



(51) International Patent Classification:
G06F 9/38 (2006.01)

(21) International Application Number:
PCT/US2016/054072

(22) International Filing Date:
28 September 2016 (28.09.2016)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
14/953,644 30 November 2015 (30.11.2015) US

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

(72) Inventor: MATHEW, Jacob; 11th Floor, Discoverer,
ITPB, Whitefield, Bangalore, KA 560066 (IN).

(74) Agents: ROZMAN, Mark J. et al.; Trop, Pruner & Hu,
P.C., 1616 S. Voss Rd., Ste. 750, Houston, Texas 77057-
2631 (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, KN, KP, KR,
KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME,
MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ,
OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA,
SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM,
TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM,
ZW.

(84) Designated States (*unless otherwise indicated, for every
kind of regional protection available*): ARIPO (BW, GH,
GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ,
TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU,
TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE,
DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU,
LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK,
SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ,
GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

- *as to the identity of the inventor (Rule 4.17(i))*
- *as to applicant's entitlement to apply for and be granted a
patent (Rule 4.17(ii))*

[Continued on next page]

(54) Title: INSTRUCTION AND LOGIC FOR IN-ORDER HANDLING IN AN OUT-OF-ORDER PROCESSOR

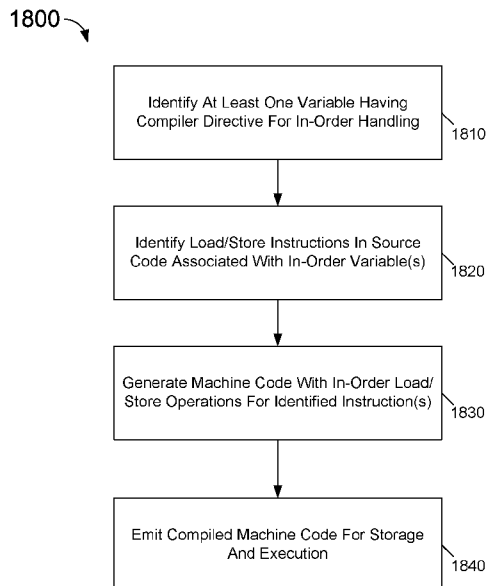


FIG. 18

(57) Abstract: In one embodiment, a processor includes a decode logic, an issue logic to issue decoded instructions, and at least one execution logic to execute issued instructions of a program. The at least one execution logic is to execute at least some instructions of the program out-of-order, and the decode logic is to decode and provide a first in-order memory instruction of the program to the issue logic. In turn, the issue logic is to order the first in-order memory instruction ahead of a second in-order memory instruction of the program. Other embodiments are described and claimed.



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

Published:

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

INSTRUCTION AND LOGIC FOR IN-ORDER HANDLING IN AN OUT-OF-ORDER PROCESSOR

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.

Background

[0002] Out-of-order execution of instructions within a processor occurs for certain compute intensive tasks like signal processing. However, in-order execution of memory mapped input/output (MMIO) accesses is required to guarantee correct execution in embedded control applications. To effect such operation, a programmer introduces special (e.g., fence/barrier) instructions to ensure in-order execution. However, this technique is error prone, and makes high-level code less portable across instruction set architectures, less readable, and less re-useable.

[0003] Historically, embedded systems such as small control units included in industrial, automotive and other specialized environments were architected with an in-order processor architecture. As more compute complex activities are performed in embedded applications, out-of-order processing architectures are being introduced, which increases complexity and suffers from backwards compatibility issues with existing code bases.

Brief Description of the Drawings

[0004] FIG. 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] FIG. 1B illustrates a data processing system, in accordance with embodiments of the present disclosure.

[0006] FIG. 1C illustrates another embodiment of a data processing system to perform operations in accordance with embodiments of the present disclosure.

[0007] FIG. 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] FIG. 3A illustrates various packed data type representations in multimedia registers, in accordance with embodiments of the present disclosure.

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

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

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

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

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

[0014] FIG. 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] FIG. 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] FIG. 5A is a block diagram of a processor, in accordance with embodiments of the present disclosure.

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

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

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

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

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

[0022] FIG. 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] FIG. 11 is a block diagram illustrating the development of IP cores, in accordance with embodiments of the present disclosure.

[0024] FIG. 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] FIG. 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] FIG. 14 is a block diagram of an instruction set architecture of a processor, in accordance with embodiments of the present disclosure.

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

[0028] FIG. 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] FIG. 17 is a block diagram of an electronic device for utilizing a processor, in accordance with embodiments of the present disclosure.

[0030] FIG. 18 is a flow diagram of a method in accordance with an embodiment of the present invention.

[0031] FIG. 19 is a flow diagram of a method in accordance with another embodiment of the present invention.

[0032] FIG. 20 is a flow diagram of a method for compiler execution in accordance with an embodiment.

[0033] FIG. 21 is a flow diagram of a method for decoding instructions in accordance with an embodiment of the present invention.

[0034] FIG. 22 is a flow diagram of a method in accordance with an embodiment of the present invention.

[0035] FIG. 23 is a block diagram of a portion of a processor in accordance with an embodiment.

Detailed Description

[0036] Embodiments provide an extended set of load and store instructions to be executed strictly in-order. Such instructions may be encoded into machine language encodings that are different from out-of-order counterparts of such load/store instructions. In embodiments, a processor guarantees to execute and commit two such control instructions only in-order. Note that other instructions of the same program may be executed out-of-order.

[0037] To create such instructions in a program, a compiler may be configured to identify appropriate load/store instructions amenable to in-order execution (as may be required for correct program operation). As one example, memory mapped input/output (MMIO) variables used in programs such as device drivers can be identified with a compiler directive (e.g., a prefix in a variable declaration) that instructs the compiler to use in-order instructions for read/write access made to those variables. In embodiments, in-order load/store instructions, e.g., control instructions, may be given a higher priority by a scheduling entity when selecting from a ready-to-execute queue of a processor.

[0038] Embodiments may provide more efficient operation and programming than instructions such as fence/barrier instructions inserted manually by a programmer. Operation using instructions as described herein also may be more efficient than providing separate cores for control and compute tasks, and/or a separate in-order mode.

[0039] Control-oriented code like a device driver involves many MMIO transactions. MMIO transactions may execute in strict program order, since bitfields in two different registers might have to be written in a certain order to enable a particular hardware function. By providing in-order instructions as described herein, e.g., for MMIO transactions, load-load, store-store, or load-store dependency may only between these MMIO transactions. By enabling compiler insertion of ordered instructions in place of code written with out-of-order instructions, readability, robustness and portability of source code is realized.

[0040] Embodiments may be used in a variety of computing contexts including but not limited to embedded systems. An interface can be used to specify blocks of addresses that are to be accessed strictly in-order, to ensure correctness. As such, embodiments enable use of out-of-order processors for an embedded control domain.

[0041] Referring now to Table 1, shown is a sample high level declaration of a MMIO variable in accordance with an embodiment, in the C language.

Table 1

volatile mmio struct _sUart Uart_attribute_ ((section ("UART_REG")))

[0042] As illustrated in Table 1, this declaration provides a compiler prefix (referred to as MMIO) as a keyword to specify that this variable is a MMIO-mapped variable and thus is to be handled in-order as described herein. In the particular example above, understand that the declaration is to define that all registers related to a particular device driver (here a universal asynchronous receiver/transmitter (UART)) mapped to a given range of addresses are to be handled accordingly.

[0043] As an example, a text editor can be used to specify the keyword to the compiler or a high-level code generation tool can be instructed to treat a block of addresses as part of a peripheral device. As a result, this in turn generates a code similar to a programmer-included code. Note that this MMIO prefix is used here as an example; other terms can also be used based on keyword availability in compiler/application properties.

[0044] Assume a code sequence is to write information to given registers, e.g., transceiver registers (uart.data and uart.trigger registers), and then poll on the uart.status register. High level source code of such operation is illustrated in Table 2.

Table 2

```

#define UART_TX_TRIGGER (1 << 4)
#define UART_TX_COMPLETE 1
void uart_send_data (char data)
{
    int c;
    Uart.data = (unsigned int) data; /*Write the data to be transmitted */
    Uart.trigger = Uart.trigger | UART_TX_TRIGGER;
    while (Uart.status != UART_TX_COMPLETE);
    return;
}

```

[0045] As seen, this source code is to write data to be transmitted into a data register (Uart_data), perform a logical operation with a trigger register and transmit the data until the trigger condition is met. More specifically, the code enables a write to the Uart.data register with the data to be transmitted, followed by instructing the Uart to start the transmit by writing to the Uart.trigger register. Then the flow waits until the Uart finishes sending the data via a one-line While loop. Note that the hardware will set Uart status ==UART_TX_COMPLETE only after the transmit operation is complete. Note that the various variables associated with the Uart definition (Uart.data, Uart.trigger, and Uart.status) are thus associated with the compiler directive and as such, a compiler is to generate in-order load/store instructions for instructions referencing these source code variables as operands. This source code, when compiled on a given machine, may be translated into machine code, e.g., into assembly language.

[0046] Table 3 below shows translated code without providing in-order instructions as described herein.

Table 3

movsx	eax, BYTE PTR [esp+0x4]
mov	ds:0x0, eax → ds:0x0 -> Uart.data register
mov	eax, ds : 0x4
or	eax, 0x10
mov	ds : 0x4, eax → dx:0x4 -> Uart.trigger register
mov	eax, ds : 0x8 → ds:0x8 -> Uart.status register
cmp	eax, 0x1
jne	17 < _uart_send_data+0x17>
ret	
nop	
nop	

[0047] An out-of-order processor can see independent threads of execution in this code, as there are no data dependencies between them. And thus there are no data hazards from the processor point of view. But in reality, the code is not independent due to sequencing requirement imposed by Uart hardware.

[0048] Embodiments provide instruction set architecture instructions to perform in-order load/store operations. Example instruction encoding includes a so-called emov instruction. In one embodiment, this instruction may be a user-level instruction to be used by a programmer to specify in-order execution of a load/store instruction. In other embodiments, the instruction may be a compiler-generated instruction (such as a machine code instruction), generated responsive to an unordered user level load/store instruction of source code. More generally, these instructions may be referred to as control instructions to denote the in-order control of such instructions. The implementation can be done with microcode in a microcode-based system, or with dedicated hardware in processors with hardware decoding.

[0049] Embodiments thus provide an out-of-order processor having an instruction set, out of which a group of instructions has an ISA level guarantee to execute and commit in order. Note that a source version of a program may provide the instructions in order. In a conventional out-of-order processor, an instruction decoder (and issue logic) may disregard this ordering. But using a processor in accordance with an embodiment, the decoder (and the issue logic) will not disregard this information for the in-order instructions described herein.

[0050] In one example, a control move instruction in accordance with an embodiment has the mnemonic, emove. An example of instruction encoding for this instruction is: emov EAX, [address]. With this encoding, the instruction may cause a move operation to move information at the identified address to the EAX register. Note that this instruction may be ordered with respect to a previous emov instruction, and certain previous load/stores (e.g., an unordered moves). That is, a control instruction in accordance with an embodiment of the present invention will inherit all the ordering rules of its standard counterpart. In addition, it will impose ordering with respect to a previous control instruction. In this case an emov instruction inherits all ordering rules of a mov instruction. In addition, it is strictly ordered with respect to previous control instructions. For example, assume a mov AX, 0xa5a5 followed by emov [0xE5010000], AX. In a processor in accordance with an embodiment of

the present invention, these two instructions may be ordered because the AX register is a common resource. This is an ordering rule inherited from a mov instruction.

[0051] Referring now to Table 4, shown is a generated assembly code in accordance with an embodiment that provides for in-order instructions. Now previously independent threads of execution are no longer independent from the processor view point, because the emov instruction ensures in-order execution among other emov instructions.

Table 4

movsx	eax, BYTE PTR [esp+0x4]
emov	ds:0x0, eax → ds:0x0 -> Uart.data register
emov	eax, ds : 0x4
or	eax, 0x10
emov	ds : 0x4, eax → dx:0x4 -> Uart.trigger register
emov	eax, ds : 0x8 → ds:0x8 -> Uart.status register
cmp	eax, 0x1
jne	17 < _uart_send_data+0x17>
ret	
nop	
nop	

[0052] 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.

[0053] 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.

[0054] Instructions used to program logic to perform embodiments of the present disclosure may be stored within a memory in the system, such as DRAM, cache, flash memory, or other storage. Furthermore, the instructions may be distributed via a network or by way of other computer-readable media. Thus a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the computer-readable medium may include any type of tangible machine-readable medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer).

[0055] 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.

[0056] 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.

[0057] 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).

[0058] 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.

[0059] 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.

[0060] 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.

[0061] 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.).

[0062] 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.

[0063] FIG. 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.

[0064] 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.

[0065] 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.

[0066] 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.

[0067] 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.

[0068] 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.

[0069] 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.

[0070] 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.

[0071] 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.

[0072] FIG. 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.

[0073] 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.

[0074] 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.

[0075] 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.

[0076] 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).

[0077] FIG. 1C illustrates another embodiment of a data processing system to perform operations in accordance with embodiments of the present disclosure. 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.

[0078] 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.

[0079] 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.

[0080] 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.

[0081] FIG. 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.

[0082] 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.

[0083] 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.

[0084] 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.

[0085] Execution block 211 may contain execution units 212, 214, 216, 218, 220, 222, 224. Execution units 212, 214, 216, 218, 220, 222, 224 may execute the instructions. Execution block 211 may include register files 208, 210 that store the integer and floating point data operand values that the micro-instructions need to execute. In one embodiment, processor 200 may comprise a number of execution units: address generation unit (AGU) 212, AGU 214, fast ALU 216, fast ALU 218, slow ALU 220, floating point ALU 222, floating point move unit 224. In another embodiment, floating point execution blocks 222, 224, may execute floating point, MMX, SIMD, and SSE, or other operations. In yet another embodiment, floating point ALU 222 may include a 64-bit by 64-bit floating point divider to execute divide, square root, and remainder micro-ops. In various embodiments, instructions involving a floating point value may be handled with the floating point hardware. In one embodiment, ALU operations may be passed to high-speed ALU execution units 216, 218. High-speed ALUs 216, 218 may execute fast operations with an effective latency of half a clock cycle. In one embodiment, most complex integer operations go to slow ALU 220 as slow ALU 220 may include integer execution hardware for long-latency type of operations, such as a multiplier, shifts, flag logic, and branch processing. Memory load/store operations may be executed by AGUs 212, 214. In one embodiment, integer ALUs 216, 218, 220 may perform integer operations on 64-bit data operands. In other embodiments, ALUs 216, 218, 220 may be implemented to support a variety of data bit sizes including sixteen, thirty-two,

128, 256, etc. Similarly, floating point units 222, 224 may be implemented to support a range of operands having bits of various widths. In one embodiment, floating point units 222, 224, may operate on 128-bit wide packed data operands in conjunction with SIMD and multimedia instructions.

[0086] 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.

[0087] 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.

[0088] In the examples of the following figures, a number of data operands may be described. FIG. 3A illustrates various packed data type representations in multimedia registers, in accordance with embodiments of the present disclosure. FIG. 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.

[0089] 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 FIG. 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 FIG. 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.

[0090] FIG. 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.

[0091] FIG. 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.

[0092] FIG. 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.

[0093] FIG. 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.

[0094] FIG. 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.

[0095] FIG. 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. FIG. 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 FIG. 4A illustrate the in-order pipeline, while the dashed lined boxes illustrate the register renaming, out-of-order issue/execution pipeline. Similarly, the solid lined boxes in FIG. 4B illustrate the in-order architecture logic, while the dashed lined boxes illustrate the register renaming logic and out-of-order issue/execution logic.

[0096] In FIG. 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.

[0097] In FIG. 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. FIG. 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.

[0098] 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.

[0099] 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.

[0100] 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.

[0101] 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.

[0102] 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.

[0103] 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).

[0104] 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.

[0105] 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.

[0106] FIG. 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.

[0107] 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.

[0108] 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.

[0109] 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.

[0110] 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.

[0111] 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.

[0112] 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.

[0113] 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.

[0114] FIG. 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.

[0115] 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.

[0116] Out-of-order execution engine 580 may be implemented in any suitable manner, such as fully or in part by out-of-order execution engine 203 as described above. Out-of-order execution engine 580 may prepare instructions received from front end 570 for execution. Out-of-order execution engine 580 may include an allocate module 582. In one embodiment, allocate module 582 may allocate resources of processor 500 or other resources, such as registers or buffers, to execute a given instruction. Allocate module 582 may make allocations in schedulers, such as a memory scheduler, fast scheduler, or floating point scheduler. Such schedulers may be represented in FIG. 5B by resource schedulers 584. Allocate module 582 may be implemented fully or in part by the allocation logic described in conjunction with FIG. 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 FIG. 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.

[0117] 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.

[0118] FIGS. 6-8 may illustrate exemplary systems suitable for including processor 500, while FIG. 9 may illustrate an exemplary system on a chip (SoC) that may include one or more of cores 502. Other system designs and implementations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, may also be suitable. In general, a huge variety of systems or electronic devices that incorporate a processor and/or other execution logic as disclosed herein may be generally suitable.

[0119] FIG. 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 FIG. 6 with broken lines.

[0120] 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. FIG. 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.

[0121] 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.

[0122] 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.

[0123] 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.

[0124] FIG. 7 illustrates a block diagram of a second system 700, in accordance with embodiments of the present disclosure. As shown in FIG. 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.

[0125] While FIG. 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.

[0126] 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 FIG. 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.

[0127] 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.

[0128] 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.

[0129] 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.

[0130] As shown in FIG. 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 FIG. 7, a system may implement a multi-drop bus or other such architecture.

[0131] FIG. 8 illustrates a block diagram of a third system 700 in accordance with embodiments of the present disclosure. Like elements in FIGS. 7 and 8 bear like reference numerals, and certain aspects of FIG. 7 have been omitted from FIG. 8 in order to avoid obscuring other aspects of FIG. 8.

[0132] FIG. 8 illustrates that processors 770, 780 may include integrated memory and I/O control logic (“CL”) 772 and 782, respectively. For at least one embodiment, CL 772, 782 may include integrated memory controller units such as that described above in connection with FIGS. 5 and 7. In addition, CL 772, 782 may also include I/O control logic. FIG. 8 illustrates that not only memories 732, 734 may be coupled to CL 772, 782, but also that I/O devices 814 may also be coupled to control logic 772, 782. Legacy I/O devices 815 may be coupled to chipset 790.

[0133] FIG. 9 illustrates a block diagram of a SoC 900, in accordance with embodiments of the present disclosure. Similar elements in FIG. 5 bear like reference numerals. Also, dashed lined boxes may represent optional features on more advanced SoCs. An interconnect units 902 may be coupled to: an application processor 910 which may include a set of one or more cores 502A-N and shared cache units 506; a system agent unit 912; a bus controller units 916; an integrated memory controller units 914; a set of one or more media processors 920 which may include integrated graphics logic 908, an image processor 924 for providing still and/or video camera functionality, an audio processor 926 for providing hardware audio acceleration, and a video processor 928 for providing video encode/decode acceleration; an static random access memory (SRAM) unit 930; a direct memory access (DMA) unit 932; and a display unit 940 for coupling to one or more external displays.

[0134] FIG. 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.

[0135] 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.

[0136] In FIG. 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 FIG. 10, including more CPUs or GPUs and other peripheral interface controllers.

[0137] 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.

