



- (51) International Patent Classification:
G06F 9/30 (2006.01) *G06F 13/16* (2006.01)
- (21) International Application Number:
PCT/US2017/035116
- (22) International Filing Date:
31 May 2017 (31.05.2017)
- (25) Filing Language: English
- (26) Publication Language: English
- (30) Priority Data:
15/201,697 05 July 2016 (05.07.2016) US
- (71) Applicant: INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, California 95054-1549 (US).
- (72) Inventors: GUILFORD, James D.; 17 Mashpee Circle, Northborough, Massachusetts 01532 (US). GOPAL, Vinodh; 15 West End Avenue, Westborough, Massachusetts 01581 (US). YAP, Kirk S.; 61 Piccadilly Way, Westborough, Massachusetts 01581 (US).
- (74) Agent: KOMENDA, J. Kyle; Patent Capital Group, 2816 Lago Vista Lane, Rockwall, Texas 75032 (US).
- (81) Designated States (*unless otherwise indicated, for every kind of national protection available*): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, 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,

(54) Title: APPARATUS FOR HARDWARE IMPLEMENTATION OF HETEROGENEOUS DECOMPRESSION PROCESSING

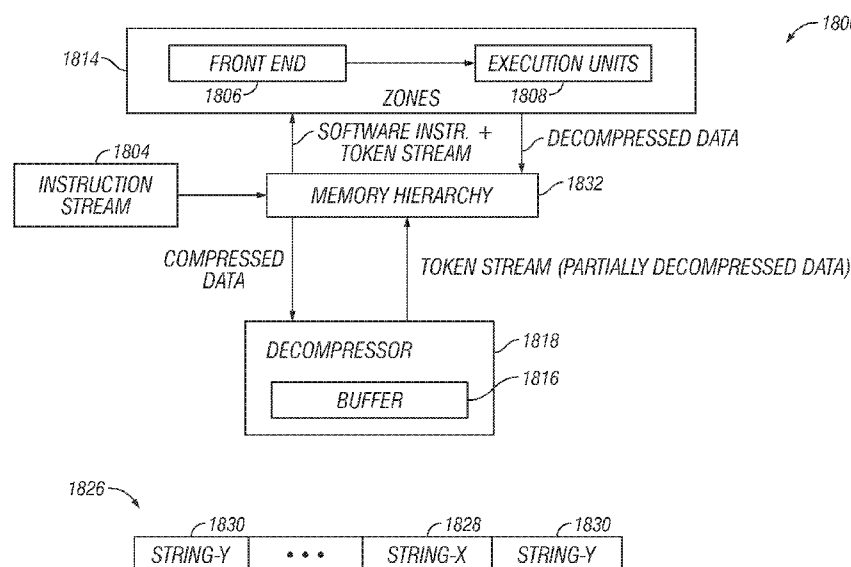


FIG. 18

(57) **Abstract:** A processor includes a memory hierarchy, buffer, and a decompressor. The decompressor includes circuitry to read elements to be decompressed according to a compression scheme, parse the elements to identify literals and matches, and, with the literals and matches, generate an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data. The intermediate token stream is to include a format for multiple tokens that are to be written in parallel with each other, and another format for tokens that include a data dependency upon themselves.



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

Published:

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

APPARATUS FOR HARDWARE IMPLEMENTATION OF HETEROGENEOUS DECOMPRESSION PROCESSING

FIELD OF THE INVENTION

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

DESCRIPTION OF RELATED ART

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

DESCRIPTION OF THE FIGURES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[0025] FIGURE 13 illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set, in accordance with embodiments of the present disclosure;

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

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

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

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

[0030] FIGURE 18 illustrates a block diagram of a system for implementing an apparatus for decompression, in accordance with embodiments of the present disclosure;

[0031] FIGURE 19 illustrates a more detailed view of a decompressor, in accordance with embodiments of the present disclosure;

[0032] FIGURES 20-23 illustrate formats and processing for token streams, according to embodiments of the present disclosure; and

[0033] FIGURE 24 illustrates an example method for decompression, in accordance with embodiments of the present disclosure.

DETAILED DESCRIPTION

[0034] The following description describes an apparatus, instruction, and processing logic for decompression processing. The decompression may be heterogeneous. Furthermore, the decompression may be faster than other solutions. In addition, the decompression may include improved bandwidth. Decompression may be performed in a processor, virtual processor, package, computer system, or other processing apparatus. Such a processing apparatus may include an out-of-order processor. In the following description, numerous specific details such as processing logic, processor types, micro-architectural conditions, events, enablement mechanisms, 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.

[0035] Although the following embodiments are described with reference to a processor, other embodiments are applicable to other types of integrated circuits and logic devices. Similar techniques and teachings of embodiments of the present disclosure may be applied to other types of circuits or semiconductor devices that may benefit from higher pipeline throughput and improved performance. The teachings of embodiments of the present disclosure are applicable to any processor or machine that performs data manipulations. However, the embodiments are not limited to processors or machines that perform 512-bit, 256-bit, 128-bit, 64-bit, 32-bit, or 16-bit data operations and may be applied to any processor and machine in which manipulation or management of data may be performed. In addition, the following description provides examples, and the accompanying drawings show various examples for the purposes of illustration. However, these examples should not be construed in a limiting sense as they are merely intended to provide examples of embodiments of the present disclosure rather than to provide an exhaustive list of all possible implementations of embodiments of the present disclosure.

[0036] 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.

[0037] 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 Discs, 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).

[0038] 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.

[0039] 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.

[0040] 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).

[0041] In one embodiment, the instruction set architecture (ISA) may be implemented by one or more micro-architectures, which may include processor logic and circuits used to implement one or more instruction sets. Accordingly, processors with

different micro-architectures may share at least a portion of a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale CA implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. Similarly, processors designed by other processor development companies, such as ARM Holdings, Ltd., MIPS, or their licensees or adopters, may share at least a portion a common instruction set, but may include different processor designs. For example, the same register architecture of the ISA may be implemented in different ways in different micro-architectures using new or well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB) and a retirement register file. In one embodiment, registers may include one or more registers, register architectures, register files, or other register sets that may or may not be addressable by a software programmer.

[0042] 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.

[0043] Scientific, financial, auto-vectorized general purpose, RMS (recognition, mining, and synthesis), and visual and multimedia applications (e.g., 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation) may require the same operation to be performed on a large number of data items. In one embodiment, Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data elements. SIMD technology may be used in processors that may logically divide the bits in a register

into a number of fixed-sized or variable-sized data elements, each of which represents a separate value. For example, in one embodiment, the bits in a 64-bit register may be organized as a source operand containing four separate 16-bit data elements, each of which represents a separate 16-bit value. This type of data may be referred to as ‘packed’ data type or ‘vector’ data type, and operands of this data type may be referred to as packed data operands or vector operands. In one embodiment, a packed data item or vector may be a sequence of packed data elements stored within a single register, and a packed data operand or a vector operand may be a source or destination operand of a SIMD instruction (or ‘packed data instruction’ or a ‘vector instruction’). In one embodiment, a SIMD instruction specifies a single vector operation to be performed on two source vector operands to generate a destination vector operand (also referred to as a result vector operand) of the same or different size, with the same or different number of data elements, and in the same or different data element order.

[0044] SIMD technology, such as that employed by the Intel® Core™ processors having an instruction set including x86, MMX™, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, and SSE4.2 instructions, ARM processors, such as the ARM Cortex® family of processors having an instruction set including the Vector Floating Point (VFP) and/or NEON instructions, and MIPS processors, such as the Loongson family of processors developed by the Institute of Computing Technology (ICT) of the Chinese Academy of Sciences, has enabled a significant improvement in application performance (Core™ and MMX™ are registered trademarks or trademarks of Intel Corporation of Santa Clara, Calif.).

[0045] In one embodiment, destination and source registers/data may be generic terms to represent the source and destination of the corresponding data or operation. In some embodiments, they may be implemented by registers, memory, or other storage areas having other names or functions than those depicted. For example, in one embodiment, “DEST1” may be a temporary storage register or other storage area, whereas “SRC1” and “SRC2” may be a first and second source storage register or other storage area, and so forth. In other embodiments, two or more of the SRC and DEST storage areas may correspond to different data storage elements within the same storage area (e.g., a SIMD register). In one embodiment, one of the source registers may also act as a destination register by, for

example, writing back the result of an operation performed on the first and second source data to one of the two source registers serving as a destination registers.

[0046] FIGURE 1A is a block diagram of an exemplary computer system formed with a processor that may include execution units to execute an instruction, in accordance with embodiments of the present disclosure. System 100 may include a component, such as a processor 102 to employ execution units including logic to perform algorithms for process data, in accordance with the present disclosure, such as in the embodiment described herein. System 100 may be representative of processing systems based on the PENTIUM[®] III, PENTIUM[®] 4, Xeon[™], Itanium[®], XScale[™] and/or StrongARM[™] microprocessors available from Intel Corporation of Santa Clara, California, although other systems (including PCs having other microprocessors, engineering workstations, set-top boxes and the like) may also be used. In one embodiment, sample system 100 may execute a version of the WINDOWS[™] operating system available from Microsoft Corporation of Redmond, Washington, although other operating systems (UNIX and Linux for example), embedded software, and/or graphical user interfaces, may also be used. Thus, embodiments of the present disclosure are not limited to any specific combination of hardware circuitry and software.

[0047] 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.

[0048] 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.

[0049] 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.

[0050] 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.

[0051] 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.

[0052] 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.

[0053] 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.

[0054] 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.

[0055] 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.

[0056] 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.

[0057] 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.

[0058] 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.

[0059] 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).

[0060] 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.

[0061] In one embodiment, SIMD coprocessor 161 comprises an execution unit 162 and a set of register files 164. One embodiment of main processor 165 comprises a decoder 165 to recognize instructions of instruction set 163 including instructions in accordance with one embodiment for execution by execution unit 162. In other embodiments, SIMD coprocessor 161 also comprises at least part of decoder 165 to decode instructions of instruction set 163. Processing core 170 may also include additional circuitry (not shown) which may be unnecessary to the understanding of embodiments of the present disclosure.

[0062] 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.

[0063] 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.

[0064] 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.

[0065] Some instructions may be converted into a single micro-op, whereas others need several micro-ops to complete the full operation. In one embodiment, if more than four micro-ops are needed to complete an instruction, decoder 228 may access microcode ROM 232 to perform the instruction. In one embodiment, an instruction may be decoded into a small number of micro-ops for processing at instruction decoder 228. In another embodiment, an instruction may be stored within microcode ROM 232 should a number of micro-ops be needed to accomplish the operation. Trace cache 230 refers to an entry point programmable logic array (PLA) to determine a correct micro-instruction pointer for reading the micro-code sequences to complete one or more instructions in accordance with one embodiment from micro-code ROM 232. After microcode ROM 232 finishes sequencing micro-ops for an instruction, front end 201 of the machine may resume fetching micro-ops from trace cache 230.

[0066] Out-of-order execution engine 203 may prepare instructions for execution. The out-of-order execution logic has a number of buffers to smooth out and re-order the flow of instructions to optimize performance as they go down the pipeline and get scheduled for execution. The allocator logic allocates the machine buffers and resources that each uop needs in order to execute. The register renaming logic renames logic registers onto entries in a register file. The allocator also allocates an entry for each uop in one of the two uop queues, one for memory operations and one for non-memory operations, in front of the instruction schedulers: memory scheduler, fast scheduler 202, slow/general floating point scheduler 204, and simple floating point scheduler 206. Uop schedulers 202, 204, 206, determine when a uop is ready to execute based on the readiness of their dependent input register operand sources and the availability of the execution resources the uops need to complete their operation. Fast scheduler 202 of one embodiment may schedule on each half of the main clock cycle while the other schedulers may only schedule once per main

processor clock cycle. The schedulers arbitrate for the dispatch ports to schedule uops for execution.

[0067] Register files 208, 210 may be arranged between schedulers 202, 204, 206, and execution units 212, 214, 216, 218, 220, 222, 224 in execution block 211. Each of register files 208, 210 perform integer and floating point operations, respectively. Each register file 208, 210, may include a bypass network that may bypass or forward just completed results that have not yet been written into the register file to new dependent uops. Integer register file 208 and floating point register file 210 may communicate data with the other. In one embodiment, integer register file 208 may be split into two separate register files, one register file for low-order thirty-two bits of data and a second register file for high order thirty-two bits of data. Floating point register file 210 may include 128-bit wide entries because floating point instructions typically have operands from 64 to 128 bits in width.

[0068] Execution block 211 may contain execution units 212, 214, 216, 218, 220, 222, 224. Execution units 212, 214, 216, 218, 220, 222, 224 may execute the instructions. Execution block 211 may include register files 208, 210 that store the integer and floating point data operand values that the micro-instructions need to execute. In one embodiment, processor 200 may comprise a number of execution units: address generation unit (AGU) 212, AGU 214, fast Arithmetic Logic Unit (ALU) 216, fast ALU 218, slow ALU 220, floating point ALU 222, floating point move unit 224. In another embodiment, floating point execution blocks 222, 224, may execute floating point, MMX, SIMD, and SSE, or other operations. In yet another embodiment, floating point ALU 222 may include a 64-bit by 64-bit floating point divider to execute divide, square root, and remainder micro-ops. In various embodiments, instructions involving a floating point value may be handled with the floating point hardware. In one embodiment, ALU operations may be passed to high-speed ALU execution units 216, 218. High-speed ALUs 216, 218 may execute fast operations with an effective latency of half a clock cycle. In one embodiment, most complex integer operations go to slow ALU 220 as slow ALU 220 may include integer execution hardware for long-latency type of operations, such as a multiplier, shifts, flag logic, and branch processing. Memory load/store operations may be executed by AGUs 212, 214. In one embodiment, integer ALUs 216, 218, 220 may perform integer operations on 64-bit data operands. In other

embodiments, ALUs 216, 218, 220 may be implemented to support a variety of data bit sizes including sixteen, thirty-two, 128, 256, etc. Similarly, floating point units 222, 224 may be implemented to support a range of operands having bits of various widths. In one embodiment, floating point units 222, 224, may operate on 128-bit wide packed data operands in conjunction with SIMD and multimedia instructions.

[0069] In one embodiment, uops schedulers 202, 204, 206, dispatch dependent operations before the parent load has finished executing. As uops may be speculatively scheduled and executed in processor 200, processor 200 may also include logic to handle memory misses. If a data load misses in the data cache, there may be dependent operations in flight in the pipeline that have left the scheduler with temporarily incorrect data. A replay mechanism tracks and re-executes instructions that use incorrect data. Only the dependent operations might need to be replayed and the independent ones may be allowed to complete. The schedulers and replay mechanism of one embodiment of a processor may also be designed to catch instruction sequences for text string comparison operations.

[0070] The term “registers” may refer to the on-board processor storage locations that may be used as part of instructions to identify operands. In other words, registers may be those that may be usable from the outside of the processor (from a programmer’s perspective). However, in some embodiments registers might not be limited to a particular type of circuit. Rather, a register may store data, provide data, and perform the functions described herein. The registers described herein may be implemented by circuitry within a processor using any number of different techniques, such as dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. In one embodiment, integer registers store 32-bit integer data. A register file of one embodiment also contains eight multimedia SIMD registers for packed data. For the discussions below, the registers may be understood to be data registers designed to hold packed data, such as 64-bit wide MMX™ registers (also referred to as ‘mm’ registers in some instances) in microprocessors enabled with MMX technology from Intel Corporation of Santa Clara, California. These MMX registers, available in both integer and floating point forms, may operate with packed data elements that accompany SIMD and SSE instructions. Similarly, 128-bit wide XMM registers relating to SSE2, SSE3, SSE4, or beyond (referred to generically as “SSEx”) technology may hold such

packed data operands. In one embodiment, in storing packed data and integer data, the registers do not need to differentiate between the two data types. In one embodiment, integer and floating point may be contained in the same register file or different register files. Furthermore, in one embodiment, floating point and integer data may be stored in different registers or the same registers.

[0071] 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.

[0072] 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.

[0073] 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.

[0074] 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.

[0075] 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.

[0076] FIGURE 3E illustrates another possible operation encoding (opcode) format 370, having forty or more bits, in accordance with embodiments of the present disclosure. Opcode format 370 corresponds with opcode format 360 and comprises an optional prefix byte 378. An instruction according to one embodiment may be encoded by one or more of fields 378, 371, and 372. Up to two operand locations per instruction may be identified by source operand identifiers 374 and 375 and by prefix byte 378. In one embodiment, prefix byte 378 may be used to identify 32-bit or 64-bit source and destination operands. In one embodiment, destination operand identifier 376 may be the same as source operand identifier 374, whereas in other embodiments they may be different. For another embodiment, destination operand identifier 376 may be the same as source operand identifier 375, whereas in other embodiments they may be different. In one embodiment, an instruction operates on one or more of the operands identified by operand identifiers 374 and 375 and one or more operands identified by operand identifiers 374 and 375 may be overwritten by the results of the instruction, whereas in other embodiments, operands identified by identifiers 374 and 375 may be written to another data element in another

register. Opcode formats 360 and 370 allow register to register, memory to register, register by memory, register by register, register by immediate, register to memory addressing specified in part by MOD fields 363 and 373 and by optional scale-index-base and displacement bytes.

[0077] FIGURE 3F illustrates yet another possible operation encoding (opcode) format, in accordance with embodiments of the present disclosure. 64-bit single instruction multiple data (SIMD) arithmetic operations may be performed through a coprocessor data processing (CDP) instruction. Operation encoding (opcode) format 380 depicts one such CDP instruction having CDP opcode fields 382 and 389. The type of CDP instruction, for another embodiment, operations may be encoded by one or more of fields 383, 384, 387, and 388. Up to three operand locations per instruction may be identified, including up to two source operand identifiers 385 and 390 and one destination operand identifier 386. One embodiment of the coprocessor may operate on eight, sixteen, thirty-two, and 64-bit values. In one embodiment, an instruction may be performed on integer data elements. In some embodiments, an instruction may be executed conditionally, using condition field 381. For some embodiments, source data sizes may be encoded by field 383. In some embodiments, Zero (Z), negative (N), carry (C), and overflow (V) detection may be done on SIMD fields. For some instructions, the type of saturation may be encoded by field 384.

[0078] FIGURE 4A is a block diagram illustrating an in-order pipeline and a register renaming stage, out-of-order issue/execution pipeline, in accordance with embodiments of the present disclosure. FIGURE 4B is a block diagram illustrating an in-order architecture core and a register renaming logic, out-of-order issue/execution logic to be included in a processor, in accordance with embodiments of the present disclosure. The solid lined boxes in FIGURE 4A illustrate the in-order pipeline, while the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline. Similarly, the solid lined boxes in FIGURE 4B illustrate the in-order architecture logic, while the dashed lined boxes illustrates the register renaming logic and out-of-order issue/execution logic.

[0079] In FIGURE 4A, a processor pipeline 400 may include a fetch stage 402, a length decode stage 404, a decode stage 406, an allocation stage 408, a renaming stage 410, a scheduling (also known as a dispatch or issue) stage 412, a register read/memory read

stage 414, an execute stage 416, a write-back/memory-write stage 418, an exception handling stage 422, and a commit stage 424.

[0080] In FIGURE 4B, arrows denote a coupling between two or more units and the direction of the arrow indicates a direction of data flow between those units. FIGURE 4B shows processor core 490 including a front end unit 430 coupled to an execution engine unit 450, and both may be coupled to a memory unit 470.

[0081] Core 490 may be a Reduced Instruction Set Computing (RISC) core, a Complex Instruction Set Computing (CISC) core, a Very Long Instruction Word (VLIW) core, or a hybrid or alternative core type. In one embodiment, core 490 may be a special-purpose core, such as, for example, a network or communication core, compression engine, graphics core, or the like.

[0082] Front end unit 430 may include a branch prediction unit 432 coupled to an instruction cache unit 434. Instruction cache unit 434 may be coupled to an instruction Translation Lookaside Buffer (TLB) 436. TLB 436 may be coupled to an instruction fetch unit 438, which is coupled to a decode unit 440. Decode unit 440 may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which may be decoded from, or which otherwise reflect, or may be derived from, the original instructions. The decoder may be implemented using various different mechanisms. Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read-only memories (ROMs), etc. In one embodiment, instruction cache unit 434 may be further coupled to a level 2 (L2) cache unit 476 in memory unit 470. Decode unit 440 may be coupled to a rename/allocator unit 452 in execution engine unit 450.

[0083] Execution engine unit 450 may include rename/allocator unit 452 coupled to a retirement unit 454 and a set of one or more scheduler units 456. Scheduler units 456 represent any number of different schedulers, including reservations stations, central instruction window, etc. Scheduler units 456 may be coupled to physical register file units 458. Each of physical register file units 458 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point,

etc., status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc. Physical register file units 458 may be overlapped by retirement unit 154 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using one or more reorder buffers and one or more retirement register files, using one or more future files, one or more history buffers, and one or more retirement register files; using register maps and a pool of registers; etc.). Generally, the architectural registers may be visible from the outside of the processor or from a programmer's perspective. The registers might not be limited to any known particular type of circuit. Various different types of registers may be suitable as long as they store and provide data as described herein. Examples of suitable registers include, but might not be limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. Retirement unit 454 and physical register file units 458 may be coupled to execution clusters 460. Execution clusters 460 may include a set of one or more execution units 162 and a set of one or more memory access units 464. Execution units 462 may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g., scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions. Scheduler units 456, physical register file units 458, and execution clusters 460 are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file unit, and/or execution cluster – and in the case of a separate memory access pipeline, certain embodiments may be implemented in which only the execution cluster of this pipeline has memory access units 464). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

[0084] 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.

[0085] 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.

[0086] 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).

[0087] 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.

[0088] While register renaming may be described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor may also include a separate instruction and data cache units 434/474 and a shared L2 cache unit 476, other embodiments may have a single internal cache for both instructions and data, such as, for

example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that may be external to the core and/or the processor. In other embodiments, all of the cache may be external to the core and/or the processor.

[0089] 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.

[0090] 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.

[0091] Processor 500 may include a memory hierarchy comprising one or more levels of caches within the cores, one or more shared cache units such as caches 506, or external memory (not shown) coupled to the set of integrated memory controller units 552. Caches 506 may include any suitable cache. In one embodiment, caches 506 may include one or more mid-level caches, such as Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof.

[0092] In various embodiments, one or more of cores 502 may perform multi-threading. System agent 510 may include components for coordinating and operating cores 502. System agent unit 510 may include for example a Power Control Unit (PCU). The PCU may be or include logic and components needed for regulating the power state of cores 502. System agent 510 may include a display engine 512 for driving one or more externally connected displays or graphics module 560. System agent 510 may include an interface 1214

for communications busses for graphics. In one embodiment, interface 1214 may be implemented by PCI Express (PCIe). In a further embodiment, interface 1214 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 1218 for providing PCIe links to other elements of a computing system. PCIe bridge 1218 may be implemented using a memory controller 1220 and coherence logic 1222.

[0093] Cores 502 may be implemented in any suitable manner. Cores 502 may be homogenous or heterogeneous in terms of architecture and/or instruction set. In one embodiment, some of cores 502 may be in-order while others may be out-of-order. In another embodiment, two or more of cores 502 may execute the same instruction set, while others may execute only a subset of that instruction set or a different instruction set.

