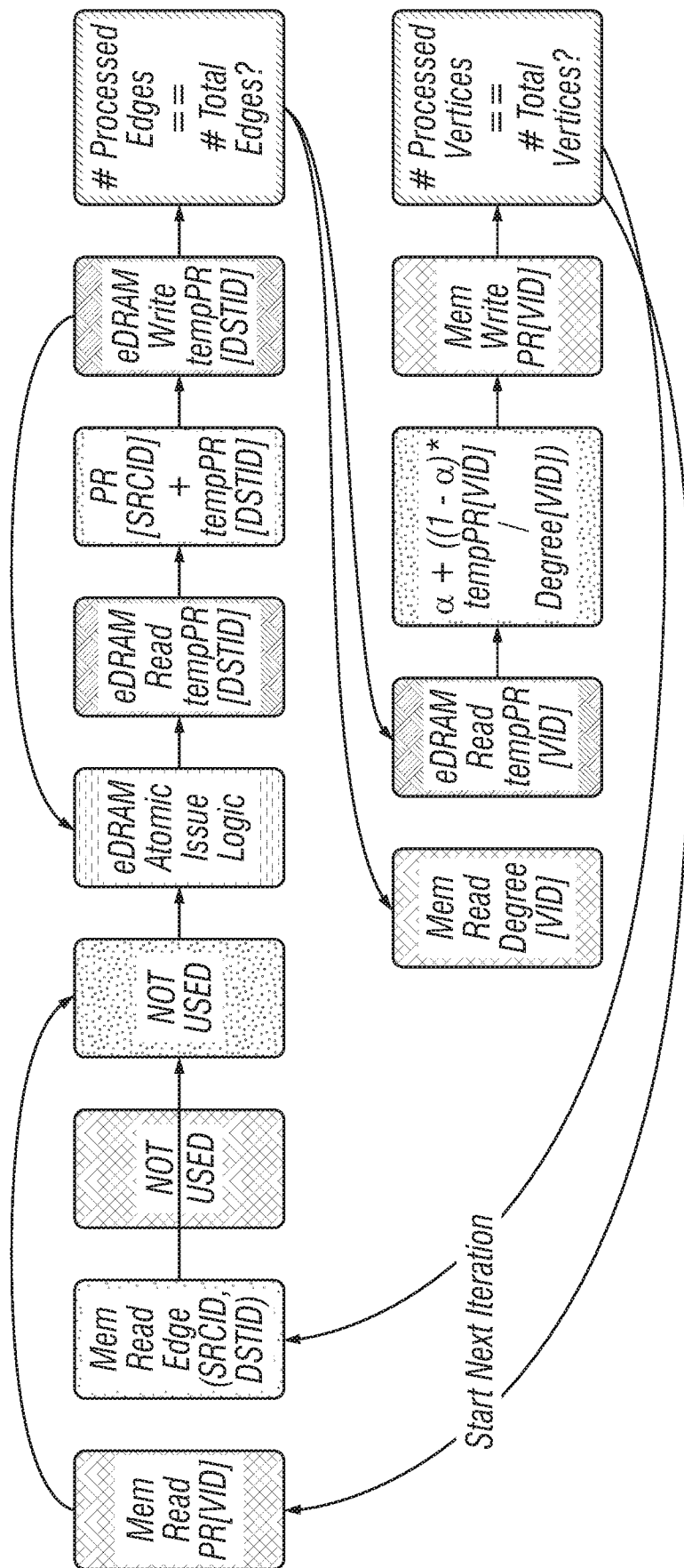


FIG. 21

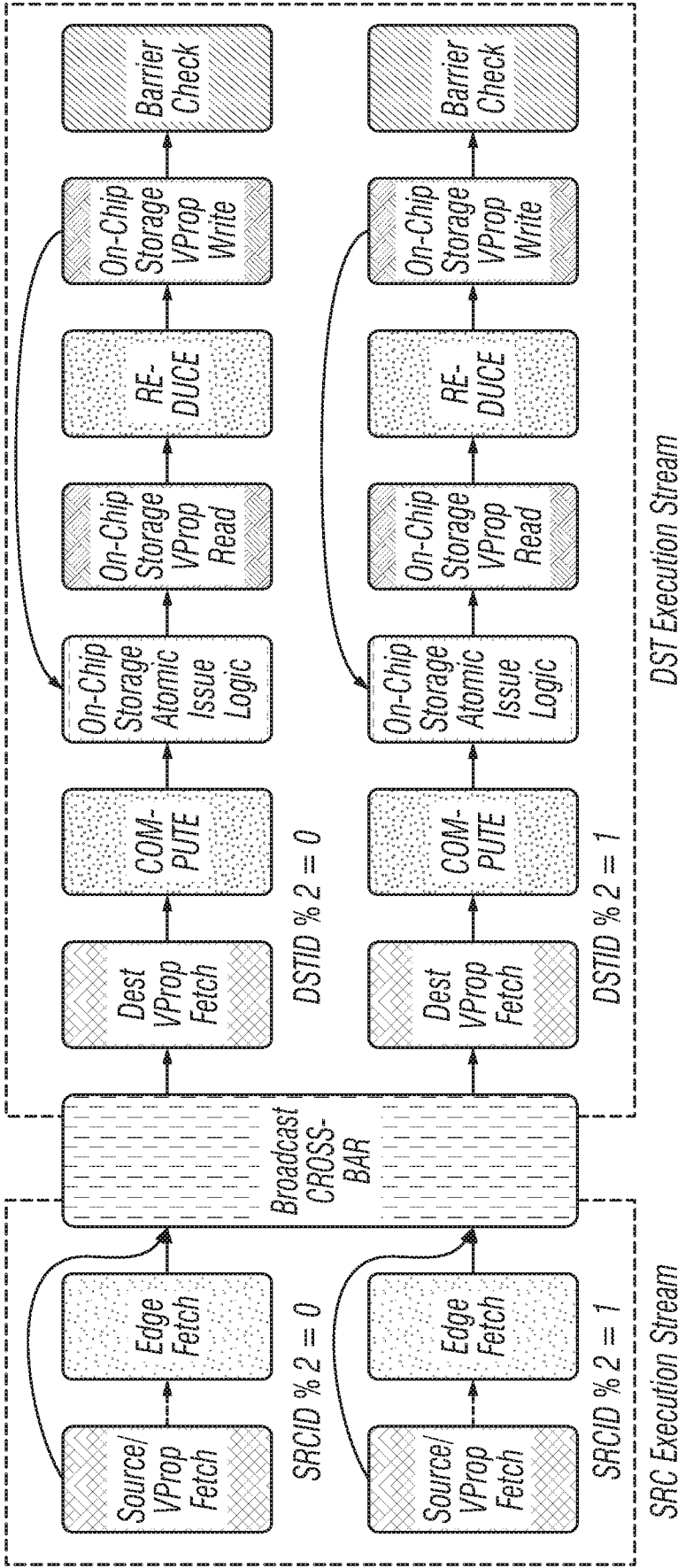
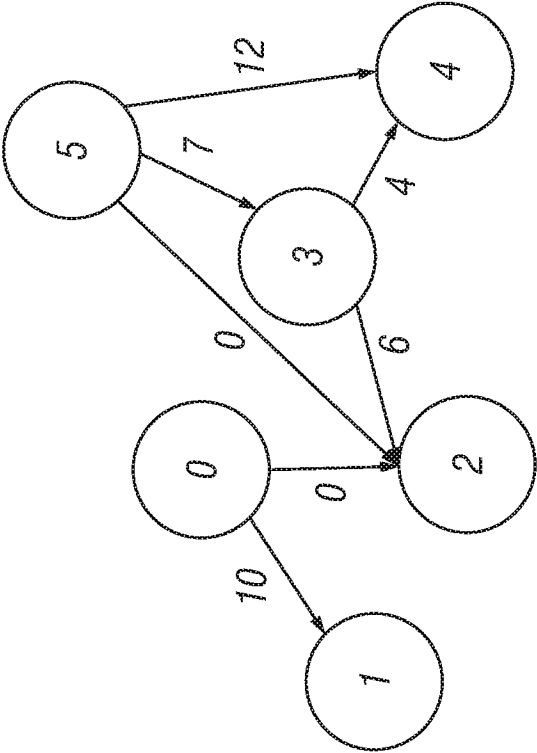


FIG. 22

SRCID					
	5	5	3	3	0
DSTID					
	4	3	2	2	1
Edge Property (Weight)					
	12	7	0	4	6
					10