[0138] FIG. 11 illustrates a block diagram illustrating the development of IP cores, in accordance with embodiments of the present disclosure. Storage 1130 may include simulation software 1120 and/or hardware or software model 1110. In one embodiment, the data representing the IP core design may be provided to storage 1130 via memory 1140 (e.g., hard disk), wired connection (e.g., internet) 1150 or wireless connection 1160. The IP core information generated by the simulation tool and model may then be transmitted to a fabrication facility where it may be fabricated by a 3rd party to perform at least one instruction in accordance with at least one embodiment.

[0139] 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.

[0140] FIG. 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 FIG. 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.

[0141] FIG. 13 is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. FIG. 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 can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler 1304 represents a compiler that is operable to generate x86 binary code 1306 (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core 1316. Similarly, FIG. 13 shows the program in the 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).

[0142] The instruction converter 1312 is used to convert the x86 binary code 1306 into alternative instruction set binary code 1311 that may be natively executed by the processor without an x86 instruction set core 1314. This converted code may or may not be the same as the alternative instruction set binary code 1310 resulting from an alternative instruction set compiler 1308; however, the converted code will accomplish the same general operation and be made up of instructions from the alternative instruction set. Thus, the 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 the x86 binary code 1306.

[0143] FIG. 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.

[0144] 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.

[0145] 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 FIG. 14, instruction set architecture 1400 may include a liquid crystal display (LCD) video interface 1425, a subscriber interface module (SIM) interface 1430, a boot ROM interface 1435, a synchronous dynamic random access memory (SDRAM) controller 1440, a flash controller 1445, and a serial peripheral interface (SPI) master unit 1450. LCD video interface 1425 may provide output of video signals from, for example, GPU 1415 and through, for example, a mobile industry processor interface (MIPI) 1490 or a high-definition multimedia interface (HDMI) 1495 to a display. Such a display may include, for example, an LCD. SIM interface 1430 may provide access to or from a SIM card or device. SDRAM controller 1440 may provide access to or from memory such as an SDRAM chip or module. Flash controller 1445 may provide access to or from memory such as flash memory or other instances of RAM. SPI master unit 1450 may provide access to or from communications modules, such as a Bluetooth module 1470, high-speed 3G modem 1475, global positioning system module 1480, or wireless module 1485 implementing a communications standard such as 802.11.

[0146] FIG. 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.

[0147] 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.

[0148] In another embodiment, memory system 1540 may include a retirement pointer 1582. Retirement pointer 1582 may store a value identifying the program order (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.

[0149] Execution entities 1565 may include any suitable number and kind of mechanisms by which a processor may execute instructions. In the example of FIG. 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.

[0150] 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.

[0151] 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.

[0152] 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.

[0153] 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.

[0154] 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.

[0155] 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.

[0156] 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.

[0157] 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.

[0158] FIG. 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 FIG. 15.

[0159] 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.

[0160] FIG. 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.

[0161] 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.

[0162] Such components may include, for example, a display 1724, a touch screen 1725, a touch pad 1730, a near field communications (NFC) unit 1745, a sensor hub 1740, a thermal sensor 1746, an express chipset (EC) 1735, a trusted platform module (TPM) 1738, BIOS/firmware/flash memory 1722, a digital signal processor 1760, a drive 1720 such as a solid state disk (SSD) or a hard disk drive (HDD), a wireless local area network (WLAN) unit 1750, a Bluetooth unit 1752, a wireless wide area network (WWAN) unit 1756, a global positioning system (GPS), a camera 1754 such as a USB 3.0 camera, or a low power double data rate (LPDDR) memory unit 1715 implemented in, for example, the LPDDR3 standard. These components may each be implemented in any suitable manner.

[0163] 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).

[0164] Referring now to FIG. 18, shown is a flow diagram of a method in accordance with an embodiment of the present invention. More specifically, method 1800 shown in FIG. 18 may be performed by compiler logic to execute on a processor, such as a static or dynamic compiler that is to generate machine code from incoming source code. As illustrated, method 1800 begins by identifying at least one variable having a compiler directive for in-order handling (block 1810). As described herein, one or more variables within code can be identified as a particular data type to be handled in order. As such, this identification may be based, e.g., on a preamble of a definition statement for the variable that identifies to the compiler that the variable is to be handled in an in-order manner.

[0165] Next, control passes to block 1820 where one or more load/store instructions in the source code can be identified as being associated with one or more of these in-order variables (if any). For example, various load/store instructions within the source code can be identified. From these identified instructions, the compiler can determine whether any of these instructions involve a variable to be handled in an in-order manner. In such case, control passes to block 1830 where machine code can be generated with in-order load/store operations for these identified instructions. In the embodiments described herein, this machine code may be implemented with special opcode types that encode such in-order operations. Finally, at block 1840 compiled machine code can be emitted from the compiler. This compiled machine code, which in an embodiment may be in an assembly language of an underlying processor architecture (e.g., an x86 processor), can be stored in a given storage. For example, this compiled program may be stored in a database. From there, the database can be accessed to obtain the code corresponding to a program, which may then be provided, e.g., to a remote source such as a consumer that seeks to download a given application from the remote database. Of course in other instances, the compiler may execute locally on a

system that is to store and then later execute the machine code. Understand while shown at this high level in the embodiment of FIG. 18, many variations and alternatives are possible.

[0166] Referring now to FIG. 19, shown is a flow diagram of a method in accordance with another embodiment of the present invention. In the embodiment of FIG. 19, method 1900 may be performed by combinations of hardware, software, and/or firmware, including hardware circuitry such as logic within a processor, including front end units such as an instruction decoder, issue logic and so forth, as well as one or more execution units such as one or more load/store execution units within the processor. As illustrated, method 1900 begins by receiving and decoding an ordered load/store instruction in the decoder (block 1910). Thereafter this decoded instruction is provided to a scheduler (block 1920). In an embodiment, this scheduler may be implemented in one or more logics or units of a front end portion of a processor, such as schedule logic, issue logic, order logic and so forth.

[0167] Next it is determined whether there is a preceding ordered load/store instruction in a scheduling block (diamond 1930). In embodiments, such scheduling block may encompass a given number of instructions that are scheduled together within an instruction scheduling window or block. If it is determined that there is one or more preceding ordered load/store instructions, control passes to block 1940 where the decoded ordered load/store instruction may be ordered after such preceding ordered load/store instruction.

[0168] Otherwise if at diamond 1930 it is determined that no such preceding instruction is present within the scheduling block, control passes to diamond 1950 to determine whether there is a succeeding ordered load/store instruction within the scheduling block. If so, control passes to block 1960 where the decoded ordered load/store instruction is ordered before the succeeding ordered load/store instruction.

[0169] Still with reference to FIG. 19, control passes from both of blocks 1940 and 1960 and diamond 1950 to block 1970. At block 1970, the decoded ordered load/store instruction can be stored in a ready queue with a priority indicator. In an embodiment in which such priority indicator is provided and used, the indicator may indicate to scheduling logic that an ordered load/store instruction is to have priority over other, non-ordered instructions.

[0170] Still with reference to FIG. 19, next at block 1980 any ordered load/store instructions within the scheduling block may be scheduled in order. In addition, these

instructions (which as discussed above may have a priority indicator) may be scheduled with priority, e.g., ahead of non-ordered instructions. Finally, at block 1990 such instructions may be executed within a given execution unit (e.g., a given load/store unit) and retired.

Understand while shown at this high level in the embodiment of FIG. 19, many variations and alternatives are possible.

[0171] As described above, many variations of instruction generation, identification and execution are possible. Referring now to FIG. 20, shown is a flow diagram of a method for compiler execution in accordance with an embodiment. As illustrated in FIG. 20, method 2000 may be performed by a compiler logic to be executed on a processor, such as a given static or run-time compiler. As seen, method 2000 begins by determining whether an instruction received within an instruction stream (e.g., of a source code program being statically or dynamically compiled) is a load/store operation (diamond 2010). If not, the given operation may be translated to machine code (block 2050). For example, a given source code operation may be translated into machine code that can be implemented as one or more assembly instructions.

[0172] Still with reference to FIG. 20, if instead at diamond 2010 it is determined that a received instruction is a load/store operation, control passes to diamond 2020 to determine whether any operand associated with the load/store operation is marked with a keyword. In an embodiment, this keyword may be a prefix code or symbol to provide a directive to the compiler that such operand (or address range) is to be handled in order. If such operand/address range is identified within the load/store operation, control passes to block 2040 where an in-order load/store instruction can be used to represent this load/store operation. Control next passes to block 2050, as discussed above. Otherwise, control passes from diamond 2020 to block 2030 where a standard load/store instruction can be used to represent the operation (and thereafter control passes to block 2050). While shown at this high level in the embodiment of FIG. 20, many variations and alternatives are possible.

[0173] Referring now to FIG. 21, shown is a flow diagram of a method for decoding instructions in accordance with an embodiment of the present invention. As seen in FIG. 21, method 2100 may be performed at least in part by hardware circuitry such as decode logic of a front end unit of a processor. As illustrated, method 2100 begins by decoding a given instruction and marking attributes and dependencies of the instruction (block 2110).

[0174] Next it is determined whether the instruction is an in-order instruction (diamond 2120). If not, control passes to block 2130 where this decoded instruction may be added to a processor data structure, such as a storage within an issue logic of the processor that is coupled downstream of the decode logic, or other data structure-like list (e.g., a re-order buffer, depending upon microarchitecture).

[0175] Note that if instead it is determined that the decoded instruction is an in-order instruction (as determined at diamond 2120), control passes to block 2140. At block 2140, the instruction may be added to an in-order decoded instruction queue which may be a first-in-first-out (FIFO) queue. Storage in this queue ensures ordering among the control instructions. That is, the first instruction in program order will be at the front of the queue, the second in the second place and so on. As above, this queue may be present in issue logic of the processor. Note that method 2100 may be performed iteratively for incoming instructions to thus identify and specially handle in-order instructions within an instruction stream including unordered instructions (and possibly ordered instructions also) and to be executed in an out-of-order processor.

[0176] Referring now to FIG. 22, shown is a flow diagram of a method in accordance with an embodiment of the present invention. More specifically, method 2200 may be performed by hardware circuitry such as issue logic of a processor to issue instructions to one or more execution units. As seen, method 2200 begins by determining whether there is an in-order instruction in-flight (meaning that the instruction is present within the processor, including in a writeback buffer) (diamond 2210). If so, control passes to block 2250 where issue logic of a processor may perform instruction issuance from the pool of standard/out-of-order instructions available in a standard issue data structure. In standard issuance of instructions, unless two instructions share the same processor resource a given instruction is issued to execution logic for execution. The instructions may execute out-of-order if the latency of the two instructions are different (e.g., addressing modes of the instructions are different); or the processor uses a write buffer between the pipeline and a bus interface unit.

[0177] Otherwise, if no in-order instruction is in-flight, control passes to diamond 2220. At diamond 2220, it can be determined whether there is any in-order instruction available within a ready queue. If not, control passes to block 2250, discussed above. If instead there is an available in-order instruction, control passes to diamond 2230 where it can be determined

whether the instruction can be issued. This determination may be based on, e.g., whether prior in-order instructions have been executed. As an example, assume one of the operands needed for the in-order instruction is being computed by a previous out-of-order instruction in the execution stage, or a prerequisite out-of-order instruction is not yet even placed in the execution stage. In general, at diamond 2230 checks are performed as on an out-of-order instruction. If it is determined at diamond 2230 that the instruction cannot be issued, control passes to block 2250 discussed above. If it is determined at diamond 2230 that the instruction can be issued, control passes to block 2240 where the instruction is issued to a selected execution unit or logic. Understand while shown at this high level in the embodiment of FIG. 22, many variations and alternatives are possible.

[0178] Referring now to FIG. 23, shown is a block diagram of a portion of a processor in accordance with an embodiment. In the embodiment of FIG. 23, portions of a pipeline of a processor 2300, which is an out-of-order processor, are shown. As illustrated, a decode unit 2310 is configured to receive incoming instructions, e.g., macro-instructions received from a fetch unit and decode such instructions into one or more micro-operations (μ ops) in a micro-coded case. For a hardware decoding case, the output of this stage is a sequence of control signals. Depending on the instruction type, decode unit 2310 provides the decoded instructions to one of multiple locations within an issue unit 2320.

[0179] More specifically as seen in FIG. 23, issue unit 2320 includes a first queue 2322 for storage of in-order instructions and a second storage 2324 for storage of other instructions. In general, these storages are to store decoded instructions and operands/information about operand location. First queue 2322 is for storage of the decoded in-order instructions, and may be implemented as a FIFO queue. Second storage 2324 is for storage of other decoded instructions. As such, in-order instructions are issued strictly in the decode order. In another embodiment, queue 2322 can be implemented as part of processor data structure 2324 (which may be a re-order buffer (ROB) in some x86 architectures) by adding an indicator to indicate that it is an in-order instruction (or not) and another, e.g., 3-4 bit token number, representing the order of arrival from decode unit 2310.

[0180] When a given instruction is selected for execution, it is provided from one of storages 2322 and 2324 to a given one of multiple execution units 2330. Depending on the type of instruction, the operation can be provided to a particular execution unit. Although the

scope of the present invention is not limited in this regard, execution units 2330 may include integer execution units, floating point execution units, vector execution units, load/store execution units, among potentially other such execution units. While shown with these limited pipeline portions for ease of illustration, understand that a full pipeline of a processor may include additional pipeline units and other logic.

[0181] The following examples pertain to further embodiments.

[0182] In one example, a processor comprises: a core comprising an out-of-order pipeline including a decode logic, an issue logic to issue decoded instructions, and at least one execution logic to execute issued instructions of a program, where the at least one execution logic is to execute at least some instructions of the program out-of-order, the decode logic to decode a first in-order memory instruction of the program and provide the decoded first in-order memory instruction to the issue logic, the issue logic to order the first in-order memory instruction ahead of a second in-order memory instruction of the program, the first in-order memory instruction an unordered instruction in a source version of the program.

[0183] In an example, the first in-order memory instruction is to enable a memory mapped input/output access for the program, where the program comprises an embedded control application.

[0184] In an example, the first in-order memory instruction and the second in-order memory instruction comprise memory mapped input/output operations.

[0185] In an example, the issue logic is to issue the first in-order memory instruction to the execution logic with a higher priority than the second in-order memory instruction.

[0186] In an example, a compiler logic is to generate the first in-order memory instruction from the unordered memory instruction of the program, based at least in part on an identifier associated with at least one operand of the unordered memory instruction.

[0187] In an example, the compiler logic is to generate the first in-order memory instruction having a different machine code than a machine code for the unordered memory instruction.

[0188] In an example, the compiler logic is to generate the first in-order memory instruction responsive to a directive indicator of a variable declaration, where an operand of the unordered memory instruction corresponds to a variable of the variable declaration.

[0189] In an example, the issue logic comprises: a first storage to store pending in-order memory instructions; and a second storage to store pending unordered instructions.

[0190] In an example, the decode logic is to provide the first in-order memory instruction to the first storage of the issue logic, the first storage comprising a first-in-first-out queue.

[0191] In an example, the execution logic comprises at least one load/store execution unit to execute the first in-order memory instruction.

[0192] Note that the above processor can be implemented using various means.

[0193] In an example, the processor comprises a SoC incorporated in a user equipment touch-enabled device.

[0194] In another example, a system comprises a display and a memory, and includes the processor of one or more of the above examples.

[0195] In another example, a method comprises: identifying a first variable having a directive indicator to indicate in-order handling; identifying an operand of a load/store instruction of a code block associated with the first variable; translating the load/store instruction to an in-order load/store instruction encoded by a first machine code, to enable execution of the first machine code by an execution logic of a processor; and storing the first machine code in a destination storage.

[0196] In an example, the method further comprises translating a second load/store instruction to an unordered load/store instruction encoded by a second machine code, the second load/store instruction not having an operand associated with a variable marked with a directive indicator.

[0197] In an example, the directive indicator comprises a variable directive in a variable declaration for the operand.

[0198] In an example, the directive indicator comprises a prefix of the variable declaration.

[0199] In an example, the method further comprises ordering the in-order load/store instruction ahead of a second in-order load/store instruction associated with the operand, the second in-order load/store instruction succeeding the load/store instruction in program order.

[0200] In an example, the in-order load/store instruction comprises a write to a device trigger register, and the second in-order load/store instruction comprises a read or a write for a device status register.

[0201] In an example, the method further comprises ordering one or more unordered load/store instructions ahead of the in-order load/store instruction.

[0202] In an example, the operand comprises a memory mapped input/output location.

[0203] In an example, the method further comprises ordering a third in-order load/store instruction of the code block ahead of the in-order load/store instruction, the third in-order load/store instruction succeeding the in-order load/store instruction in program order and associated with a different operand than the operand of the in-order load/store instruction.

[0204] In another example, a computer readable medium including instructions is to perform the method of any of the above examples.

[0205] In another example, a computer readable medium including data is to be used by at least one machine to fabricate at least one integrated circuit to perform the method of any one of the above examples.

[0206] In another example, an apparatus comprises means for performing the method of any one of the above examples.

[0207] In another example, a system comprises: a processor including a first core having: a decode logic to decode instructions; an issue logic to issue decoded instructions, the issue logic including a first queue to store in-order memory access instructions and a second storage to store unordered instructions; and at least one execution logic to execute issued instructions of a program, where the issue logic is to store a decoded first in-order memory access instruction of the program and a decoded second in-order memory access instruction of the program in the first queue and store one or more unordered instructions of the program in the second storage, and provide the decoded first in-order memory access instruction to the at least one execution logic ahead of the decoded second in-order memory access instruction

of the program and unordered with respect to one or more of the one or more unordered instructions. The system may further include a dynamic random access memory coupled to the processor.

[0208] In an example, a compiler logic is to generate the first in-order memory access instruction from an unordered memory access instruction, based at least in part on an identifier associated with at least one operand of the unordered memory access instruction, the first in-order memory access instruction having a different machine code than a machine code for the unordered memory access instruction.

[0209] In an example, the at least one operand comprises a memory-mapped input/output location to be accessed during the program, the program comprising a device driver for an embedded controller, where the first core comprises an out-of-order pipeline.

[0210] In another example, an apparatus comprises: means for identifying a first variable having a directive indicator to indicate in-order handling; means for identifying an operand of a load/store instruction of a code block associated with the first variable; means for translating the load/store instruction to an in-order load/store instruction encoded by a first machine code, to enable execution of the first machine code by execution means of the apparatus; and means for storing the first machine code in a destination storage means.

[0211] In an example, the apparatus further comprises means for translating a second load/store instruction to an unordered load/store instruction encoded by a second machine code, the second load/store instruction not having an operand associated with a variable marked with a directive indicator.

[0212] Embodiments may be used in many different types of systems. For example, in one embodiment a communication device can be arranged to perform the various methods and techniques described herein. Of course, the scope of the present invention is not limited to a communication device, and instead other embodiments can be directed to other types of apparatus for processing instructions, or one or more machine readable media including instructions that in response to being executed on a computing device, cause the device to carry out one or more of the methods and techniques described herein.

[0213] Embodiments may be implemented in code and may be stored on a non-transitory storage medium having stored thereon instructions which can be used to program a system to

perform the instructions. Embodiments also may be implemented in data and may be stored on a non-transitory storage medium, which if used by at least one machine, causes the at least one machine to fabricate at least one integrated circuit to perform one or more operations. Still further embodiments may be implemented in a computer readable storage medium including information that, when manufactured into a SoC or other processor, is to configure the SoC or other processor to perform one or more operations. The storage medium may include, but is not limited to, any type of disk including floppy disks, optical disks, solid state drives (SSDs), 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.