[0094] Processor 500 may include a general-purpose processor, such as a Core™ i3, i5, i7, 2 Duo and Quad, Xeon™, Itanium™, XScale™ or StrongARM™ processor, which may be available from Intel Corporation, of Santa Clara, Calif. Processor 500 may be provided from another company, such as ARM Holdings, Ltd, MIPS, etc. Processor 500 may be a special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, co-processor, embedded processor, or the like. Processor 500 may be implemented on one or more chips. Processor 500 may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

[0095] In one embodiment, a given one of caches 506 may be shared by multiple ones of cores 502. In another embodiment, a given one of caches 506 may be dedicated to one of cores 502. The assignment of caches 506 to cores 502 may be handled by a cache controller or other suitable mechanism. A given one of caches 506 may be shared by two or more cores 502 by implementing time-slices of a given cache 506.

[0096] Graphics module 560 may implement an integrated graphics processing subsystem. In one embodiment, graphics module 560 may include a graphics processor. Furthermore, graphics module 560 may include a media engine 565. Media engine 565 may provide media encoding and video decoding.

[0097] FIGURE 5B is a block diagram of an example implementation of a core 502, in accordance with embodiments of the present disclosure. Core 502 may include a front end 570 communicatively coupled to an out-of-order engine 580. Core 502 may be communicatively coupled to other portions of processor 500 through cache hierarchy 503.

[0098] Front end 570 may be implemented in any suitable manner, such as fully or in part by front end 201 as described above. In one embodiment, front end 570 may communicate with other portions of processor 500 through cache hierarchy 503. In a further embodiment, front end 570 may fetch instructions from portions of processor 500 and prepare the instructions to be used later in the processor pipeline as they are passed to out-of-order execution engine 580.

[0099] Out-of-order execution engine 580 may be implemented in any suitable manner, such as fully or in part by out-of-order execution engine 203 as described above. Out-of-order execution engine 580 may prepare instructions received from front end 570 for execution. Out-of-order execution engine 580 may include an allocate module 1282. In one embodiment, allocate module 1282 may allocate resources of processor 500 or other resources, such as registers or buffers, to execute a given instruction. Allocate module 1282 may make allocations in schedulers, such as a memory scheduler, fast scheduler, or floating point scheduler. Such schedulers may be represented in FIGURE 5B by resource schedulers 584. Allocate module 1282 may be implemented fully or in part by the allocation logic described in conjunction with FIGURE 2. Resource schedulers 584 may determine when an instruction is ready to execute based on the readiness of a given resource's sources and the availability of execution resources needed to execute an instruction. Resource schedulers 584 may be implemented by, for example, schedulers 202, 204, 206 as discussed above. Resource schedulers 584 may schedule the execution of instructions upon one or more resources. In one embodiment, such resources may be internal to core 502, and may be illustrated, for example, as resources 586. In another embodiment, such resources may be external to core 502 and may be accessible by, for example, cache hierarchy 503. Resources may include, for example, memory, caches, register files, or registers. Resources internal to core 502 may be represented by resources 586 in FIGURE 5B. As necessary, values written to or read from resources 586 may be coordinated with other portions of processor 500 through, for example, cache hierarchy 503. As instructions are assigned resources, they may

be placed into a reorder buffer 588. Reorder buffer 588 may track instructions as they are executed and may selectively reorder their execution based upon any suitable criteria of processor 500. In one embodiment, reorder buffer 588 may identify instructions or a series of instructions that may be executed independently. Such instructions or a series of instructions may be executed in parallel from other such instructions. Parallel execution in core 502 may be performed by any suitable number of separate execution blocks or virtual processors. In one embodiment, shared resources—such as memory, registers, and caches—may be accessible to multiple virtual processors within a given core 502. In other embodiments, shared resources may be accessible to multiple processing entities within processor 500.

[0100] Cache hierarchy 503 may be implemented in any suitable manner. For example, cache hierarchy 503 may include one or more lower or mid-level caches, such as caches 572, 574. In one embodiment, cache hierarchy 503 may include an LLC 595 communicatively coupled to caches 572, 574. In another embodiment, LLC 595 may be implemented in a module 590 accessible to all processing entities of processor 500. In a further embodiment, module 590 may be implemented in an uncore module of processors from Intel, Inc. Module 590 may include portions or subsystems of processor 500 necessary for the execution of core 502 but might not be implemented within core 502. Besides LLC 595, Module 590 may include, for example, hardware interfaces, memory coherency coordinators, interprocessor interconnects, instruction pipelines, or memory controllers. Access to RAM 599 available to processor 500 may be made through module 590 and, more specifically, LLC 595. Furthermore, other instances of core 502 may similarly access module 590. Coordination of the instances of core 502 may be facilitated in part through module 590.

[0101] FIGURES 6-8 may illustrate exemplary systems suitable for including processor 500, while FIGURE 9 may illustrate an exemplary System on a Chip (SoC) that may include one or more of cores 502. Other system designs and implementations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, DSPs, graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, may also be suitable.

In general, a huge variety of systems or electronic devices that incorporate a processor and/or other execution logic as disclosed herein may be generally suitable.

[0102] FIGURE 6 illustrates a block diagram of a system 600, in accordance with embodiments of the present disclosure. System 600 may include one or more processors 610, 615, which may be coupled to Graphics Memory Controller Hub (GMCH) 620. The optional nature of additional processors 615 is denoted in FIGURE 6 with broken lines.

[0103] Each processor 610,615 may be some version of processor 500. However, it should be noted that integrated graphics logic and integrated memory control units might not exist in processors 610,615. FIGURE 6 illustrates that GMCH 620 may be coupled to a memory 640 that may be, for example, a dynamic random access memory (DRAM). The DRAM may, for at least one embodiment, be associated with a non-volatile cache.

[0104] GMCH 620 may be a chipset, or a portion of a chipset. GMCH 620 may communicate with processors 610, 615 and control interaction between processors 610, 615 and memory 640. GMCH 620 may also act as an accelerated bus interface between the processors 610, 615 and other elements of system 600. In one embodiment, GMCH 620 communicates with processors 610, 615 via a multi-drop bus, such as a frontside bus (FSB) 695.

[0105] Furthermore, GMCH 620 may be coupled to a display 645 (such as a flat panel display). In one embodiment, GMCH 620 may include an integrated graphics accelerator. GMCH 620 may be further coupled to an input/output (I/O) controller hub (ICH) 650, which may be used to couple various peripheral devices to system 600. External graphics device 660 may include be a discrete graphics device coupled to ICH 650 along with another peripheral device 670.

[0106] In other embodiments, additional or different processors may also be present in system 600. For example, additional processors 610, 615 may include additional processors that may be the same as processor 610, additional processors that may be heterogeneous or asymmetric to processor 610, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor. There may be a variety of differences between the physical resources 610, 615 in terms of a spectrum of metrics of merit including architectural, micro-architectural, thermal, power consumption characteristics, and the like. These differences may effectively

manifest themselves as asymmetry and heterogeneity amongst processors 610, 615. For at least one embodiment, various processors 610, 615 may reside in the same die package.

[0107] FIGURE 7 illustrates a block diagram of a second system 700, in accordance with embodiments of the present disclosure. As shown in FIGURE 7, multiprocessor system 700 may include a point-to-point interconnect system, and may include a first processor 770 and a second processor 780 coupled via a point-to-point interconnect 750. Each of processors 770 and 780 may be some version of processor 500 as one or more of processors 610, 615.

[0108] 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.

[0109] Processors 770 and 780 are shown including integrated memory controller units 772 and 782, respectively. Processor 770 may also include as part of its bus controller units point-to-point (P-P) interfaces 776 and 778; similarly, second processor 780 may include P-P interfaces 786 and 788. Processors 770, 780 may exchange information via a point-to-point (P-P) interface 750 using P-P interface circuits 778, 788. As shown in FIGURE 7, IMCs 772 and 782 may couple the processors to respective memories, namely a memory 732 and a memory 734, which in one embodiment may be portions of main memory locally attached to the respective processors.

[0110] Processors 770, 780 may each exchange information with a chipset 790 via individual P-P interfaces 752, 754 using point to point interface circuits 776, 794, 786, 798. In one embodiment, chipset 790 may also exchange information with a high-performance graphics circuit 738 via a high-performance graphics interface 739.

[0111] A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode.

[0112] Chipset 790 may be coupled to a first bus 716 via an interface 796. In one embodiment, first bus 716 may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present disclosure is not so limited.

[0113] As shown in FIGURE 7, various I/O devices 714 may be coupled to first bus 716, along with a bus bridge 718 which couples first bus 716 to a second bus 720. In one embodiment, second bus 720 may be a Low Pin Count (LPC) bus. Various devices may be coupled to second bus 720 including, for example, a keyboard and/or mouse 722, communication devices 727 and a storage unit 728 such as a disk drive or other mass storage device which may include instructions/code and data 730, in one embodiment. Further, an audio I/O 724 may be coupled to second bus 720. Note that other architectures may be possible. For example, instead of the point-to-point architecture of FIGURE 7, a system may implement a multi-drop bus or other such architecture.

[0114] 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.

[0115] FIGURE 8 illustrates that processors 870, 880 may include integrated memory and I/O Control Logic ("CL") 872 and 882, respectively. For at least one embodiment, CL 872, 882 may include integrated memory controller units such as that described above in connection with FIGURES 5 and 7. In addition, CL 872, 882 may also include I/O control logic. FIGURE 8 illustrates that not only memories 832, 834 may be coupled to CL 872, 882, but also that I/O devices 814 may also be coupled to control logic 872, 882. Legacy I/O devices 815 may be coupled to chipset 890.

[0116] FIGURE 9 illustrates a block diagram of a SoC 900, in accordance with embodiments of the present disclosure. Similar elements in FIGURE 5 bear like reference numerals. Also, dashed lined boxes may represent optional features on more advanced SoCs. An interconnect units 902 may be coupled to: an application processor 910 which may include a set of one or more cores 902A-N and shared cache units 906; a system agent unit 910; a bus controller units 916; an integrated memory controller units 914; a set or one or more media processors 920 which may include integrated graphics logic 908, an image processor 924 for providing still and/or video camera functionality, an audio processor 926 for providing hardware audio acceleration, and a video processor 928 for providing video encode/decode acceleration; an SRAM unit 930; a DMA unit 932; and a display unit 940 for coupling to one or more external displays.

[0117] 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.

[0118] 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.

[0119] 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.

[0120] 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.

[0121] FIGURE 11 illustrates a block diagram illustrating the development of IP cores, in accordance with embodiments of the present disclosure. Storage 1130 may include simulation software 1120 and/or hardware or software model 1110. In one embodiment, the data representing the IP core design may be provided to storage 1130 via memory 1140 (e.g., hard disk), wired connection (e.g., internet) 1150 or wireless connection 1160. The IP core information generated by the simulation tool and model may then be transmitted to a fabrication facility where it may be fabricated by a third party to perform at least one instruction in accordance with at least one embodiment.

[0122] In some embodiments, one or more instructions may correspond to a first type or architecture (e.g., x86) and be translated or emulated on a processor of a different type or architecture (e.g., ARM). An instruction, according to one embodiment, may therefore be performed on any processor or processor type, including ARM, x86, MIPS, a GPU, or other processor type or architecture.

[0123] FIGURE 12 illustrates how an instruction of a first type may be emulated by a processor of a different type, in accordance with embodiments of the present disclosure. In FIGURE 12, program 1205 contains some instructions that may perform the same or substantially the same function as an instruction according to one embodiment. However the instructions of program 1205 may be of a type and/or format that is different from or incompatible with processor 1215, meaning the instructions of the type in program 1205 may not be able to execute natively by the processor 1215. However, with the help of emulation logic, 1210, the instructions of program 1205 may be translated into instructions that may be natively be executed by the processor 1215. In one embodiment, the emulation logic may be embodied in hardware. In another embodiment, the emulation logic may be embodied in a tangible, machine-readable medium containing software to translate instructions of the type in program 1205 into the type natively executable by processor 1215. In other embodiments, emulation logic may be a combination of fixed-function or programmable

hardware and a program stored on a tangible, machine-readable medium. In one embodiment, the processor contains the emulation logic, whereas in other embodiments, the emulation logic exists outside of the processor and may be provided by a third party. In one embodiment, the processor may load the emulation logic embodied in a tangible, machine-readable medium containing software by executing microcode or firmware contained in or associated with the processor.

[0124] FIGURE 13 illustrates a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set, in accordance with embodiments of the present disclosure. In the illustrated embodiment, the instruction converter may be a software instruction converter, although the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIGURE 13 shows a program in a high level language 1302 may be compiled using an x86 compiler 1304 to generate x86 binary code 1306 that may be natively executed by a processor with at least one x86 instruction set core 1316. The processor with at least one x86 instruction set core 1316 represents any processor that may perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. x86 compiler 1304 represents a compiler that may be operable to generate x86 binary code 1306 (e.g., object code) that may, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core 1316. Similarly, FIGURE 13 shows the program in high level language 1302 may be compiled using an alternative instruction set compiler 1308 to generate alternative instruction set binary code 1310 that may be natively executed by a processor without at least one x86 instruction set core 1314 (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, CA). Instruction converter 1312 may be used to convert x86 binary code 1306 into code that may be natively executed by the processor without an x86 instruction set core 1314. This converted code might not be the

same as alternative instruction set binary code 1310; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, instruction converter 1312 represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute x86 binary code 1306.

[0125] FIGURE 14 is a block diagram of an instruction set architecture 1400 of a processor, in accordance with embodiments of the present disclosure. Instruction set architecture 1400 may include any suitable number or kind of components.

[0126] For example, instruction set architecture 1400 may include processing entities such as one or more cores 1406, 1407 and a graphics processing unit 1415. Cores 1406, 1407 may be communicatively coupled to the rest of instruction set architecture 1400 through any suitable mechanism, such as through a bus or cache. In one embodiment, cores 1406, 1407 may be communicatively coupled through an L2 cache control 1408, which may include a bus interface unit 1409 and an L2 cache 1410. Cores 1406, 1407 and graphics processing unit 1415 may be communicatively coupled to each other and to the remainder of instruction set architecture 1400 through interconnect 1410. In one embodiment, graphics processing unit 1415 may use a video code 1420 defining the manner in which particular video signals will be encoded and decoded for output.

[0127] Instruction set architecture 1400 may also include any number or kind of interfaces, controllers, or other mechanisms for interfacing or communicating with other portions of an electronic device or system. Such mechanisms may facilitate interaction with, for example, peripherals, communications devices, other processors, or memory. In the example of FIGURE 14, instruction set architecture 1400 may include an LCD video interface 1425, a Subscriber Interface Module (SIM) interface 1430, a boot ROM interface 1435, an SDRAM controller 1440, a flash controller 1445, and a Serial Peripheral Interface (SPI) master unit 1450. LCD video interface 1425 may provide output of video signals from, for example, GPU 1415 and through, for example, a Mobile Industry Processor Interface (MIPI) 1490 or a High-Definition Multimedia Interface (HDMI) 1495 to a display. Such a display may include, for example, an LCD. SIM interface 1430 may provide access to or from a SIM card or device. SDRAM controller 1440 may provide access to or from memory such as an SDRAM chip or

module. Flash controller 1445 may provide access to or from memory such as flash memory or other instances of RAM. SPI master unit 1450 may provide access to or from communications modules, such as a Bluetooth module 1470, high-speed 3G modem 1475, global positioning system module 1480, or wireless module 1485 implementing a communications standard such as 802.11.

[0128] 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.

[0129] Instruction architecture 1500 may include a memory system 1540 communicatively coupled to one or more execution entities 1565. Furthermore, instruction architecture 1500 may include a caching and bus interface unit such as unit 1510 communicatively coupled to execution entities 1565 and memory system 1540. In one embodiment, loading of instructions into execution entities 1564 may be performed by one or more stages of execution. Such stages may include, for example, instruction prefetch stage 1530, dual instruction decode stage 1550, register rename stage 155, issue stage 1560, and writeback stage 1570.

[0130] In one embodiment, memory system 1540 may include an executed instruction pointer 1580. Executed instruction pointer 1580 may store a value identifying the oldest, undispatched instruction within a batch of instructions. The oldest instruction may correspond to the lowest Program Order (PO) value. A PO may include a unique number of an instruction. Such an instruction may be a single instruction within a thread represented by multiple strands. A PO may be used in ordering instructions to ensure correct execution semantics of code. A PO may be reconstructed by mechanisms such as evaluating increments to PO encoded in the instruction rather than an absolute value. Such a reconstructed PO may be known as an "RPO." Although a PO may be referenced herein, such a PO may be used interchangeably with an RPO. A strand may include a sequence of instructions that are data dependent upon each other. The strand may be arranged by a binary translator at compilation time. Hardware executing a strand may execute the instructions of a given strand in order according to PO of the various instructions. A thread

may include multiple strands such that instructions of different strands may depend upon each other. A PO of a given strand may be the PO of the oldest instruction in the strand which has not yet been dispatched to execution from an issue stage. Accordingly, given a thread of multiple strands, each strand including instructions ordered by PO, executed instruction pointer 1580 may store the oldest—illustrated by the lowest number—PO in the thread.

[0131] 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.

[0132] 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.

[0133] 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.

[0134] To further facilitate its functions, bus interface unit 1520 may include an interrupt control and distribution unit 1511 for generating interrupts and other

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

[0135] Memory system 1540 may include any suitable number and kind of mechanisms for storing information for the processing needs of instruction architecture 1500. In one embodiment, memory system 1504 may include a load store unit 1530 for storing information such as buffers written to or read back from memory or registers. In another embodiment, memory system 1504 may include a translation lookaside buffer (TLB) 1545 that provides look-up of address values between physical and virtual addresses. In yet another embodiment, bus interface unit 1520 may include a Memory Management Unit (MMU) 1544 for facilitating access to virtual memory. In still yet another embodiment, memory system 1504 may include a prefetcher 1543 for requesting instructions from memory before such instructions are actually needed to be executed, in order to reduce latency.

[0136] The operation of instruction architecture 1500 to execute an instruction may be performed through different stages. For example, using unit 1510 instruction prefetch stage 1530 may access an instruction through prefetcher 1543. Instructions retrieved may be stored in instruction cache 1532. Prefetch stage 1530 may enable an option 1531 for fast-loop mode, wherein a series of instructions forming a loop that is small enough to fit within a given cache are executed. In one embodiment, such an execution may be performed without needing to access additional instructions from, for example, instruction cache 1532. Determination of what instructions to prefetch may be made by, for example, branch prediction unit 1535, which may access indications of execution in global history 1536, indications of target addresses 1537, or contents of a return stack 1538 to determine which

of branches 1557 of code will be executed next. Such branches may be possibly prefetched as a result. Branches 1557 may be produced through other stages of operation as described below. Instruction prefetch stage 1530 may provide instructions as well as any predictions about future instructions to dual instruction decode stage.

[0137] Dual instruction decode stage 1550 may translate a received instruction into microcode-based instructions that may be executed. Dual instruction decode stage 1550 may simultaneously decode two instructions per clock cycle. Furthermore, dual instruction decode stage 1550 may pass its results to register rename stage 1555. In addition, dual instruction decode stage 1550 may determine any resulting branches from its decoding and eventual execution of the microcode. Such results may be input into branches 1557.

[0138] Register rename stage 1555 may translate references to virtual registers or other resources into references to physical registers or resources. Register rename stage 1555 may include indications of such mapping in a register pool 1556. Register rename stage 1555 may alter the instructions as received and send the result to issue stage 1560.

[0139] Issue stage 1560 may issue or dispatch commands to execution entities 1565. Such issuance may be performed in an out-of-order fashion. In one embodiment, multiple instructions may be held at issue stage 1560 before being executed. Issue stage 1560 may include an instruction queue 1561 for holding such multiple commands. Instructions may be issued by issue stage 1560 to a particular processing entity 1565 based upon any acceptable criteria, such as availability or suitability of resources for execution of a given instruction. In one embodiment, issue stage 1560 may reorder the instructions within instruction queue 1561 such that the first instructions received might not be the first instructions executed. Based upon the ordering of instruction queue 1561, additional branching information may be provided to branches 1557. Issue stage 1560 may pass instructions to executing entities 1565 for execution.

[0140] Upon execution, writeback stage 1570 may write data into registers, queues, or other structures of instruction set architecture 1500 to communicate the completion of a given command. Depending upon the order of instructions arranged in issue stage 1560, the operation of writeback stage 1570 may enable additional instructions to be executed. Performance of instruction set architecture 1500 may be monitored or debugged by trace unit 1575.

[0141] 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.

[0142] Execution pipeline 1600 may include any suitable combination of steps or operations. In 1605, predictions of the branch that is to be executed next may be made. In one embodiment, such predictions may be based upon previous executions of instructions and the results thereof. In 1610, instructions corresponding to the predicted branch of execution may be loaded into an instruction cache. In 1615, one or more such instructions in the instruction cache may be fetched for execution. In 1620, the instructions that have been fetched may be decoded into microcode or more specific machine language. In one embodiment, multiple instructions may be simultaneously decoded. In 1625, references to registers or other resources within the decoded instructions may be reassigned. For example, references to virtual registers may be replaced with references to corresponding physical registers. In 1630, the instructions may be dispatched to queues for execution. In 1640, the instructions may be executed. Such execution may be performed in any suitable manner. In 1650, the instructions may be issued to a suitable execution entity. The manner in which the instruction is executed may depend upon the specific entity executing the instruction. For example, at 1655, an ALU may perform arithmetic functions. The ALU may utilize a single clock cycle for its operation, as well as two shifters. In one embodiment, two ALUs may be employed, and thus two instructions may be executed at 1655. At 1660, a determination of a resulting branch may be made. A program counter may be used to designate the destination to which the branch will be made. 1660 may be executed within a single clock cycle. At 1665, floating point arithmetic may be performed by one or more FPUs. The floating point operation may require multiple clock cycles to execute, such as two to ten cycles. At 1670, multiplication and division operations may be performed. Such operations may be performed in four clock cycles. At 1675, loading and storing operations to registers or other portions of pipeline 1600 may be performed. The operations may include loading and storing addresses. Such operations may be performed in four clock cycles. At 1680, write-back operations may be performed as required by the resulting operations of 1655-1675.

[0143] 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.

[0144] 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.

[0145] 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 DSP 1760, a drive 1720 such as a Solid State Disk (SSD) or a Hard Disk Drive (HDD), a wireless local area network (WLAN) unit 1750, a Bluetooth unit 1752, a Wireless Wide Area Network (WWAN) unit 1756, a Global Positioning System (GPS), a camera 1754 such as a USB 3.0 camera, or a Low Power Double Data Rate (LPDDR) memory unit 1715 implemented in, for example, the LPDDR3 standard. These components may each be implemented in any suitable manner.

[0146] Furthermore, in various embodiments other components may be communicatively coupled to processor 1710 through the components discussed above. For example, an accelerometer 1741, Ambient Light Sensor (ALS) 1742, compass 1743, and gyroscope 1744 may be communicatively coupled to sensor hub 1740. A thermal sensor 1739, fan 1737, keyboard 1746, and touch pad 1730 may be communicatively coupled to EC 1735. Speaker 1763, headphones 1764, and a microphone 1765 may be communicatively coupled to an audio unit 1764, which may in turn be communicatively coupled to DSP 1760. Audio unit 1764 may include, for example, an audio codec and a class D amplifier. A SIM card 1757 may be communicatively coupled to WWAN unit 1756. Components such as WLAN unit 1750 and Bluetooth unit 1752, as well as WWAN unit 1756 may be implemented in a Next Generation Form Factor (NGFF).

