



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

(21) International Application Number:
PCT/US2017/038204

(22) International Filing Date:
20 June 2017 (20.06.2017)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
62/353,937 23 June 2016 (23.06.2016) US
15/224,587 31 July 2016 (31.07.2016) US

(71) Applicant: **MICROSOFT TECHNOLOGY LICENSING, LLC** [US/US]; One Microsoft Way, Redmond, Washington 98052-6399 (US).

(72) Inventors: **SMITH, Aaron L.**; Microsoft Technology Licensing, LLC, One Microsoft Way, Redmond, Washington 98052-6399 (US). **GRAY, Jan S.**; Microsoft Technology Licensing, LLC, One Microsoft Way, Redmond, Washington 98052-6399 (US).

(74) Agent: **MINHAS, Sandip** et al.; Microsoft Technology Licensing, LLC, One Microsoft Way, Redmond, Washington 98052-6399 (US).

(81) Designated States (unless otherwise indicated, for every kind of national protection available): AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA,

(54) Title: LOAD-STORE QUEUE FOR BLOCK-BASED PROCESSOR

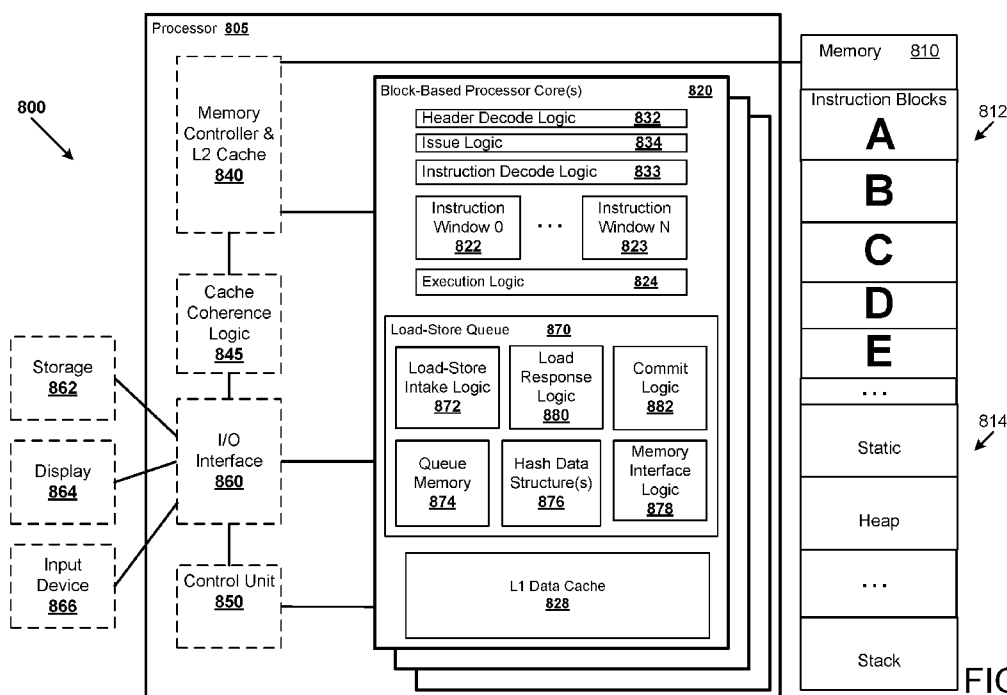


FIG. 8

(57) Abstract: Technology related to load-store queues for block-based processor architectures is disclosed. In one example of the disclosed technology, a processor includes issue logic and a load-store buffer. The issue logic can be configured to issue load and store instructions out of program order. Each of the load and store instructions can include an identifier specifying a relative program order of the respective instruction. The load-store buffer can be configured to enqueue the issued load and store instructions; generate hash values for addresses of the load and store instructions; and update a hash data structure using the generated hash values for the issued store instructions as an index of the hash data structure. For the load instructions, the hash data structure can be searched to generate load response data for the load instructions. The generated load response data can be forwarded to an execution unit of the processor.

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

Published:

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

LOAD-STORE QUEUE FOR BLOCK-BASED PROCESSOR

BACKGROUND

[001] Microprocessors have benefitted from continuing gains in transistor count, integrated circuit cost, manufacturing capital, clock frequency, and energy efficiency due to continued transistor scaling predicted by Moore's law, with little change in associated processor Instruction Set Architectures (ISAs). However, the benefits realized from photolithographic scaling, which drove the semiconductor industry over the last 40 years, are slowing or even reversing. Reduced Instruction Set Computing (RISC) architectures have been the dominant paradigm in processor design for many years. Out-of-order superscalar implementations have not exhibited sustained improvement in area or performance. Accordingly, there is ample opportunity for improvements in processor ISAs to extend performance improvements.

SUMMARY

[002] Methods, systems, apparatus, and computer-readable storage devices are disclosed for a load-store queue of a block-based processor instruction set architecture (BB-ISA). The described techniques and tools can potentially improve processor performance and can be implemented separately, or in various combinations with each other. As will be described more fully below, the described techniques and tools can be implemented in a digital signal processor, microprocessor, application-specific integrated circuit (ASIC), a soft processor (e.g., a microprocessor core implemented in a field programmable gate array (FPGA) using reconfigurable logic), programmable logic, or other suitable logic circuitry. As will be readily apparent to one of ordinary skill in the art, the disclosed technology can be implemented in various computing platforms, including, but not limited to, servers, mainframes, cellphones, smartphones, PDAs, handheld devices, handheld computers, touch screen tablet devices, tablet computers, wearable computers, and laptop computers.

[003] In some examples of the disclosed technology, a processor includes issue logic and a load-store buffer. The issue logic can be configured to issue load and store instructions out of program order. Each of the load and store instructions can include an identifier specifying a relative program order of the respective instruction. The load-store buffer can be configured to enqueue the issued load and store instructions; generate hash values for addresses of the load and store instructions; and update a hash data structure using the generated hash values of the store instructions as an index of the hash table. For the load

instructions, the hash data structure can be searched to generate load response data for the load instructions. The generated load response data can be forwarded to an execution unit of the processor.

[004] This Summary is provided to introduce a selection of concepts in a simplified form that are further described below in the Detailed Description. This Summary is not intended to identify key features or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter. The foregoing and other objects, features, and advantages of the disclosed subject matter will become more apparent from the following detailed description, which proceeds with reference to the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

[005] FIG. 1 illustrates a block-based processor including multiple processor cores, as can be used in some examples of the disclosed technology.

[006] FIG. 2 illustrates a block-based processor core, as can be used in some examples of the disclosed technology.

[007] FIG. 3 illustrates a number of instruction blocks, according to certain examples of disclosed technology.

[008] FIG. 4 illustrates portions of source code and respective instruction blocks.

[009] FIG. 5 illustrates block-based processor headers and instructions, as can be used in some examples of the disclosed technology.

[010] FIG. 6 is a flowchart illustrating an example of a progression of states of a processor core of a block-based processor.

[011] FIG. 7 illustrates an example snippet of instructions of a program for a block-based processor.

[012] FIG. 8 illustrates an example system including a load-store queue for executing an instruction block on a block-based processor core, as can be used in some examples of the disclosed technology.

[013] FIG. 9 is a flowchart illustrating an example method of processing instructions received by a load-store queue, as can be performed in some examples of the disclosed technology.

[014] FIG. 10 is a flowchart illustrating an example method of load response logic of a load-store queue, as can be performed in some examples of the disclosed technology.

[015] FIG. 11 illustrates example data structures that can be used by a load-store queue, as can be used in some examples of the disclosed technology.

[016] FIG. 12 is a flowchart illustrating an example method of commit logic of a load-store queue, as can be performed in some examples of the disclosed technology.

[017] FIGS. 13-14 are flowcharts illustrating example methods of executing an instruction block on a block-based processor core, as can be performed in some examples of the disclosed technology.

[018] FIG. 15 is a block diagram illustrating a suitable computing environment for implementing some embodiments of the disclosed technology.

[019] FIG. 16 is a block diagram outlining an example FPGA microarchitecture as can be used in some examples of the disclosed technology.

[020] FIG. 17 illustrates example reconfigurable logic in a reconfigurable logic block as can be used in certain examples of the disclosed technology.

DETAILED DESCRIPTION

I. General Considerations

[021] This disclosure is set forth in the context of representative embodiments that are not intended to be limiting in any way.