[0214] While the present invention has been described with respect to a limited number of embodiments, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this present invention.

What is claimed is:

- 1 1. A processor comprising:
2 a core comprising an out-of-order pipeline including a decode logic, an issue logic to
3 issue decoded instructions, and at least one execution logic to execute issued instructions of a
4 program, wherein the at least one execution logic is to execute at least some instructions of
5 the program out-of-order, the decode logic to decode a first in-order memory instruction of
6 the program and provide the decoded first in-order memory instruction to the issue logic, the
7 issue logic to order the first in-order memory instruction ahead of a second in-order memory
8 instruction of the program, the first in-order memory instruction an unordered instruction in a
9 source version of the program.
- 1 2. The processor of claim 1, wherein the first in-order memory instruction is to enable a
2 memory mapped input/output access for the program, wherein the program comprises an
3 embedded control application.
- 1 3. The processor of claim 1, wherein the first in-order memory instruction and the
2 second in-order memory instruction comprise memory mapped input/output operations.
- 1 4. The processor of claim 1, wherein the issue logic is to issue the first in-order memory
2 instruction to the execution logic with a higher priority than the second in-order memory
3 instruction.
- 1 5. The processor of claim 1, wherein a compiler logic is to generate the first in-order
2 memory instruction from the unordered memory instruction of the program, based at least in
3 part on an identifier associated with at least one operand of the unordered memory
4 instruction.
- 1 6. The processor of claim 5, wherein the compiler logic is to generate the first in-order
2 memory instruction having a different machine code than a machine code for the unordered
3 memory instruction.

1 7. The processor of claim 5, wherein the compiler logic is to generate the first in-order
2 memory instruction responsive to a directive indicator of a variable declaration, wherein an
3 operand of the unordered memory instruction corresponds to a variable of the variable
4 declaration.

1 8. The processor of claim 1, wherein the issue logic comprises:
2 a first storage to store pending in-order memory instructions; and
3 a second storage to store pending unordered instructions.

1 9. The processor of claim 8, wherein the decode logic is to provide the first in-order
2 memory instruction to the first storage of the issue logic, the first storage comprising a first-
3 in-first-out queue.

1 10. The processor of claim 1, wherein the execution logic comprises at least one
2 load/store execution unit to execute the first in-order memory instruction.

1 11. A method comprising:
2 identifying a first variable having a directive indicator to indicate in-order handling;
3 identifying an operand of a load/store instruction of a code block associated with the
4 first variable;
5 translating the load/store instruction to an in-order load/store instruction encoded by a
6 first machine code, to enable execution of the first machine code by an execution logic of a
7 processor; and
8 storing the first machine code in a destination storage.

1 12. The method of claim 11, further comprising translating a second load/store instruction
2 to an unordered load/store instruction encoded by a second machine code, the second
3 load/store instruction not having an operand associated with a variable marked with a
4 directive indicator.

1 13. The method of claim 11, wherein the directive indicator comprises a variable directive
2 in a variable declaration for the operand.

- 1 14. The method of claim 13, wherein the directive indicator comprises a prefix of the
2 variable declaration.
- 1 15. The method of claim 11, further comprising ordering the in-order load/store
2 instruction ahead of a second in-order load/store instruction associated with the operand, the
3 second in-order load/store instruction succeeding the load/store instruction in program order.
- 1 16. The method of claim 15, wherein the in-order load/store instruction comprises a write
2 to a device trigger register, and the second in-order load/store instruction comprises a read or
3 a write for a device status register.
- 1 17. The method of claim 15, further comprising ordering one or more unordered
2 load/store instructions ahead of the in-order load/store instruction.
- 1 18. The method of claim 11, wherein the operand comprises a memory mapped
2 input/output location.
- 1 19. The method of claim 11, further comprising ordering a third in-order load/store
2 instruction of the code block ahead of the in-order load/store instruction, the third in-order
3 load/store instruction succeeding the in-order load/store instruction in program order and
4 associated with a different operand than the operand of the in-order load/store instruction.
- 1 20. A computer-readable storage medium including computer-readable instructions, when
2 executed, to implement a method as claimed in any one of claims 11 to 19.
- 1 21. A system comprising:
2 a processor including a first core having:
3 a decode logic to decode instructions;
4 an issue logic to issue decoded instructions, the issue logic including a first
5 queue to store in-order memory access instructions and a second storage to store unordered
6 instructions; and

at least one execution logic to execute issued instructions of a program,
wherein the issue logic is to store a decoded first in-order memory access instruction of the
program and a decoded second in-order memory access instruction of the program in the first
queue and store one or more unordered instructions of the program in the second storage, and
provide the decoded first in-order memory access instruction to the at least one execution
logic ahead of the decoded second in-order memory access instruction of the program and
unordered with respect to one or more of the one or more unordered instructions; and
a dynamic random access memory coupled to the processor.

22. The system of claim 21, wherein a compiler logic is to generate the first in-order
memory access instruction from an unordered memory access instruction, based at least in
part on an identifier associated with at least one operand of the unordered memory access
instruction, the first in-order memory access instruction having a different machine code than
a machine code for the unordered memory access instruction.

23. The system of claim 21, wherein the at least one operand comprises a memory-
mapped input/output location to be accessed during the program, the program comprising a
device driver for an embedded controller, wherein the first core comprises an out-of-order
pipeline.

24. An apparatus comprising:
means for identifying a first variable having a directive indicator to indicate in-order
handling;
means for identifying an operand of a load/store instruction of a code block associated
with the first variable;
means for translating the load/store instruction to an in-order load/store instruction
encoded by a first machine code, to enable execution of the first machine code by execution
means of the apparatus; and
means for storing the first machine code in a destination storage means.

25. The apparatus of claim 24, further comprising means for translating a second
load/store instruction to an unordered load/store instruction encoded by a second machine

- 3 code, the second load/store instruction not having an operand associated with a variable
- 4 marked with a directive indicator.

1/29

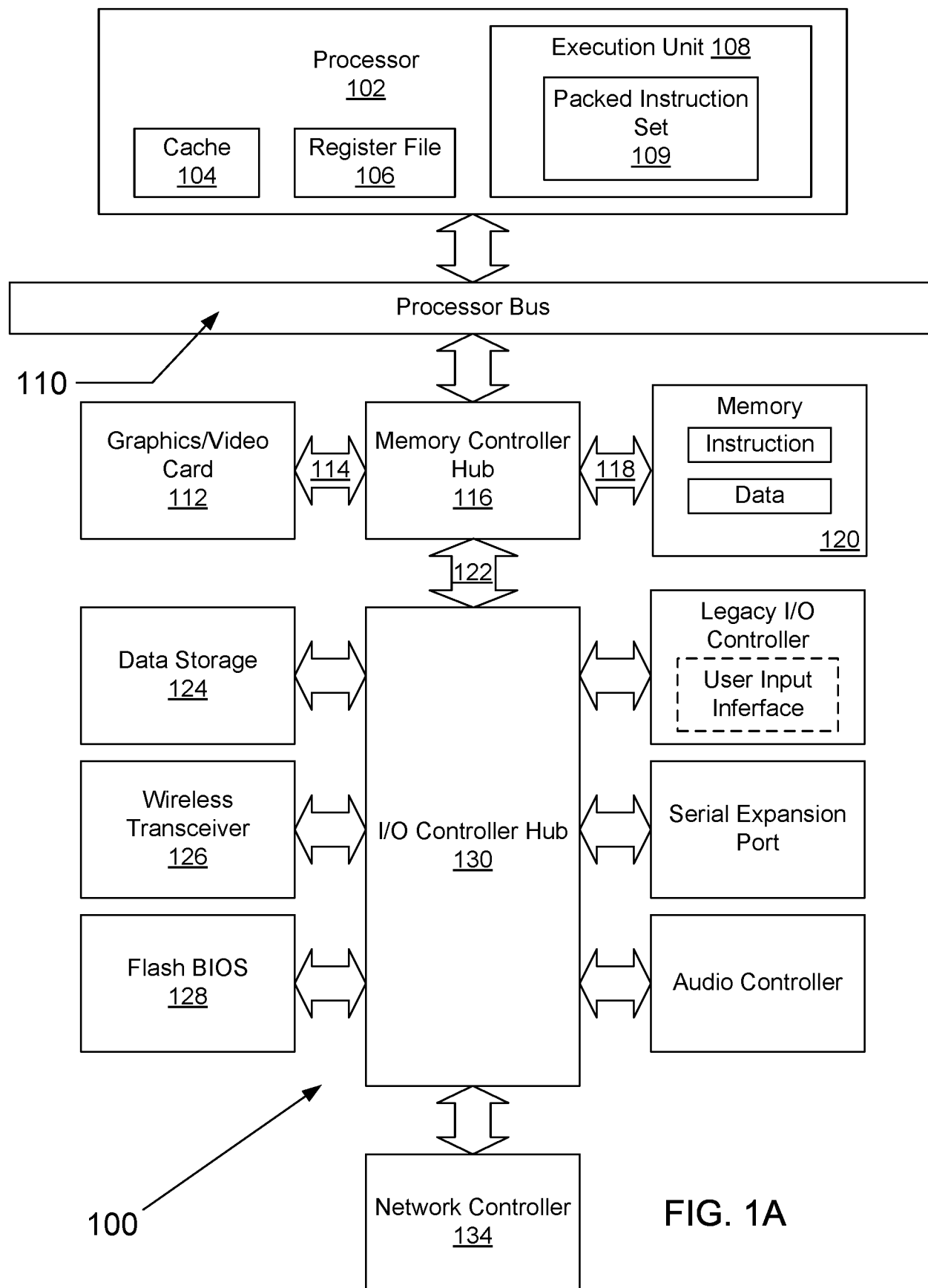


FIG. 1A

2/29

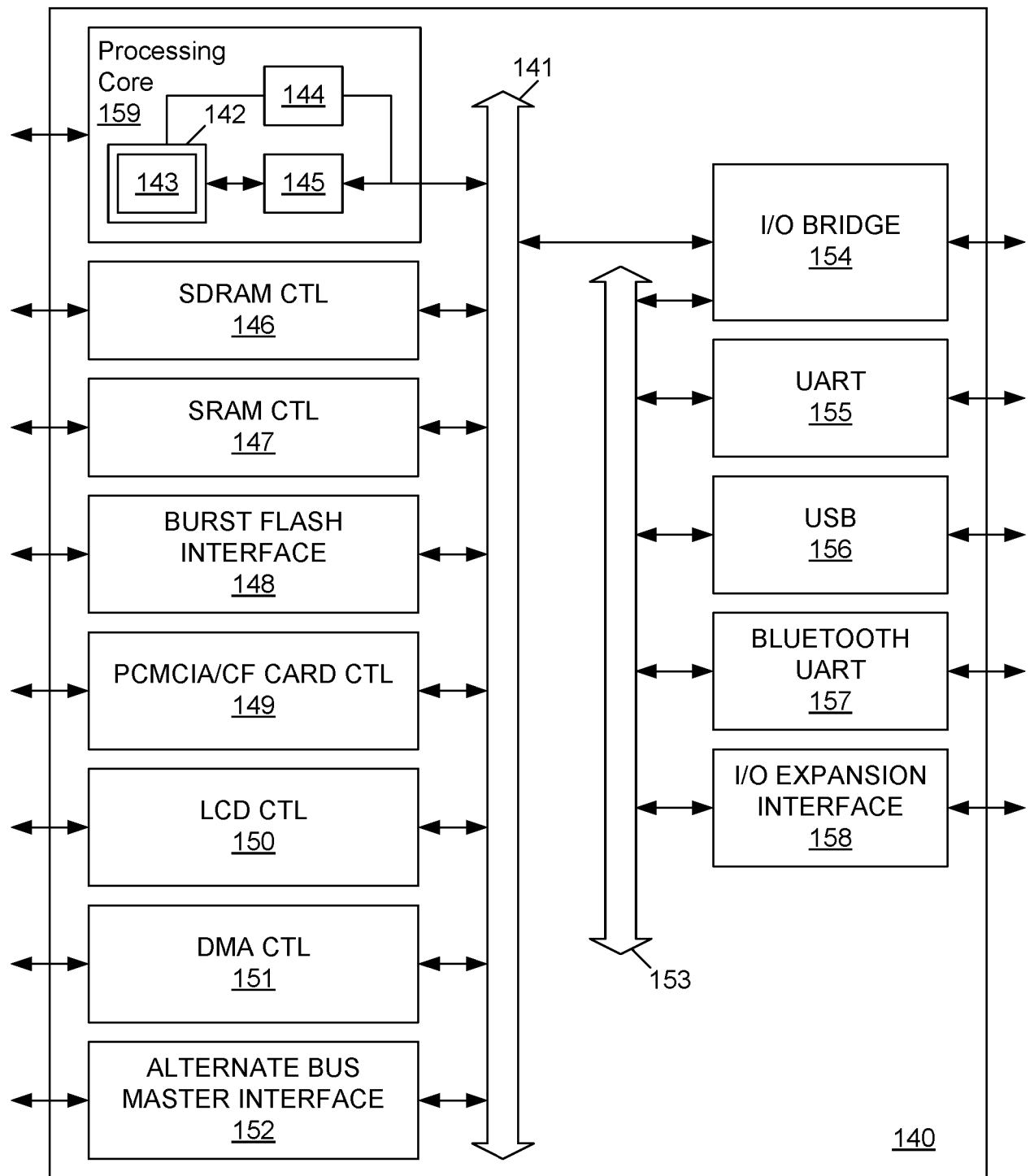


FIG. 1B

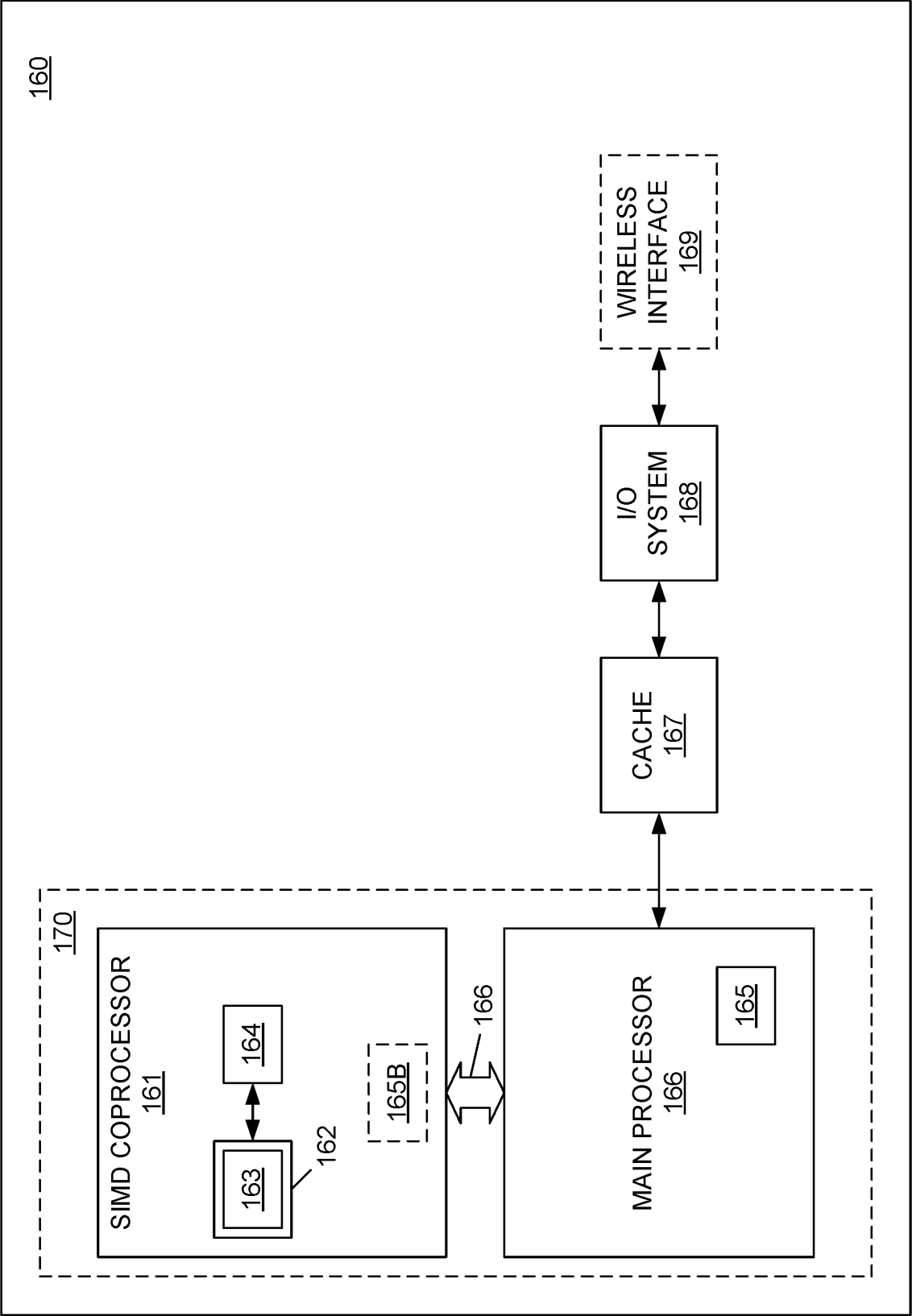
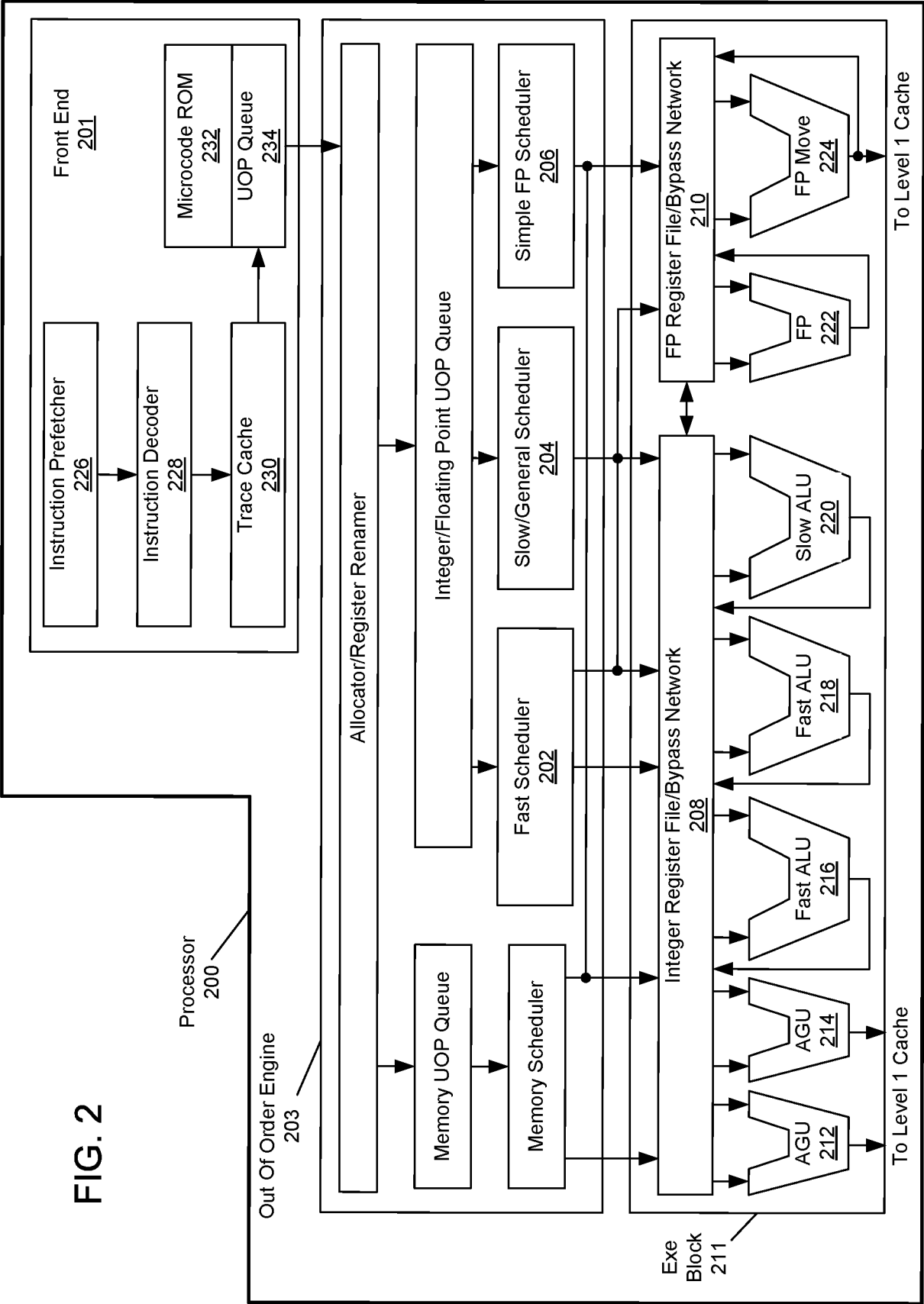