FIG. 23

Example Graph



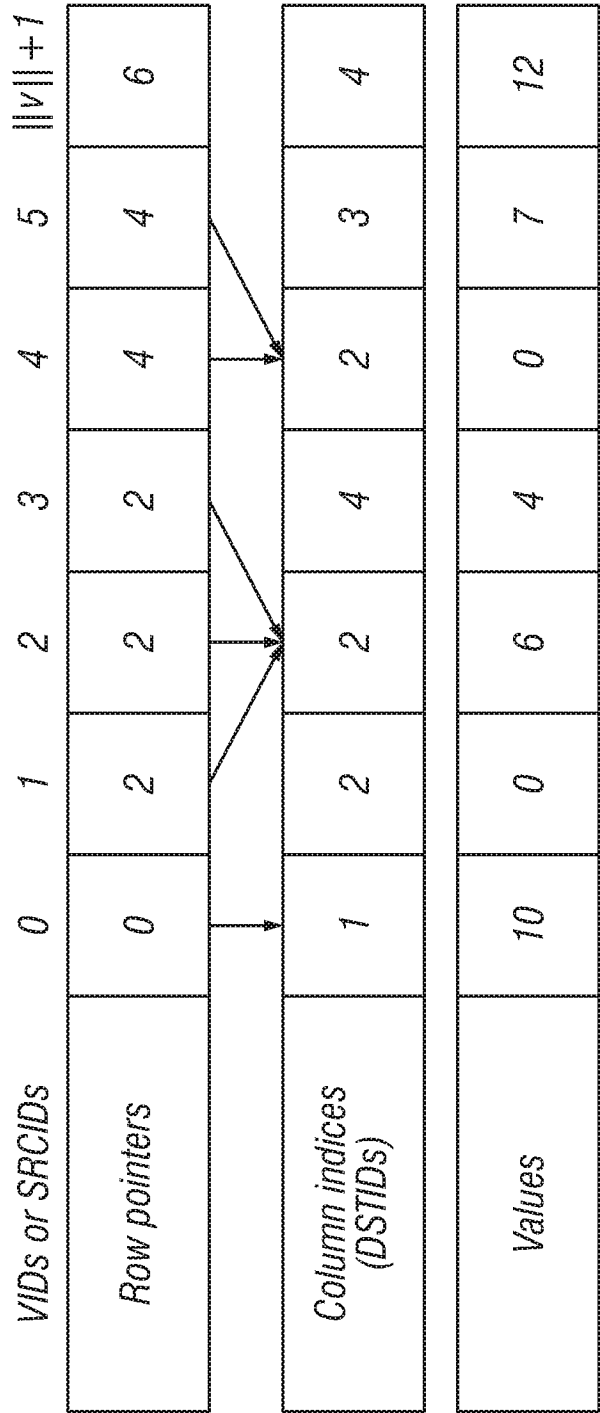