[022] As used in this application the singular forms “a,” “an,” and “the” include the plural forms unless the context clearly dictates otherwise. Additionally, the term “includes” means “comprises.” Further, the term “coupled” encompasses mechanical, electrical, magnetic, optical, as well as other practical ways of coupling or linking items together, and does not exclude the presence of intermediate elements between the coupled items. Furthermore, as used herein, the term “and/or” means any one item or combination of items in the phrase.

[023] The systems, methods, and apparatus described herein should not be construed as being limiting in any way. Instead, this disclosure is directed toward all novel and non-obvious features and aspects of the various disclosed embodiments, alone and in various combinations and subcombinations with one another. The disclosed systems, methods, and apparatus are not limited to any specific aspect or feature or combinations thereof, nor do the disclosed things and methods require that any one or more specific advantages be present or problems be solved. Furthermore, any features or aspects of the disclosed embodiments can be used in various combinations and subcombinations with one another.

[024] Although the operations of some of the disclosed methods are described in a particular, sequential order for convenient presentation, it should be understood that this manner of description encompasses rearrangement, unless a particular ordering is required by specific language set forth below. For example, operations described sequentially may

in some cases be rearranged or performed concurrently. Moreover, for the sake of simplicity, the attached figures may not show the various ways in which the disclosed things and methods can be used in conjunction with other things and methods. Additionally, the description sometimes uses terms like “produce,” “generate,” “display,” “receive,” “emit,” “verify,” “execute,” and “initiate” to describe the disclosed methods. These terms are high-level descriptions of the actual operations that are performed. The actual operations that correspond to these terms will vary depending on the particular implementation and are readily discernible by one of ordinary skill in the art.

[025] Theories of operation, scientific principles, or other theoretical descriptions presented herein in reference to the apparatus or methods of this disclosure have been provided for the purposes of better understanding and are not intended to be limiting in scope. The apparatus and methods in the appended claims are not limited to those apparatus and methods that function in the manner described by such theories of operation.

[026] Any of the disclosed methods can be implemented as computer-executable instructions stored on one or more computer-readable media (*e.g.*, computer-readable media, such as one or more optical media discs, volatile memory components (such as DRAM or SRAM), or nonvolatile memory components (such as hard drives)) and executed on a computer (*e.g.*, any commercially available computer, including smart phones or other mobile devices that include computing hardware). Any of the computer-executable instructions for implementing the disclosed techniques, as well as any data created and used during implementation of the disclosed embodiments, can be stored on one or more computer-readable media (*e.g.*, computer-readable storage media). The computer-executable instructions can be part of, for example, a dedicated software application or a software application that is accessed or downloaded via a web browser or other software application (such as a remote computing application). Such software can be executed, for example, on a single local computer (*e.g.*, with general-purpose and/or block-based processors executing on any suitable commercially available computer) or in a network environment (*e.g.*, via the Internet, a wide-area network, a local-area network, a client-server network (such as a cloud computing network), or other such network) using one or more network computers.

[027] For clarity, only certain selected aspects of the software-based implementations are described. Other details that are well known in the art are omitted. For example, it should be understood that the disclosed technology is not limited to any specific computer

language or program. For instance, the disclosed technology can be implemented by software written in C, C++, Java, or any other suitable programming language. Likewise, the disclosed technology is not limited to any particular computer or type of hardware. Certain details of suitable computers and hardware are well-known and need not be set forth in detail in this disclosure.

[028] Furthermore, any of the software-based embodiments (comprising, for example, computer-executable instructions for causing a computer to perform any of the disclosed methods) can be uploaded, downloaded, or remotely accessed through a suitable communication means. Such suitable communication means include, for example, the Internet, the World Wide Web, an intranet, software applications, cable (including fiber optic cable), magnetic communications, electromagnetic communications (including RF, microwave, and infrared communications), electronic communications, or other such communication means.

II. Introduction to the Disclosed Technologies

[029] Superscalar out-of-order microarchitectures employ substantial circuit resources to rename registers, schedule instructions in dataflow order, clean up after miss-speculation, and retire results in-order for precise exceptions. This includes expensive energy-consuming circuits, such as deep, many-ported register files, many-ported content-accessible memories (CAMs) for dataflow instruction scheduling wakeup, and many-wide bus multiplexers and bypass networks, all of which are resource intensive. For example, FPGA-based implementations of multi-read, multi-write-port random-access memories (RAMs) typically require a mix of replication, multi-cycle operation, clock doubling, bank interleaving, live-value tables, and other expensive techniques.

[030] The disclosed technologies can realize energy efficiency and/or performance enhancement through application of techniques including high instruction-level parallelism (ILP), out-of-order, superscalar execution, while avoiding substantial complexity and overhead in both processor hardware and associated software. In some examples of the disclosed technology, a block-based processor comprising multiple processor cores uses an Explicit Data Graph Execution (EDGE) ISA designed for area- and energy-efficient, high-ILP execution. In some examples, use of EDGE architectures and associated compilers finesses away much of the register renaming, CAMs, and complexity. In some examples, the respective cores of the block-based processor can store or cache fetched and decoded instructions that may be repeatedly executed, and the fetched and decoded instructions can be reused to potentially achieve reduced power

and/or increased performance.

[031] In certain examples of the disclosed technology, an EDGE ISA can eliminate the need for one or more complex architectural features, including register renaming, dataflow analysis, misspeculation recovery, and in-order retirement while supporting mainstream programming languages such as C and C++. In certain examples of the disclosed technology, a block-based processor executes a plurality of two or more instructions as an atomic block. Block-based instructions can be used to express semantics of program data flow and/or instruction flow in a more explicit fashion, allowing for improved compiler and processor performance. In certain examples of the disclosed technology, an explicit data graph execution instruction set architecture (EDGE ISA) includes information about program control flow that can be used to improve detection of improper control flow instructions, thereby increasing performance, saving memory resources, and/or and saving energy.

[032] In some examples of the disclosed technology, instructions organized within instruction blocks are fetched, executed, and committed atomically. Intermediate results produced by the instructions within an atomic instruction block are buffered locally until the instruction block is committed. When the instruction block is committed, updates to the visible architectural state resulting from executing the instructions of the instruction block are made visible to other instruction blocks. Instructions inside blocks execute in dataflow order, with intermediate results directly forwarded to the instructions which consume them, which reduces or eliminates using register renaming and provides power-efficient out-of-order execution. A compiler can be used to explicitly encode data dependencies through the ISA, reducing or eliminating burdening processor core control logic from rediscovering dependencies at runtime. Using predicated execution, intra-block branches can be converted to dataflow instructions, and dependencies, other than memory dependencies, can be limited to direct data dependencies. Disclosed target form encoding techniques allow instructions within a block to communicate their operands directly via operand buffers, reducing accesses to a power-hungry, multi-ported physical register files.

[033] Between instruction blocks, instructions can communicate using visible architectural state such as memory and registers. Thus, by utilizing a hybrid dataflow execution model, EDGE architectures can still support imperative programming languages and sequential memory semantics, but desirably also enjoy the benefits of out-of-order execution with near in-order power efficiency and complexity.

[034] Out-of-order processors can include hardware to track in-flight load and store

instructions for accessing memory. In-flight instructions can include loads that have issued but have not responded with load data and stores that have issued but have not been written back to the memory. Typical out-of-order processors may use a content addressable memory (CAM) to track the in-flight load and store instructions and to
5 identify potential data dependencies. While CAMs can achieve high performance, they can be expensive in terms of chip area and energy usage. For example, a CAM can have a different comparator for each entry in the CAM so that all entries in the CAM can be tested in a single cycle. Each of the different comparators consumes chip area and energy. In some chip implementation technologies, a CAM having many such entries may
10 consume a prohibitively large area and/or fraction of available logic resources.