FIG. 1C



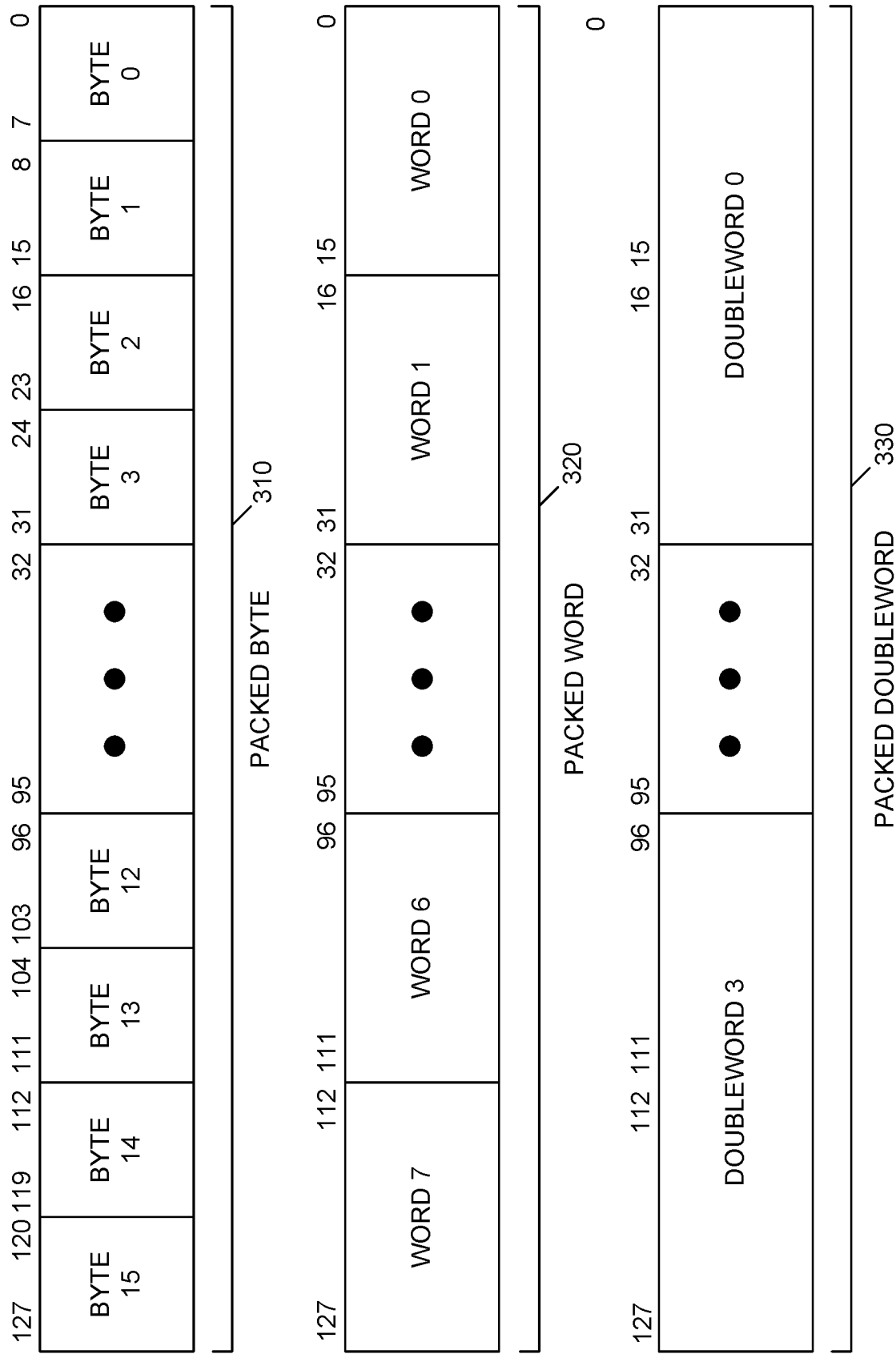


FIG. 3A

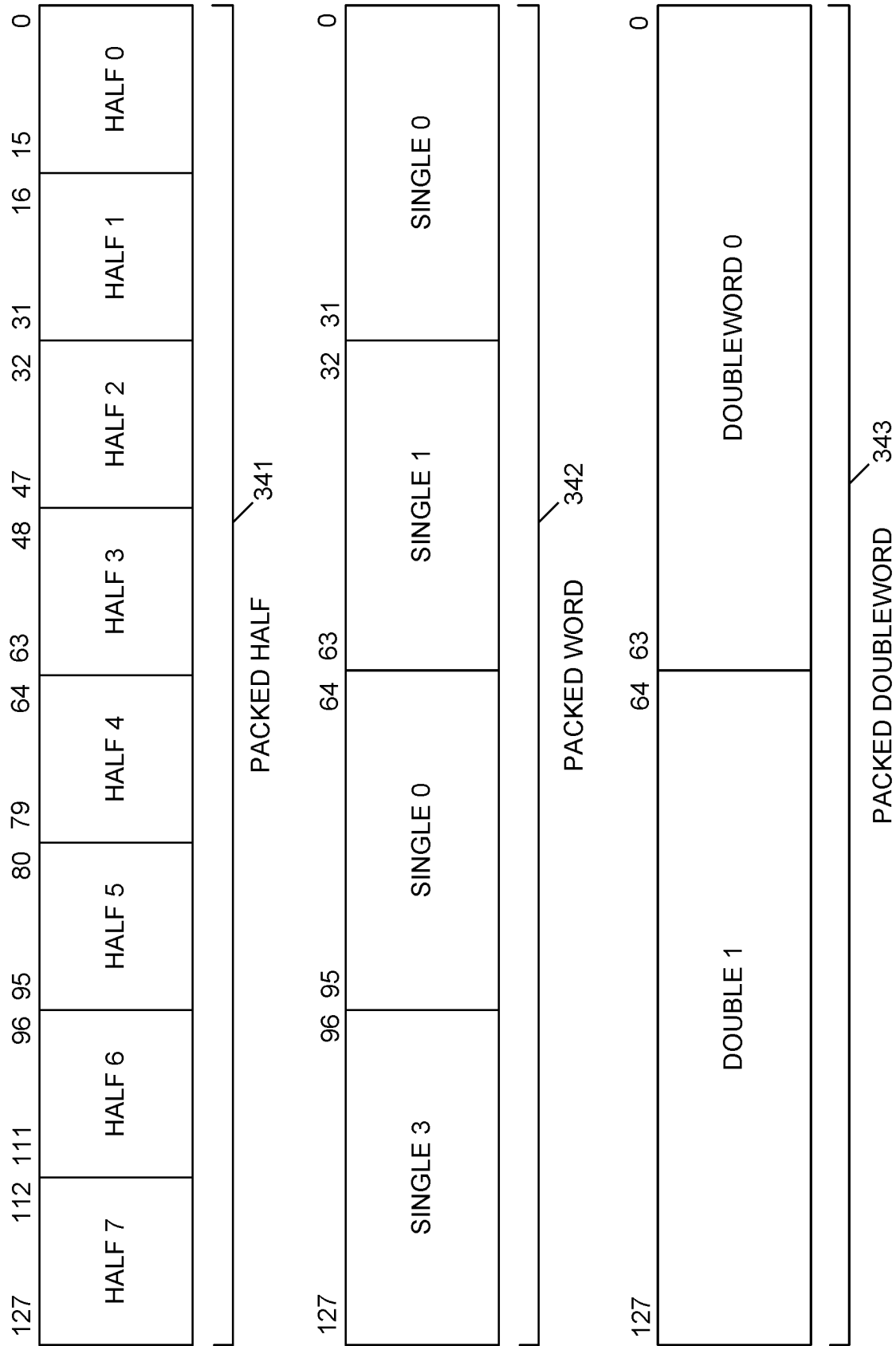
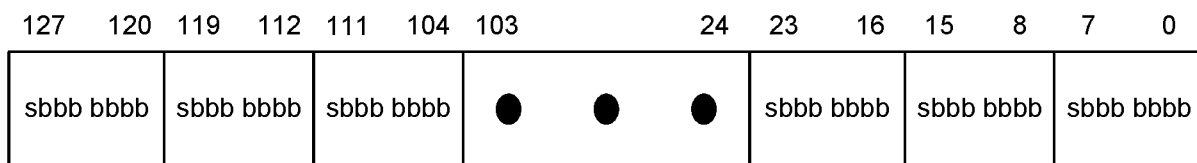
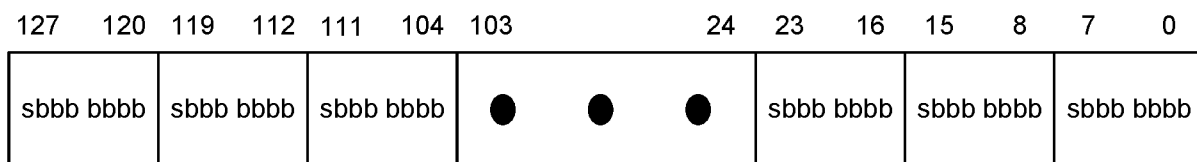