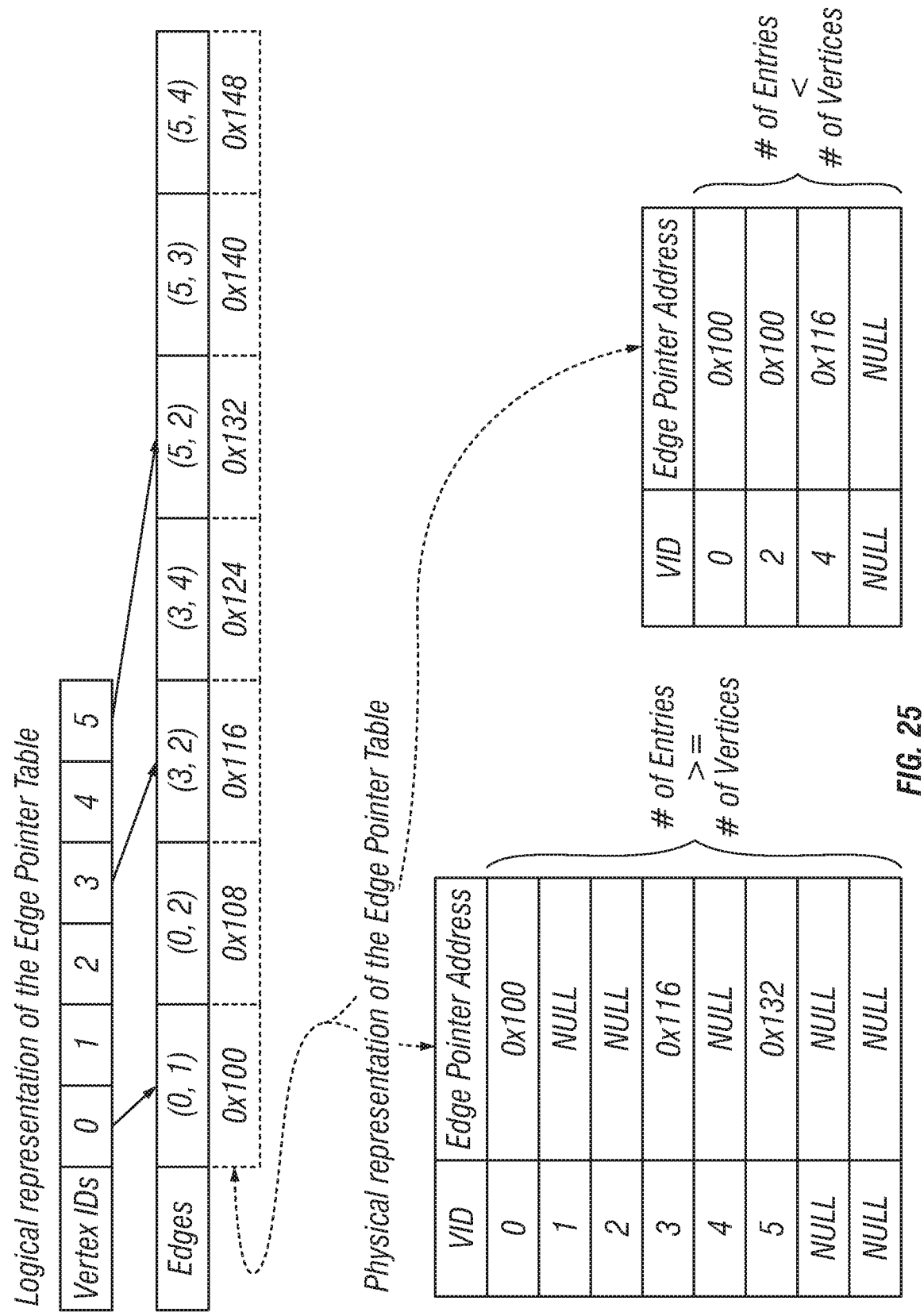