[0147] FIGURE 18 illustrates a block diagram of a system 1800 for implementing heterogeneous decompression processing, in accordance with embodiments of the present disclosure. In one embodiment, decompression of any the results of any LZ77-based compression algorithm may be performed. In another embodiment, decompression may be performed using a heterogeneous partitioning of the computations needed for decompression. Some computations may be performed by specific hardware on system 1800, while other computations may be performed by software executed through existing execution pipelines. Such software may be executed through execution of instructions therein. The specific hardware may include a fixed-function hardware engine. The software may be executed through SIMD processing. In one embodiment, system 1800 may perform heterogeneous decompression processing based upon extensions to or instructions of SIMD-formats.

[0148] In one embodiment, such data compression may include the LZ4 data compression scheme. In another embodiment, the data compression may include a lossless compression scheme. In other embodiments, the data compression may include any compression scheme in which a number of literals are recorded within one record. For example, the data compression may include LZ77-based compression formats or the Snappy data compression scheme. In yet other embodiments, the data compression may include data compressions schemes in which parameters must be set with an earlier portion of the data compression scheme, wherein the parameters describe or define aspects of later portions of the data compression scheme. Such parameters might describe or define, for example, length of data that was not found to be repeated from earlier data, length of data that was found to be repeats of earlier data, and offsets of data that was found to be repeats of earlier data. For any given input byte from an input data stream, data compression may be performed. The input data stream may result in multiple compressed data sequences. As the compression algorithm may find repeated substrings and replace them with backward references in the form of relative distance offsets, a compressed data stream may include a series of elements two types. The first type may include literal bytes, and the second type may include pointers to replicated strings. A pointer may be represented as a pair in the format <length, backward distance offset>.

[0149] In one embodiment, decompression performed by system 1800 may include two or more steps. First, system 1800 may parse a compressed input stream into tokens. Such tokens may include literals. Second, system 1800 may copy a specified number of bytes from the literals to an output stream. The number of bytes may be specified according to the compression scheme. Decompression by system 1800 may contrast decompression in other systems that are implemented entirely in software or entirely in hardware. In one embodiment, parsing the input stream into tokens may be performed in a hardware component of system 1800, such as a hardware decompressor 1818. In another embodiment, copying specified bytes to an output stream may be performed by software, including instructions executed through a traditional multi-purpose execution pipeline. In a further embodiment, tokens may be parsed by hardware decompressor 1818 (or another suitable part of system 1800) into a plurality of execution streams for execution by cores. Each stream may be generated to include tokens formatted according to a desired or specified scheme. Furthermore, tokens may include different formats within a given stream. As discussed in further detail below, some tokens may be in a “fast” format option and some tokens may be in a “slow” format option.

[0150] Decompression may be performed by any suitable portion of system 1800. In one embodiment, decompression may be performed in hardware. Such hardware may include an accelerator, a module, co-processor, a peripheral on a communication bus, or a memory controller. For example, system 1800 may implement such hardware in decompressor 1818. Decompressor 1818 may be included inside processor 1802 or may be implemented outside processor 1802 and communicatively coupled to processor 1802. Decompressor 1818 may be implemented by any suitable combination of digital circuitry and analog circuitry. Decompressor 1818 may include various execution units especially designed to perform decompression according to the embodiments of this disclosure.

[0151] Decompression may be performed based upon any suitable mechanism. In one embodiment, decompression performed by decompressor 1818 may be initiated by receiving jobs from processor 1802. Such jobs may be initiated by processor 1802 writing control and status registers (CSRs) to indicate parameters of decompression, what type of decompression is to be performed, and where such data to be decompressed is located. Some parameters and data to be compressed may be located in, for example, memory

hierarchy 1832. Memory hierarchy 1832 may include any suitable number of combinations of physical memory, caches, or other storage. In one embodiment, decompression might be performed based upon a determination that instructions in instruction stream 1804 to be executed by processor 1802 are for performing decompression. In such embodiments, execution of decompression might be offloaded to decompressor 1818. In another embodiment, decompression might be performed by analysis by processor 1802 that decompression will be advantageous. In yet another embodiment, decompression might be specified by use of an application programming interface, driver, or other mechanism. The resulting data from data compression may be sent by compression module 1818 to memory hierarchy 1832. The results may be read, sent to recipients, or otherwise utilized or processed by cores 1814. While cores 1814 could themselves perform the second step of the decompression, performance of the first step of the decompression might be slower than on hardware especially designed to perform the decompression such as decompressor 1818. Accordingly, in one embodiment, parsing steps of decompression may be offloaded to compression module 1818.

[0152] Decompressor 1818 may be implemented in any suitable manner. In one embodiment, compression module 1818 may include a buffer 1816 for outputting data to memory hierarchy 1832. By using such a buffer 1816, the entire contents of buffer 1816 may be written more efficiently to memory hierarchy 1832 than, for example, performing direct memory access writes to random locations in memory hierarchy as each data element is produced.

[0153] Although a single instance of decompressor 1818 is illustrated, system 1800 may include any suitable number and kind of decompressors 1818. Moreover, decompressor 1818 may include any suitable number and kind of decompressor circuits. Each decompressor 1818 or circuit thereof may be able to perform one or more kinds of decompression algorithms. The particular decompressor 1818 or circuit within decompressor 1818 corresponding to a given decompression algorithm may be selected according to instructions executed specifying the decompression or an evaluation of the type of data to be decompressed, or an evaluation of the type of operation that is to be performed once decompression is complete.

[0154] For example, as shown in FIGURE 19, decompressor 1818 may include several parser circuits 1902. Each parser circuit 1902 may correspond to a different algorithm. An input stream, such as data 1920 from instruction stream 1804, may be divided and assigned to a given parser circuit 1902.

[0155] Decompressor 1818 may produce a token stream that is appropriate for software to accept and convert in the decompression second step to decompressed data. The format of the token stream may depend upon the particular decompression algorithm. Furthermore, the format of the token stream may be selected as having an appropriate size for software to perform the second step of decompression. Such an appropriate size may be based upon the capabilities of the host processor. For example, scalars may be used when support for SIMD is unavailable. Furthermore, different formats may have different trade-offs between size and processing speed. For example, when the processor supports both scalar and SIMD formats, the SIMD format may be larger but would allow faster overall processing. The output format may be chosen from a variety of output formats. The formats may be produced by, for example, formatter circuit 1904A for scalars 1906 or formatter circuit 1904B for SIMD 1908. In one embodiment, the formats may include a formatter circuit 1904C for which access of a cache 1910 in a memory hierarchy may be made directly. Formatter circuit 1904C may access the cache 1920 directly and perform memory copies directly, such as with a point-to-point interface.

[0156] Returning to FIGURE 18, the data that was compressed may be represented by data 1826. Data 1826 may be shown as it was originally generated, before being compressed. For example, data 1826 may include a sequence of data including a first string, "string-x" 1828 followed by another string, "string-y" 1830. The same string string-y 1830 may have appeared in previous data (shown as part of data 1826, but in actuality appearing in previous data). When data was originally compressed, the compression module may have reached string-x 1828 and determined that that string-x 1828 did not appear in data to be compressed before such a point. String-x 1828 may be considered a literal, in that its values will be passed into the data compression sequence literally, with its values preserved. However, the appearance of an additional instance of string-y 1830 represented an opportunity to compress the data, rather than literally repeat its contents into the data compression sequence. Following a compression method such as LZ4, or a modified version

thereof, the compressed data sequence may have been encoded with a reference to the previous instance of string-y 1830 in the input data (such as in data previously encountered in the input stream), rather than its literal contents. Such an encoding may be represented by one or more fields representing the offset of such a previous appearance of string-y 1830, as well as one or more fields representing the width of string-y 1830. In such a case, string-y 1830 may be referred to the match, and an offset of the match may be encoded (where it previously appeared in the input data) and a width of the match in the compressed data sequence, rather than the literal contents of the match.

[0157] System 1800 may otherwise fetch, dispatch, execute, and retire instructions, such as those in instruction stream 1804 in a sequential or out-of-order manner. System 1800 may utilize any suitable number or kind of out-of-order processors or processing entities. System 1800 may illustrate elements of such a processor 1802, which may include any processor core, logical processor, processor, or other processing entity such as those illustrated in FIGURES 1-17. System 1800 may include a front end 1806 communicatively coupled to one or more execution units 1808. Front end 1812 may process instructions so that they may be executed by execution units 1808. Front end 1812 may include allocate units, decode instructions, perform binary translation, and dispatch units. Moreover, processor 1802 may include retirement units to reclaim resources used by instructions. One or more elements, such as front end 1812 and execution units 1808, may be included in one or more cores 1814. Cores 1814 may execute instructions to perform the second half of decompression, such as copying literals from designated offsets within the input stream.

[0158] As an instruction stream 1804 is executed, some data may be compressed as indicated by, for example, implicit or explicit instructions in instruction stream 1804. Such data may be stored in for example, memory hierarchy 1832. Decompression may be offloaded in part to decompressor 1818. The resulting token stream from decompression may be passed back into memory hierarchy 1832. Additional software instructions may be executed to, for example, copy literals into memory hierarchy 1832. Such additional instructions may be executed in cores 1814 and result in decompressed data.

[0159] The token stream output from decompressor 1818 may be implemented in any suitable manner.

[0160] FIGURE 20 illustrates a format of the token stream output from decompressor 1818, in accordance with embodiments of the present disclosure. As discussed above, a token may be created based upon its probable use as either “fast” or “slow.” Although these designations are used in this disclosure, more than two such categorizations may be used when implementing system 1800. The “fast” token format may represent data that may be copied faster by software in the second step than content from the “slow” token format. The CPU processing of “fast” records will take less time than the CPU processing of “slow” records. The output stream of tokens may include a heterogeneous mixture of tokens in the “fast” and the “slow” format. More “fast” records are likely to be used than “slow” records.

[0161] For example, assume eight bytes are copied at a time using the “fast” tokens. Given a run of literals (“aaaaaaaa”), the first “a” is encoded as a literal, and this is followed by a match with a distance of one and a length of nine. The length is larger in size than the distance. Copying data after the current point to be written byte by byte may be performed correctly. While the first “a” would be copied as a literal, this copy is independent of the next bit, which is the copy of length 9 and distance 1. Multiple bytes will be copied from a distance of 1. In this example, the “a” would be encoded as a literal, and then a fast record would be written with the literal as a partial record (i.e. with the higher lanes having 0 size), and this would be followed by a slow record for the copy of nine at a distance of one. The match may be encoded as a slow record as the copy is being performed eight bytes at a time—eight bytes are read and eight bytes are written at a time. The match with a length of nine at a distance of one would read one byte of valid data and seven bytes of incorrect data. The read in effect interferes with itself. Some of the data that is being read is being written by the same 8-byte copy. However, most of the time match tokens will include fast records. A slow token may occur for matches when the distance is less than the length and the distance is also less than the read-or-write byte size (in this case, eight). If the distance is greater than or equal to the length, then all of the data that needs to be read exists before the copy begins. Furthermore, if the distance is greater than the read-or-write byte size, then the match can be encoded in a series of 8-byte fast record copies.

[0162] Partial “fast” records may be used in cases when there are data dependencies. For example, during a parallel read of data, lane 4 of a read may attempt to

read data to be written by lane 1. Thus, a parallel fetch of both lanes as part of an eight-byte simultaneous operation may require a fetch of lane 4 before the write to lane 1. Thus, the read may result in unuseful or garbage data. If a reference to a previous match overlapped an earlier lane for such a data-dependent read, a partial fast-path token may be issued. The problematic copy would be put into the first lane of a new fast record. , then a new token issued for the problematic read. The past-path record thus might not be full. In another example, data in a later lane (in some particular fast record) might be reading data written from an earlier lane. In such a case, a partially full fast record might be emitted and later move the copy needed into a new record. Hypothetically, use of slow records might be avoided completely if a fast record with only a single lane were used, and where the lane in each record only copied a single byte. However, this would cause bloat in the size of the output, as approximately 25 bytes of intermediate or metadata would be generated for every such byte of actual data.

[0163] Consider another use case for a match where the length is 256 and the distance is 9. In one embodiment, 32 fast records may be produced wherein 31 of them have only one lane used. This approach may minimize use of slow records (i.e. to only use them when absolutely necessary). However, this may cause bloat and overhead that, under some design constraints, may exceed whatever savings are gained by avoiding slow records. For example, processing 32 fast records actually takes longer than processing a single slow record. In such a case, the system may choose to use a slow record under the sole condition of (distance < length), ignoring the other relationship. Using the present example, the system using such a scheme would generate one slow record rather than the thirty-two fast ones.

[0164] As shown in FIGURE 20, the intermediate token stream may include a series of byte-aligned records. The fast record may include a series of qwords of addresses and then a qword of a mask, where the masks are 8 bytes. Each byte may have a value of (1, 3, 7, ..., FF) indicating how many bytes should be copied from the corresponding address. If the address is not valid, then the corresponding byte in the mask is 0. By definition, addr0 may be always considered valid, and thus mask byte 0 might never be set to '0'. In the slow record, the size or length field may specify the copy length. For data such as literals, the qword addresses may point to a buffer containing the literal data. For other data, such as a

match, the address may point to previously decompressed data. Such decompressed data may be in, for example, history of past execution. Copying literals or copying match data from earlier results might thus be handled in the same manner. The software processing such a record might merely use such addresses to load the appropriate literal or match without having any indication of whether a given address is a specific one of a literal or a match.

[0165] Using the format shown in FIGURE 20, the number of slow records or copy operations used may be reduced or minimized compared to other approaches. However, there will be partial fast records.

[0166] FIGURE 21 illustrates another format of the token stream output from decompressor 1818, in accordance with embodiments of the present disclosure. The token format in FIGURE 21 may vary from the token format shown in FIGURE 20 in that literals may be allowed to appear in the slow format. As shown, the slow record may include a qword that specifies the address of the literal, rather than an offset. The length may still be specified. As a result, the slow record might be larger in size (ten bytes rather than six). The slow record may be able to be either a reference to previously decompressed data (which may contain a previous match) or a literal.

[0167] When forming the intermediate token stream output from the parsers, when a dependent token arrives (wherein copying a value requires reading from another location for which a write is being performed), rather than flushing a partial fast record, the oldest tokens from the fast record are rewritten as slow records until the dependency is gone. For example, if the dependent token is in lane 4 and dependent upon a token in lane 1, tokens in lanes 0-1 are rewritten as slow tokens. In another example, tokens in lanes 0-3 are rewritten as slow tokens. After tokens are rewritten into slow tokens and the dependent token is now an independent token, the remaining tokens may be issued in the fast format.

[0168] In comparison to the format shown in FIGURE 20, as an advantage, all fast record tokens might be full. That is, all lanes (such as all eight lanes) might be used. As a disadvantage, more slow records might be used. Moreover, the slow records may be larger. Formerly partial records might include a zero-byte mask. As such 8 bytes of data might be effectively loaded but then discarded.

[0169] Returning to FIGURE 20, in one embodiment, the formats in FIGURE 20 may be amended such that, for partial records, a valid address (copied from earlier in the record) is placed in the higher address qwords. Accordingly, a partial record is modified. In a further embodiment, the valid address is inserted and as a result the load occurs. However, the results of this load may be later discarded. Thus, this might be considered a useless load. Nevertheless, in such an embodiment, fast path computations do not need to compute the values of mask for gather operations.

[0170] In another embodiment, the formats in FIGURE 20 may be amended such that the records are aligned between elements that are larger than bytes. While the records are byte aligned in FIGURE 20, they are not necessarily aligned by larger elements, such as qwords. In a further embodiment, bytes may be inserted between masks and addresses in the fast record. In another, further embodiment, the byte added between the masks and the addresses in the fast record would explicitly contain the gather mask.

[0171] FIGURE 22 illustrates another format of the token output from decompressor 1818, in accordance with embodiments of the present disclosure. In the example shown in FIGURE 22, 16-bit fields may be used. However, variations of this format (and those formats in the other figures) may include use of 32-bit fields. Each lane may be two bytes wide.

[0172] Using this format, the literal data may be stored in the record. As such, the literal data might not need to be loaded as part of a gather operation. The distances may be stored in 16-bit fields so that less space is needed.

[0173] In the fast record portion of this format, eight masks may be used. Eight bits may serve as a mask. Each bit may correspond to lanes of data. A byte of the masks may identify which bytes in a lane (such as a ZMM qword lane) contain valid data. Valid values may include, for example, 0x00, 0x01, 0x03, 0x07, etc. Each bit of the K byte may indicate whether or not the corresponding lane is a reference to previously decompressed data. This previously decompressed data may have been made by a previous match, by literals, or a combination thereof. A bit set to zero will indicate that the corresponding lane does not contain a reference to a previously decompressed data but instead contains a literal. The L_n bytes may be used to store lane information. In one embodiment, the lane bytes may include one or two bytes of literal data. In another embodiment, the lane bytes may include

a reference value. The reference value may be defined by (64k (or other size of addressing) - distance). For example, a distance of one would be represented as 0xFFFF.

[0174] For the slow record portion of this format, a first constant byte of zero may be maintained. A second byte may specify a size (S) that is the number of bytes to be copied, less one. The remaining bytes may specify a distance backwards for the source of the copy. The sizes of these fields may be adjusted as necessary. For example, in the slow record, the length may be two bytes in size and the distance may be one byte in size.

[0175] Using these formats, typically copying would be used with the fast records. A gather instruction would load a ZMM register with data, compress it using the mask, and the write to the output. Data dependencies or overlapped copies (wherein the destination bytes overlap the source bytes) would disrupt this process such that the slow format would be used. If an odd number of literals is required, then only a single byte is wasted (rather than three, five, or seven). For a 64-bit address, only a 16-bit offset is needed.

[0176] Under ideal conditions, a negative offset is returned for writing decompressed data. Such a negative offset means that the data needed for the load can be easily gathered. The actual register that is pointing to the destination for copying can identify 2^{16} bits backwards. Thus, a positive offset can be used to identify such a register.

[0177] Returning to FIGURE 19, operation of formatter circuits 1904 may be made by any suitable number or kind of circuitry. Formatter circuits 1904 may operate to iteratively push entries into an 8-entry or lane or accumulator. When eight entries are gathered, a fast-record may be written and the accumulator set to empty. If a conflict arises due to dependency, and there is a partial record in the accumulator, the fast-record is written and the logic of the circuitry attempts the write again. If there is a conflict and the accumulator is empty, the formatter circuit writes a slow-record.

[0178] If such circuitry of the formatter circuit were written in VHDL or microcode, or if the circuitry of the formatter circuit were written into pseudo-code, an example of such pseudo-code could include:

```
// inputs -> "is_lit", "len", and "dist"
while (len != 0) {
  if (is_lit) {
    // is literal, no conflict
```

```

size = min(len, 2);
push_fast_lit(lits, size);
} else {
// is reference to a previously decompressed data
size = min(len, 8);
if (dist >= m_size_fast_entries + size) {
// no conflict
push_fast_ref(_64K - dist + m_size_fast_entries, size);
} else {
if (m_num_fast_entries != 0) {
// flush fast rec, and try again
write_fast();
continue;
}
// write slow rec
size = min(len, 256);
write_slow_rec(size, dist);
}}
len -= size;
if (m_num_fast_entries == 8)
write_fast();
}}
```

[0179] Given inputs of whether a literal is specified, a length, and a distance, the formatter circuit may operate as long as the input length is not zero. If the literal is specified, then a size may be specified as a lesser of the length and two. The literal may be pushed into the fast buffer. The process of pushing into the buffer may take as operands the literals and the determined size. Furthermore, this process may include setting masks of the fast entries ("Masks[m_num_fast_entries]") (the number of fast entries created thus far may be referred to as m_num_fast_entries) to a "size-bit" mask. For example, if the size is two, then the mask is 0x03. The process may then include writing one or two bytes (as appropriate) to L[m_num_fast_entries].

[0180] Otherwise, if the literal is not specified, then a reference to previously decompressed data may have been specified instead. The size may be set to the lesser of the length and eight. If the distance is greater than the size of the fast entries yet created plus the previously calculated size, then there is no conflict and the reference to previously decompressed data may be pushed on to the fast buffer. Otherwise, if the number of fast entries yet created is not zero, then the fast buffer may be flushed and the entire process repeated. A slow record may be created, wherein the size is the lesser of the length and 256, and the slow buffer or record may be written with the size and distance.

[0181] If the write buffer is full, the write buffer may be written. The length may be decremented from the size.

[0182] The process of pushing a reference to previously decompressed data to the fast buffer ("push_fast_ref(offset, size)") may include setting masks of the fast entries yet created to the size-bit mask, and setting K at the bit corresponding the number of fast entries yet created. The offset may be written to the lane bytes indexed at the number of fast entries yet created.

[0183] In each case where the fast buffer is populated, the calculated size may be added to the size of fast entries yet created and the number of fast entries yet created may be incremented.

[0184] As discussed above, once the intermediate token stream is received by the rest of system 1800, software may execute the actual copying of values to memory to finalize decompression. The operation of a copy loop in such software may be illustrated according to the following examples.

[0185] For example, a token stream may include three bytes of literals (A B C), a three byte reference (offset O1), a five byte reference (offset O2), five bytes of literals (D E F G H), and a dependent reference that cannot be encoded. A resultant record may include (LSB on the right) K as (00001100) as a gather mask, masks as (00 02 03 03 1F 07 01 03) as a compression mask and lane bytes as (00 0H GF ED 02 01 0C BA).

[0186] The first two fields might be loaded into general purpose registers and in mask registers. The lane bytes may be loaded into a ZMM register with an instruction to expand words into qwords. Accordingly, a read of an XMM register would expand using zero-fill operations each of the eight word lanes into eight qword lanes.

[0187] FIGURE 23 illustrates register content results for the processing of a fast token stream, according to embodiments of the present disclosure. In such results, for example the “0 GF” notation may indicate that the literals “GF” are in the lower two bytes while the remainder bytes (six such bytes) are zero. “0 O1” and “0 O2” may refer to the 16-bit offset-1 or offset-2 being 0-extended to fill a qword.

[0188] From these results, a gather may be performed using the gather mask. An instruction for such an operation may be specified as, for example, “vpgatherqq” with the parameters “zmm{G_Mask}” with an output. In some architectures, the same zmm register cannot be both a source and a destination, and as such the output may be a copy of the source. A control bit for a given lane may specify that when the bit is 0, the output register is unchanged, and when the bit is 1, the output register is written with the result of the operation. In the present example, offsets are in some lanes and data are in other lanes. While use of the same register as the source and destination may be convenient, it might not be allowed given the particular architecture. Accordingly, a copy of the register may be made and use the original as the source and the other as the destination. The lanes containing literal data (and a 0 mask bit) will be unchanged, so they will contain the desired literal data, and the lanes with a 1 mask bit will be loaded with the data read from the gather instruction. In the instruction, the operands are given as “[zmm + gpr]”, where the zmm register is the register with data to be read, containing offsets in the appropriate lane, and the gpr (general purpose register) points to the output buffer (or the place to write in the output buffer minus 64K, so that the offsets are all positive values). [zmm + gpr] may specify the input, i.e., the data to be read. Then the gather operation adds (for each lane as selected by the mask bits) the offset from that lane to the address in the GPR to get the actual address from which to read the data.

[0189] After the gather, only lanes two and three might be changed, which previously only held references or offsets. Rather than including offsets, they now include five and three bytes of the actual data. For example, the zmm register may specify (0 00 0 0H 0 GF 0 ED D2 D1 0 0C 0 BA).