FIG. 3B

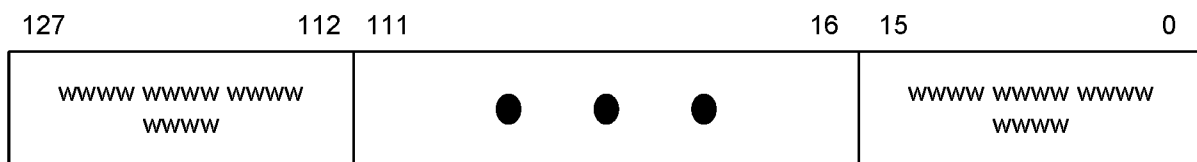
7/29



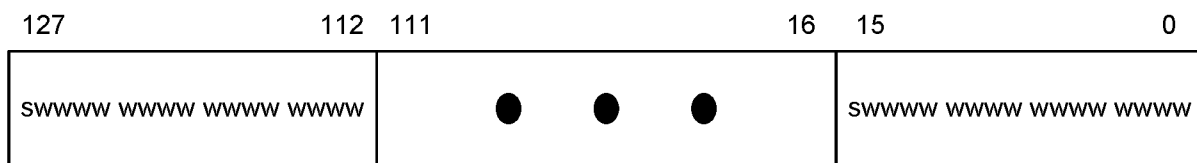
Unsigned Packed Byte Representation 344



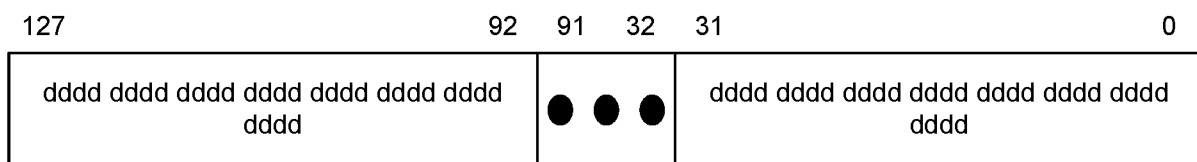
Signed Packed Byte Representation 345



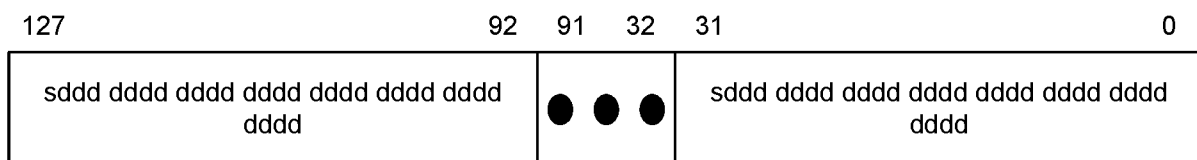
Unsigned Packed Word Representation 346



Signed Packed Word Representation 347



Unsigned Packed Doubleword Representation 348



Signed Packed Doubleword Representation 349

FIG. 3C

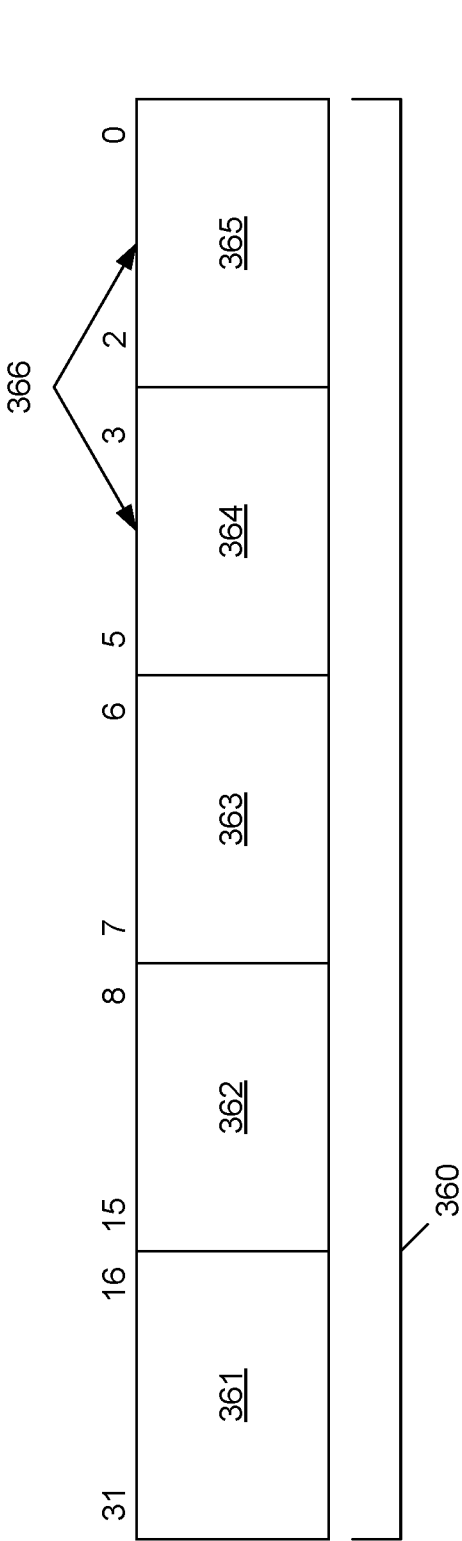


FIG. 3D

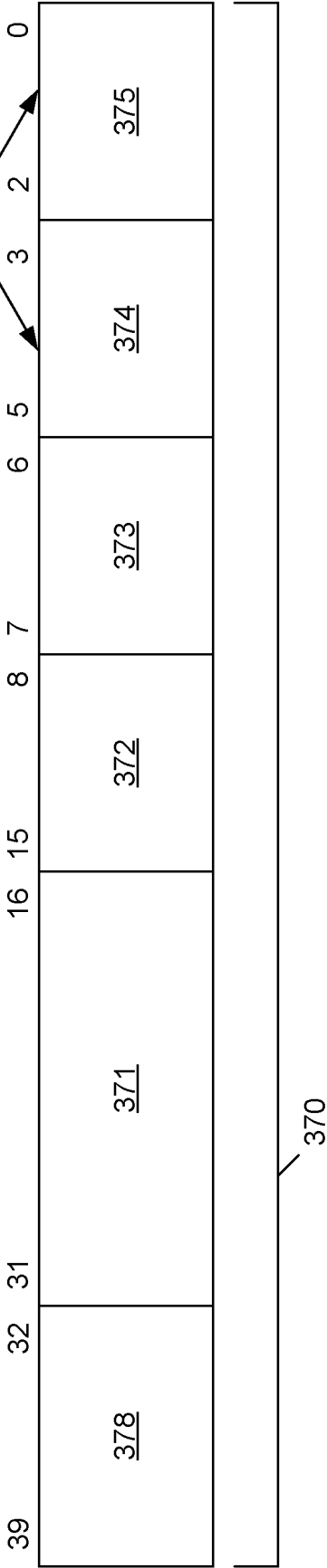


FIG. 3E

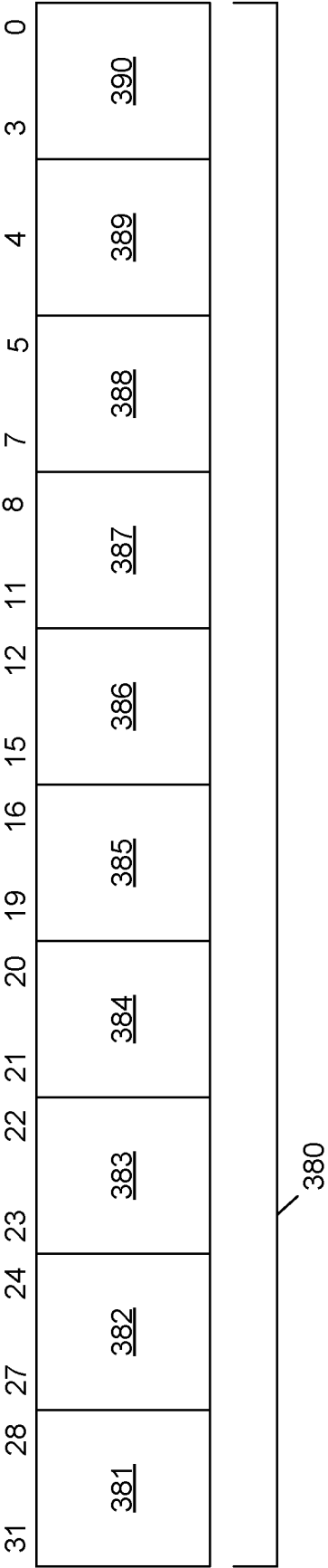
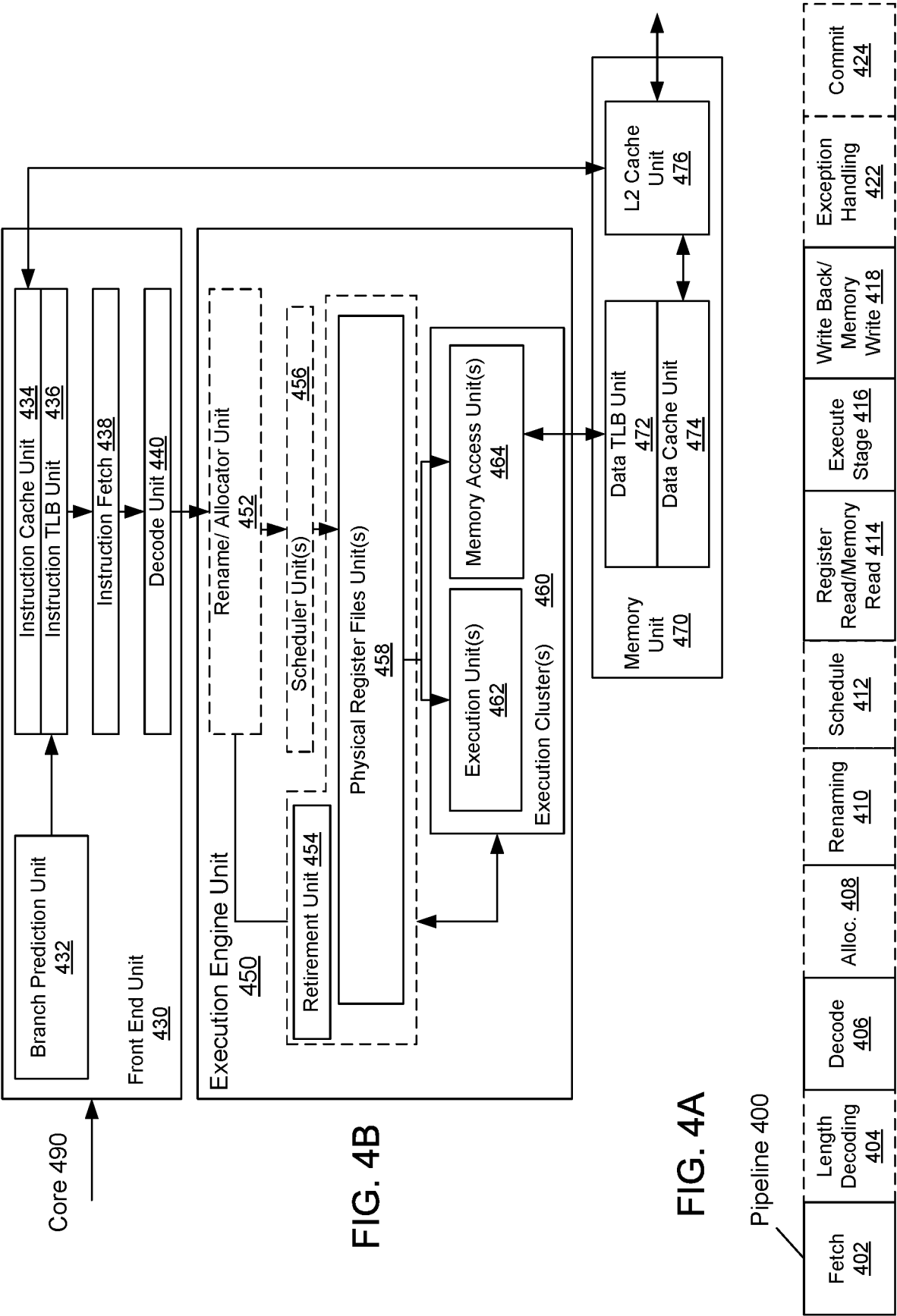