[035] As disclosed herein, a processor can track the in-flight, out-of-order load and store instructions without using a CAM so that the processor can be made smaller and more energy efficient. In particular, the processor can include a load-store queue that enqueues the in-flight load and store instructions in a local buffer. One or more hash functions can
15 be used to generate hash values for addresses of respective load and store instructions to help correlate loads and stores so that queued stores occurring earlier in program order may forward their store data to loads occurring relatively later in program order. For example, a first hash function can be used to generate hash values for addresses of respective load and store instructions. For each new store instruction arriving at the load-
20 store queue, the hash value can be used to store a reference to the respective enqueued store instructions in a hash table. For each enqueued load instruction, the hash table can be searched for store instructions with a matching hash value of the respective load instructions and matching the address of the respective load instructions. Additionally or alternatively, a second hash function can be used to generate hash values for the addresses
25 of the respective load and store instructions. A Bloom filter data structure can track all of the hash values generated by the second hash function for each new store instruction arriving at the load-store queue. For each enqueued load instruction, the Bloom filter data structure can be analyzed to determine whether the hash values of the enqueued store instructions matches the hash value of the enqueued load instruction. If there is a match,
30 the local buffer can be searched for store instructions occurring earlier in program order with a matching hash value of the respective load instructions and matching the address of the respective load instructions. The results of the search(es) can be used to generate response data for the load instruction and the response data can be forwarded to an execution unit of the processor. As will be readily understood to one of ordinary skill in

the relevant art, a spectrum of implementations of the disclosed technology are possible with various area, performance, and power tradeoffs.

III. Example Block-Based Processor

[036] FIG. 1 is a block diagram 10 of a block-based processor 100 as can be
5 implemented in some examples of the disclosed technology. The processor 100 is configured to execute atomic blocks of instructions according to an instruction set architecture (ISA), which describes a number of aspects of processor operation, including a register model, a number of defined operations performed by block-based instructions, a memory model, interrupts, and other architectural features. The block-based processor
10 includes a plurality of processing cores 110, including a processor core 111.

[037] As shown in FIG. 1, the processor cores are connected to each other via core interconnect 120. The core interconnect 120 carries data and control signals between individual ones of the cores 110, a memory interface 140, and an input/output (I/O) interface 145. The core interconnect 120 can transmit and receive signals using electrical,
15 optical, magnetic, or other suitable communication technology and can provide communication connections arranged according to a number of different topologies, depending on a particular desired configuration. For example, the core interconnect 120 can have a crossbar, a bus, a point-to-point bus, or other suitable topology. In some examples, any one of the cores 110 can be connected to any of the other cores, while in
20 other examples, some cores are only connected to a subset of the other cores. For example, each core may only be connected to a nearest 4, 8, or 20 neighboring cores. The core interconnect 120 can be used to transmit input/output data to and from the cores, as well as transmit control signals and other information signals to and from the cores. For example, each of the cores 110 can receive and transmit semaphores that indicate the execution
25 status of instructions currently being executed by each of the respective cores. In some examples, the core interconnect 120 is implemented as wires connecting the cores 110, and memory system, while in other examples, the core interconnect can include circuitry for multiplexing data signals on the interconnect wire(s), switch and/or routing components, including active signal drivers and repeaters, or other suitable circuitry. In
30 some examples of the disclosed technology, signals transmitted within and to/from the processor 100 are not limited to full swing electrical digital signals, but the processor can be configured to include differential signals, pulsed signals, or other suitable signals for transmitting data and control signals.

[038] In the example of FIG. 1, the memory interface 140 of the processor includes

interface logic that is used to connect to additional memory, for example, memory located on another integrated circuit besides the processor 100. As shown in FIG. 1 an external memory system 150 includes an L2 cache 152 and main memory 155. In some examples the L2 cache can be implemented using static RAM (SRAM) and the main memory 155 can be implemented using dynamic RAM (DRAM). In some examples the memory system 150 is included on the same integrated circuit as the other components of the processor 100. In some examples, the memory interface 140 includes a direct memory access (DMA) controller allowing transfer of blocks of data in memory without using register file(s) and/or the processor 100. In some examples, the memory interface 140 can include a memory management unit (MMU) for managing and allocating virtual memory, expanding the available main memory 155.

[039] The I/O interface 145 includes circuitry for receiving and sending input and output signals to other components, such as hardware interrupts, system control signals, peripheral interfaces, co-processor control and/or data signals (*e.g.*, signals for a graphics processing unit, floating point coprocessor, physics processing unit, digital signal processor, or other co-processing components), clock signals, semaphores, or other suitable I/O signals. The I/O signals may be synchronous or asynchronous. In some examples, all or a portion of the I/O interface is implemented using memory-mapped I/O techniques in conjunction with the memory interface 140.

[040] The block-based processor 100 can also include a control unit 160. The control unit can communicate with the processing cores 110, the I/O interface 145, and the memory interface 140 via the core interconnect 120 or a side-band interconnect (not shown). The control unit 160 supervises operation of the processor 100. Operations that can be performed by the control unit 160 can include allocation and de-allocation of cores for performing instruction processing, control of input data and output data between any of the cores, register files, the memory interface 140, and/or the I/O interface 145, modification of execution flow, and verifying target location(s) of branch instructions, instruction headers, and other changes in control flow. The control unit 160 can also process hardware interrupts, and control reading and writing of special system registers, for example the program counter stored in one or more register file(s). In some examples of the disclosed technology, the control unit 160 is at least partially implemented using one or more of the processing cores 110, while in other examples, the control unit 160 is implemented using a non-block-based processing core (*e.g.*, a general-purpose RISC processing core coupled to memory). In some examples, the control unit 160 is

implemented at least in part using one or more of: hardwired finite state machines, programmable microcode, programmable gate arrays, or other suitable control circuits. In alternative examples, control unit functionality can be performed by one or more of the cores 110.

- 5 **[041]** The control unit 160 includes a scheduler that is used to allocate instruction blocks to the processor cores 110. As used herein, scheduler allocation refers to hardware for directing operation of instruction blocks, including initiating instruction block mapping, fetching, decoding, execution, committing, aborting, idling, and refreshing an instruction block. In some examples, the hardware receives signals generated using computer-
10 executable instructions to direct operation of the instruction scheduler. Processor cores 110 are assigned to instruction blocks during instruction block mapping. The recited stages of instruction operation are for illustrative purposes, and in some examples of the disclosed technology, certain operations can be combined, omitted, separated into multiple operations, or additional operations added.
- 15 **[042]** The block-based processor 100 also includes a clock generator 170, which distributes one or more clock signals to various components within the processor (*e.g.*, the cores 110, interconnect 120, memory interface 140, and I/O interface 145). In some examples of the disclosed technology, all of the components share a common clock, while in other examples different components use a different clock, for example, a clock signal
20 having differing clock frequencies. In some examples, a portion of the clock is gated to allow power savings when some of the processor components are not in use. In some examples, the clock signals are generated using a phase-locked loop (PLL) to generate a signal of fixed, constant frequency and duty cycle. Circuitry that receives the clock signals can be triggered on a single edge (*e.g.*, a rising edge) while in other examples, at least
25 some of the receiving circuitry is triggered by rising and falling clock edges. In some examples, the clock signal can be transmitted optically or wirelessly.

IV. Example Block-Based Processor Core

- [043]** FIG. 2 is a block diagram 200 further detailing an example microarchitecture for the block-based processor 100, and in particular, an instance of one of the block-based
30 processor cores (processor core 111), as can be used in certain examples of the disclosed technology. For ease of explanation, the exemplary block-based processor core 111 is illustrated with five stages: instruction fetch (IF), decode (DC), operand fetch, execute (EX), and memory/data access (LS). However, it will be readily understood by one of ordinary skill in the relevant art that modifications to the illustrated microarchitecture,

such as adding/removing stages, adding/removing units that perform operations, and other implementation details can be modified to suit a particular application for a block-based processor.

[044] In some examples of the disclosed technology, the processor core 111 can be used
5 to execute and commit an instruction block of a program. An instruction block is an atomic collection of block-based-processor instructions that includes an instruction block header and a plurality of instructions. As will be discussed further below, the instruction block header can include information describing an execution mode of the instruction block and information that can be used to further define semantics of one or more of the
10 plurality of instructions within the instruction block. Depending on the particular ISA and processor hardware used, the instruction block header can also be used, during execution of the instructions, to improve performance of executing an instruction block by, for example, allowing for early fetching of instructions and/or data, improved branch prediction, speculative execution, improved energy efficiency, and improved code
15 compactness.