[0190] Next, the results may be compressed using the compression mask. This may take all of the bytes in the zmm register that have a bit set to one in the mask and concatenate the results, eliminating the (non-leading) zeroed bytes in the zmm. As a result,

(00000 HGFED D2 D1 CBA) may be written to memory. The output pointer may be advanced by the pop count (the count of 1-bits) of the compression mask register.

[0191] In one embodiment, the output of the write may be to the pointer of the output buffer plus 64k. This makes the output buffer pointer point to a spot 64k before the actual output buffer address. This may allow use of positive offsets rather than negative values. This may be to facilitate “base+offset” wherein “base-offset” usage will not function correctly. Moreover, padding in the conversion from word to qwords might not perform sign extension.

[0192] FIGURE 24 is an illustration of an example method 2400 for decompression, in accordance with embodiments of the present disclosure. Method 2400 may be performed in part, for example, by decompressor 1818 and may yield data sequences for output to recipients such as core 1814 through memory hierarchy 1832. Method 2400 may be performed in part, for example, by cores 1814 to copy values from tokens generated by decompressor 1818 into memory.

[0193] Method 2400 may begin at any suitable point and may execute in any suitable order. In one embodiment, method 2400 may begin at 2405. In various embodiments, method 2400 may be performed during the execution of an out-of-order processor. Steps of method 2400 may be performed in parallel with one another.

[0194] At 2405, data to be decompressed may be identified. The data may be determined to be decompressed by a given compression scheme. At 2410, elements in the stream of data to be decompressed may be identified. The elements may arrive, one at a time or in blocks, to a decompressor. 2405 and 2410 may repeat and execute in parallel with other steps of method 2400.

[0195] At 2415, in one embodiment the elements may be parsed according to a specified algorithm. Moreover, a destination format (such as scalar or SIMD) may be determined for the data.

[0196] At 2420, as part of the parsing, it may be determined whether there is any data dependency or overlap among a first N bytes, wherein the N bytes are to be otherwise written together in a fast record to a token stream. If there are no data dependencies, at 2425 N bytes may be written together in a fast record to a token stream. If there are data dependencies, at 2430 an available number of bytes available to be written to a fast record

may be written. Unavailable bytes may be masked out or omitted, depending upon formatting.

[0197] At 2435, the fast record stream may be issued. At 2440, if needed, slow byte records may be created to overcome data dependencies. These may also be issued.

[0198] At 2445, it may be determined whether additional elements are to be processed into the token stream. If so, method 2400 may repeat at 2420. Otherwise, method 2400 may proceed to 2450.

[0199] At 2450, the token stream may be received by software. At 2455, the software may copy elements from the token stream according to literal identifiers, length, offset, and other suitable parameters. The elements may be copied into memory according to the parameters by software. 2450 and 2455 may repeat and operate independently from other steps of method 2400 as needed.

[0200] At 2460, it may be determined whether method 2400 will terminate or optionally repeat at, for example, 2405.

[0201] Method 2400 may be initiated by any suitable criteria. Furthermore, although method 2400 describes an operation of particular elements, method 2400 may be performed by any suitable combination or type of elements. For example, method 2400 may be implemented by the elements illustrated in FIGURES 1-23 or any other system operable to implement method 2400. As such, the preferred initialization point for method 2400 and the order of the elements comprising method 2400 may depend on the implementation chosen. In some embodiments, some elements may be optionally omitted, reorganized, repeated, or combined.

[0202] Embodiments of the mechanisms disclosed herein may be implemented in hardware, software, firmware, or a combination of such implementation approaches. Embodiments of the disclosure may be implemented as computer programs or program code executing on programmable systems comprising at least one processor, a storage system (including volatile and non-volatile memory and/or storage elements), at least one input device, and at least one output device.

[0203] Program code may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a

processing system may include any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

[0204] The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

[0205] One or more aspects of at least one embodiment may be implemented by representative instructions stored on a machine-readable medium which represents various logic within the processor, which when read by a machine causes the machine to fabricate logic to perform the techniques described herein. Such representations, known as “IP cores” may be stored on a tangible, machine-readable medium and supplied to various customers or manufacturing facilities to load into the fabrication machines that actually make the logic or processor.

[0206] 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.

[0207] 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.

[0208] 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.

[0209] 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.

[0210] Embodiments of the present disclosure include a processor. The processor may include a memory hierarchy. In combination with any of the above embodiments, the processor may include a buffer for input to the memory hierarchy. In combination with any of the above embodiments, the processor may include a decompressor. In combination with any of the above embodiments, the processor may include circuitry to read elements to be decompressed according to a compression scheme, parse the elements to identify literals and matches, and, with the literals and matches, generate an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data. In combination with any of the above embodiments, the intermediate token stream is to include a first format for multiple tokens that are to be written in parallel with each other and a second format for tokens that include a data dependency upon themselves. In combination with any of the above embodiments, the decompressor further includes

circuitry to generate the second format with an address. In combination with any of the above embodiments, multiple tokens that are to be written in parallel with each other are to be included in N lanes. In combination with any of the above embodiments, the decompressor further includes circuitry to generate the second format with an address, determine that a data dependency exists within the elements to be decompressed, and, based upon the determination that the data dependency exists within the elements to be decompressed, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed. In combination with any of the above embodiments, the decompressor further includes circuitry to generate the second format with an address, determine that a data dependency exists within the elements to be decompressed, based upon the determination of the data dependency, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed, and write a token that caused the data dependency in the first format. In combination with any of the above embodiments, the multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format. In combination with any of the above embodiments, the decompressor further includes circuitry to determine that a data dependency exists within the elements to be decompressed, based upon the determination of the data dependency, populate the N lanes with less than N tokens, and copy a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency. In combination with any of the above embodiments, the decompressor further includes circuitry to insert a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression. In combination with any of the above embodiments, the decompressor further includes circuitry to store literal data into the first format.

[0211] Embodiments of the present disclosure include a system. The system may include a processor. The processor may include a memory hierarchy. In combination with any of the above embodiments, the processor may include a buffer for input to the memory hierarchy. In combination with any of the above embodiments, the processor may include a decompressor. In combination with any of the above embodiments, the processor may include circuitry to read elements to be decompressed according to a compression scheme,

parse the elements to identify literals and matches, and, with the literals and matches, generate an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data. In combination with any of the above embodiments, the intermediate token stream is to include a first format for multiple tokens that are to be written in parallel with each other and a second format for tokens that include a data dependency upon themselves. In combination with any of the above embodiments, the decompressor further includes circuitry to generate the second format with an address. In combination with any of the above embodiments, multiple tokens that are to be written in parallel with each other are to be included in N lanes. In combination with any of the above embodiments, the decompressor further includes circuitry to generate the second format with an address, determine that a data dependency exists within the elements to be decompressed, and, based upon the determination that the data dependency exists within the elements to be decompressed, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed. In combination with any of the above embodiments, the decompressor further includes circuitry to generate the second format with an address, determine that a data dependency exists within the elements to be decompressed, based upon the determination of the data dependency, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed, and write a token that caused the data dependency in the first format. In combination with any of the above embodiments, the multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format. In combination with any of the above embodiments, the decompressor further includes circuitry to determine that a data dependency exists within the elements to be decompressed, based upon the determination of the data dependency, populate the N lanes with less than N tokens, and copy a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency. In combination with any of the above embodiments, the decompressor further includes circuitry to insert a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression. In combination with any of the above embodiments, the decompressor further includes circuitry to store literal data into the first format.

[0212] Embodiments of the present disclosure include a method operating within a processor. The method may include reading elements to be decompressed according to a compression scheme, parsing the elements to identify literals and matches, and, with the literals and matches, generating an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data. In combination with any of the above embodiments, the intermediate token stream is to include a first format for multiple tokens that are to be written in parallel with each other and a second format for tokens that include a data dependency upon themselves. In combination with any of the above embodiments, the method further includes formatting the second format with an address. In combination with any of the above embodiments, multiple tokens that are to be written in parallel with each other are to be included in N lanes. In combination with any of the above embodiments, the method further includes generating the second format with an address, determining that a data dependency exists within the elements to be decompressed, and, based upon the determination that the data dependency exists within the elements to be decompressed, rewriting oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed. In combination with any of the above embodiments, multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format. In combination with any of the above embodiments, the method further comprises determining that a data dependency exists within the elements to be decompressed, based upon the determination of the data dependency, populating the N lanes with less than N tokens, and copying a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency. In combination with any of the above embodiments, the method further comprises inserting a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression. In combination with any of the above embodiments, the method further includes storing literal data into the first format. In combination with any of the above embodiments, the method further includes storing literal data into the first format.

[0213] Embodiments of the present disclosure include an apparatus. The apparatus may include means for reading elements to be decompressed according to a compression

scheme, means for parsing the elements to identify literals and matches, and, with the literals and matches, means for generating an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data. In combination with any of the above embodiments, the intermediate token stream is to include a first format for multiple tokens that are to be written in parallel with each other and a second format for tokens that include a data dependency upon themselves. In combination with any of the above embodiments, the apparatus further includes means for formatting the second format with an address. In combination with any of the above embodiments, multiple tokens that are to be written in parallel with each other are to be included in N lanes. In combination with any of the above embodiments, the apparatus further includes means for generating the second format with an address, means for determining that a data dependency exists within the elements to be decompressed, and, based upon the determination that the data dependency exists within the elements to be decompressed, means for rewriting oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed. In combination with any of the above embodiments, multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format. In combination with any of the above embodiments, the apparatus further comprises means for determining that a data dependency exists within the elements to be decompressed, based upon the determination of the data dependency, means for populating the N lanes with less than N tokens, and means for copying a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency. In combination with any of the above embodiments, the apparatus further comprises means for inserting a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression. In combination with any of the above embodiments, the apparatus further includes means for storing literal data into the first format. In combination with any of the above embodiments, the apparatus further includes means for storing literal data into the first format.

CLAIMS

What is claimed is:

1. A processor, comprising:
a memory hierarchy;
a buffer for input to the memory hierarchy; and
a decompressor, including circuitry to:
read elements to be decompressed according to a compression scheme;
parse the elements to identify literals and matches; and
with the literals and matches, generate an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data;
wherein the intermediate token stream is to include:
a first format for multiple tokens that are to be written in parallel with each other; and
a second format for tokens that include a data dependency upon themselves.
2. The processor of Claim 1, wherein the decompressor further includes circuitry to generate the second format with an address.
3. The processor of Claim 1, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes; and
the decompressor further includes circuitry to:
generate the second format with an address;
determine that a data dependency exists within the elements to be decompressed; and
based upon the determination that the data dependency exists within the elements to be decompressed, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed.

4. The processor of Claim 1, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes; and
the decompressor further includes circuitry to:
generate the second format with an address;
determine that a data dependency exists within the elements to be decompressed;
based upon the determination of the data dependency, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed; and
write a token that caused the data dependency in the first format.

5. The processor of Claim 1, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format;
the decompressor further includes circuitry to:
determine that a data dependency exists within the elements to be decompressed;
based upon the determination of the data dependency, populate the N lanes with less than N tokens;
copy a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency.

6. The processor of Claim 1, wherein the decompressor further includes circuitry to insert a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression.

7. The processor of Claim 1, wherein the decompressor further includes circuitry to store literal data into the first format.

8. A system, comprising:
- a processor;
 - a memory hierarchy;
 - a buffer for input to the memory hierarchy; and
 - a decompressor, including circuitry to:
 - read elements to be decompressed according to a compression scheme;
 - parse the elements to identify literals and matches; and
 - with the literals and matches, generate an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data;
- wherein the intermediate token stream is to include:
- a first format for multiple tokens that are to be written in parallel with each other; and
 - a second format for tokens that include a data dependency upon themselves.
9. The system of Claim 8, wherein the decompressor further includes circuitry to generate the second format with an address.
10. The system of Claim 8, wherein:
- multiple tokens that are to be written in parallel with each other are to be included in N lanes; and
 - the decompressor further includes circuitry to:
 - generate the second format with an address;
 - determine that a data dependency exists within the elements to be decompressed; and
 - based upon the determination that the data dependency exists within the elements to be decompressed, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed.

11. The system of Claim 8, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes; and
the decompressor further includes circuitry to:
generate the second format with an address;
determine that a data dependency exists within the elements to be decompressed;
based upon the determination of the data dependency, rewrite oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed; and
write a token that caused the data dependency in the first format.

12. The system of Claim 8, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format;
the decompressor further includes circuitry to:
determine that a data dependency exists within the elements to be decompressed;
based upon the determination of the data dependency, populate the N lanes with less than N tokens; and
copy a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency.

13. The system of Claim 8, wherein the decompressor further includes circuitry to insert a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression.

14. The system of Claim 8, wherein the decompressor further includes circuitry to store literal data into the first format.

15. A method comprising, within a processor:
reading elements to be decompressed according to a compression scheme;
parsing the elements to identify literals and matches; and
with the literals and matches, generating an intermediate token stream formatted for software-based copying of the literals and matches to produce decompressed data;
wherein the intermediate token stream is to include:

a first format for multiple tokens that are to be written in parallel with each other; and

a second format for tokens that include a data dependency upon themselves.

16. The method of Claim 15, further comprising generating the second format with an address.

17. The method of Claim 15, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes; and

further comprising:

generating the second format with an address;

determining that a data dependency exists within the elements to be decompressed; and

based upon the determination that the data dependency exists within the elements to be decompressed, rewriting oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed.

18. The method of Claim 15, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes and written according to the first format; and

the method further comprises:

determining that a data dependency exists within the elements to be decompressed;

based upon the determination of the data dependency, populating the N lanes with less than N tokens; and

copying a valid address from a populated token into an unpopulated lane based upon the determination of the data dependency.

19. The method of Claim 15, further comprising inserting a byte into a mask of the first format that includes a gather mask to identify which elements represented in the first format are to be copied upon decompression.

20. The method of Claim 15, further comprising storing literal data into the first format.

21. The method of Claim 15, wherein:
multiple tokens that are to be written in parallel with each other are to be included in N lanes; and

the method further includes:

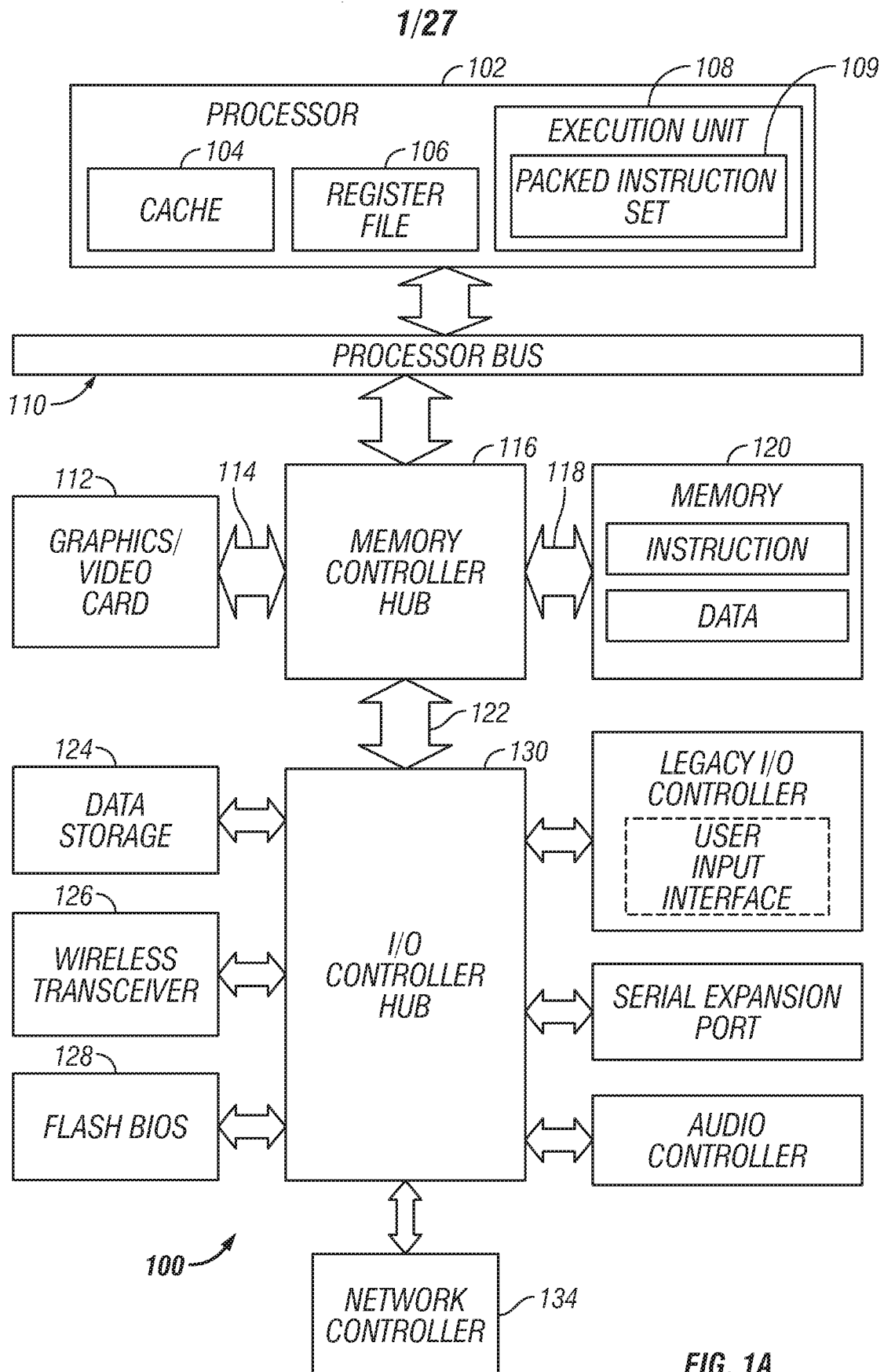
generating the second format with an address;

determining that a data dependency exists within the elements to be decompressed;

based upon the determination of the data dependency, rewriting oldest tokens from up to N unflushed tokens of the first format into the second format until the data dependency is removed; and

writing a token that cause the data dependency in the first format.