FIG. 3F



11/29

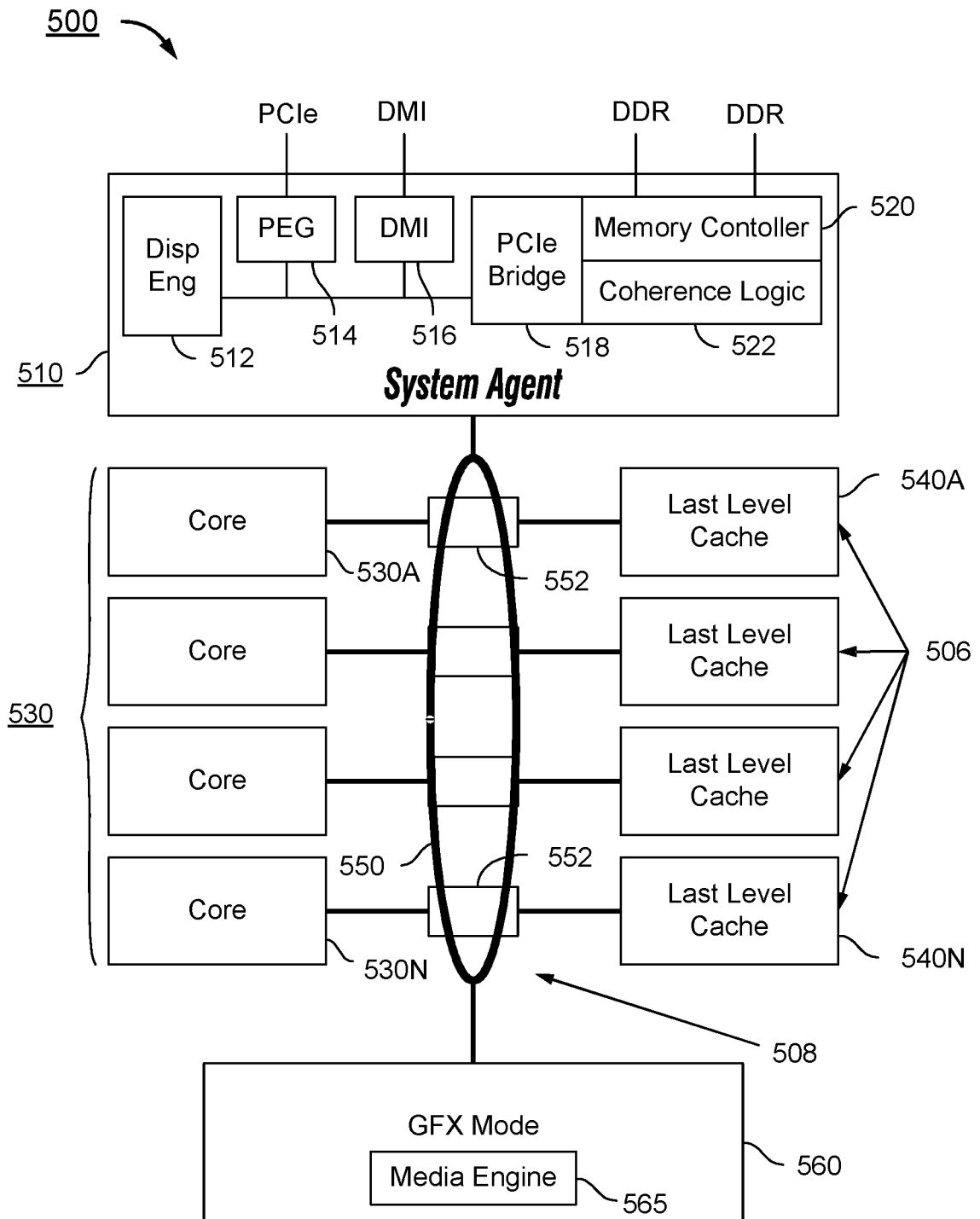


FIG. 5A

12/29

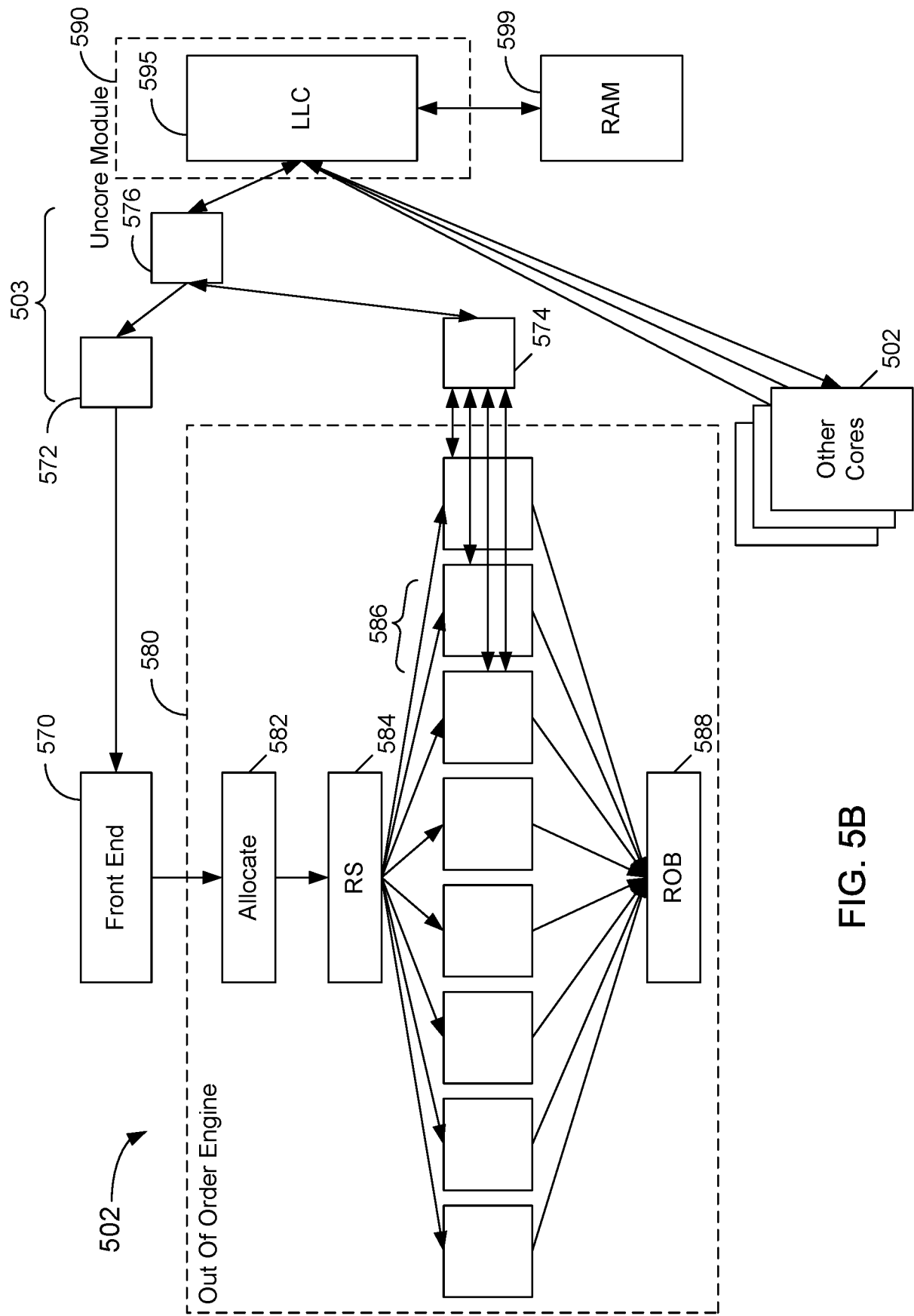


FIG. 5B

13/29

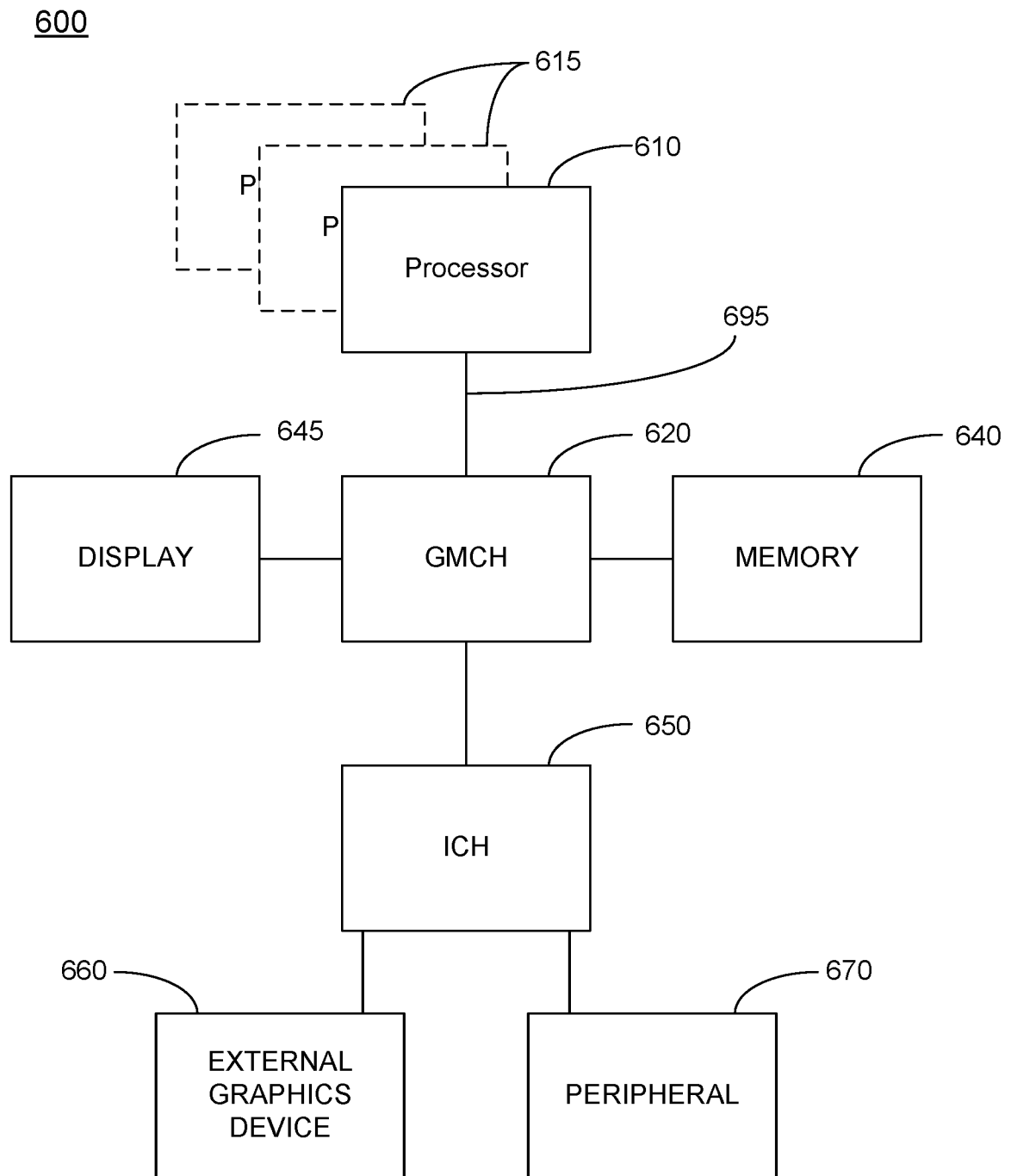


FIG. 6

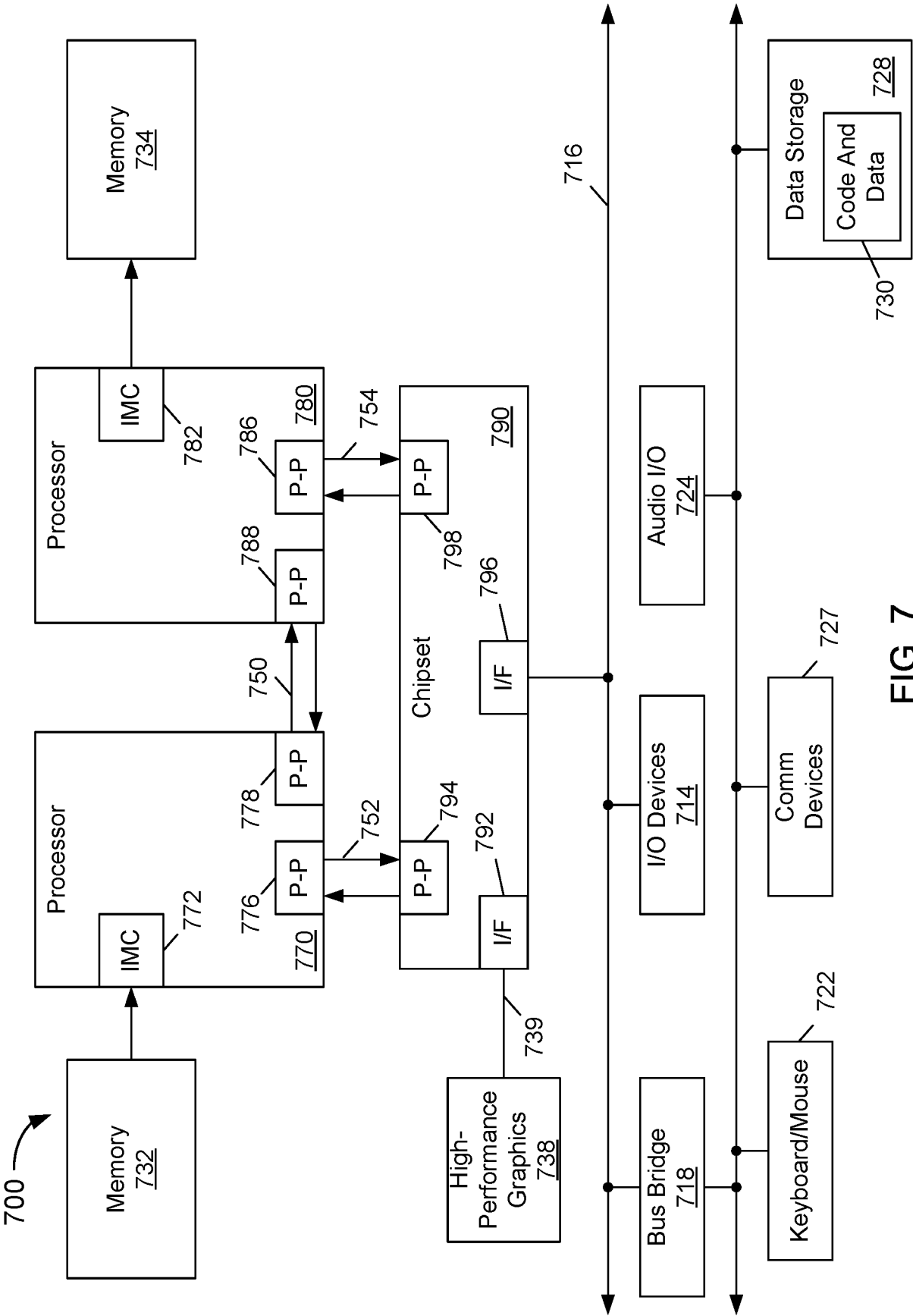


FIG. 7

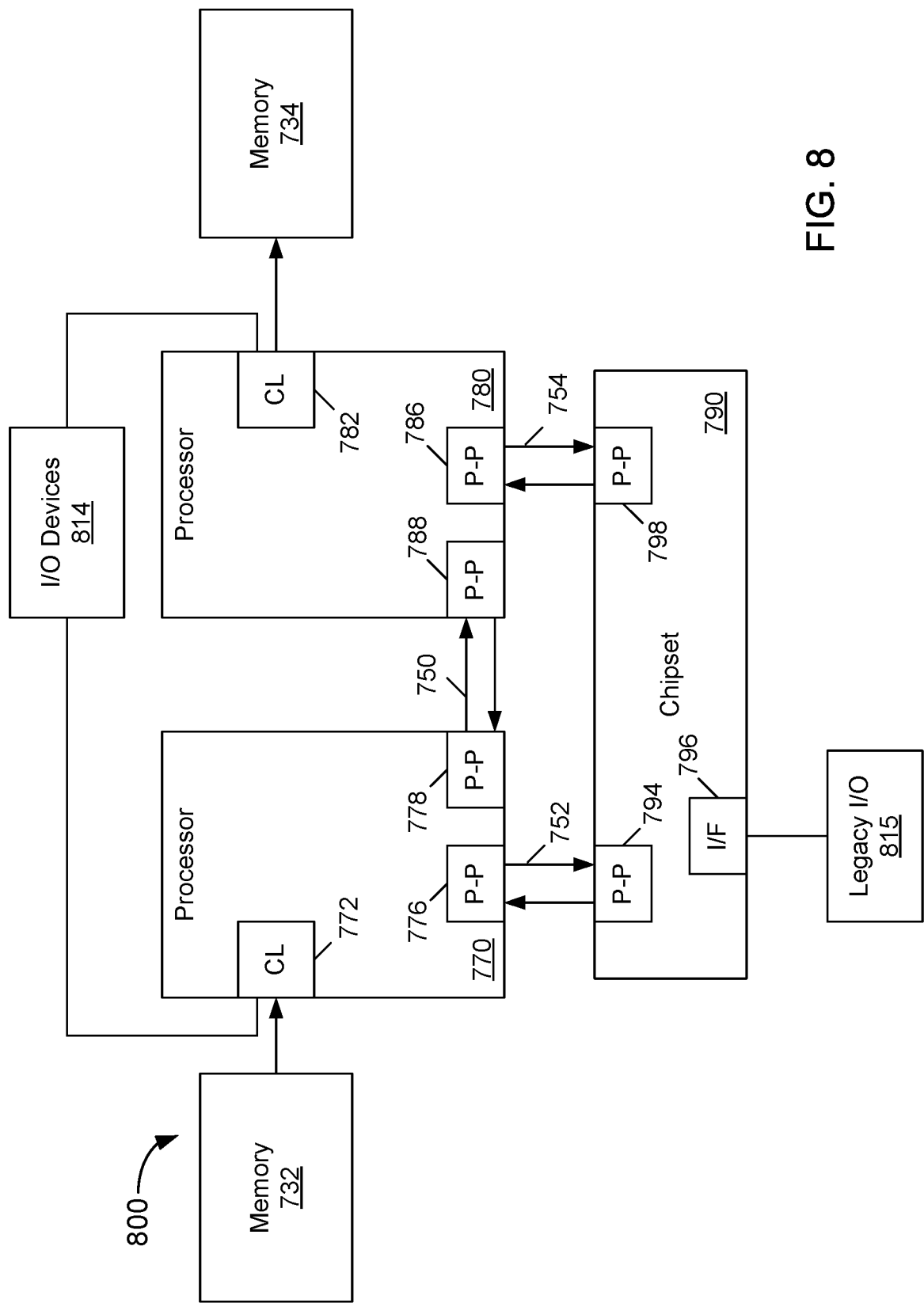


FIG. 8

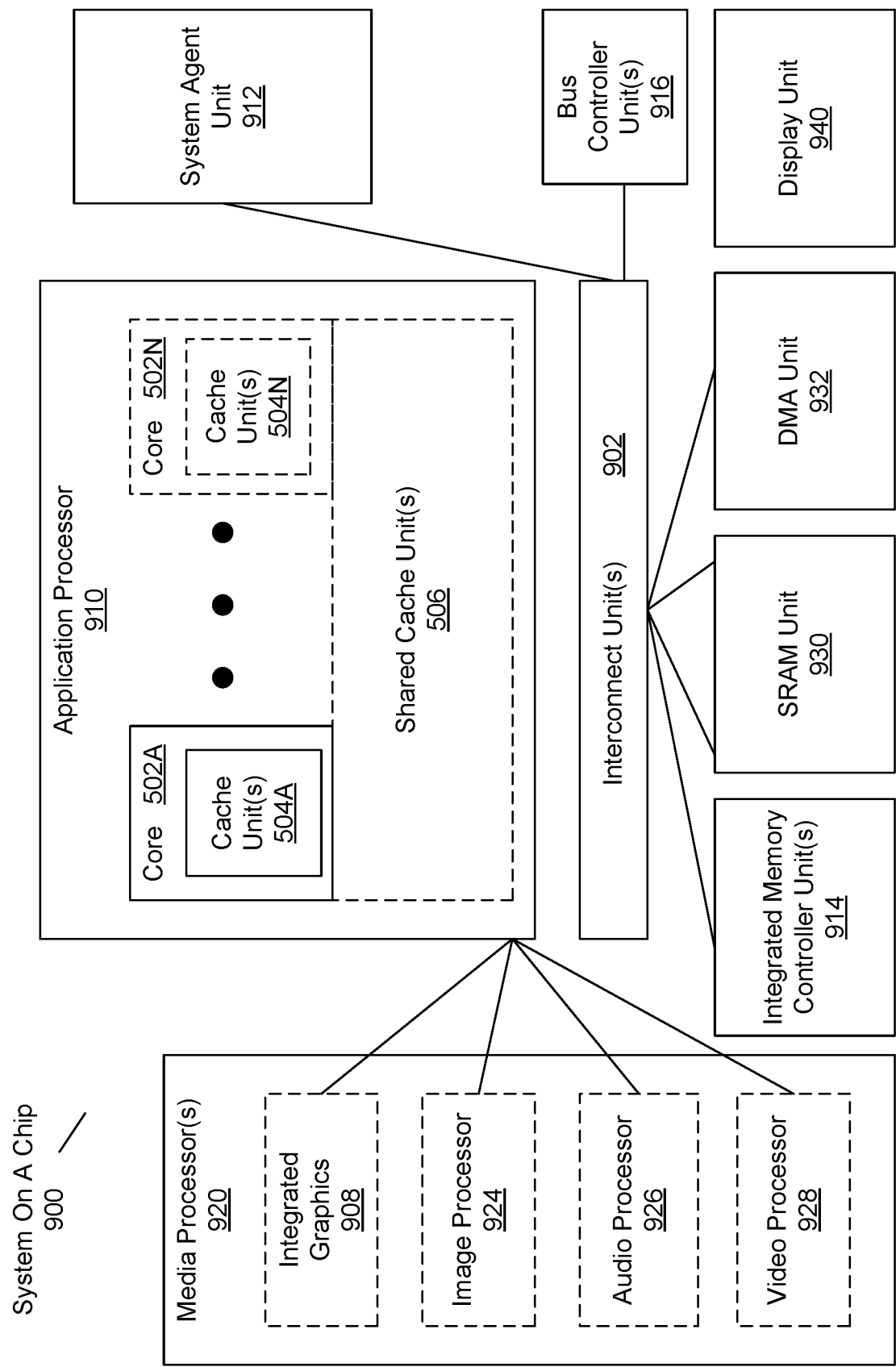


FIG. 9

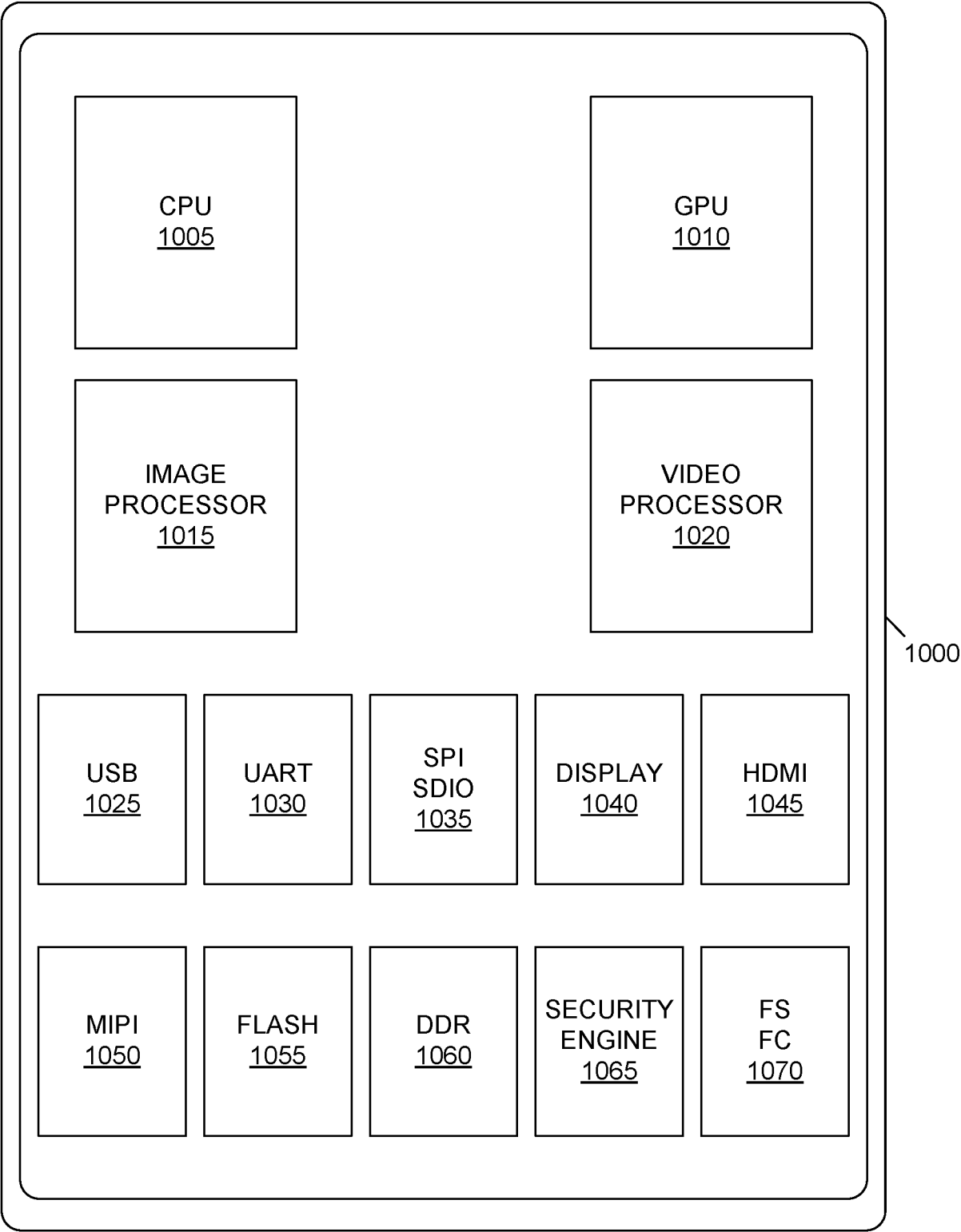


FIG. 10

18/29

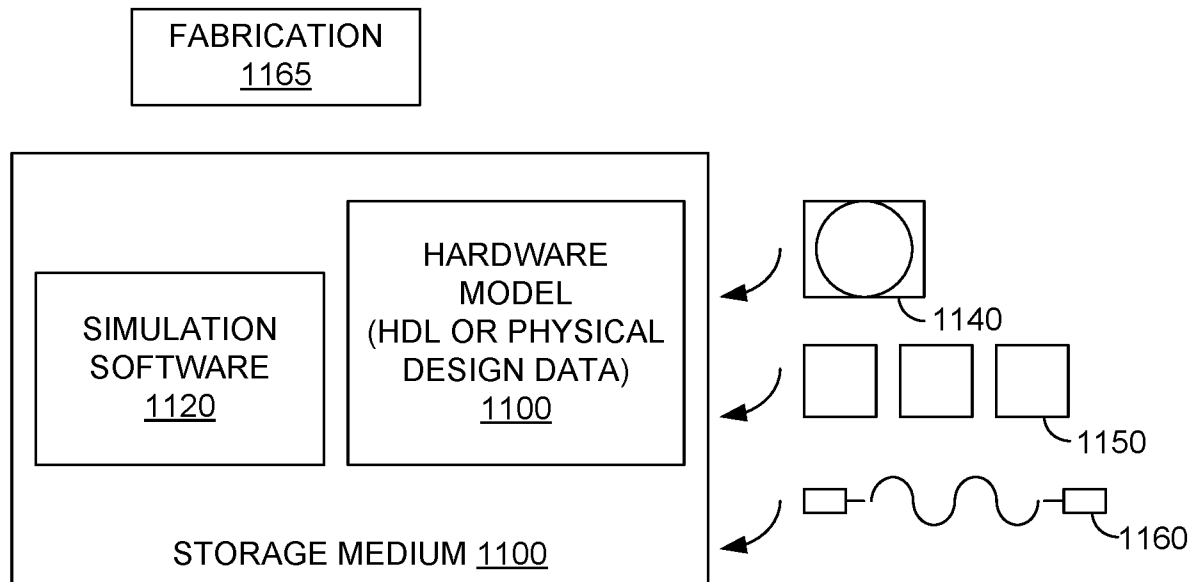


FIG. 11

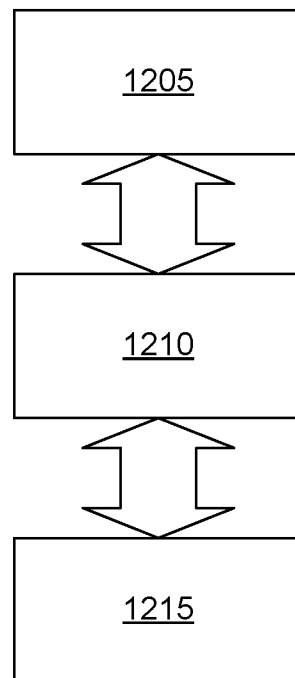
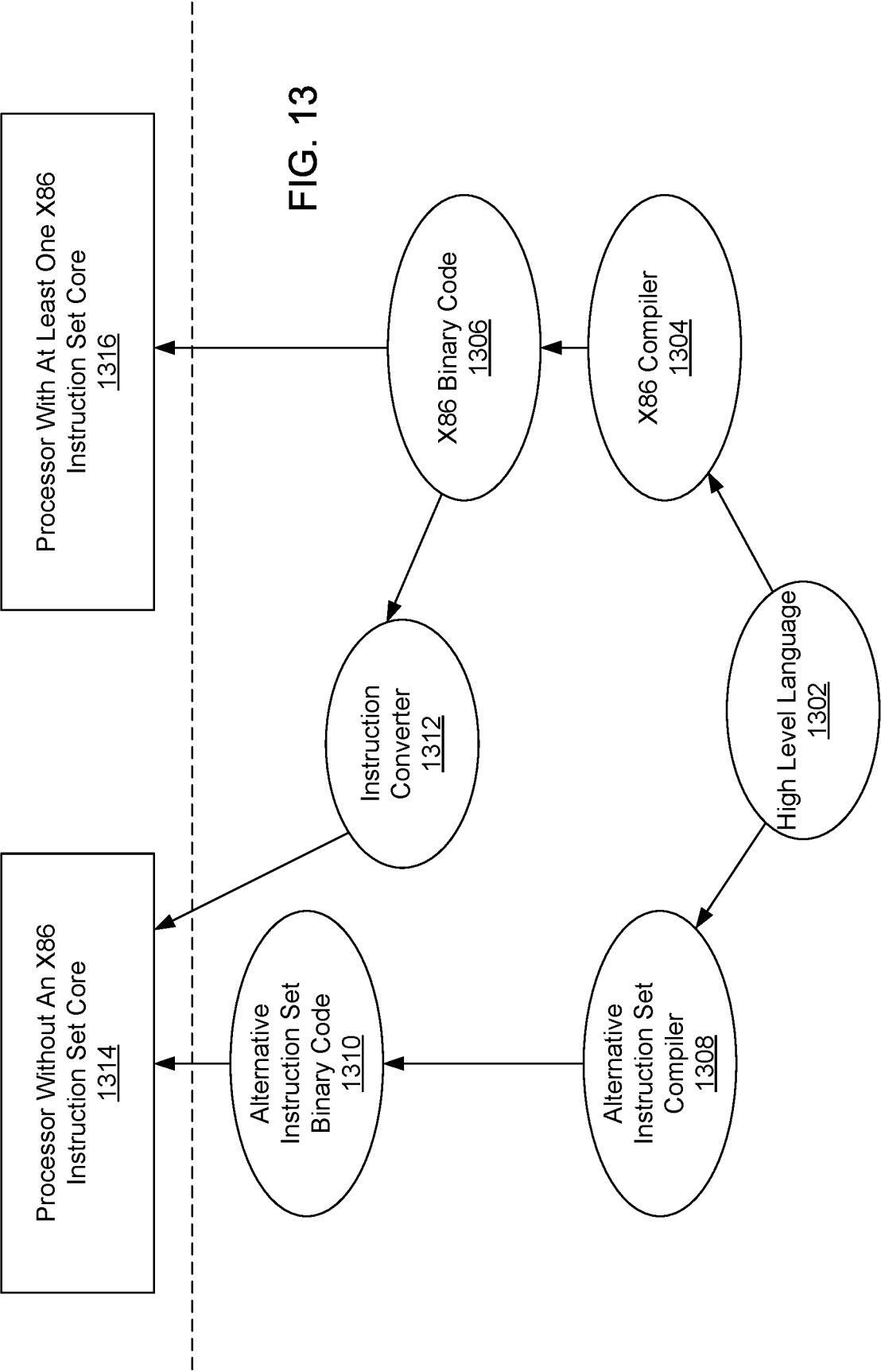