[045] The instructions of the instruction block can be dataflow instructions that explicitly encode relationships between producer-consumer instructions of the instruction block. In particular, an instruction can communicate a result directly to a targeted instruction through an operand buffer that is reserved only for the targeted instruction. The
20 intermediate results stored in the operand buffers are generally not visible to cores outside of the executing core because the block-atomic execution model only passes final results between the instruction blocks. The final results from executing the instructions of the atomic instruction block are made visible outside of the executing core when the instruction block is committed. Thus, the visible architectural state generated by each
25 instruction block can appear as a single transaction outside of the executing core, and the intermediate results are typically not observable outside of the executing core.

[046] As shown in FIG. 2, the processor core 111 includes a control unit 205, which can receive control signals from other cores and generate control signals to regulate core operation and schedules the flow of instructions within the core using an instruction
30 scheduler 206. The control unit 205 can include state access logic 207 for examining core status and/or configuring operating modes of the processor core 111. The control unit 205 can include execution control logic 208 for generating control signals during one or more operating modes of the processor core 111. Operations that can be performed by the control unit 205 and/or instruction scheduler 206 can include allocation and de-allocation

of cores for performing instruction processing, control of input data and output data between any of the cores, register files, the memory interface 140, and/or the I/O interface 145. The control unit 205 can also process hardware interrupts, and control reading and writing of special system registers, for example the program counter stored in one or more register file(s). In other examples of the disclosed technology, the control unit 205 and/or instruction scheduler 206 are implemented using a non-block-based processing core (e.g., a general-purpose RISC processing core coupled to memory). In some examples, the control unit 205, instruction scheduler 206, state access logic 207, and/or execution control logic 208 are implemented at least in part using one or more of: hardwired finite state machines, programmable microcode, programmable gate arrays, or other suitable control circuits.

[047] The control unit 205 can decode the instruction block header to obtain information about the instruction block. For example, execution modes of the instruction block can be specified in the instruction block header through various execution flags. The decoded execution mode can be stored in registers of the execution control logic 208. Based on the execution mode, the execution control logic 208 can generate control signals to regulate core operation and schedule the flow of instructions within the core 111, such as by using the instruction scheduler 206. For example, during a default execution mode, the execution control logic 208 can sequence the instructions of one or more instruction blocks executing on one or more instruction windows (e.g., 210, 211) of the processor core 111. Specifically, each of the instructions can be sequenced through the instruction fetch, decode, operand fetch, execute, and memory/data access stages so that the instructions of an instruction block can be pipelined and executed in parallel. The instructions are ready to execute when their operands are available, and the instruction scheduler 206 can select the order in which to execute the instructions.

[048] The state access logic 207 can include an interface for other cores and/or a processor-level control unit (such as the control unit 160 of FIG. 1) to communicate with and access state of the core 111. For example, the state access logic 207 can be connected to a core interconnect (such as the core interconnect 120 of FIG. 1) and the other cores can communicate via control signals, messages, reading and writing registers, and the like.

[049] The state access logic 207 can include control state registers or other logic for modifying and/or examining modes and/or status of an instruction block and/or core status. As an example, the core status can indicate whether an instruction block is mapped to the core 111 or an instruction window (e.g., instruction windows 210, 211) of the core

111, whether an instruction block is resident on the core 111, whether an instruction block is executing on the core 111, whether the instruction block is ready to commit, whether the instruction block is performing a commit, and whether the instruction block is idle. As another example, the status of an instruction block can include a token or flag indicating the instruction block is the oldest instruction block executing and a flag indicating the instruction block is executing speculatively.

[050] The control state registers (CSRs) can be mapped to unique memory locations that are reserved for use by the block-based processor. For example, CSRs of the control unit 160 (FIG. 1) can be assigned to a first range of addresses, CSRs of the memory interface 140 (FIG. 1) can be assigned to a second range of addresses, a first processor core can be assigned to a third range of addresses, a second processor core can be assigned to a fourth range of addresses, and so forth. In one embodiment, the CSRs can be accessed using general purpose memory read and write instructions of the block-based processor. Additionally or alternatively, the CSRs can be accessed using specific read and write instructions (e.g., the instructions have opcodes different from the memory read and write instructions) for the CSRs. Thus, one core can examine the configuration state of a different core by reading from an address corresponding to the different core's CSRs. Similarly, one core can modify the configuration state of a different core by writing to an address corresponding to the different core's CSRs. Additionally or alternatively, the CSRs can be accessed by shifting commands into the state access logic 207 through serial scan chains. In this manner, one core can examine the state access logic 207 of a different core and one core can modify the state access logic 207 or modes of a different core.

[051] Each of the instruction windows 210 and 211 can receive instructions and data from one or more of input ports 220, 221, and 222 which connect to an interconnect bus and instruction cache 227, which in turn is connected to the instruction decoders 228 and 229. Additional control signals can also be received on an additional input port 225. Each of the instruction decoders 228 and 229 decodes instructions for an instruction block and stores the decoded instructions within a memory store 215 and 216 located in each respective instruction window 210 and 211.

[052] The processor core 111 further includes a register file 230 coupled to an L1 (level one) cache 235. The register file 230 stores data for registers defined in the block-based processor architecture, and can have one or more read ports and one or more write ports. For example, a register file may include two or more write ports for storing data in the register file, as well as having a plurality of read ports for reading data from individual

registers within the register file. In some examples, a single instruction window (*e.g.*, instruction window 210) can access only one port of the register file at a time, while in other examples, the instruction window 210 can access one read port and one write port, or can access two or more read ports and/or write ports simultaneously. In some examples, the register file 230 can include 64 registers, each of the registers holding a word of 32 bits of data. (This application will refer to 32-bits of data as a word, unless otherwise specified.) In some examples, some of the registers within the register file 230 may be allocated to special purposes. For example, some of the registers can be dedicated as system registers examples of which include registers storing constant values (*e.g.*, an all zero word), program counter(s) (PC), which indicate the current address of a program thread that is being executed, a physical core number, a logical core number, a core assignment topology, core control flags, a processor topology, or other suitable dedicated purpose. In some examples, there are multiple program counter registers, one or each program counter, to allow for concurrent execution of multiple execution threads across one or more processor cores and/or processors. In some examples, program counters are implemented as designated memory locations instead of as registers in a register file. In some examples, use of the system registers may be restricted by the operating system or other supervisory computer instructions. In some examples, the register file 230 is implemented as an array of flip-flops, while in other examples, the register file can be implemented using latches, SRAM, or other forms of memory storage. The ISA specification for a given processor, for example processor 100, specifies how registers within the register file 230 are defined and used.

[053] In some examples, the processor 100 includes a global register file that is shared by a plurality of the processor cores. In some examples, individual register files associated with a processor core can be combined to form a larger file, statically or dynamically, depending on the processor ISA and configuration.

[054] As shown in FIG. 2, the memory store 215 of the instruction window 210 includes a number of decoded instructions 241, a left operand (LOP) buffer 242, a right operand (ROP) buffer 243, and an instruction scoreboard 245. In some examples of the disclosed technology, each instruction of the instruction block is decomposed into a row of decoded instructions, left and right operands, and scoreboard data, as shown in FIG. 2. The decoded instructions 241 can include partially- or fully-decoded versions of instructions stored as bit-level control signals. The operand buffers 242 and 243 store operands (*e.g.*, register values received from the register file 230, data received from memory, immediate

operands coded within an instruction, operands calculated by an earlier-issued instruction, or other operand values) until their respective decoded instructions are ready to execute. Instruction operands are read from the operand buffers 242 and 243, not the register file.

[055] The memory store 216 of the second instruction window 211 stores similar
5 instruction information (decoded instructions, operands, and scoreboard) as the memory store 215, but is not shown in FIG. 2 for the sake of simplicity. Instruction blocks can be executed by the second instruction window 211 concurrently or sequentially with respect to the first instruction window, subject to ISA constraints and as directed by the control unit 205.

10 [056] In some examples of the disclosed technology, front-end pipeline stages IF and DC can run decoupled from the back-end pipelines stages (IS, EX, LS). In one embodiment, the control unit can fetch and decode two instructions per clock cycle into each of the instruction windows 210 and 211. In alternative embodiments, the control unit can fetch and decode one, four, or another number of instructions per clock cycle into a
15 corresponding number of instruction windows. The control unit 205 provides instruction window dataflow scheduling logic to monitor the ready state of each decoded instruction's inputs (*e.g.*, each respective instruction's predicate(s) and operand(s) using the scoreboard 245. When all of the inputs for a particular decoded instruction are ready, the instruction is ready to issue. The control logic 205 then initiates execution of one or more next
20 instruction(s) (*e.g.*, the lowest numbered ready instruction) each cycle and its decoded instruction and input operands are sent to one or more of functional units 260 for execution. The decoded instruction can also encode a number of ready events. The scheduler in the control logic 205 accepts these and/or events from other sources and updates the ready state of other instructions in the window. Thus execution proceeds,
25 starting with the processor core's 111 ready zero input instructions, instructions that are targeted by the zero input instructions, and so forth.