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

**FIG. 1A**

2/27

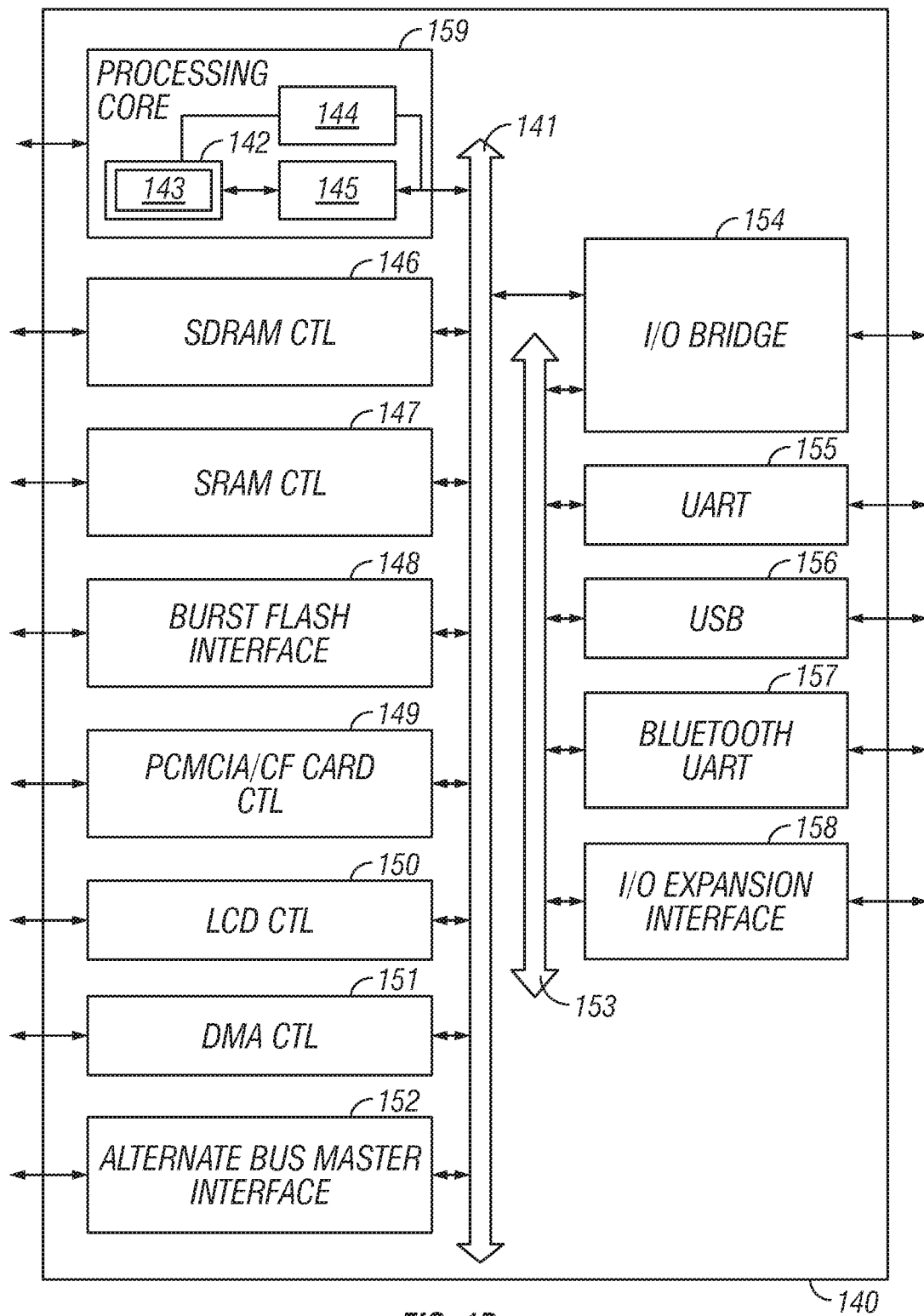


FIG. 1B

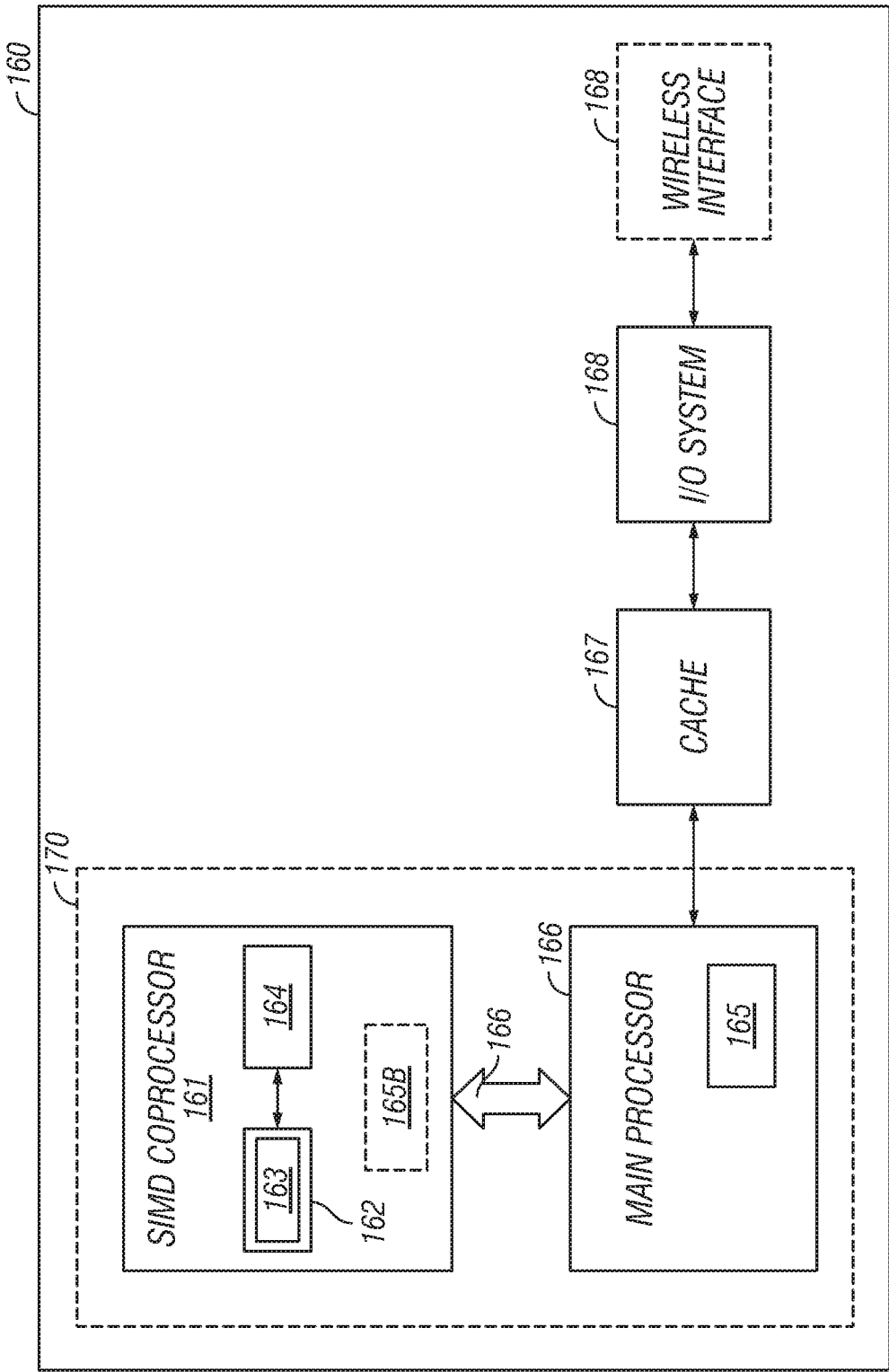


FIG. 1C

4/27

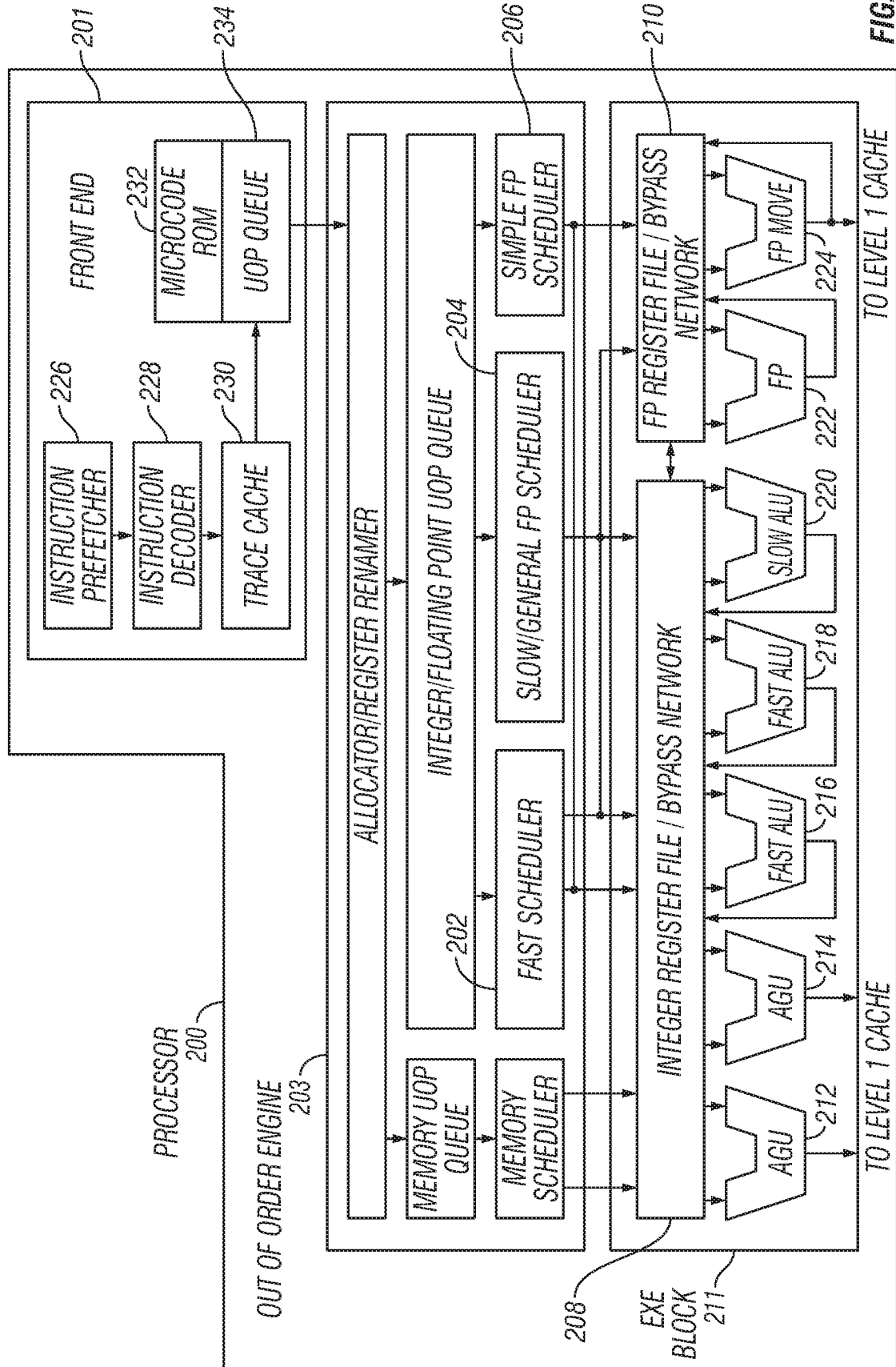


FIG. 2

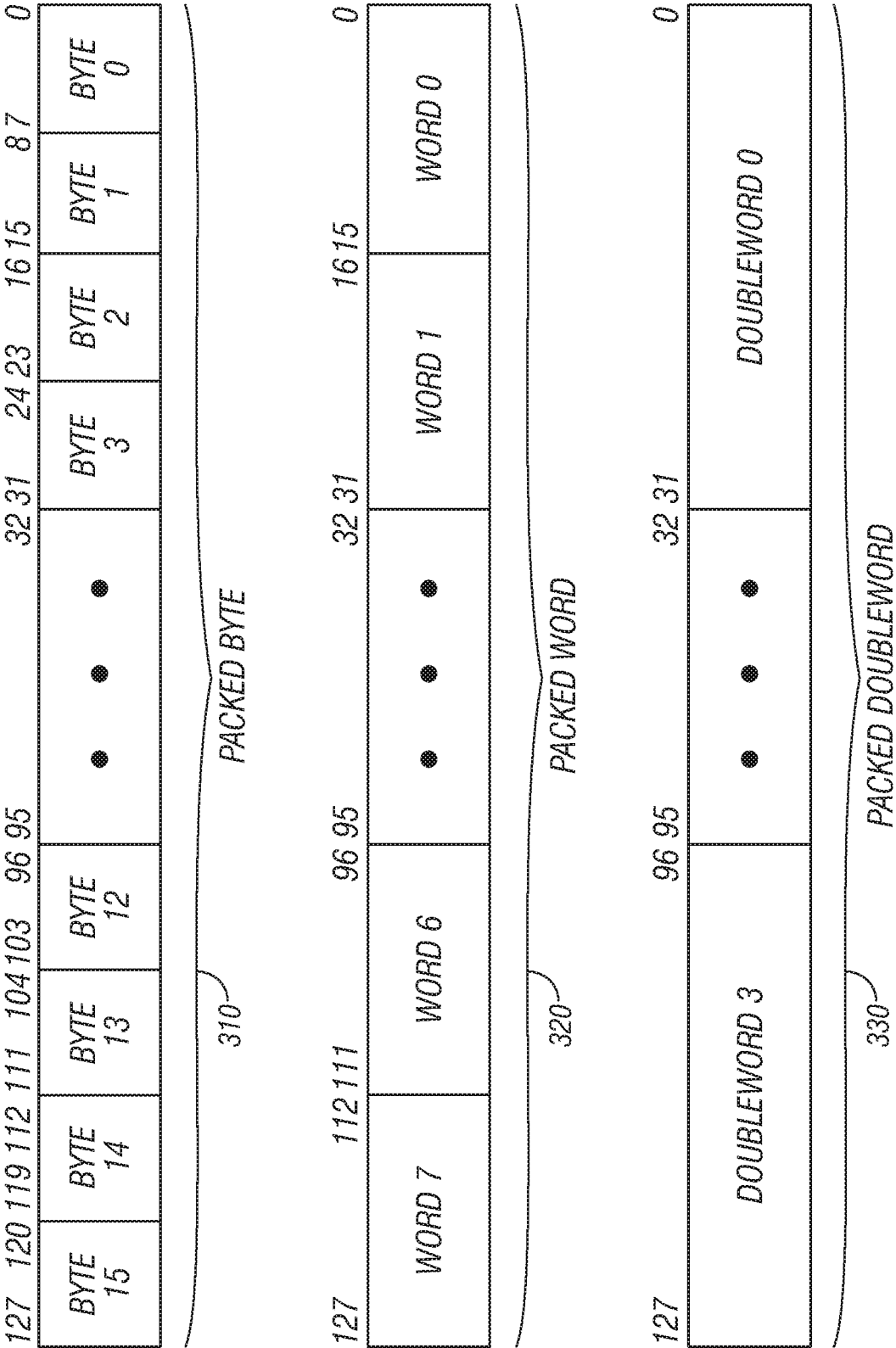
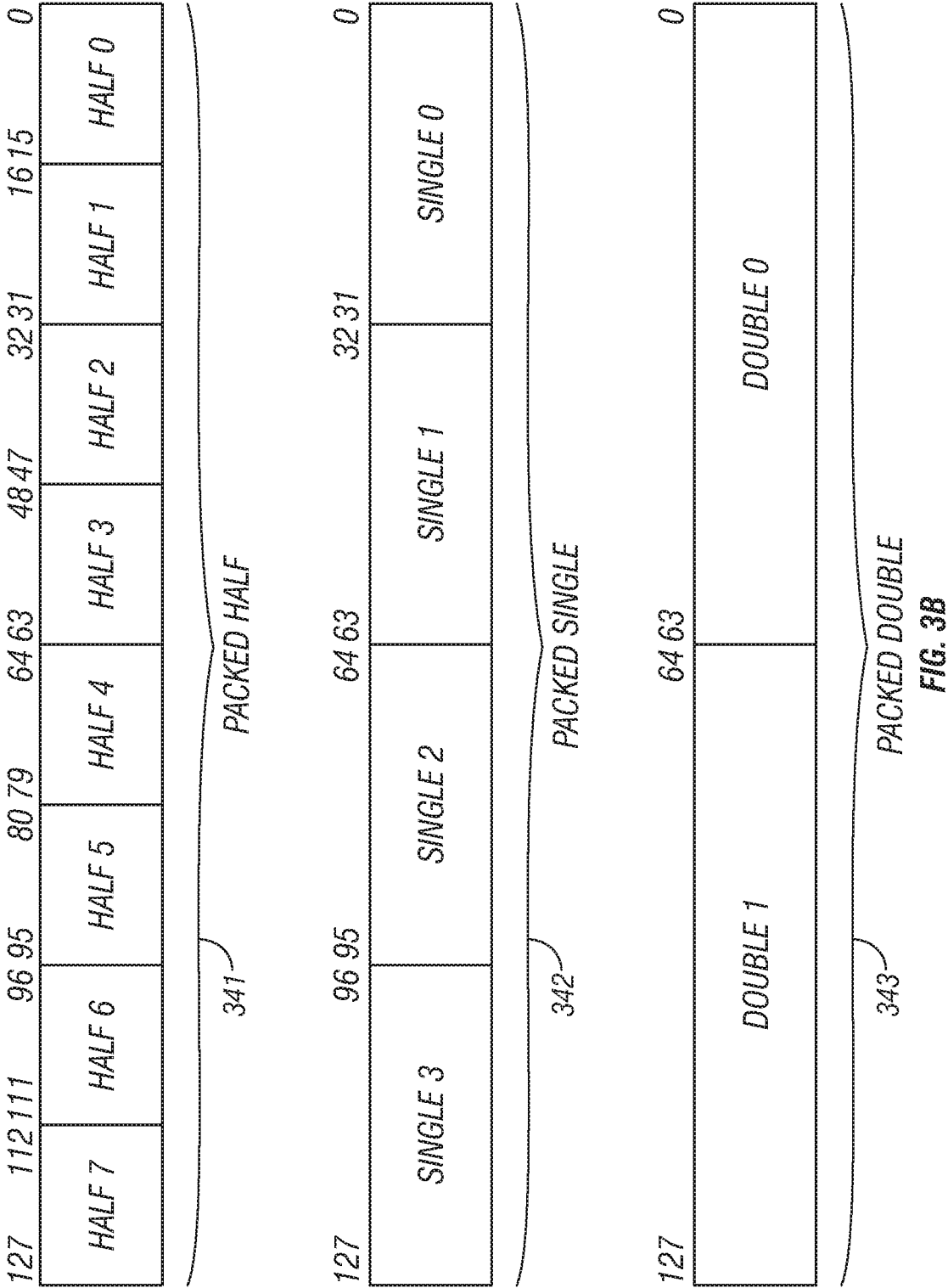
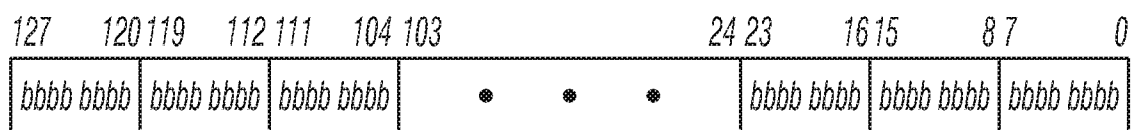
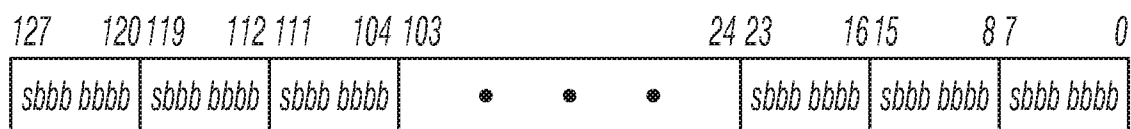


FIG. 3A

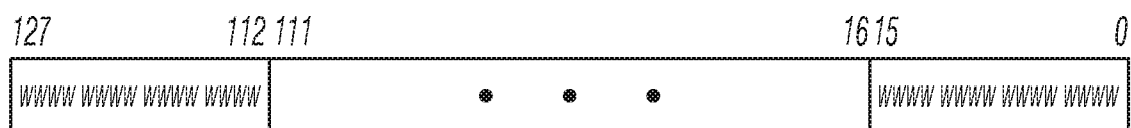


7/27

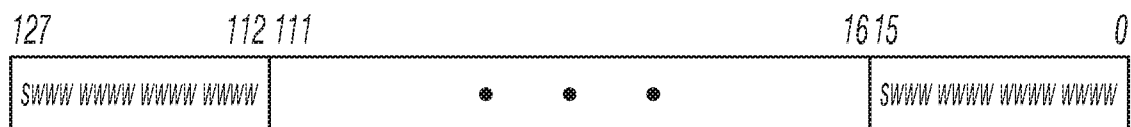
UNSIGNED PACKED BYTE REPRESENTATION 344



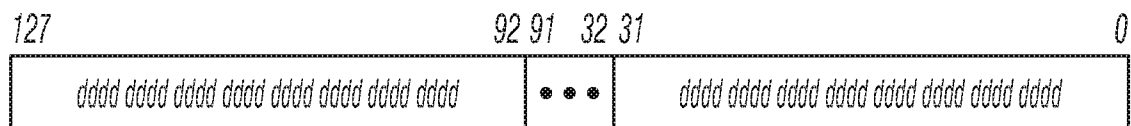
SIGNED PACKED BYTE REPRESENTATION 345



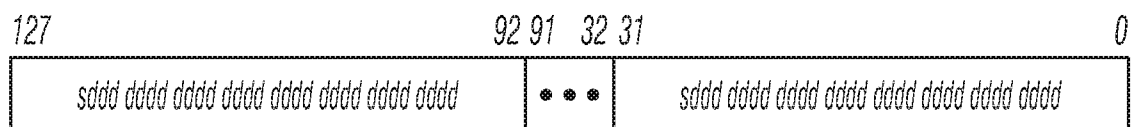
UNSIGNED PACKED WORD REPRESENTATION 346



SIGNED PACKED WORD REPRESENTATION 347

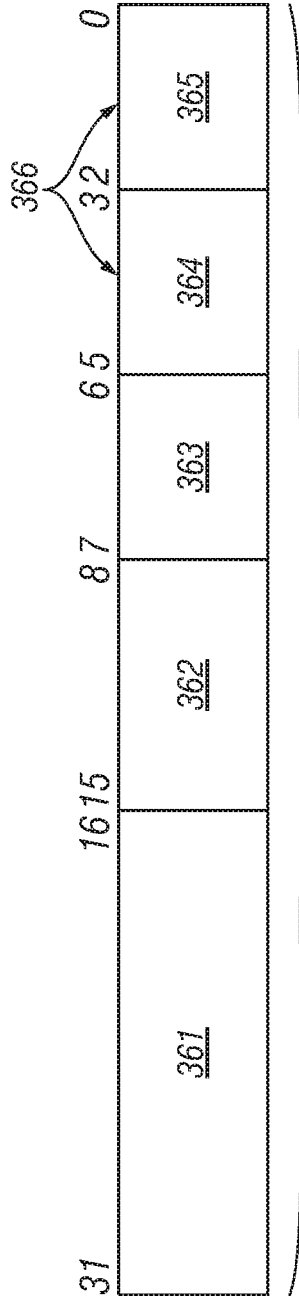


UNSIGNED PACKED DOUBLEWORD REPRESENTATION 348



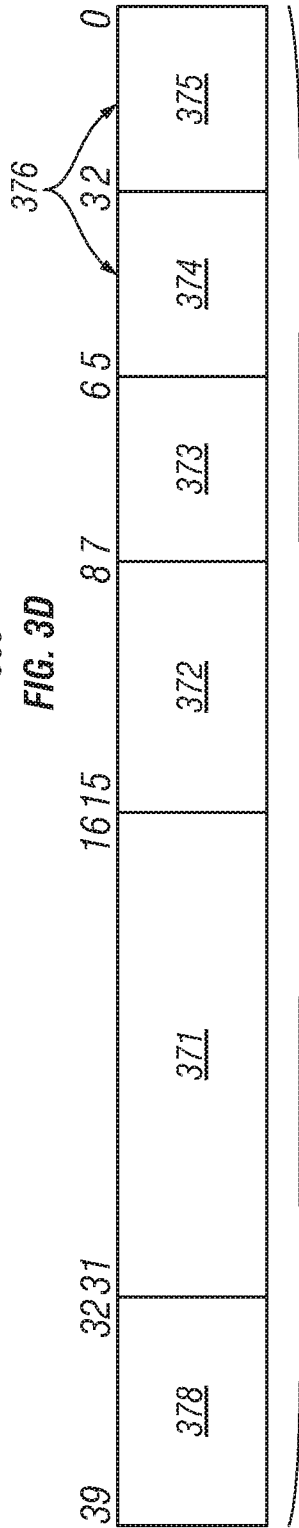
SIGNED PACKED DOUBLEWORD REPRESENTATION 349