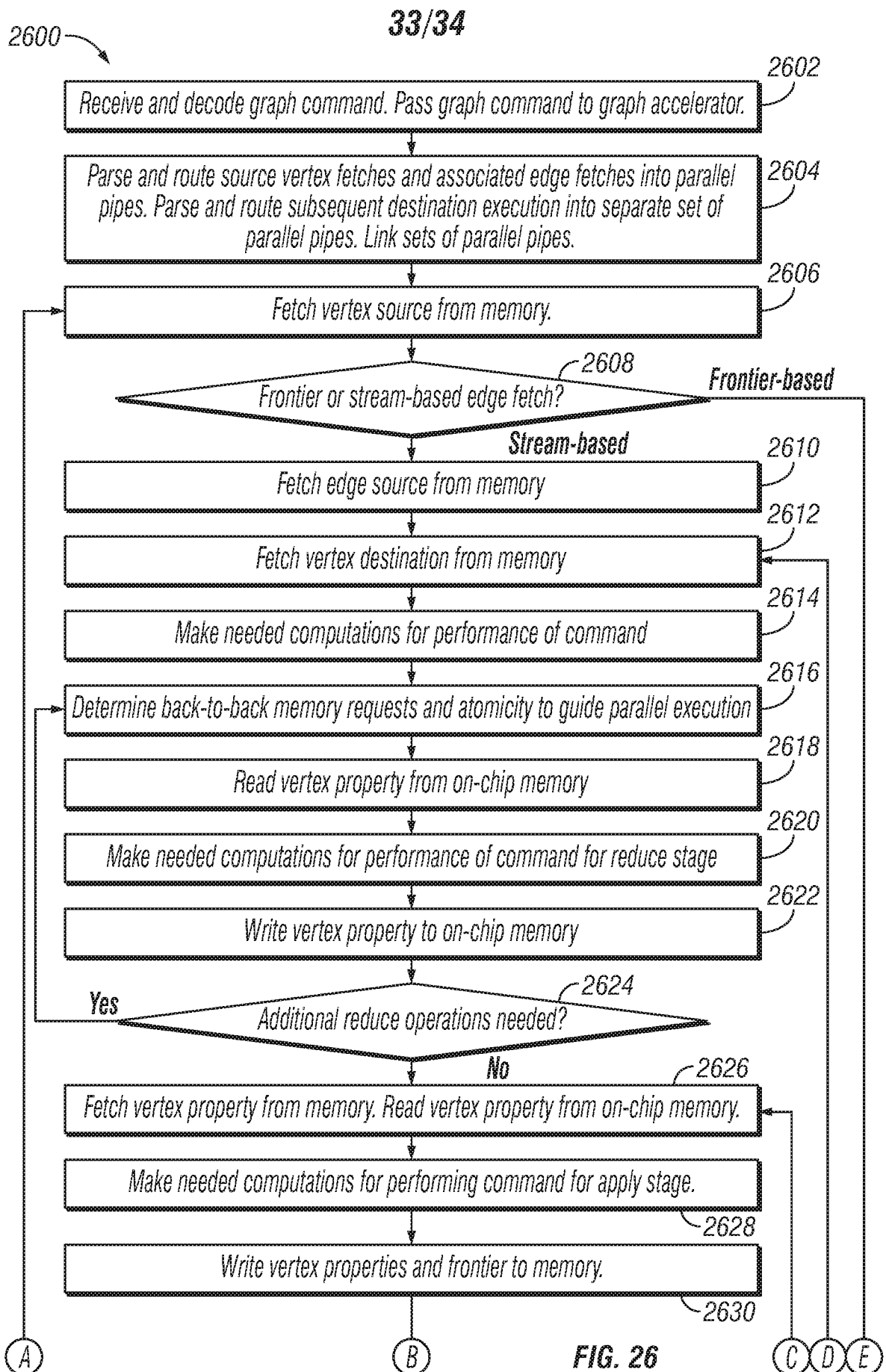
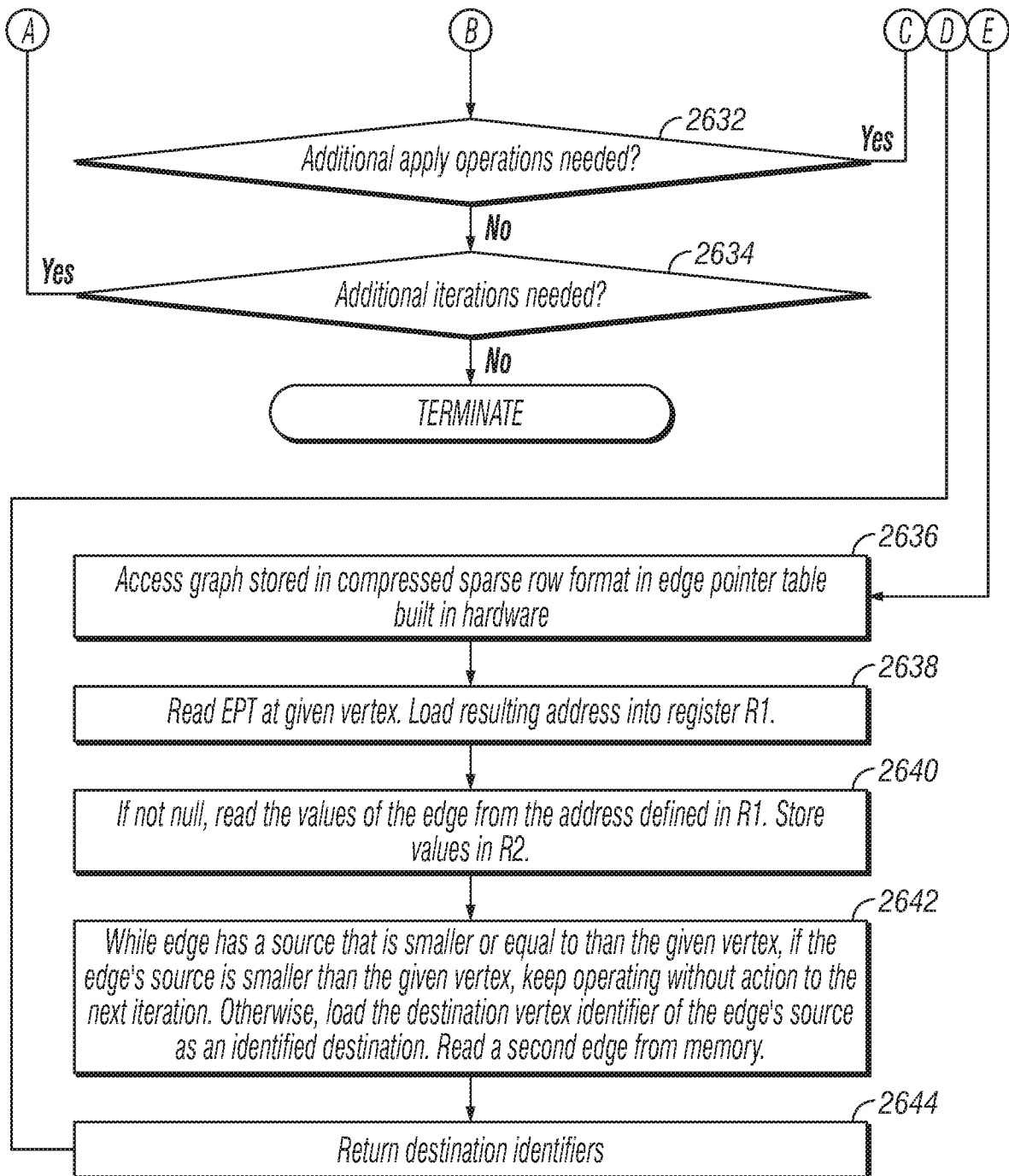