[057] The decoded instructions 241 need not execute in the same order in which they are arranged within the memory store 215 of the instruction window 210. Rather, the instruction scoreboard 245 is used to track dependencies of the decoded instructions and,
30 when the dependencies have been met, the associated individual decoded instruction is scheduled for execution. For example, a reference to a respective instruction can be pushed onto a ready queue when the dependencies have been met for the respective instruction, and instructions can be scheduled in a first-in first-out (FIFO) order from the ready queue. Information stored in the scoreboard 245 can include, but is not limited to,

the associated instruction's execution predicate (such as whether the instruction is waiting for a predicate bit to be calculated and whether the instruction executes if the predicate bit is true or false), availability of operands to the instruction, or other prerequisites required before executing the associated individual instruction.

- 5 **[058]** In one embodiment, the scoreboard 245 can include decoded ready state, which is initialized by the instruction decoder 228, and active ready state, which is initialized by the control unit 205 during execution of the instructions. For example, the decoded ready state can encode whether a respective instruction has been decoded, awaits a predicate and/or some operand(s), perhaps via a broadcast channel, or is immediately ready to issue. The
- 10 active ready state can encode whether a respective instruction awaits a predicate and/or some operand(s), is ready to issue, or has already issued. The decoded ready state can be cleared on a block reset or a block refresh. Upon branching to a new instruction block, the decoded ready state and the active ready state is cleared (a block or core reset). However, when an instruction block is re-executed on the core, such as when it branches back to
- 15 itself (a block refresh), only active ready state is cleared. Block refreshes can occur immediately (when an instruction block branches to itself) or after executing a number of other intervening instruction blocks. The decoded ready state for the instruction block can thus be preserved so that it is not necessary to re-fetch and decode the block's instructions. Hence, block refresh can be used to save time and energy in loops and other repeating
- 20 program structures.

- [059]** The number of instructions that are stored in each instruction window generally corresponds to the number of instructions within an instruction block. In some examples, the number of instructions within an instruction block can be 32, 64, 128, 1024, or another number of instructions. In some examples of the disclosed technology, an instruction
- 25 block is allocated across multiple instruction windows within a processor core. In some examples, the instruction windows 210, 211 can be logically partitioned so that multiple instruction blocks can be executed on a single processor core. For example, one, two, four, or another number of instruction blocks can be executed on one core. The respective instruction blocks can be executed concurrently or sequentially with each other.

- 30 **[060]** Instructions can be allocated and scheduled using the control unit 205 located within the processor core 111. The control unit 205 orchestrates fetching of instructions from memory, decoding of the instructions, execution of instructions once they have been loaded into a respective instruction window, data flow into/out of the processor core 111, and control signals input and output by the processor core. For example, the control unit

205 can include the ready queue, as described above, for use in scheduling instructions. The instructions stored in the memory store 215 and 216 located in each respective instruction window 210 and 211 can be executed atomically. Thus, updates to the visible architectural state (such as writes to the register file 230 and the memory) affected by the
5 executed instructions can be buffered locally within the core until the instructions are committed. The control unit 205 can determine when instructions are ready to be committed, sequence the commit logic, and issue a commit signal. For example, a commit phase for an instruction block can begin when all register writes are buffered, all writes to memory are buffered, and a branch target is calculated. The instruction block can be
10 committed when updates to the visible architectural state are complete. For example, an instruction block can be committed when the register writes are written to the register file, the stores are sent to a load-store unit or memory controller, and the commit signal is generated. The control unit 205 also controls, at least in part, allocation of functional units 260 to each of the respective instructions windows.

15 **[061]** As shown in FIG. 2, a first router 250, which has a number of execution pipeline registers 255, is used to send data from either of the instruction windows 210 and 211 to one or more of the functional units 260, which can include but are not limited to, integer ALUs (arithmetic logic units) (*e.g.*, integer ALUs 264 and 265), floating point units (*e.g.*, floating point ALU 267), shift/rotate logic (*e.g.*, barrel shifter 268), or other suitable
20 execution units, which can including graphics functions, physics functions, and other mathematical operations. Data from the functional units 260 can then be routed through a second router 270 to outputs 290, 291, and 292, routed back to an operand buffer (*e.g.* LOP buffer 242 and/or ROP buffer 243), or fed back to another functional unit, depending on the requirements of the particular instruction being executed. The second router 270
25 can include a load-store queue 275, which can be used to process memory instructions, a data cache 277, which stores data being output from the core to memory, and load-store pipeline register 278.

[062] The core also includes control outputs 295 which are used to indicate, for example, when execution of all of the instructions for one or more of the instruction windows 210 or
30 211 has completed. When execution of an instruction block is complete, the instruction block is designated as “committed” and signals from the control outputs 295 can in turn can be used by other cores within the block-based processor 100 and/or by the control unit 160 to initiate scheduling, fetching, and execution of other instruction blocks. Both the first router 250 and the second router 270 can send data back to the instruction (for

example, as operands for other instructions within an instruction block).

[063] As will be readily understood to one of ordinary skill in the relevant art, the components within an individual core are not limited to those shown in FIG. 2, but can be varied according to the requirements of a particular application. For example, a core may
5 have fewer or more instruction windows, a single instruction decoder might be shared by two or more instruction windows, and the number of and type of functional units used can be varied, depending on the particular targeted application for the block-based processor. Other considerations that apply in selecting and allocating resources with an instruction core include performance requirements, energy usage requirements, integrated circuit die,
10 process technology, and/or cost.

[064] It will be readily apparent to one of ordinary skill in the relevant art that trade-offs can be made in processor performance by the design and allocation of resources within the instruction window (*e.g.*, instruction window 210) and control logic 205 of the processor cores 110. The area, clock period, capabilities, and limitations substantially determine the
15 realized performance of the individual cores 110 and the throughput of the block-based processor cores 110.

[065] The instruction scheduler 206 can have diverse functionality. In certain higher performance examples, the instruction scheduler is highly concurrent. For example, each cycle, the decoder(s) write instructions' decoded ready state and decoded instructions into
20 one or more instruction windows, selects the next instruction to issue, and, in response the back end sends ready events—either target-ready events targeting a specific instruction's input slot (predicate, left operand, right operand, *etc.*), or broadcast-ready events targeting all instructions. The per-instruction ready state bits, together with the decoded ready state can be used to determine that the instruction is ready to issue.

[066] In some examples, the instruction scheduler 206 is implemented using storage (e.g., first-in first-out (FIFO) queues, content addressable memories (CAMs)) storing data indicating information used to schedule execution of instruction blocks according to the disclosed technology. For example, data regarding instruction dependencies, transfers of control, speculation, branch prediction, and/or data loads and stores are arranged in storage
30 to facilitate determinations in mapping instruction blocks to processor cores. For example, instruction block dependencies can be associated with a tag that is stored in a FIFO or CAM and later accessed by selection logic used to map instruction blocks to one or more processor cores. In some examples, the instruction scheduler 206 is implemented using a general purpose processor coupled to memory, the memory being configured to store data

for scheduling instruction blocks. In some examples, instruction scheduler 206 is implemented using a special purpose processor or using a block-based processor core coupled to the memory. In some examples, the instruction scheduler 206 is implemented as a finite state machine coupled to the memory. In some examples, an operating system
5 executing on a processor (e.g., a general purpose processor or a block-based processor core) generates priorities, predictions, and other data that can be used at least in part to schedule instruction blocks with the instruction scheduler 206. As will be readily apparent to one of ordinary skill in the relevant art, other circuit structures, implemented in an integrated circuit, programmable logic, or other suitable logic can be used to implement
10 hardware for the instruction scheduler 206.