FIG. 3C



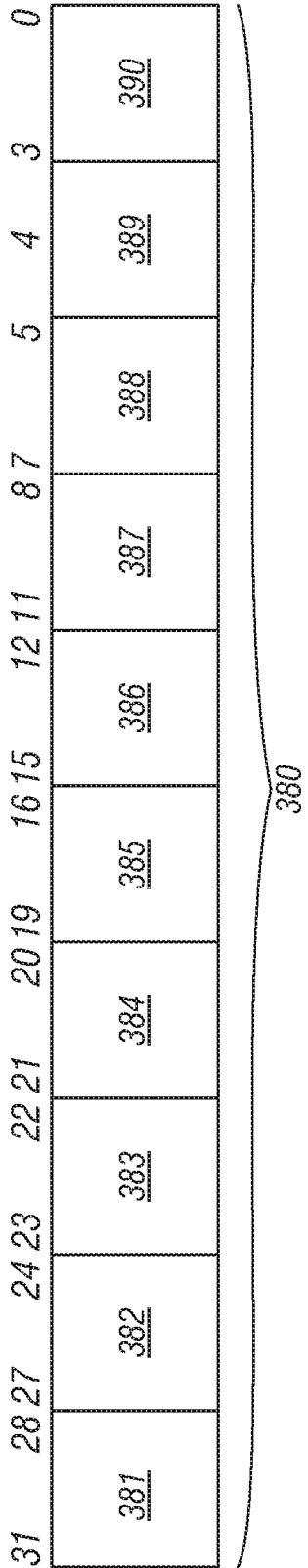
360

FIG. 3D



370

FIG. 3E



380

FIG. 3F

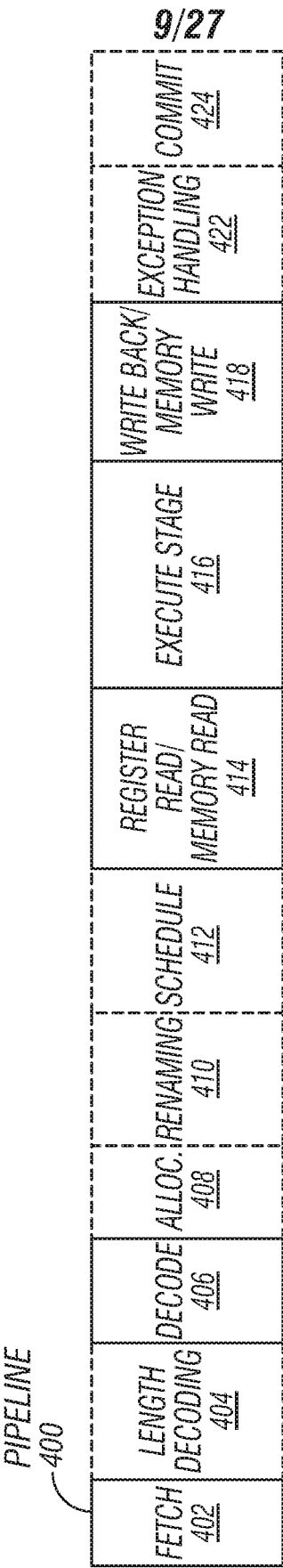


FIG. 4A

10/27

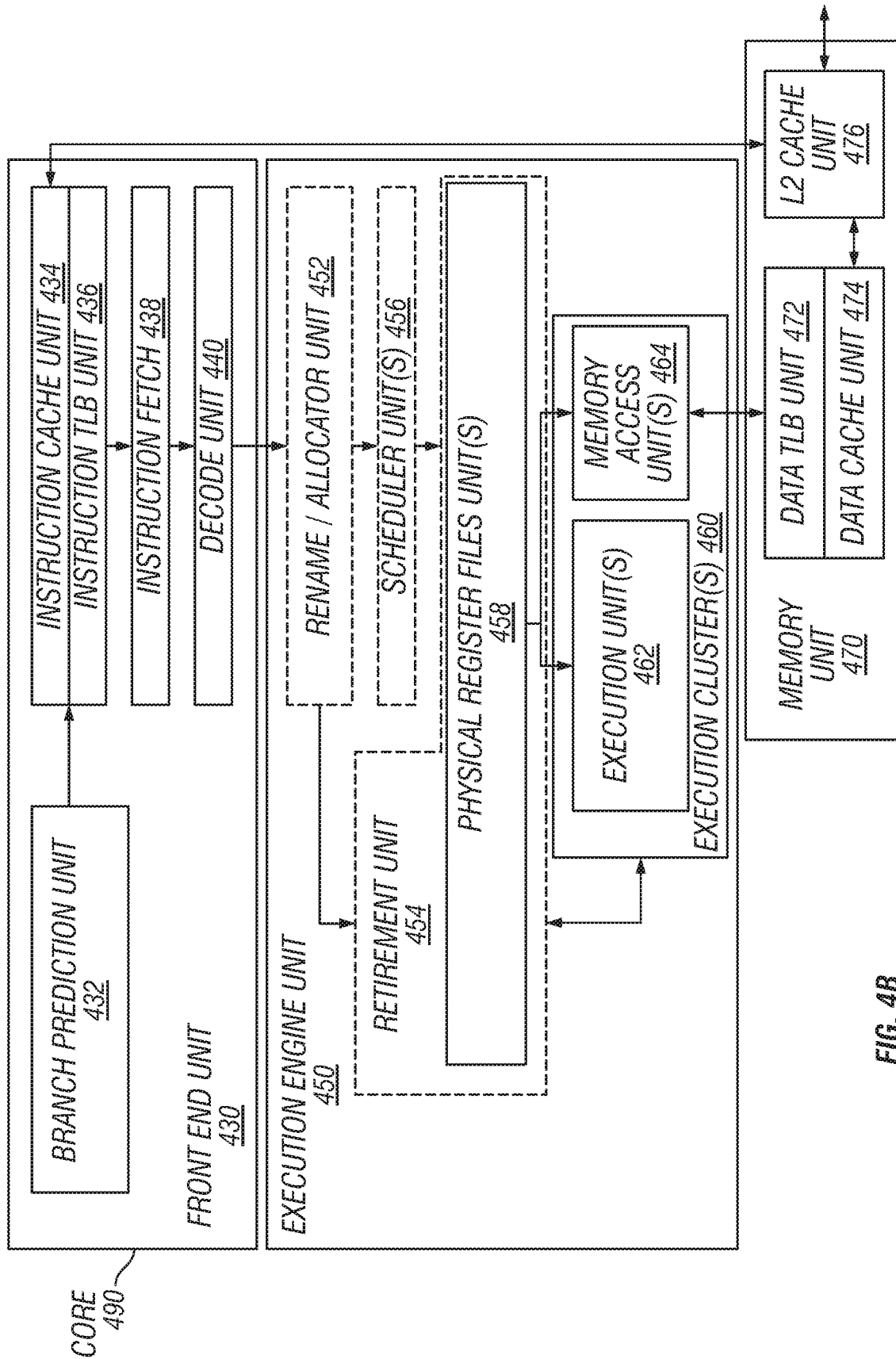


FIG. 4B

11/27

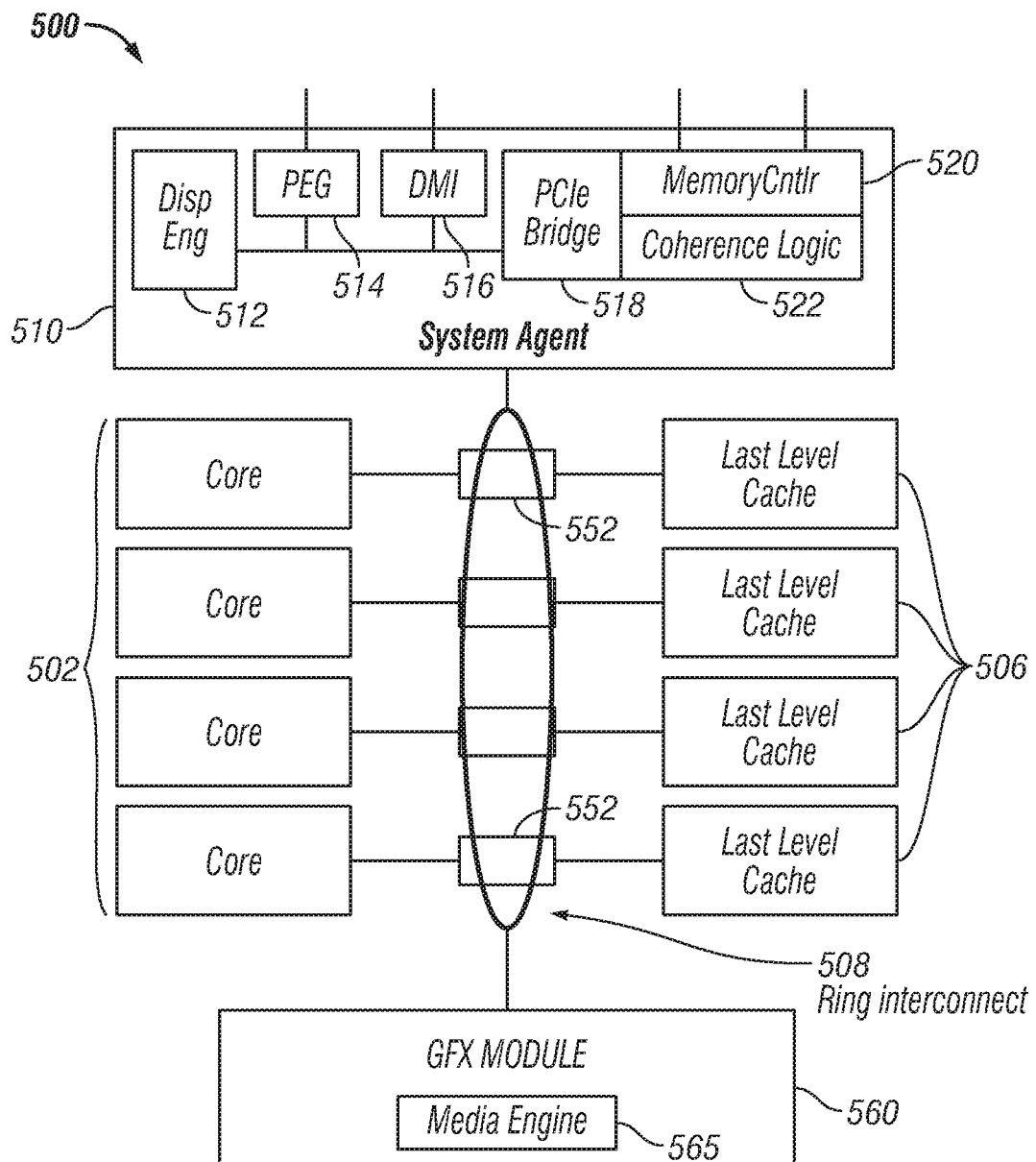


FIG. 5A

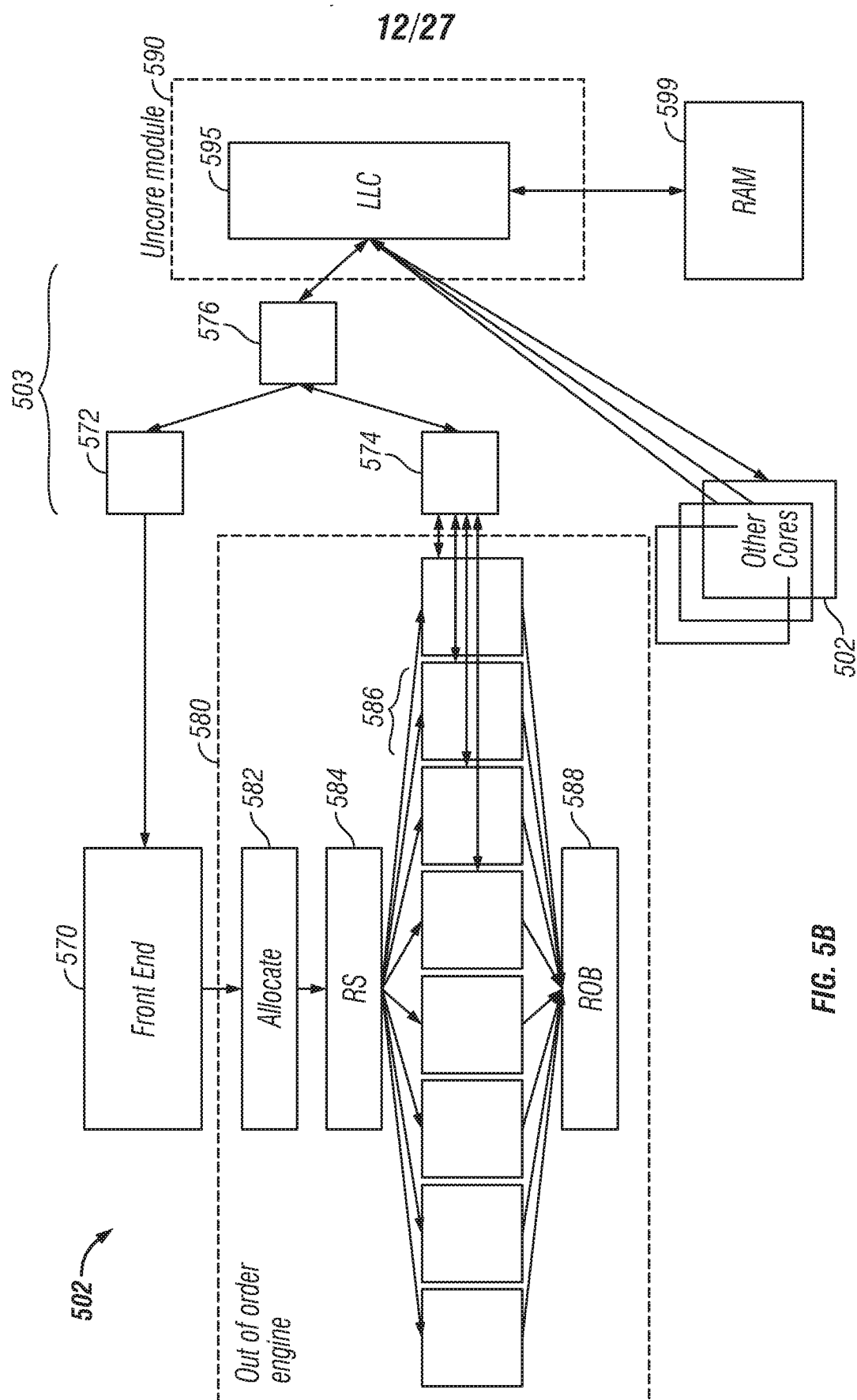


FIG. 5B

13/27

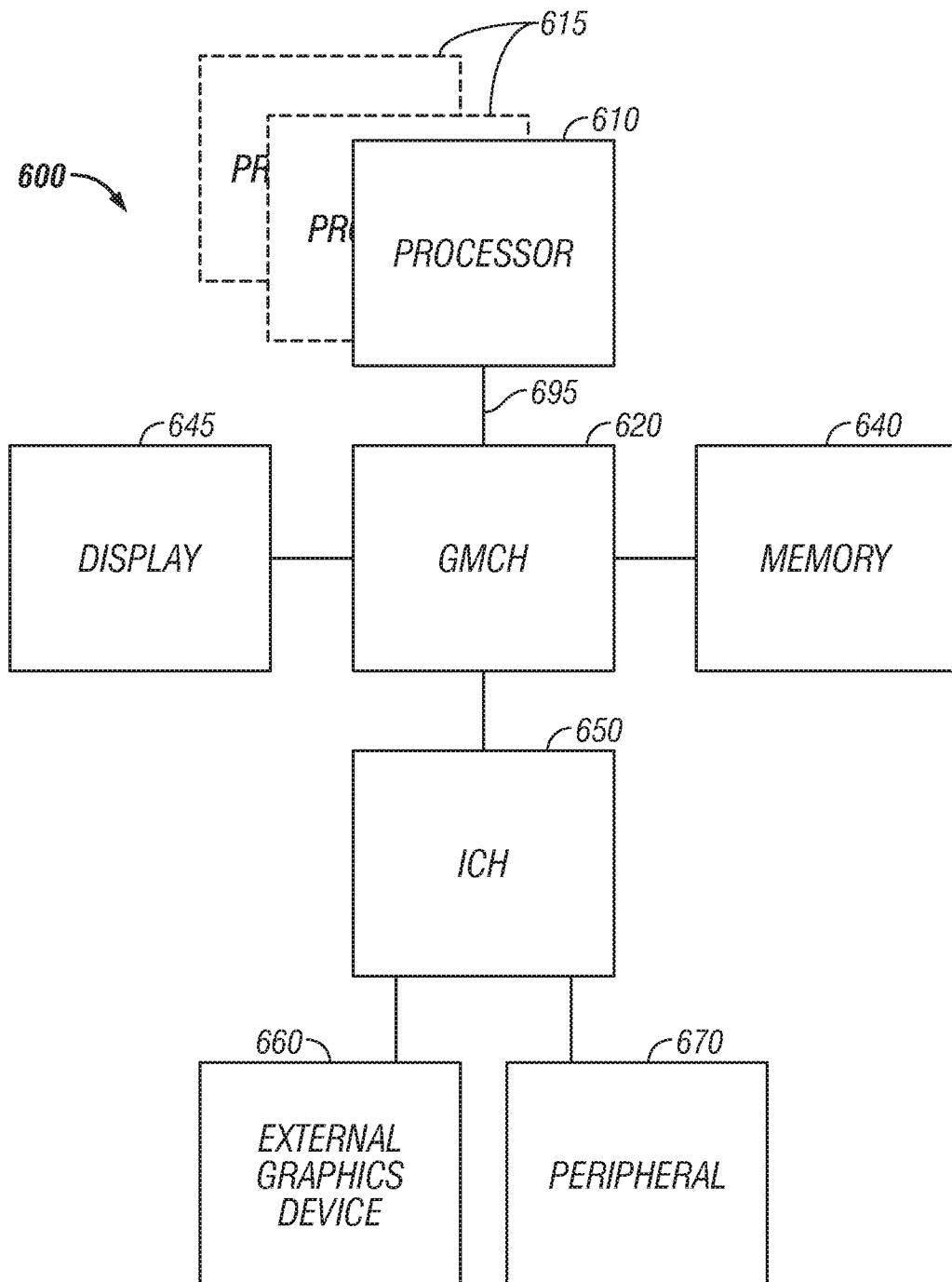


FIG. 6

14/27

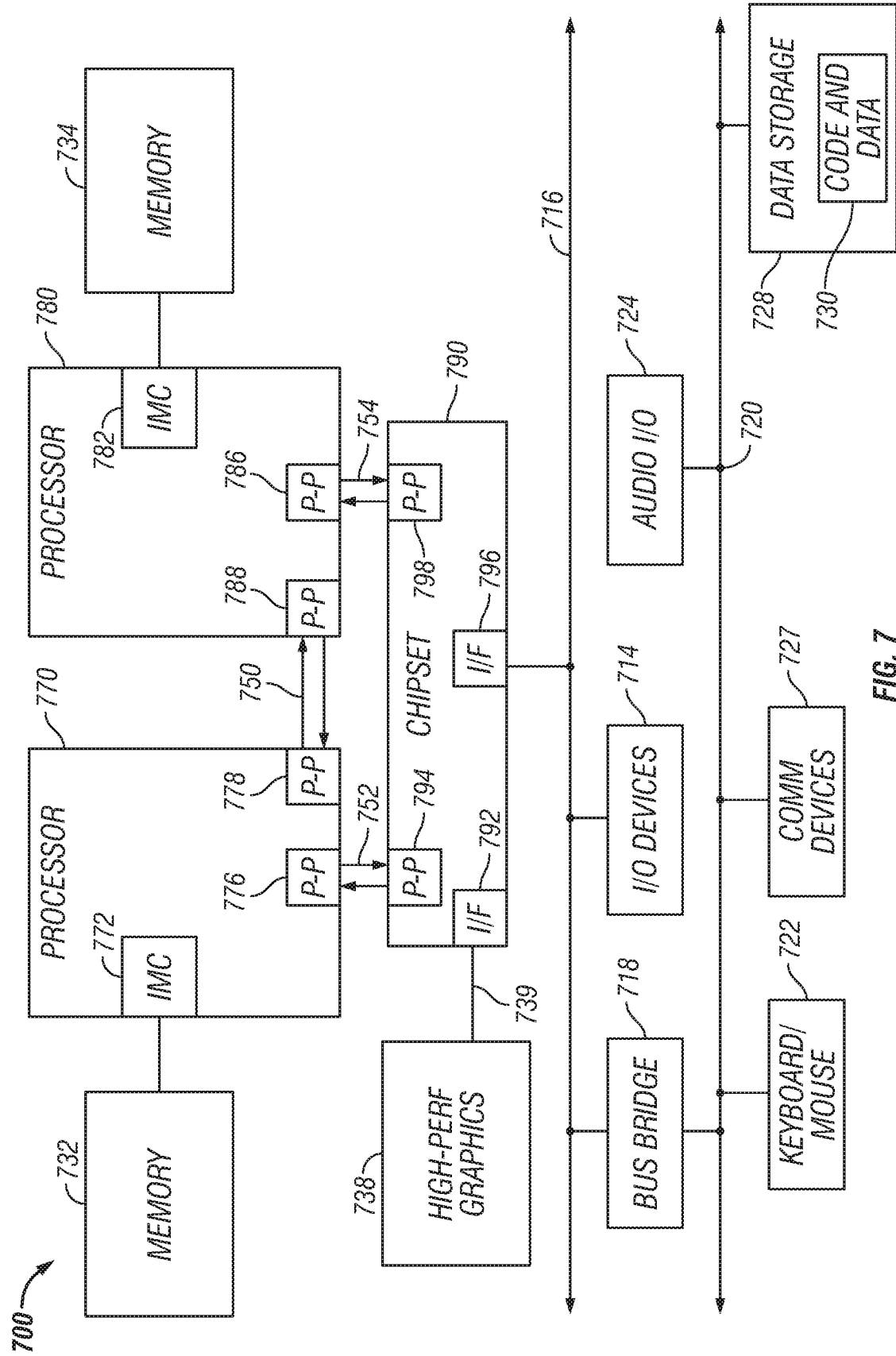


FIG. 7

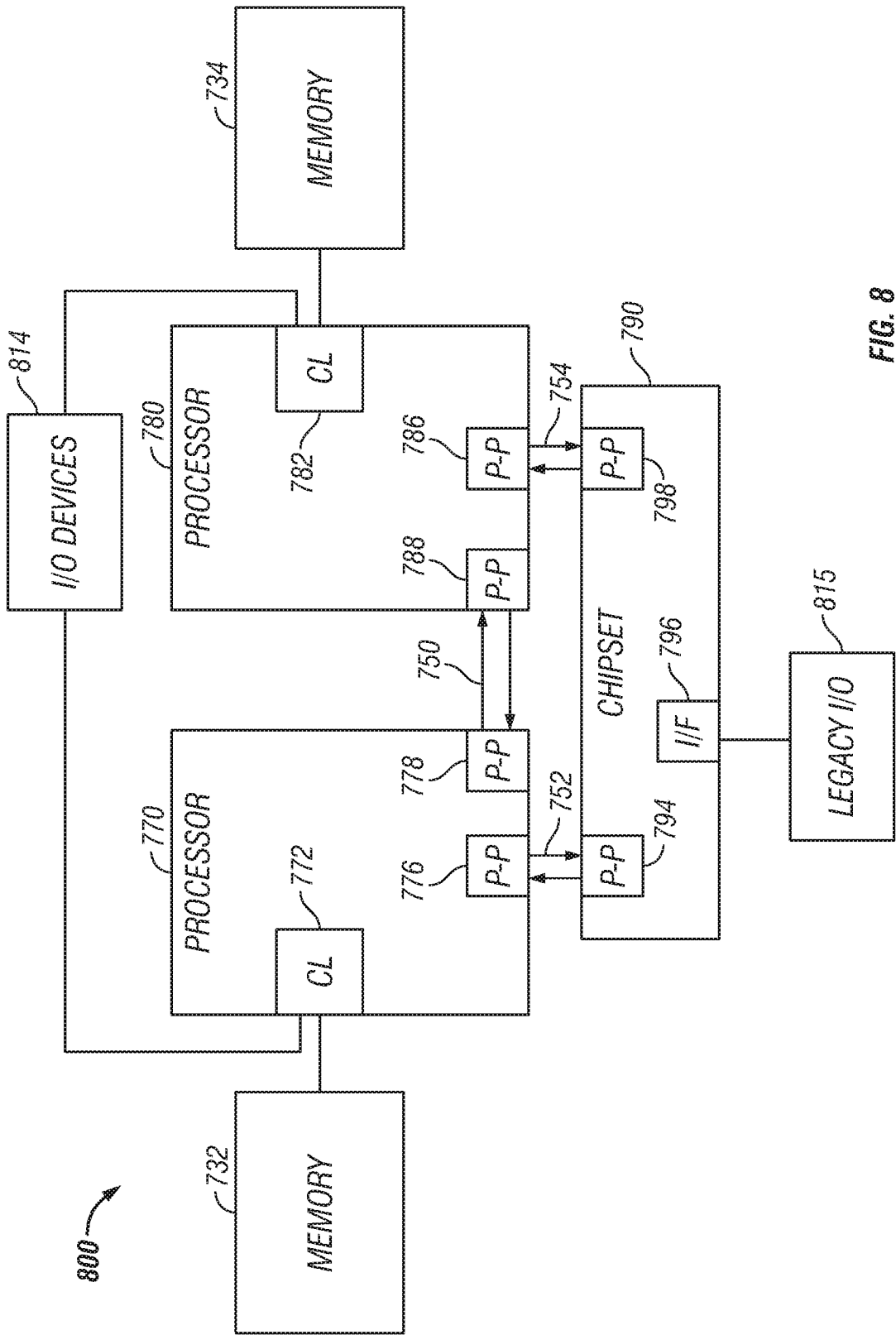


FIG. 8

16/27

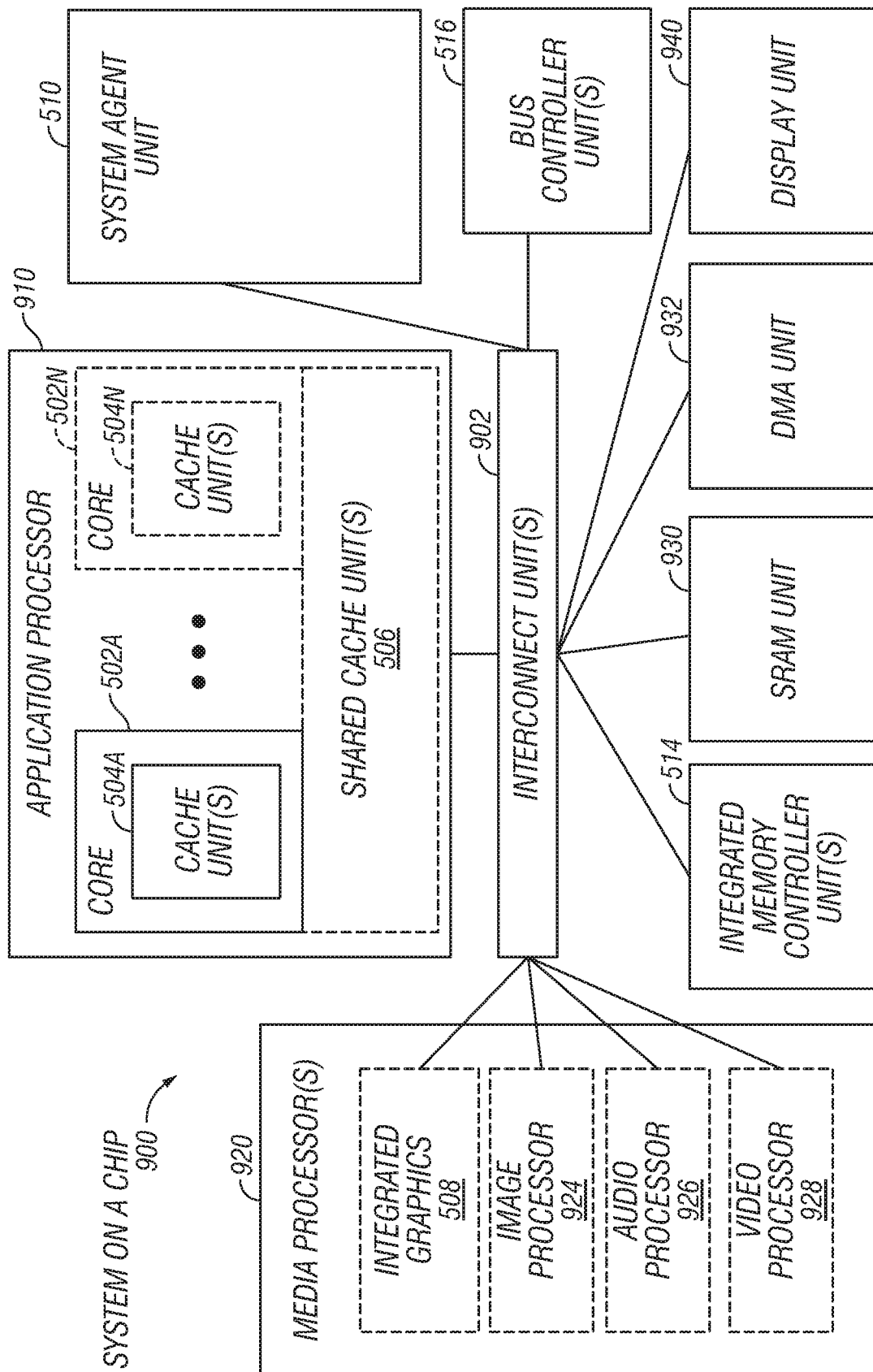


FIG. 9

17/27

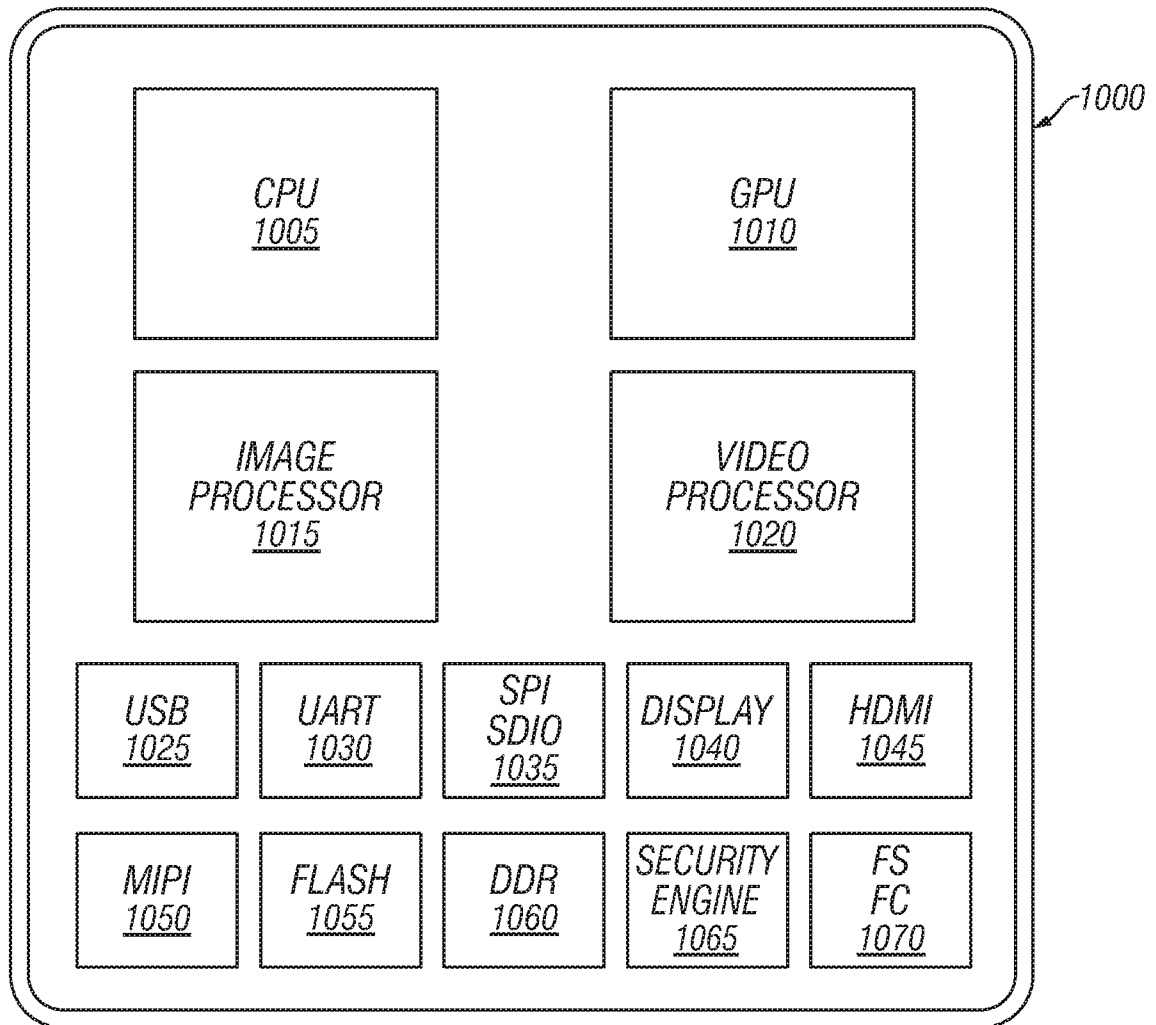


FIG. 10

18/27

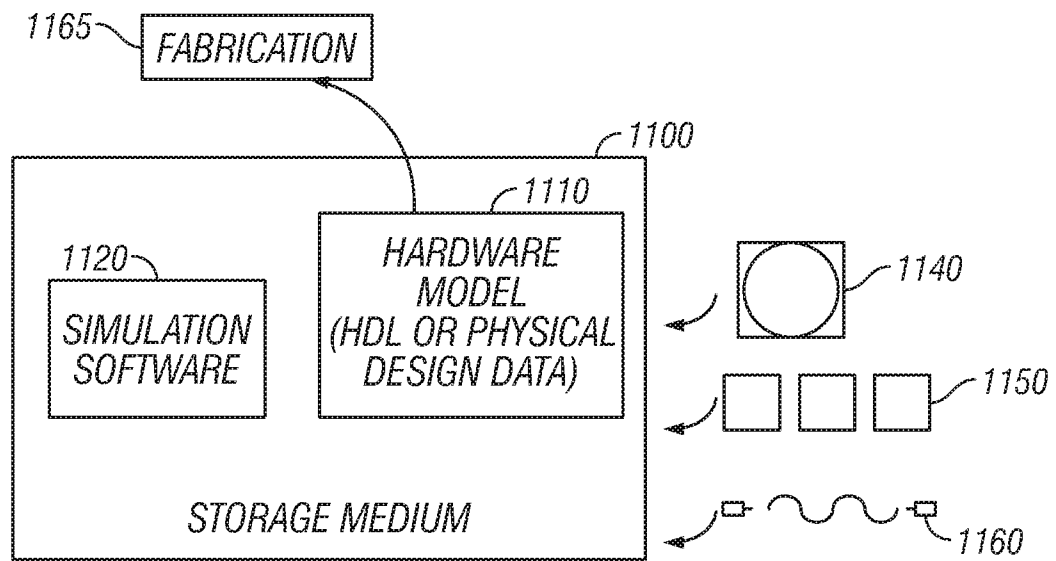


FIG. 11

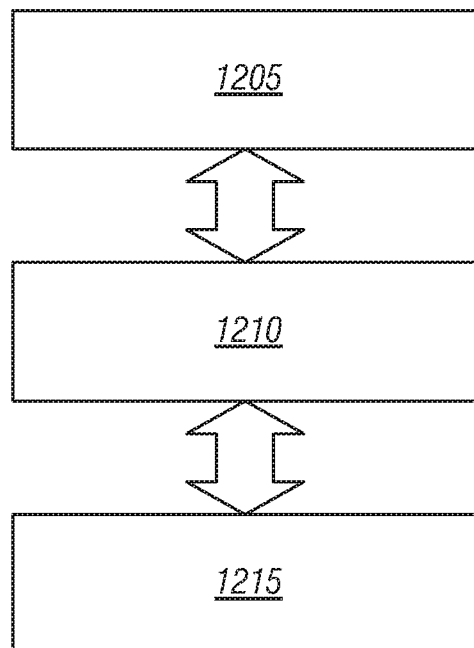


FIG. 12

19/27

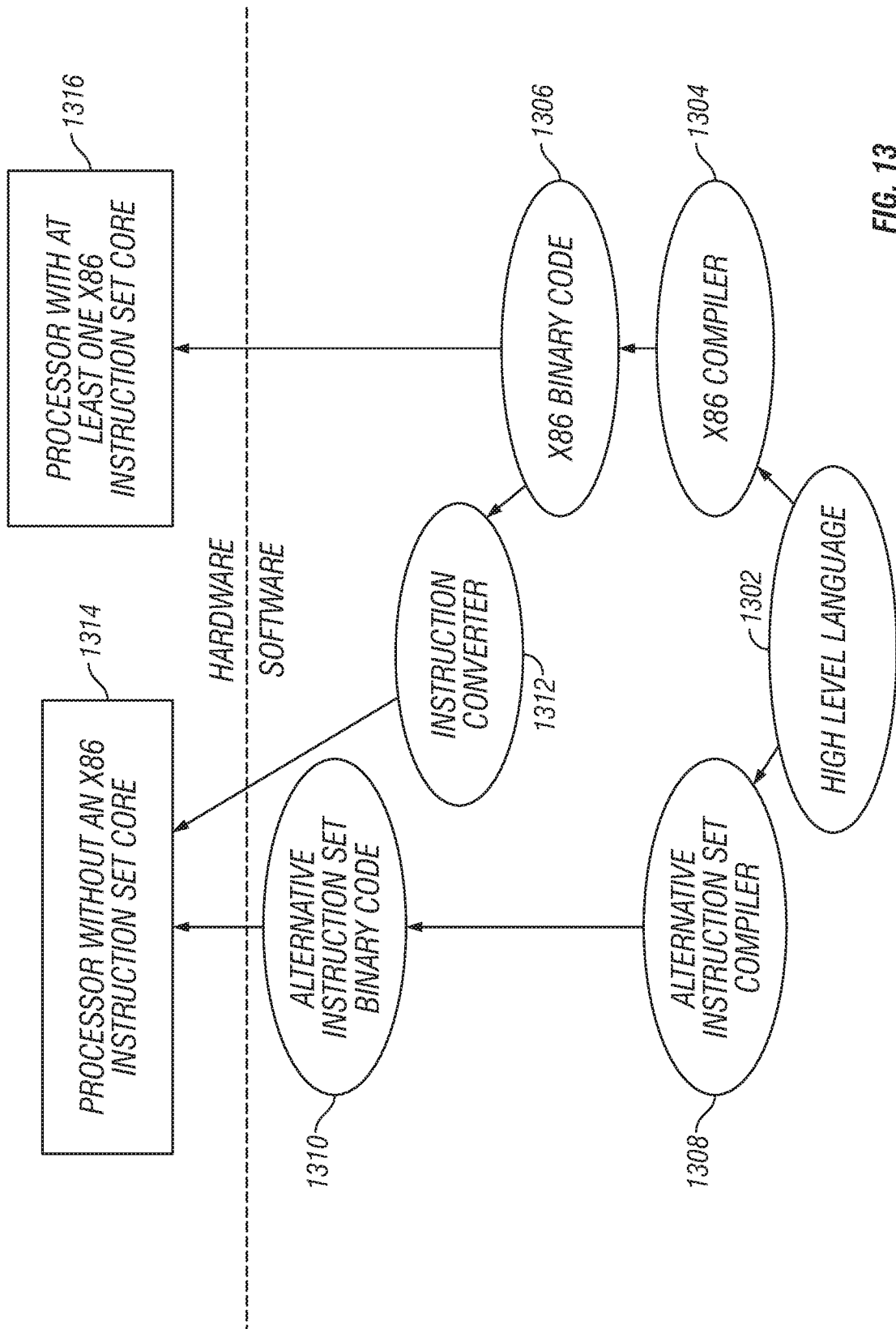


FIG. 13

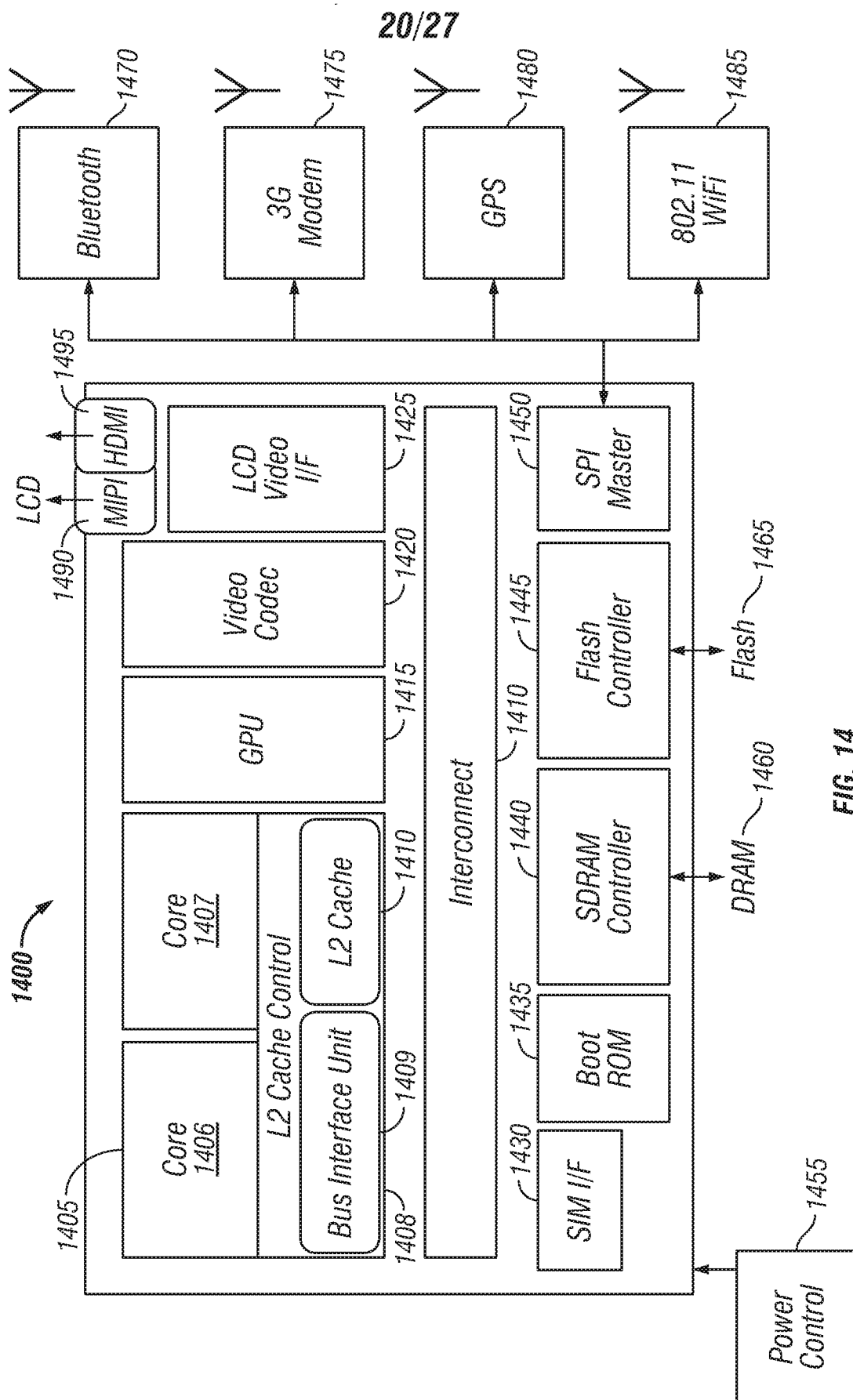


FIG. 14

21/27

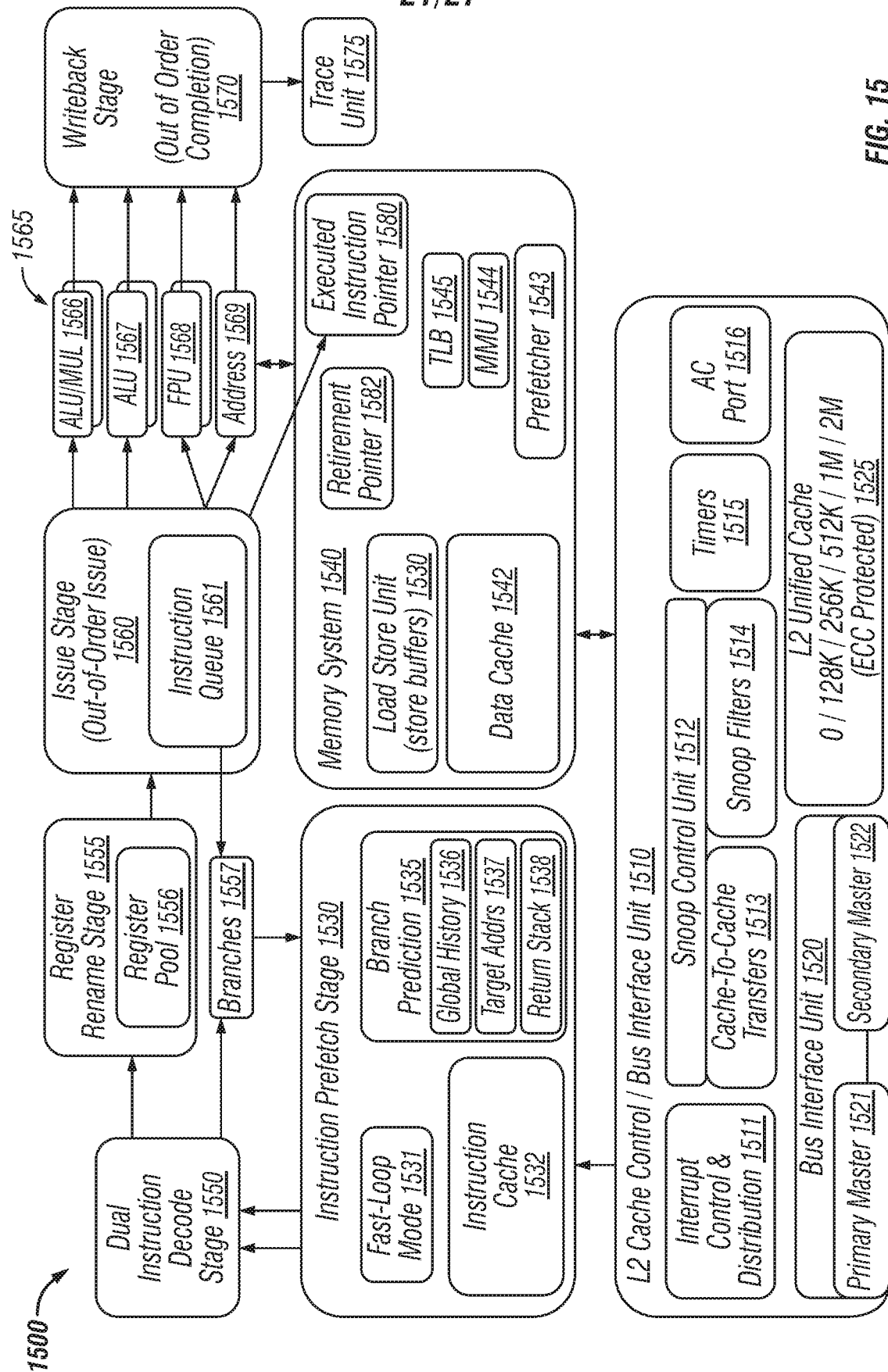


FIG. 15

22/27

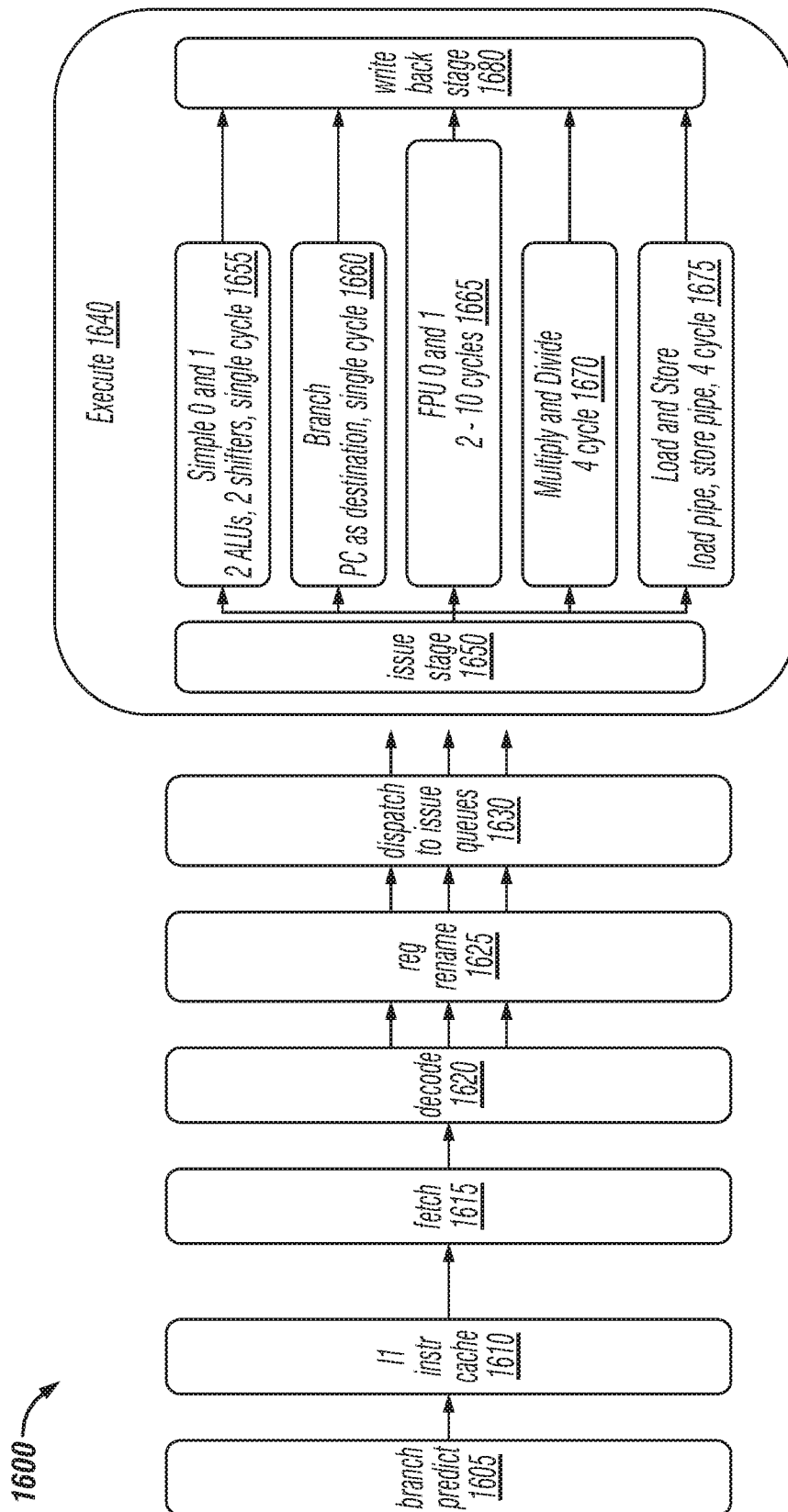
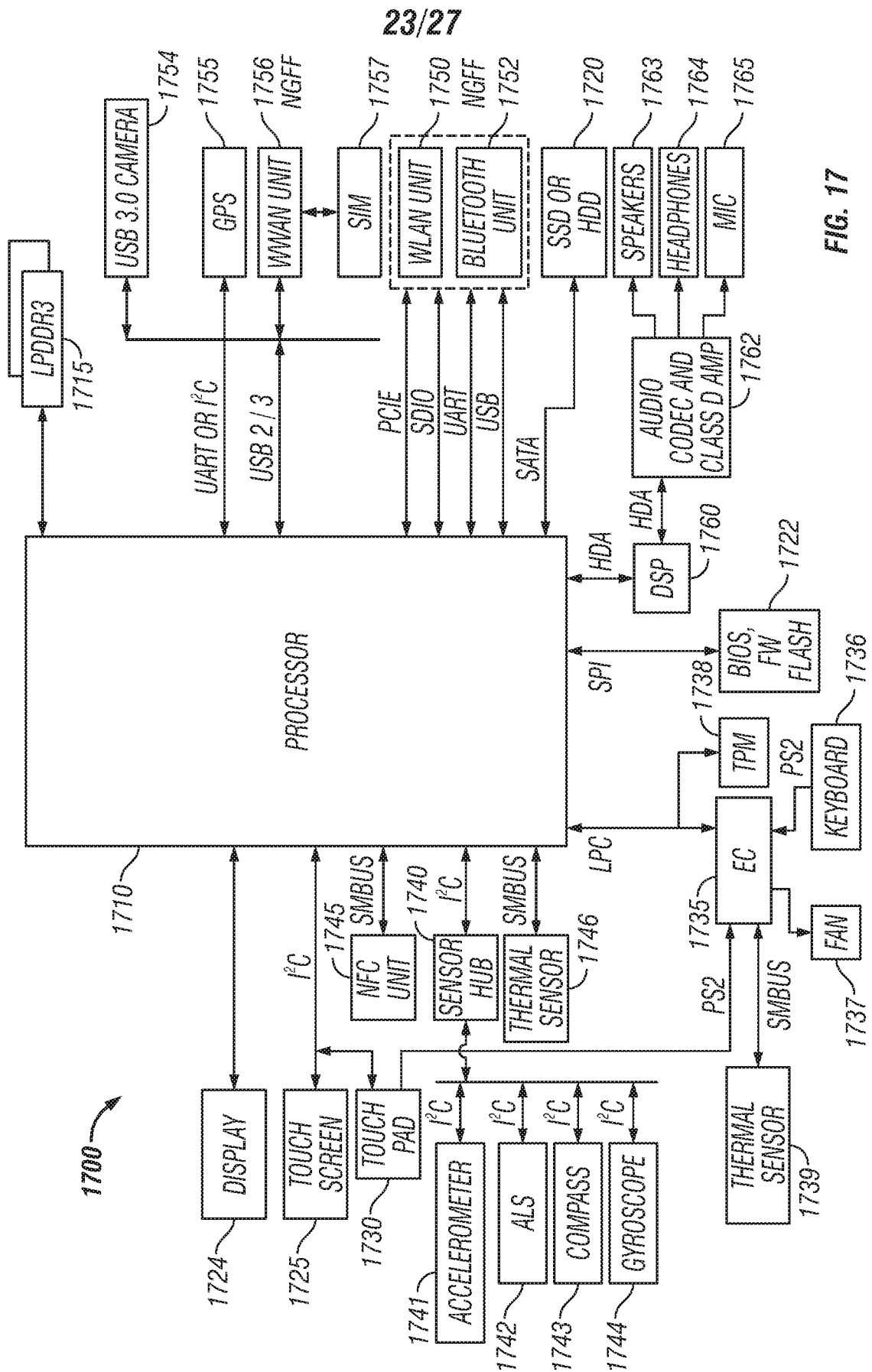


FIG. 16

**FIG. 17**

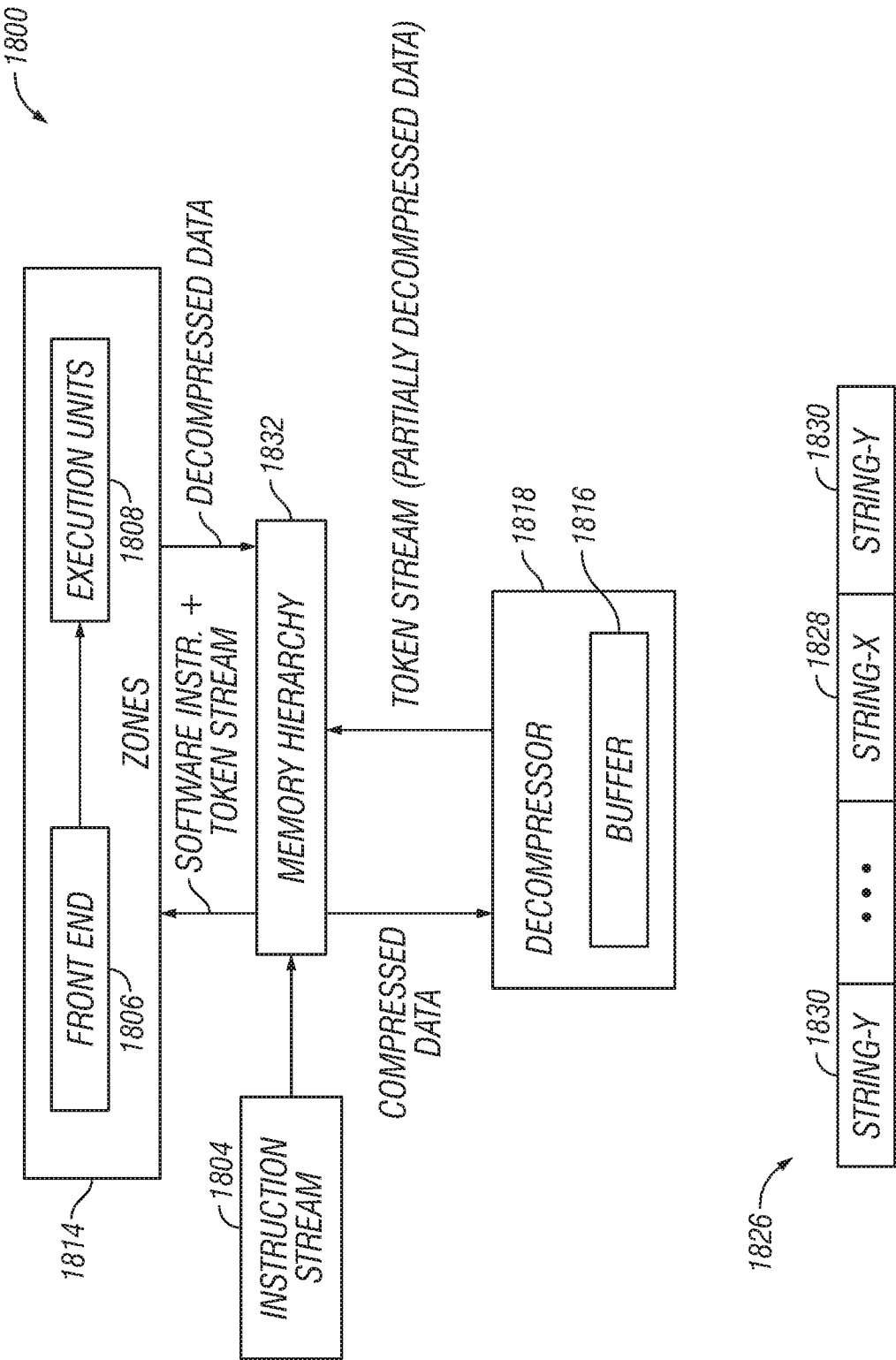


FIG. 18

25/27

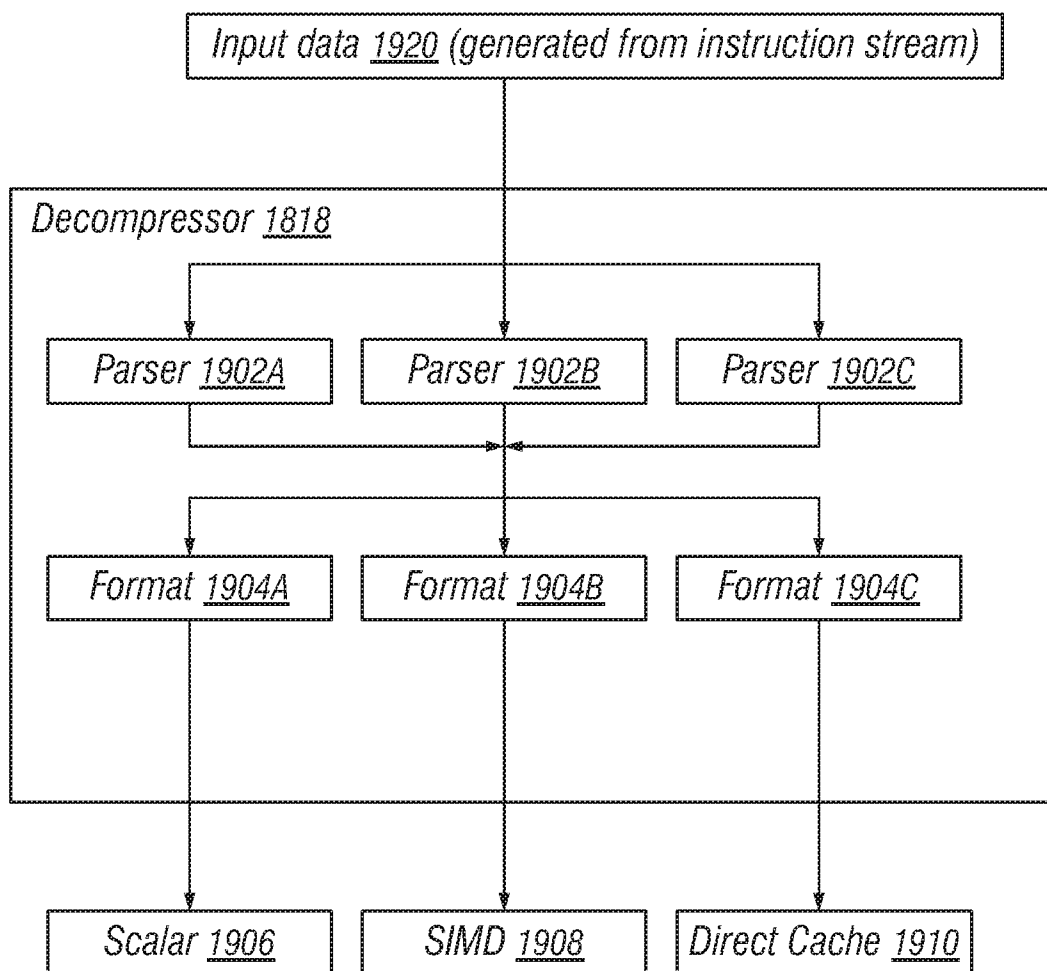


FIG. 19

26/27

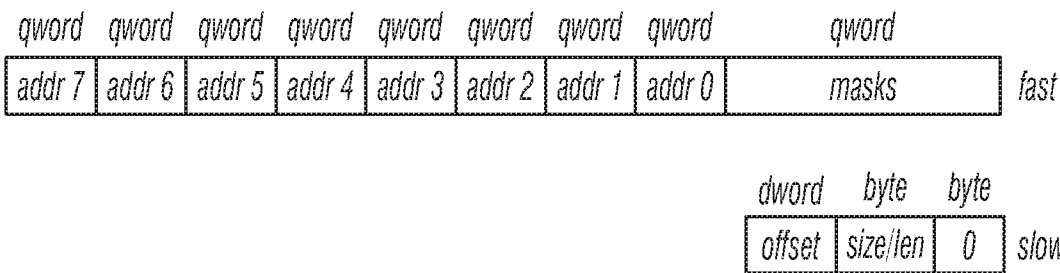


FIG. 20

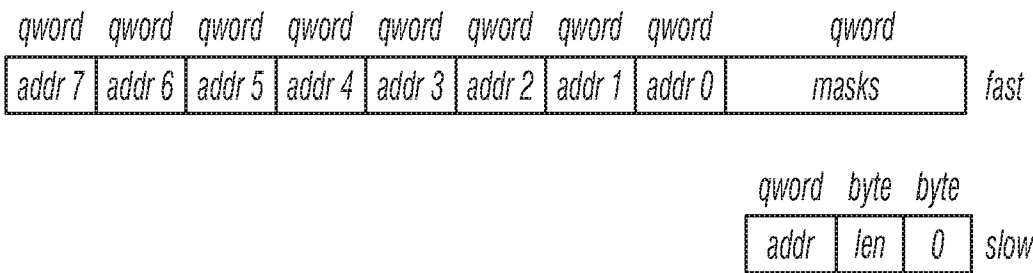


FIG. 21

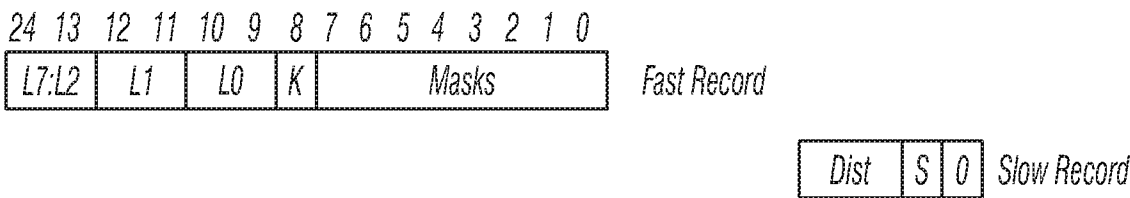


FIG. 22

C_mask	00	01	03	03	1F	07	01	03
G_mask	0	0	0	0	1	0	0	0
zmm	0 00	0 0H	0 GF	0 ED	0 02	0 01	0 0C	0 BA

FIG. 23

27/27

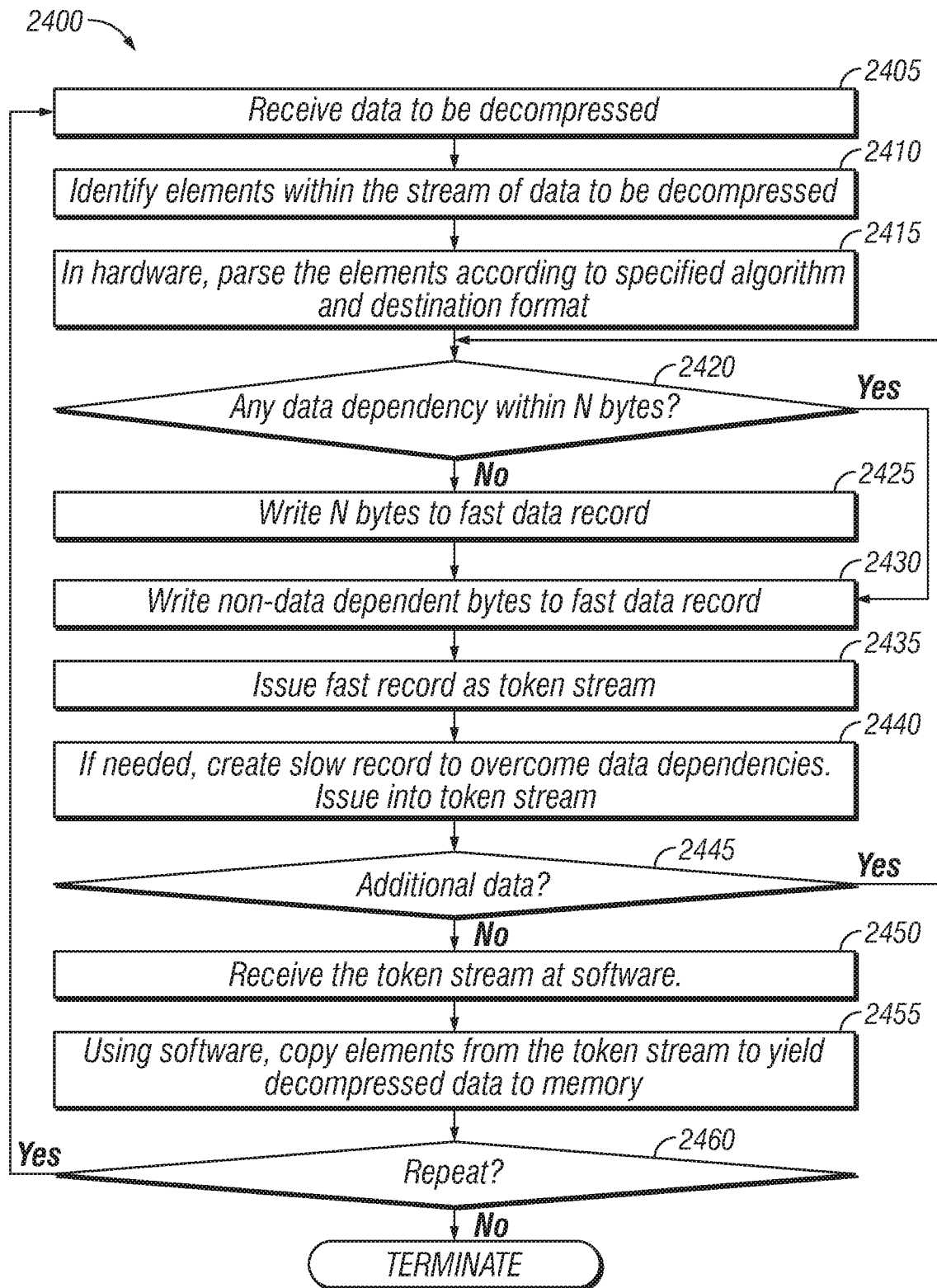


FIG. 24

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2017/035116**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/30(2006.01)i, G06F 13/16(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; H03M 7/30; H04N 7/167; G06F 12/00; G06F 13/16