FIG. 24





34/34



INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/019845**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i, G06F 9/38(2006.01)i**

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 9/30; G06F 15/16; G06F 17/30; G06F 9/46; G06F 9/38

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

eKOMPASS(KIPO internal) & Keywords: accelerator circuit, instruction, graph analytic function, vertice, edge, datatype, stream, property, parallel execution pipeline, table, identifier, pointer address

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2016-0063037 A1 (THE JOHNS HOPKINS UNIVERSITY) 03 March 2016	1, 3-4, 7-8, 10-11
Y	See paragraphs [0040]-[0086]; claim 1; and figures 3A-5E.	, 13-14, 17-18, 20
Y		2, 5-6, 9, 12, 15-16, 19
Y	US 2003-0140160 A1 (URI RAZ et al.) 24 July 2003	2, 5-6, 9, 12, 15-16, 19
	See paragraphs [0042], [0050]; and figures 3-4.	
A	US 2014-0137130 A1 (INTERNATIONAL BUSINESS MACHINES CORPORATION) 15 May 2014	1-20
	See paragraphs [0057]-[0075]; and figures 3-5C.	
A	US 2014-0136553 A1 (INTERNATIONAL BUSINESS MACHINES CORP.) 15 May 2014	1-20
	See paragraphs [0056]-[0075]; and figures 3-5C.	
A	US 2013-0024479 A1 (NAN GONG et al.) 24 January 2013	1-20
	See paragraphs [0026]-[0061]; and figures 1-14.	



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

26 May 2017 (26.05.2017)

Date of mailing of the international search report

26 May 2017 (26.05.2017)

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

CHIN, Sang Bum

Telephone No. +82-42-481-8398



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/019845

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2016-0063037 A1	03/03/2016	None	
US 2003-0140160 A1	24/07/2003	AU 2957501 A	31/07/2001
		AU 3282501 A	31/07/2001
		AU 5104699 A	14/02/2000
		AU 9308701 A	08/04/2002
		AU 9308801 A	08/04/2002
		CA 2361342 A1	03/02/2000
		EP 1097416 A2	09/05/2001
		EP 1330738 A1	30/07/2003
		EP 1332432 A1	06/08/2003
		JP 04607320 B2	05/01/2011
		JP 2002-521745 A	16/07/2002
		US 2001-0034736 A1	25/10/2001
		US 2001-0037399 A1	01/11/2001
		US 2001-0037400 A1	01/11/2001
		US 2001-0044850 A1	22/11/2001
		US 2002-0042833 A1	11/04/2002
		US 2002-0138640 A1	26/09/2002
		US 6311221 B1	30/10/2001
		US 6574618 B2	03/06/2003
		US 7197570 B2	27/03/2007
		US 7606924 B2	20/10/2009
		WO 00-05637 A2	03/02/2000
		WO 00-05637 A3	12/10/2000
		WO 01-53940 A2	26/07/2001
		WO 01-53940 A3	28/03/2002
		WO 01-54375 A2	26/07/2001
		WO 01-54375 A3	28/02/2002
		WO 02-27492 A1	04/04/2002
		WO 02-27556 A1	04/04/2002
US 2014-0137130 A1	15/05/2014	US 2014-0137129 A1	15/05/2014
		US 9116738 B2	25/08/2015
		US 9122513 B2	01/09/2015
US 2014-0136553 A1	15/05/2014	US 2014-0136555 A1	15/05/2014
US 2013-0024479 A1	24/01/2013	US 8521782 B2	27/08/2013