[067] In some cases, the scheduler 206 accepts events for target instructions that have not yet been decoded and must also inhibit reissue of issued ready instructions. Instructions can be non-predicated, or predicated (based on a true or false condition). A predicated instruction does not become ready until it is targeted by another instruction's predicate
15 result, and that result matches the predicate condition. If the associated predicate does not match, the instruction never issues. In some examples, predicated instructions may be issued and executed speculatively. In some examples, a processor may subsequently check that speculatively issued and executed instructions were correctly speculated. In some examples a misspeculated issued instruction and the specific transitive closure of
20 instructions in the block that consume its outputs may be re-executed, or misspeculated side effects annulled. In some examples, discovery of a misspeculated instruction leads to the complete roll back and re-execution of an entire block of instructions.

V. Example Stream of Instruction Blocks

[068] Turning now to the diagram 300 of FIG. 3, a portion 310 of a stream of block-based instructions, including a number of variable length instruction blocks 311–315 (A–
25 E) is illustrated. The stream of instructions can be used to implement user application, system services, or any other suitable use. In the example shown in FIG. 3, each instruction block begins with an instruction header, which is followed by a varying number of instructions. For example, the instruction block 311 includes a header 320 and
30 twenty instructions 321. The particular instruction header 320 illustrated includes a number of data fields that control, in part, execution of the instructions within the instruction block, and also allow for improved performance enhancement techniques including, for example branch prediction, speculative execution, lazy evaluation, and/or other techniques. The instruction header 320 also includes an ID bit which indicates that

the header is an instruction header and not an instruction. The instruction header 320 also includes an indication of the instruction block size. The instruction block size can be in larger chunks of instructions than one, for example, the number of 4-instruction chunks contained within the instruction block. In other words, the size of the block is shifted 4 bits in order to compress header space allocated to specifying instruction block size. Thus, a size value of 0 indicates a minimally-sized instruction block which is a block header followed by four instructions. In some examples, the instruction block size is expressed as a number of bytes, as a number of words, as a number of n-word chunks, as an address, as an address offset, or using other suitable expressions for describing the size of instruction blocks. In some examples, the instruction block size is indicated by a terminating bit pattern in the instruction block header and/or footer.

[069] The instruction block header 320 can also include execution flags, which indicate special instruction execution requirements. For example, branch prediction or memory dependence prediction can be inhibited for certain instruction blocks, depending on the particular application.

[070] In some examples of the disclosed technology, the instruction header 320 includes one or more identification bits that indicate that the encoded data is an instruction header. For example, in some block-based processor ISAs, a single ID bit in the least significant bit space is always set to the binary value 1 to indicate the beginning of a valid instruction block. In other examples, different bit encodings can be used for the identification bit(s). In some examples, the instruction header 320 includes information indicating a particular version of the ISA for which the associated instruction block is encoded.

[071] The block instruction header can also include a number of block exit types for use in, for example, branch prediction, control flow determination, and/or bad jump detection.

The exit type can indicate what the type of branch instructions are, for example: sequential branch instructions, which point to the next contiguous instruction block in memory; offset instructions, which are branches to another instruction block at a memory address calculated relative to an offset; subroutine calls, or subroutine returns. By encoding the branch exit types in the instruction header, the branch predictor can begin operation, at least partially, before branch instructions within the same instruction block have been fetched and/or decoded.

[072] The instruction block header 320 also includes a store mask which identifies the load-store queue identifiers that are assigned to store operations for the instruction block. The instruction block header can also include a write mask, which identifies which global

register(s) the associated instruction block may write. In some examples a block-based processor architecture can include not only scalar instructions, but also single-instruction multiple-data (SIMD) instructions, that allow for operations with a larger number of data operands within a single instruction.

5 VI. Example Block Instruction Target Encoding

[073] FIG. 4 is a diagram 400 depicting an example of two portions 410 and 415 of C language source code and their respective instruction blocks 420 and 425 (in assembly language), illustrating how block-based instructions can explicitly encode their targets. The high-level C language source code can be translated to the low-level assembly language and machine code by a compiler whose target is a block-based processor. A high-level language can abstract out many of the details of the underlying computer architecture so that a programmer can focus on functionality of the program. In contrast, the machine code encodes the program according to the target computer's ISA so that it can be executed on the target computer, using the computer's hardware resources.

10 Assembly language is a human-readable form of machine code.

[074] In the following examples, the assembly language instructions use the following nomenclature: "I[<number>]" specifies the number of the instruction within the instruction block where the numbering begins at zero for the instruction following the instruction header and the instruction number is incremented for each successive instruction; the operation of the instruction (such as READ, ADDI, DIV, and the like) follows the instruction number; optional values (such as the immediate value 1) or references to registers (such as R0 for register 0) follow the operation; and optional targets that are to receive the results of the instruction follow the values and/or operation. Each of the targets can be to another instruction, a broadcast channel to other instructions, or a register that can be visible to another instruction block when the instruction block is committed. An example of an instruction target is T[1R] which targets the right operand of instruction 1. An example of a register target is W[R0], where the target is written to register 0.

20

[075] In the diagram 400, the first two READ instructions 430 and 431 of the instruction block 420 target the right (T[2R]) and left (T[2L]) operands, respectively, of the ADD instruction 432. In the illustrated ISA, the read instruction is the only instruction that reads from the global register file; however any instruction can target, the global register file. When the ADD instruction 432 receives the result of both register reads it will become ready and execute.

30

[076] When the TLEI (test-less-than-equal-immediate) instruction 433 receives its single

input operand from the ADD, it will become ready and execute. The test then produces a predicate operand that is broadcast on channel one (B[1P]) to all instructions listening on the broadcast channel, which in this example are the two predicated branch instructions (BRO P1t 434 and BRO P1f 435). In the assembly language of the diagram 400, “P1f”
 5 indicates the instruction is predicated (the “P”) on a false result (the “f”) being transmitted on broadcast channel 1 (the “1”), and “P1t” indicates the instruction is predicated on a true result being transmitted on broadcast channel 1. The branch that receives a matching predicate will fire.

[077] A dependence graph 440 for the instruction block 420 is also illustrated, as an
 10 array 450 of instruction nodes and their corresponding operand targets 455 and 456. This illustrates the correspondence between the block instructions 420, the corresponding instruction window entries, and the underlying dataflow graph represented by the instructions. Here decoded instructions READ 430 and READ 431 are ready to issue, as they have no input dependencies. As they issue and execute, the values read from registers
 15 R6 and R7 are written into the right and left operand buffers of ADD 432, marking the left and right operands of ADD 432 “ready.” As a result, the ADD 432 instruction becomes ready, issues to an ALU, executes, and the sum is written to the left operand of TLEI 433.

[078] As a comparison, a conventional out-of-order RISC or CISC processor would dynamically build the dependence graph at runtime, using additional hardware
 20 complexity, power, area and reducing clock frequency and performance. However, the dependence graph is known statically at compile time and an EDGE compiler can directly encode the producer-consumer relations between the instructions through the ISA, freeing the microarchitecture from rediscovering them dynamically. This can potentially enable a simpler microarchitecture, reducing area, power and boosting frequency and performance.

25 VII. Example Block-Based Instruction Formats

[079] FIG. 5 is a diagram illustrating generalized examples of instruction formats for an instruction header 510, a generic instruction 520, a branch instruction 530, a load instruction 540, and a store instruction 550. Each of the instruction headers or instructions is labeled according to the number of bits. For example the instruction header 510 includes
 30 four 32-bit words and is labeled from its least significant bit (lsb) (bit 0) up to its most significant bit (msb) (bit 127). As shown, the instruction header includes a write mask field, a store mask field, a number of exit type fields, a number of execution flag fields (X flags), an instruction block size field, and an instruction header ID bit (the least significant bit of the instruction header).

[080] The execution flag fields can indicate special instruction execution modes. For example, an “inhibit branch predictor” flag can be used to inhibit branch prediction for the instruction block when the flag is set. As another example, an “inhibit memory dependence prediction” flag can be used to inhibit memory dependence prediction for the instruction block when the flag is set. As another example, a “break after block” flag can be used to halt an instruction thread and raise an interrupt when the instruction block is committed. As another example, a “break before block” flag can be used to halt an instruction thread and raise an interrupt when the instruction block header is decoded and before the instructions of the instruction block are executed.