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), Google Search Engine & Keywords: memory hierarchy, buffer, decompressor, parse, literals, matches, intermediate token stream, format, parallel, dependency

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	EVANGELIA SITARIDI et al., 'Massively-Parallel Lossless Data Decompression', Distributed, Parallel, and Cluster Computing, 02 June 2016, pages 1-10. <URL: https://arxiv.org/abs/1606.00519 > See pages 1-6; and figures 6-8.	1-22
A	US 2002-0091905 A1 (PETER D. GEIGER et al.) 11 July 2002 See paragraphs [0061]-[0103]; and figures 1-6.	1-22
A	US 2004-0264696 A1 (GORDON J. COCKBURN et al.) 30 December 2004 See paragraphs [0039]-[0067]; and figures 1-4.	1-22
A	US 2009-0077252 A1 (NADIM Y. ABDO et al.) 19 March 2009 See paragraphs [0044]-[0053]; and figures 3(A)-3(C).	1-22
A	US 5572206 A (JOHN W. MILLER et al.) 05 November 1996 See column 4, line 32 - column 9, line 7; and figures 2-4.	1-22



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

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

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

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

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

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

"&" document member of the same patent family

Date of the actual completion of the international search

23 August 2017 (23.08.2017)

Date of mailing of the international search report

25 August 2017 (25.08.2017)

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

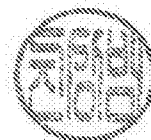


Facsimile No. +82-42-481-8578

Authorized officer

CHIN, Sang Bum

Telephone No. +82-42-481-8398