FIG. 12



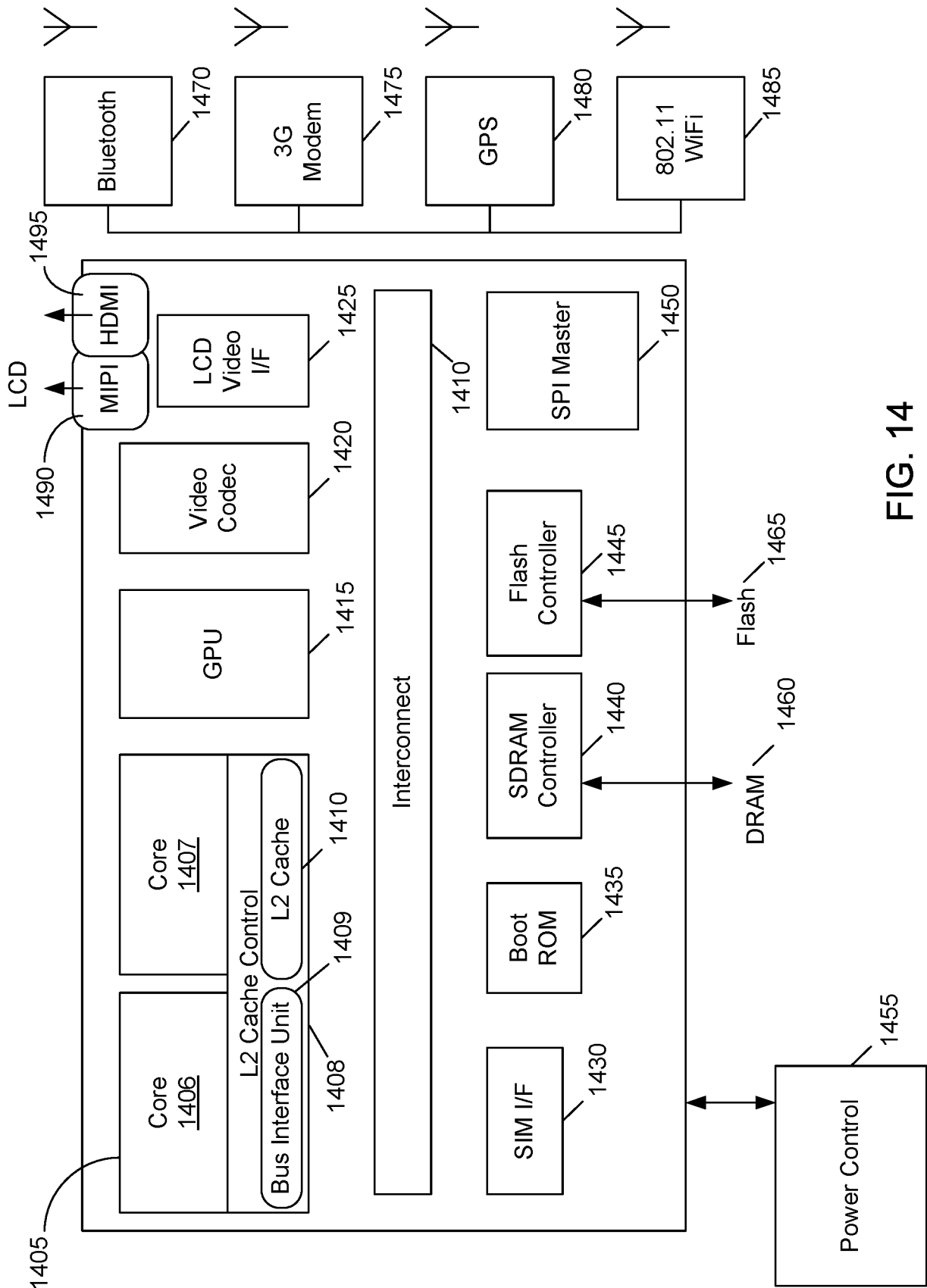


FIG. 14

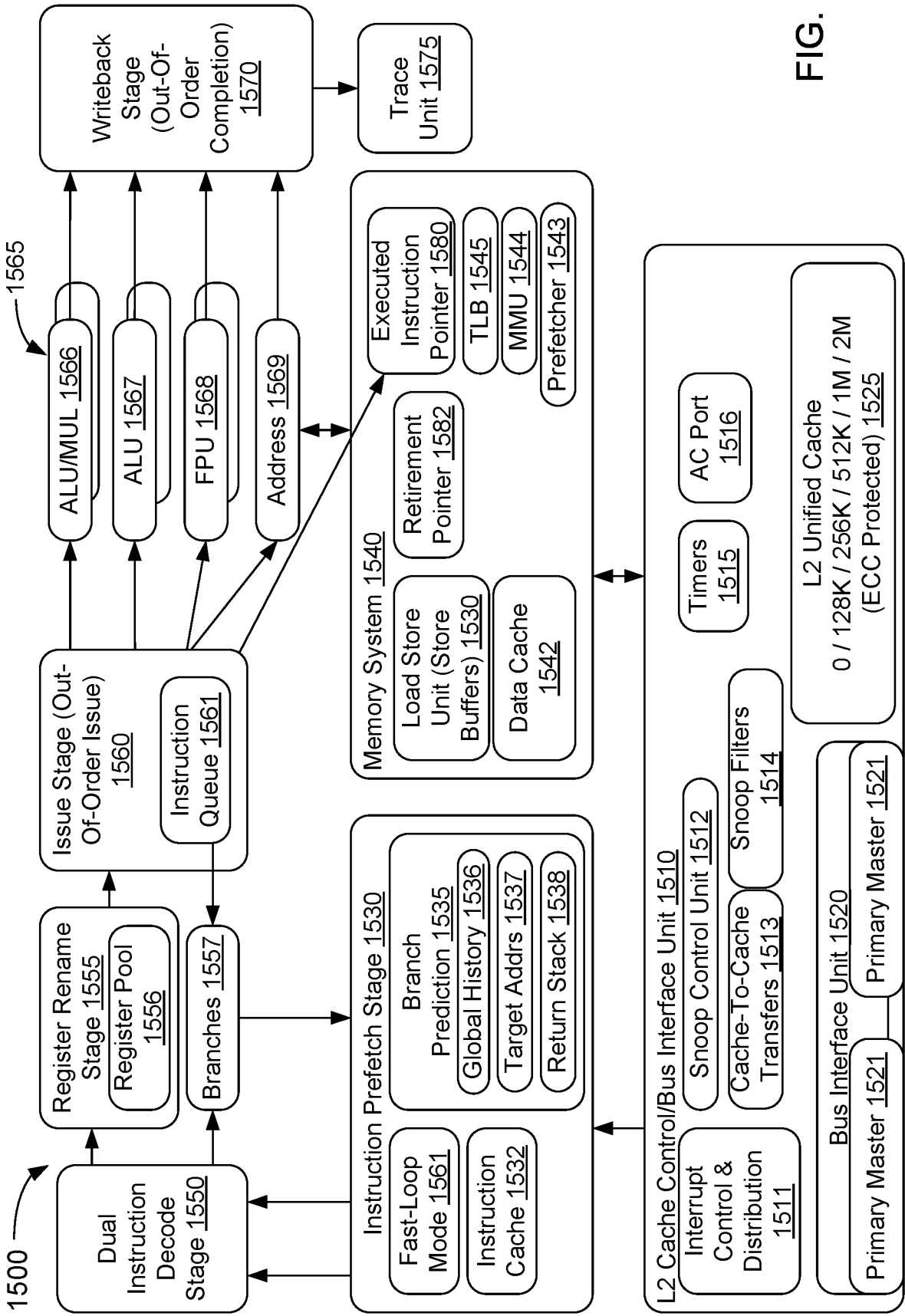


FIG. 15

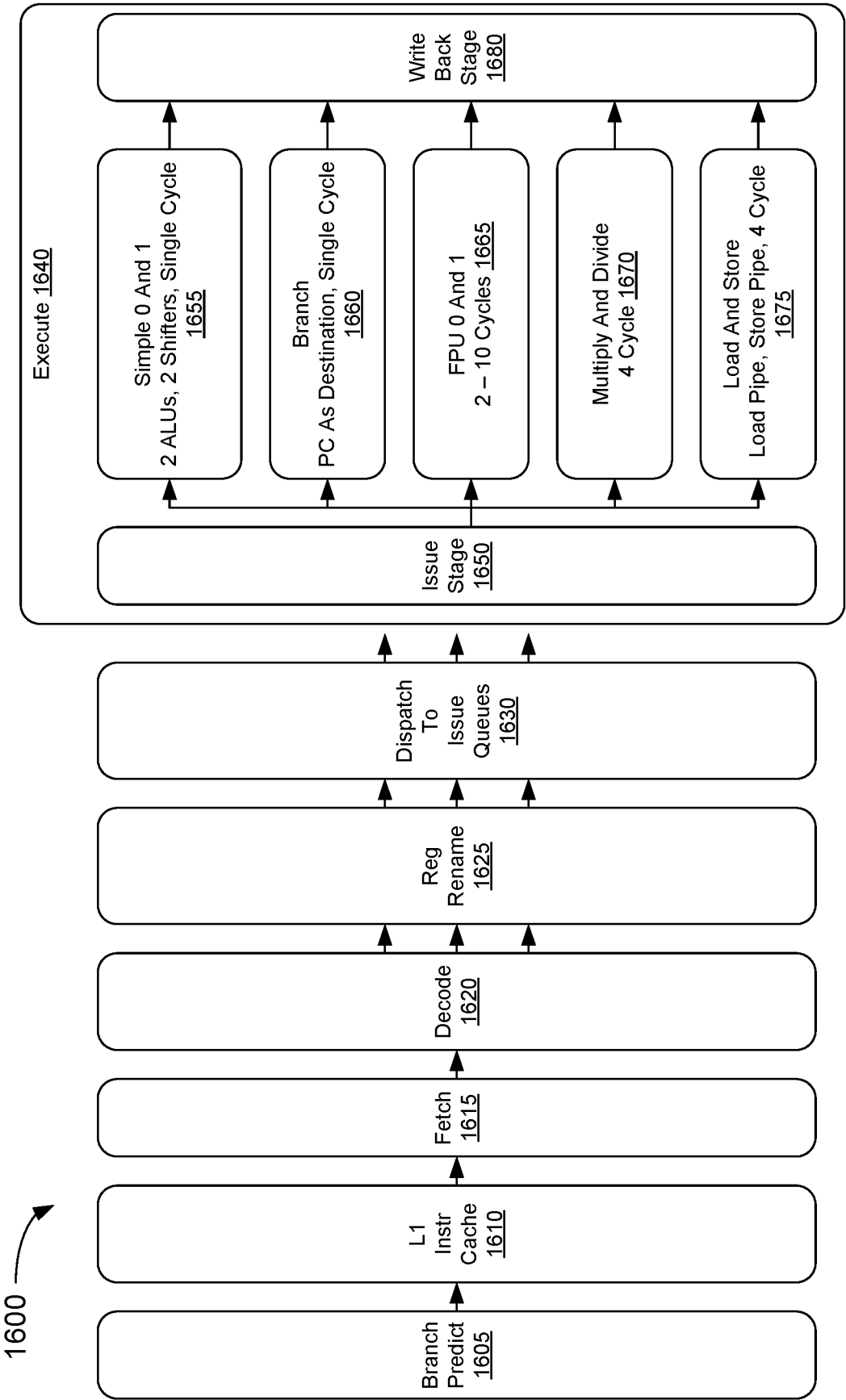


FIG. 16

23/29

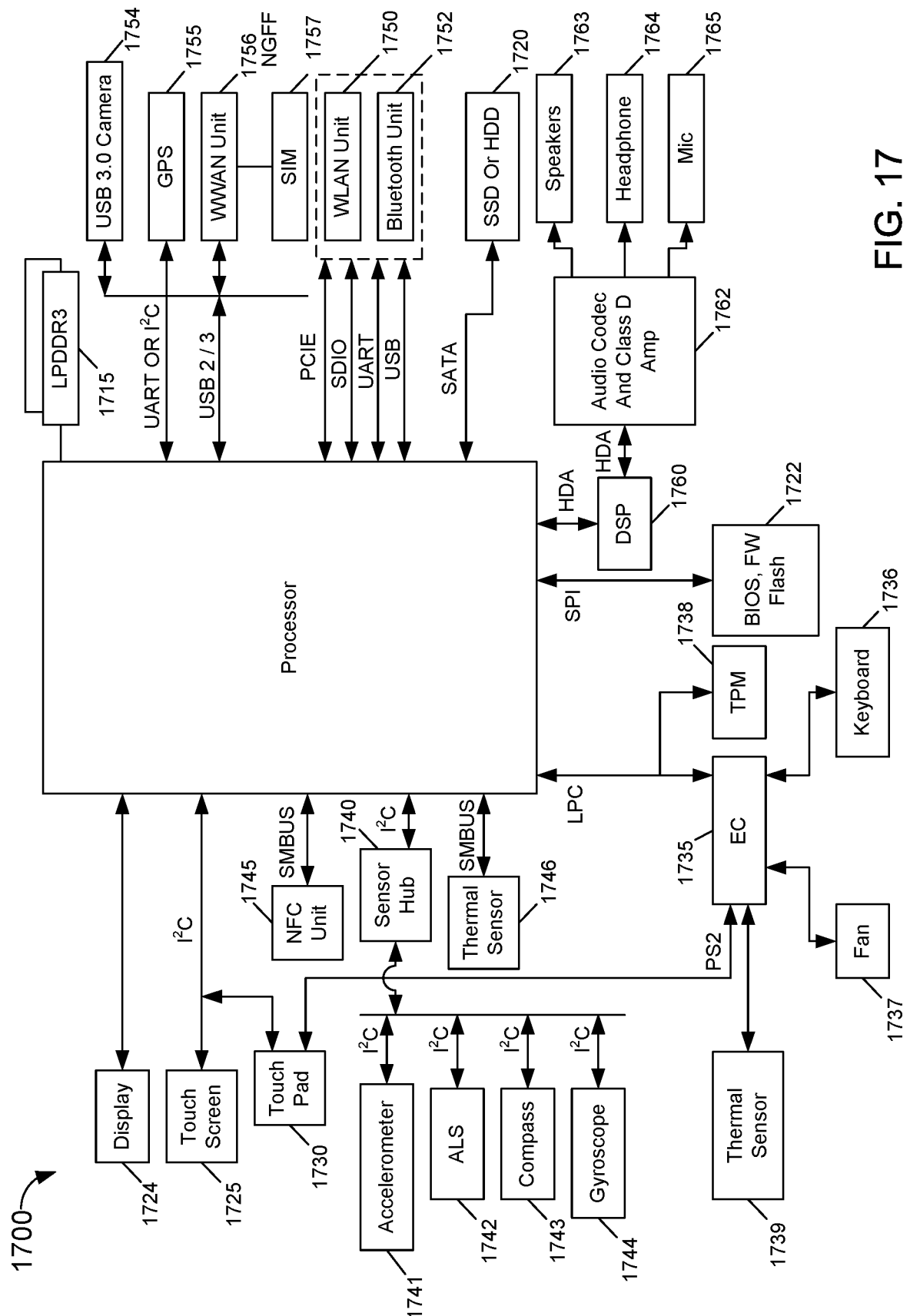


FIG. 17

24/29

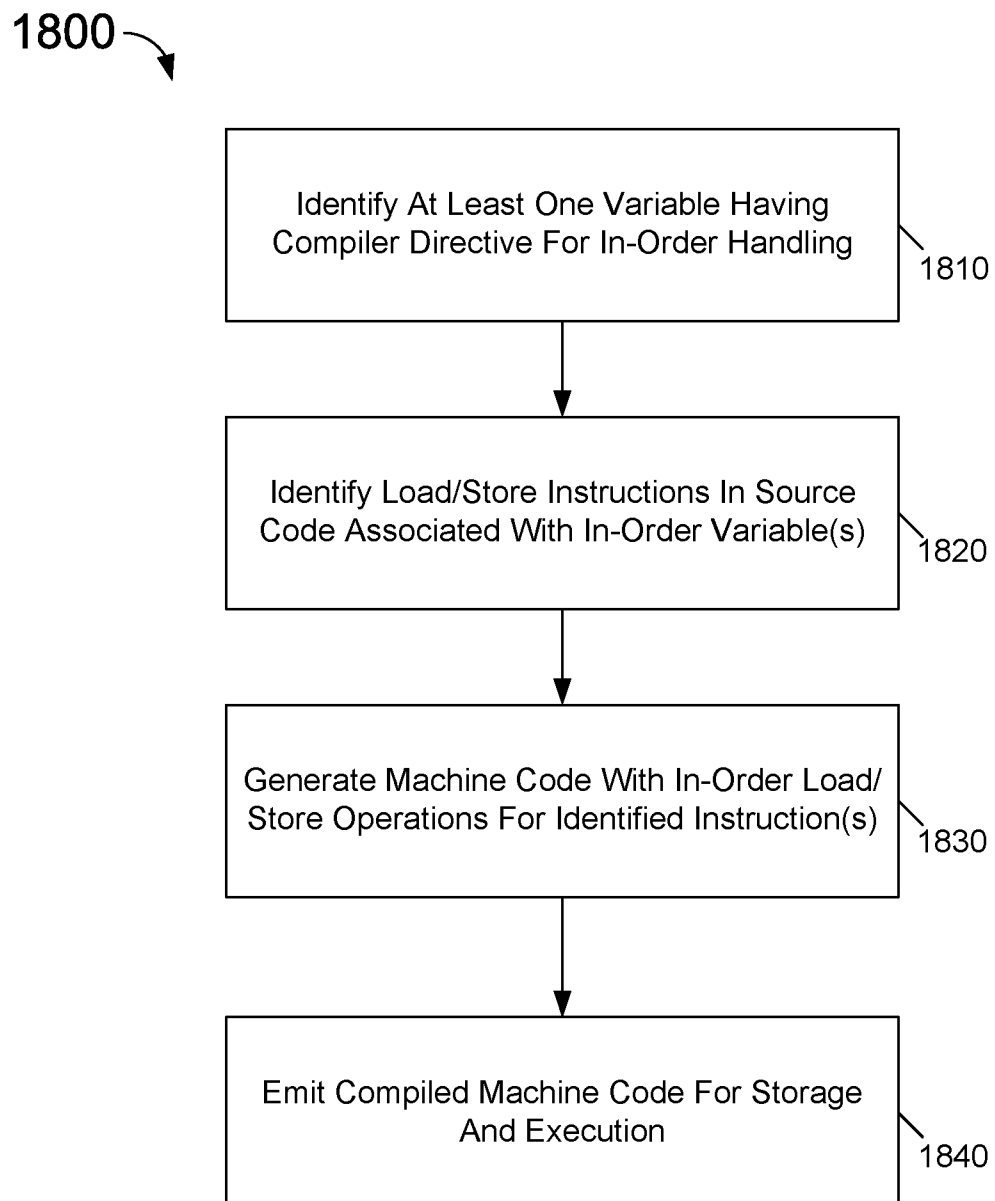


FIG. 18

25/29

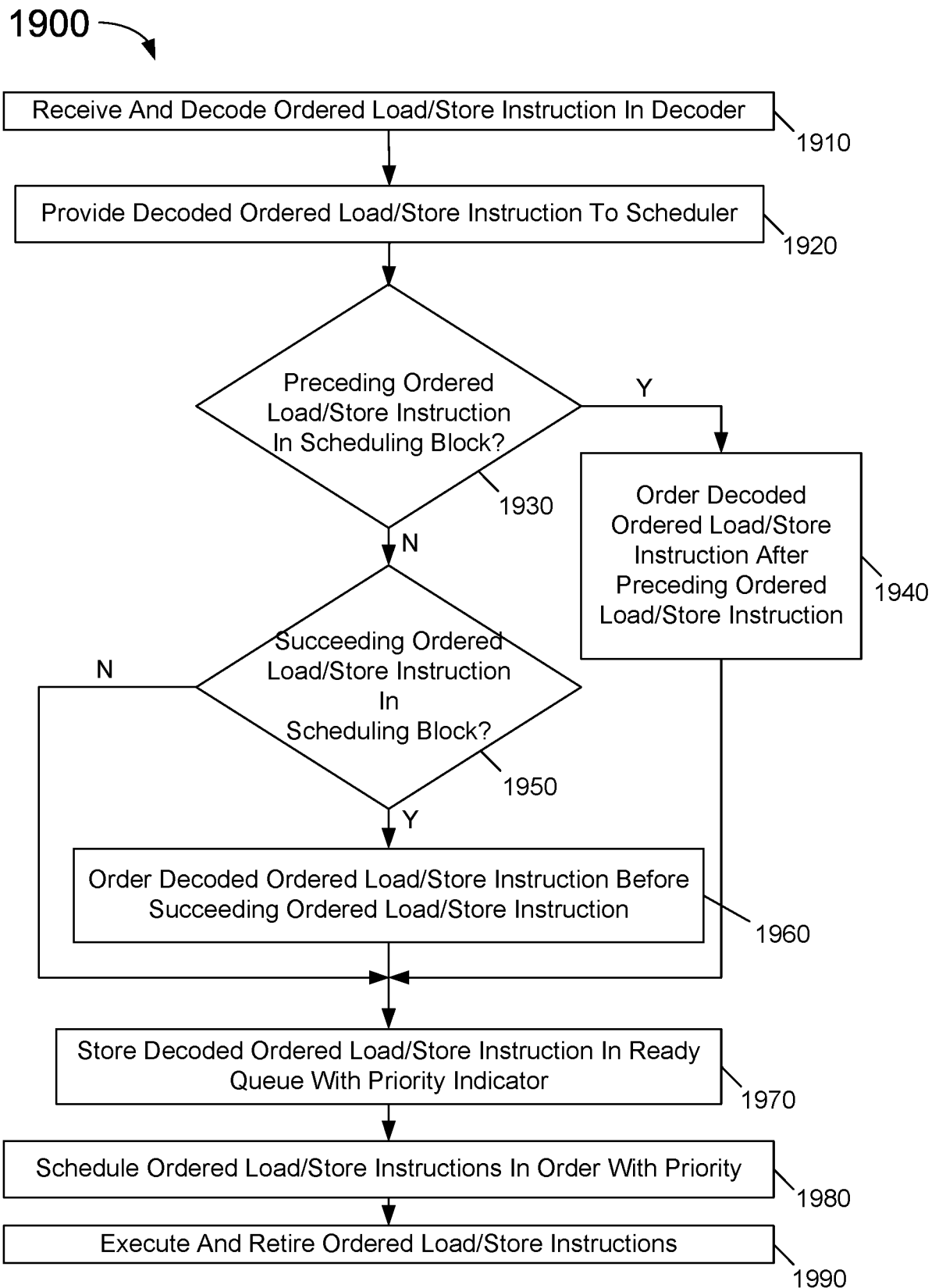


FIG. 19

26/29

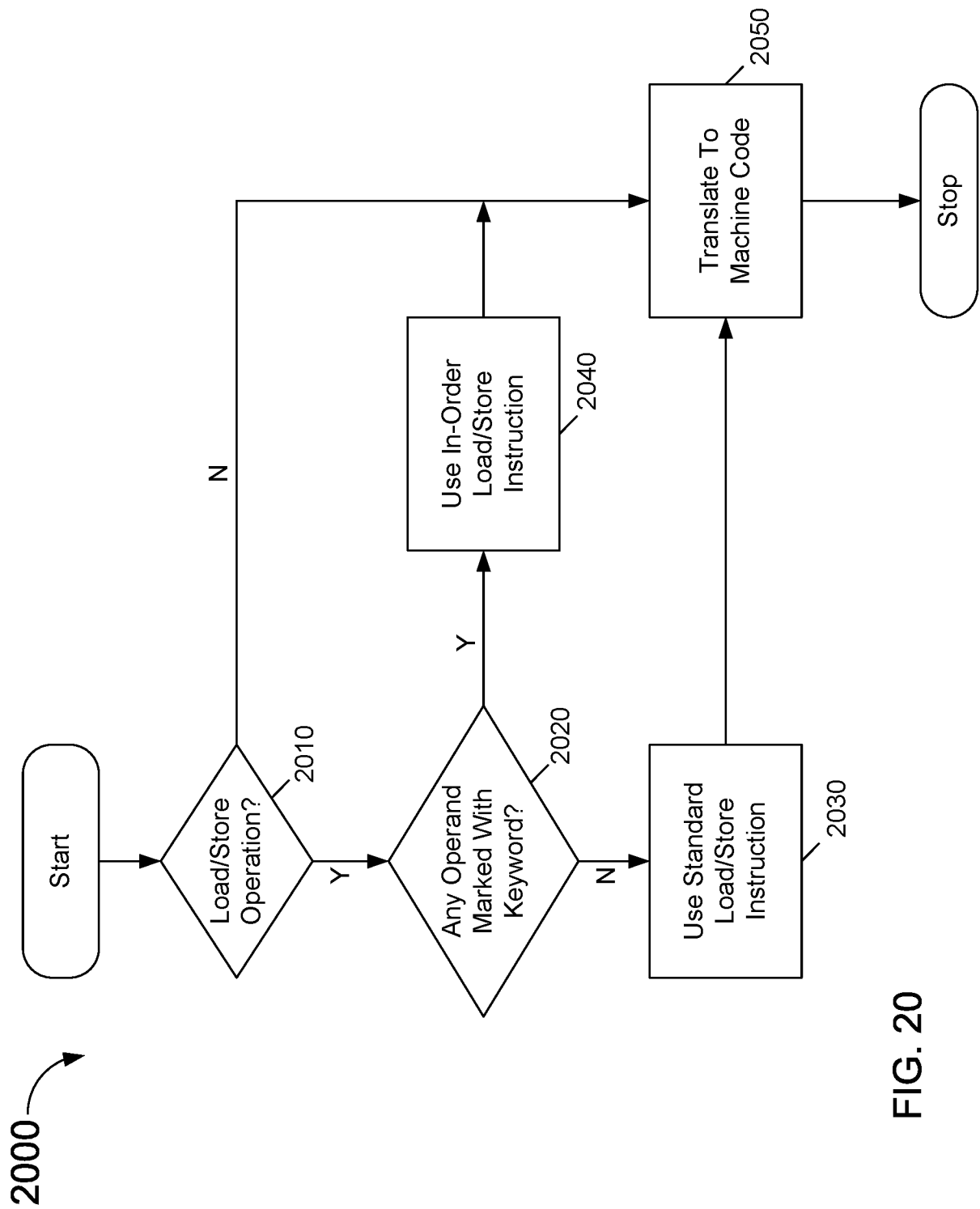


FIG. 20

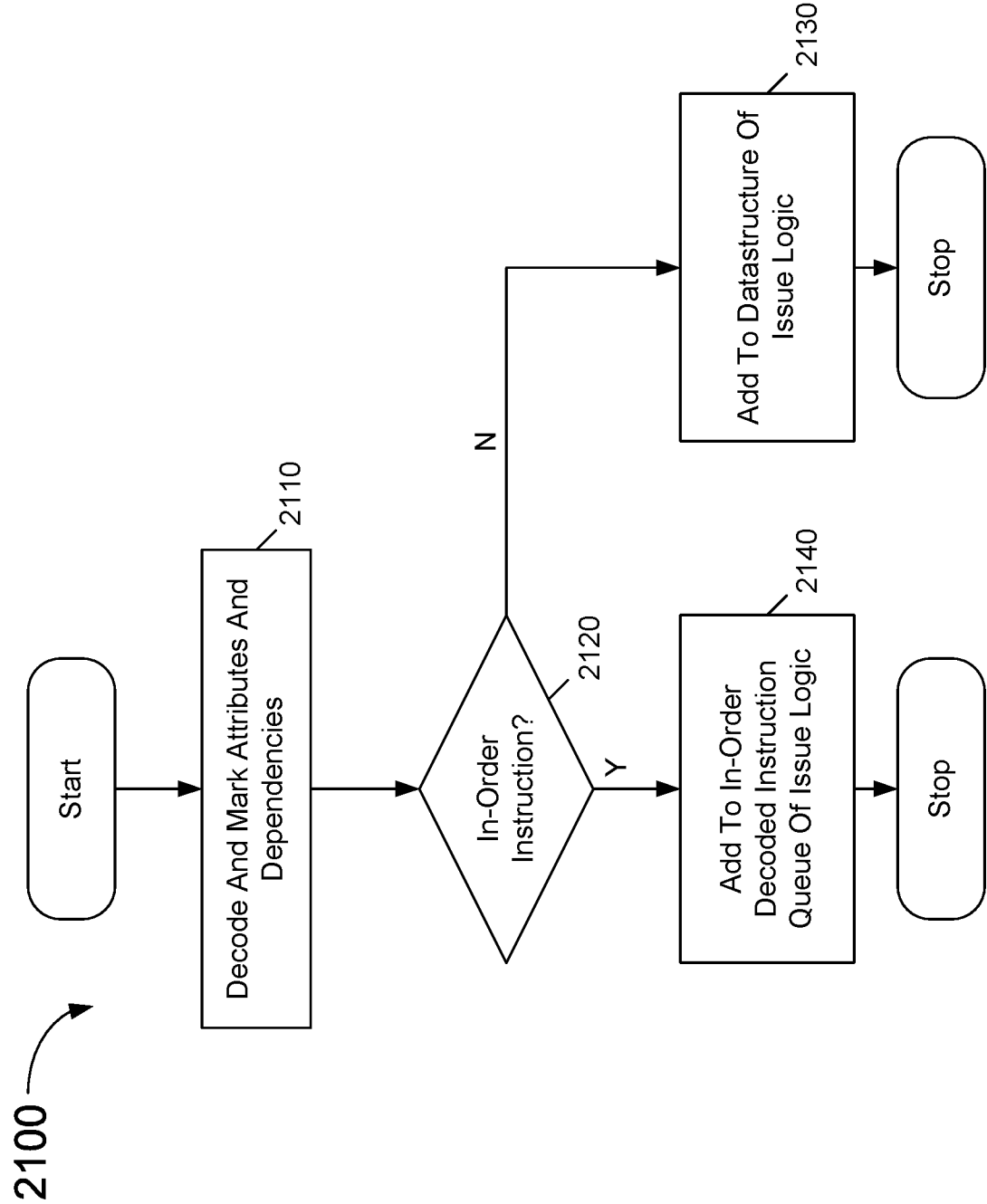


FIG. 21

28/29

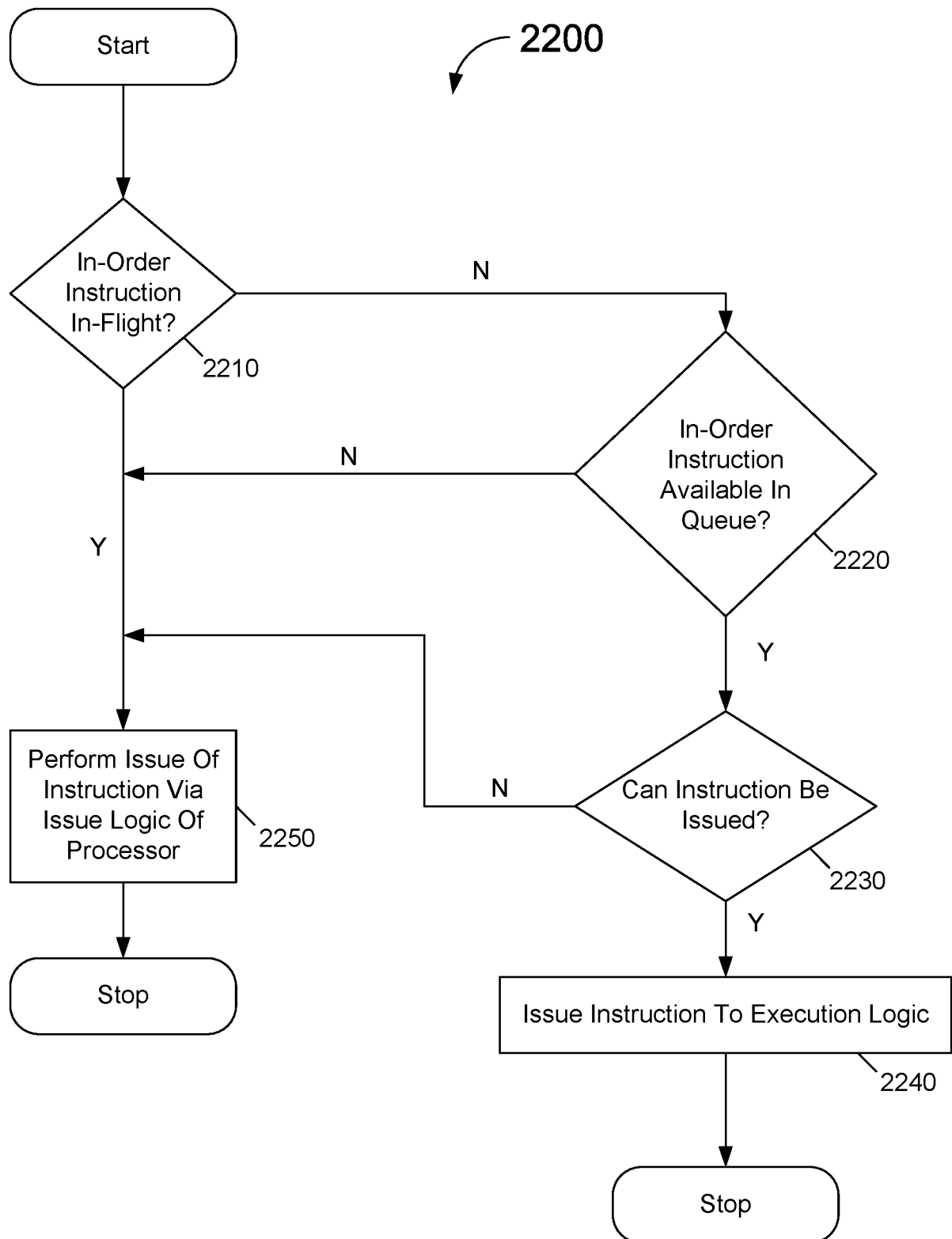


FIG. 22

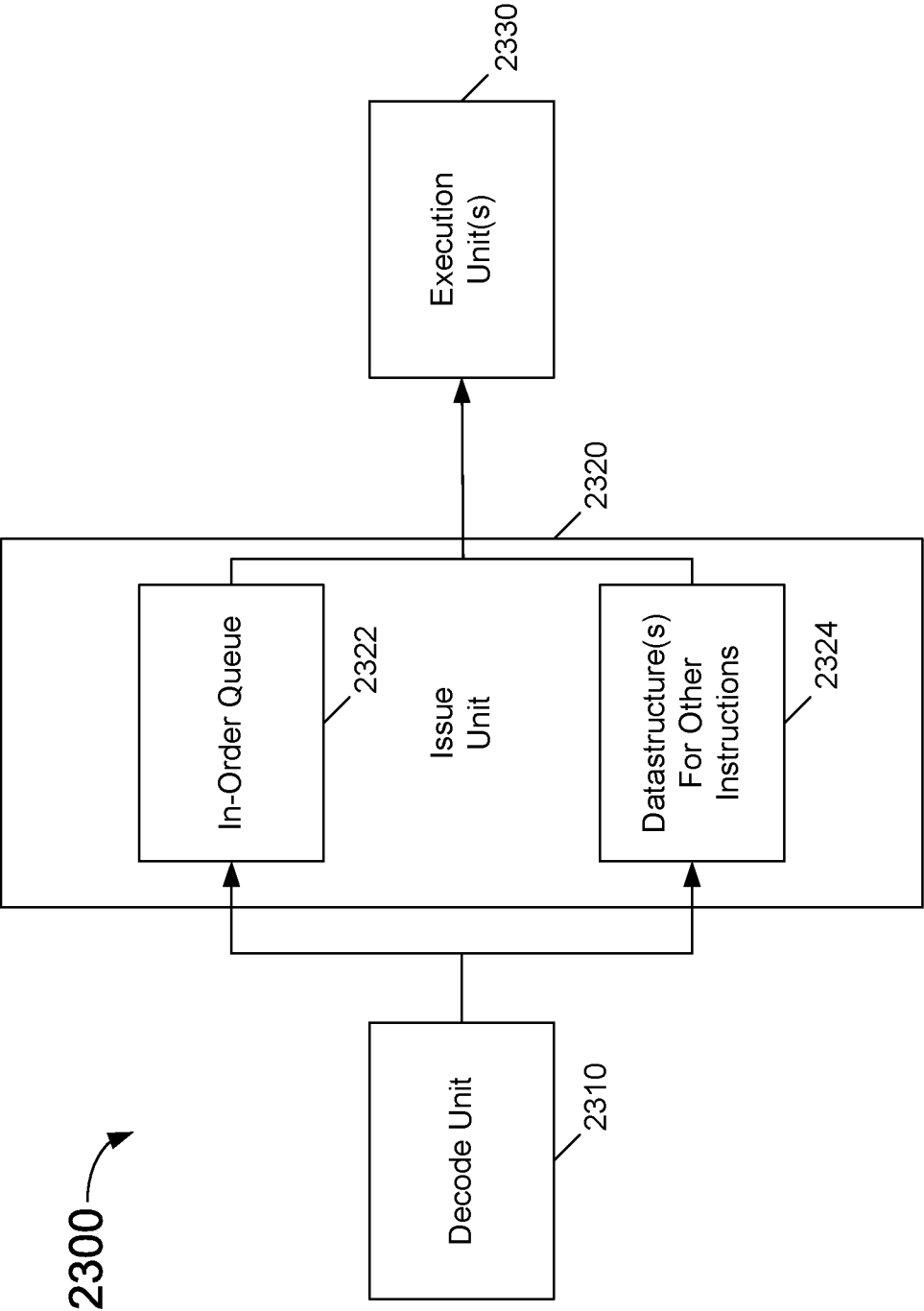


FIG. 23

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2016/054072**A. CLASSIFICATION OF SUBJECT MATTER****G06F 9/38(2006.01)i**

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 9/38; G06F 9/30; G06F 15/00; G06F 9/46; G06F 12/00; G06F 9/312

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

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

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

eKOMPASS(KIPO internal) & keywords: out-of-order, in-order, load/store instruction, decode, machine code, memory mapped input/output**C. DOCUMENTS CONSIDERED TO BE RELEVANT**

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 6192465 B1 (JAMES S. ROBERTS) 20 February 2001 See column 3, lines 16-52; column 4, line 23 - column 11, line 60; column 12, line 65 - column 14, line 14; claims 1-2, 7, 13; and figures 1-2B, 6.	1-4,8-10
Y		5-7,11-25
Y	US 2012-0124586 A1 (DINIEL HOPPER et al.) 17 May 2012 See paragraphs [0021]-[0024], [0026]-[0051]; claims 1-20; and figures 1, 3-4.	5-7,11-25
A	US 2013-0046956 A1 (THANG M. TRAN et al.) 21 February 2013 See paragraphs [0040]-[0047]; and figures 5-7.	1-25