[081] The exit type fields include data that can be used to indicate the types of control flow and/or synchronization instructions encoded within the instruction block. For example, the exit type fields can indicate that the instruction block includes one or more of the following: sequential branch instructions, offset branch instructions, indirect branch instructions, call instructions, return instructions, and/or break instructions. In some examples, the branch instructions can be any control flow instructions for transferring control flow between instruction blocks, including relative and/or absolute addresses, and using a conditional or unconditional predicate. The exit type fields can be used for branch prediction and speculative execution in addition to determining implicit control flow instructions. In some examples, up to six exit types can be encoded in the exit type fields, and the correspondence between fields and corresponding explicit or implicit control flow instructions can be determined by, for example, examining control flow instructions in the instruction block.

[082] The illustrated generic block instruction 520 is stored as one 32-bit word and includes an opcode field, a predicate field, a broadcast ID field (BID), a first target field (T1), and a second target field (T2). For instructions with more consumers than target fields, a compiler can build a fanout tree using move instructions, or it can assign high-fanout instructions to broadcasts. Broadcasts support sending an operand over a lightweight network to any number of consumer instructions in a core. A broadcast identifier can be encoded in the generic block instruction 520.

[083] While the generic instruction format outlined by the generic instruction 520 can represent some or all instructions processed by a block-based processor, it will be readily understood by one of skill in the art that, even for a particular example of an ISA, one or more of the instruction fields may deviate from the generic format for particular instructions. The opcode field specifies the length or width of the instruction 520 and the

operation(s) performed by the instruction 520, such as memory load/store, register read/write, add, subtract, multiply, divide, shift, rotate, system operations, or other suitable instructions.

[084] The predicate field specifies the condition under which the instruction will execute.

- 5 For example, the predicate field can specify the value “true,” and the instruction will only execute if a corresponding condition flag matches the specified predicate value. In some examples, the predicate field specifies, at least in part, a field, operand, or other resource which is used to compare the predicate, while in other examples, the execution is predicated on a flag set by a previous instruction (*e.g.*, the preceding instruction in the instruction block). In some examples, the predicate field can specify that the instruction will always, or never, be executed. Thus, use of the predicate field can allow for denser object code, improved energy efficiency, and improved processor performance, by reducing the number of branch instructions.

- [085]** The target fields T1 and T2 specifying the instructions to which the results of the block-based instruction are sent. For example, an ADD instruction at instruction slot 5 can specify that its computed result will be sent to instructions at slots 3 and 10. Depending on the particular instruction and ISA, one or both of the illustrated target fields can be replaced by other information, for example, the first target field T1 can be replaced by an immediate operand, an additional opcode, specify two targets, *etc.*

- 20 **[086]** The branch instruction 530 includes an opcode field, a predicate field, a broadcast ID field (BID), and an offset field. The opcode and predicate fields are similar in format and function as described regarding the generic instruction. The offset can be expressed in units of four instructions, thus extending the memory address range over which a branch can be executed. The predicate shown with the generic instruction 520 and the branch instruction 530 can be used to avoid additional branching within an instruction block. For example, execution of a particular instruction can be predicated on the result of a previous instruction (*e.g.*, a comparison of two operands). If the predicate is false, the instruction will not commit values calculated by the particular instruction. If the predicate value does not match the required predicate, the instruction does not issue. For example, a BRO_F (predicated false) instruction will issue if it is sent a false predicate value.

[087] It should be readily understood that, as used herein, the term “branch instruction” is not limited to changing program execution to a relative memory location, but also includes jumps to an absolute or symbolic memory location, subroutine calls and returns, and other instructions that can modify the execution flow. In some examples, the

execution flow is modified by changing the value of a system register (*e.g.*, a program counter PC or instruction pointer), while in other examples, the execution flow can be changed by modifying a value stored at a designated location in memory. In some examples, a jump register branch instruction is used to jump to a memory location stored
5 in a register. In some examples, subroutine calls and returns are implemented using jump and link and jump register instructions, respectively.

[088] The load instruction 540 is used for retrieving data stored at a target address of memory so that the data can be used by a processor core. The target address of the data can be calculated dynamically at runtime. For example, the address can be a sum of an
10 operand of the load instruction 540 and an immediate field of the load instruction 540. As another example, the address can be a sum of an operand of the load instruction 540 and a sign-extended and/or shifted immediate field of the load instruction 540. As another example, the address of the data can be a sum of two operands of the load instruction 540. The load instruction 540 can include a load-store identifier field (LSID) to provide a
15 relative program ordering of the load within an instruction block. For example, the compiler can assign an LSID to each load and store of the instruction block at compile-time. The amount and type of data can be retrieved and/or formatted in various ways. For example, the data can be formatted as a signed or unsigned value and the amount or size of the data retrieved can vary. Different opcodes can be used to identify the type of load
20 instruction 540, such as as a load unsigned byte, load signed byte, load double-word, load unsigned half-word, load signed half-word, load unsigned word, and load signed word, for example. The output of the load instruction 540 can be directed to a target instruction as indicated by a target field (T0).

[089] A predicated load instruction is a load instruction that conditionally executes based
25 on whether a result associated with the instruction matches a predicate test value. For example, the result can be delivered to an operand of the predicated load instruction from another instruction, and the predicate test value can be encoded in a field of the predicated load instruction. As a specific example, the load instruction 540 can be a predicated load instruction when one or more bits of the predicate field (PR) are non-zero. For example,
30 the predicate field can be two bits wide where one bit is used to indicate that the instruction is predicated and one bit is used to indicate the predicate test value. Specifically, the encodings “00” can indicate the load instruction 540 is not predicated; “10” can indicate the load instruction 540 is predicated on a false condition (*e.g.*, the predicate test value is a “0”); “11” can indicate the load instruction 540 is predicated on a

true condition (e.g., the predicate test value is a “0”); and “10” can be reserved. Thus, a two-bit predicate field can be used to compare a received result to a true or false condition. A wider predicate field can be used to compare the received result to a larger number.

[090] In one example, the result to be compared to the predicate test value can be passed
 5 to the instruction via one or more broadcast operands or channels. The broadcast channel of the predicate can be identified within the load instruction 540 using a broadcast identifier field (BID). For example, the broadcast identifier field can be two-bits wide to encode four possible broadcast channels on which to receive the value to compare to the predicate test value. As a specific example, if the value received on the identified
 10 broadcast channel matches the predicate test value, the load instruction 540 is executed. However, if the value received on the identified broadcast channel does not match the predicate test value, the load instruction 540 is not executed.

[091] The load instruction 540 can be relatively slow to execute compared to other instructions because it is used to retrieve data from memory, and memory accesses can be
 15 relatively slow. For example, operations that occur entirely within a processor core can be relatively faster because the logic circuits of the processor core are relatively closer together and faster than the circuits in main memory. Memory may be shared by multiple processor cores of a processor, so it can be relatively far from a particular processor core and the memory may be larger than a processor core making it relatively slower. As
 20 described in more detail below, a load-store queue can be used to manage completion of the load instruction 540.

[092] As a specific example of a 32-bit load instruction 540, the opcode field can be encoded in bits [31:25]; the predicate field can be encoded in bits [24:23]; the broadcast identifier field can be encoded in bits [22:21]; the LSID field can be encoded in bits
 25 [20:16]; the immediate field can be encoded in bits [15:9]; and the target field can be encoded in bits [8:0].

[093] The store instruction 550 is used for storing data at a target address of the memory. The target address of the data can be calculated dynamically at runtime. For example, the address can be a sum of a first operand of the store instruction 550 and an immediate field
 30 of the store instruction 550. As another example, the address can be a sum of an operand of the store instruction 550 and a sign-extended and/or shifted immediate field of the store instruction 550. As another example, the address of the data can be a sum of two operands of the store instruction 550. The store instruction 550 can include a load-store identifier field (LSID) to provide a relative program ordering of the store within an instruction

block. The amount of data to be stored can vary based on an opcode of the store instruction 550, such as a store byte, store half-word, store word, and store double-word, for example. The data to be stored at the memory location can be input from a second operand of the store instruction 550. The second operand can be generated by another instruction or encoded as a field of the store instruction 550.