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/035116

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2002-0091905 A1	11/07/2002	AU 2747600 A	18/08/2000
		EP 1157470 A1	28/11/2001
		EP 1157470 B1	12/11/2003
		EP 1164706 A2	19/12/2001
		EP 1164706 A3	23/03/2005
		JP 04475820 B2	09/06/2010
		JP 2002-536863 A	29/10/2002
		US 2001-0038642 A1	08/11/2001
		US 2001-0054131 A1	20/12/2001
		US 2002-0010819 A1	24/01/2002
		US 2002-0101367 A1	01/08/2002
		US 2003-0058873 A1	27/03/2003
		US 2009-0125698 A1	14/05/2009
		US 2012-0212498 A1	23/08/2012
		US 5838334 A	17/11/1998
		US 5995120 A	30/11/1999
		US 6002411 A	14/12/1999
		US 6067098 A	23/05/2000
		US 6108014 A	22/08/2000
		US 6145069 A	07/11/2000
		US 6170047 B1	02/01/2001
		US 6173381 B1	09/01/2001
		US 6208273 B1	27/03/2001
		US 6370631 B1	09/04/2002
		US 6819271 B2	16/11/2004
		US 6822589 B1	23/11/2004
		US 6879266 B1	12/04/2005
		US 6885319 B2	26/04/2005
		US 7129860 B2	31/10/2006
		US 7190284 B1	13/03/2007
		US 7538694 B2	26/05/2009
		US 8176288 B2	08/05/2012
		US 8711164 B2	29/04/2014
		WO 00-45516 A1	03/08/2000
		WO 99-52093 A1	14/10/1999
US 2004-0264696 A1	30/12/2004	US 7500103 B2	03/03/2009
US 2009-0077252 A1	19/03/2009	CN 101803203 A	11/08/2010
		CN 101803203 B	16/10/2013
		EP 2198522 A2	23/06/2010
		EP 2198522 B1	23/10/2013
		JP 05566888 B2	06/08/2014
		JP 2010-539606 A	16/12/2010
		US 8819288 B2	26/08/2014
		WO 2009-036350 A2	19/03/2009
		WO 2009-036350 A3	07/05/2009
US 5572206 A	05/11/1996	EP 0691628 A2	10/01/1996

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2017/035116Patent document
cited in search reportPublication
datePatent family
member(s)Publication
date

JP 08-069370 A

12/03/1996