A	US 2009-0013135 A1 (DOUGLAS C. BURGER et al.) 08 January 2009 See paragraphs [0028]-[0041]; claim 1; and figures 2-3.	1-25
A	US 5467473 A (JAMES A. KAHLE et al.) 14 November 1995 See column 5, line 46 - column 7, line 18; and figures 2-4.	1-25



Further documents are listed in the continuation of Box C.



See patent family annex.

* Special categories of cited documents:

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

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

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

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

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

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

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

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

"&" document member of the same patent family

Date of the actual completion of the international search

27 December 2016 (27.12.2016)

Date of mailing of the international search report

02 January 2017 (02.01.2017)

Name and mailing address of the ISA/KR

International Application Division

Korean Intellectual Property Office

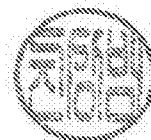
189 Cheongsa-ro, Seo-gu, Daejeon, 35208, Republic of Korea

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/US2016/054072

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 6192465 B1	20/02/2001	None	
US 2012-0124586 A1	17/05/2012	US 8769539 B2	01/07/2014
US 2013-0046956 A1	21/02/2013	US 2013-0046957 A1	21/02/2013
		US 8966229 B2	24/02/2015
		US 9110656 B2	18/08/2015
US 2009-0013135 A1	08/01/2009	US 8447911 B2	21/05/2013
		WO 2009-006595 A1	08/01/2009
US 05467473 A	14/11/1995	EP 0605869 A1	13/07/1994
		JP 07-160501 A	23/06/1995
		JP 2597811 B2	09/04/1997