[094] A predicated store instruction is a store instruction that conditionally executes based on whether a result associated with the instruction matches a predicate test value. For example, the result can be delivered to an operand of the predicated store instruction from another instruction, and the predicate test value can be encoded in a field of the predicated store instruction. For example, the store instruction 550 can be a predicated store instruction when one or more bits of the predicate field (PR) are non-zero. The result to be compared to the predicate test value can be passed to the instruction via one or more broadcast operands or channels. The broadcast channel of the predicate can be identified within the store instruction 550 using a broadcast identifier field (BID). As a specific example, if the value received on the identified broadcast channel matches the predicate test value, the store instruction 550 is executed. However, if the value received on the identified broadcast channel does not match the predicate test value, the store instruction 550 is not executed.

[095] As a specific example of a 32-bit store instruction 550, the opcode field can be encoded in bits [31:25]; the predicate field can be encoded in bits [24:23]; the broadcast identifier field can be encoded in bits [22:21]; the LSID field can be encoded in bits [20:16]; and the immediate field can be encoded in bits [15:9]. The bits [8:1] can be reserved for additional functions or for future use.

VIII. Example States of a Processor Core

[096] FIG. 6 is a flowchart illustrating an example of a progression of states 600 of a processor core of a block-based computer. The block-based computer is composed of multiple processor cores that are collectively used to run or execute a software program. The program can be written in a variety of high-level languages and then compiled for the block-based processor using a compiler that targets the block-based processor. The compiler can emit code that, when run or executed on the block-based processor, will perform the functionality specified by the high-level program. The compiled code can be stored in a computer-readable memory that can be accessed by the block-based processor. The compiled code can include a stream of instructions grouped into a series of instruction blocks. During execution, one or more of the instruction blocks can be executed by the

block-based processor to perform the functionality of the program. Typically, the program will include more instruction blocks than can be executed on the cores at any one time. Thus, blocks of the program are mapped to respective cores, the cores perform the work specified by the blocks, and then the blocks on respective cores are replaced with different blocks until the program is complete. As one example, a single core can be used to execute all of the blocks of a program. Some of the instruction blocks may be executed more than once, such as during a loop or a subroutine of the program. An “instance” of an instruction block can be created for each time the instruction block will be executed. Thus, each repetition of an instruction block can use a different instance of the instruction block. As the program is run, the respective instruction blocks can be mapped to and executed on the processor cores based on architectural constraints, available hardware resources, and the dynamic flow of the program. During execution of the program, the respective processor cores can transition through a progression of states 600, so that one core can be in one state and another core can be in a different state.

[097] At state 605, a state of a respective processor core can be unmapped. An unmapped processor core is a core that is not currently assigned to execute an instance of an instruction block. For example, the processor core can be unmapped before the program begins execution on the block-based computer. As another example, the processor core can be unmapped after the program begins executing but not all of the cores are being used. In particular, the instruction blocks of the program are executed, at least in part, according to the dynamic flow of the program. Some parts of the program may flow generally serially or sequentially, such as when a later instruction block depends on results from an earlier instruction block. Other parts of the program may have a more parallel flow, such as when multiple instruction blocks can execute at the same time without using the results of the other blocks executing in parallel. Fewer cores can be used to execute the program during more sequential streams of the program and more cores can be used to execute the program during more parallel streams of the program.

[098] At state 610, the state of the respective processor core can be mapped. A mapped processor core is a core that is currently assigned to execute an instance of an instruction block. When the instruction block is mapped to a specific processor core, the instruction block is in-flight. An in-flight instruction block is a block that is targeted to a particular core of the block-based processor, and the block will be or is executing, either speculatively or non-speculatively, on the particular processor core. In particular, the in-flight instruction blocks correspond to the instruction blocks mapped to processor cores in

states 610–650. A block executes non-speculatively when it is known during mapping of the block that the program will use the work provided by the executing instruction block. A block executes speculatively when it is not known during mapping whether the program will or will not use the work provided by the executing instruction block. Executing a block speculatively can potentially increase performance, such as when the speculative block is started earlier than if the block were to be started after or when it is known that the work of the block will be used. However, executing speculatively can potentially increase the energy used when executing the program, such as when the speculative work is not used by the program.

5 [099] A block-based processor includes a finite number of homogeneous or heterogeneous processor cores. A typical program can include more instruction blocks than can fit onto the processor cores. Thus, the respective instruction blocks of a program will generally share the processor cores with the other instruction blocks of the program. In other words, a given core may execute the instructions of several different instruction blocks during the execution of a program. Having a finite number of processor cores also means that execution of the program may stall or be delayed when all of the processor cores are busy executing instruction blocks and no new cores are available for dispatch. When a processor core becomes available, an instance of an instruction block can be mapped to the processor core.

10 [0100] An instruction block scheduler can assign which instruction block will execute on which processor core and when the instruction block will be executed. The mapping can be based on a variety of factors, such as a target energy to be used for the execution, the number and configuration of the processor cores, the current and/or former usage of the processor cores, the dynamic flow of the program, whether speculative execution is enabled, a confidence level that a speculative block will be executed, and other factors. An instance of an instruction block can be mapped to a processor core that is currently available (such as when no instruction block is currently executing on it). In one embodiment, the instance of the instruction block can be mapped to a processor core that is currently busy (such as when the core is executing a different instance of an instruction block) and the later-mapped instance can begin when the earlier-mapped instance is complete.

15 [0101] At state 620, the state of the respective processor core can be fetch. For example, the IF pipeline stage of the processor core can be active during the fetch state. Fetching an instruction block can include transferring instructions of the block from memory (such as

the L1 cache, the L2 cache, or main memory) to the processor core, and reading instructions from local buffers of the processor core so that the instructions can be decoded. For example, the instructions of the instruction block can be loaded into an instruction cache, buffer, or registers of the processor core. Multiple instructions of the instruction block can be fetched in parallel (e.g., at the same time) during the same clock cycle. The fetch state can be multiple cycles long and can overlap with the decode (630) and execute (640) states when the processor core is pipelined.

[0102] When instructions of the instruction block are loaded onto the processor core, the instruction block is resident on the processor core. The instruction block is partially resident when some, but not all, instructions of the instruction block are loaded. The instruction block is fully resident when all instructions of the instruction block are loaded. The instruction block will be resident on the processor core until the processor core is reset or a different instruction block is fetched onto the processor core. In particular, an instruction block is resident in the processor core when the core is in states 620–670.

[0103] At state 630, the state of the respective processor core can be decode. For example, the DC pipeline stage of the processor core can be active during the fetch state. During the decode state, instructions of the instruction block are being decoded so that they can be stored in the memory store of the instruction window of the processor core. In particular, the instructions can be transformed from relatively compact machine code, to a less compact representation that can be used to control hardware resources of the processor core. Predicated load and predicated store instructions can be identified during the decode state. The decode state can be multiple cycles long and can overlap with the fetch (620) and execute (640) states when the processor core is pipelined. After an instruction of the instruction block is decoded, it can be executed when all dependencies of the instruction are met.

[0104] At state 640, the state of the respective processor core can be execute. During the execute state, instructions of the instruction block are being executed. In particular, the EX and/or LS pipeline stages of the processor core can be active during the execute state. Data associated with load and/or store instructions can be fetched and/or pre-fetched during the execute state. The instruction block can be executing speculatively or non-speculatively. A speculative block can execute to completion or it can be terminated prior to completion, such as when it is determined that work performed by the speculative block will not be used. When an instruction block is terminated, the processor can transition to the abort state. A speculative block can complete when it is determined the work of the block will

be used, all register writes are buffered, all writes to memory are buffered, and a branch target is calculated, for example. A non-speculative block can execute to completion when all register writes are buffered, all writes to memory are buffered, and a branch target is calculated, for example. The execute state can be multiple cycles long and can overlap
5 with the fetch (620) and decode (630) states when the processor core is pipelined. When the instruction block is complete, the processor can transition to the commit state.

[0105] At state 650, the state of the respective processor core can be commit or abort. During commit, the work of the instructions of the instruction block can be atomically committed so that other blocks can use the work of the instructions. In particular, the
10 commit state can include a commit phase where locally buffered architectural state is written to architectural state that is visible to or accessible by other processor cores. As one example, stores to memory can be buffered in a load-store queue during execution of the block, and the stores can be written to memory during the commit phase. When the visible architectural state is updated, a commit signal can be issued and the processor core
15 can be released so that another instruction block can be executed on the processor core. Alternatively, the commit phase can overlap with execution of the next block and the load-store queue can be used to maintain a consistent view of memory. For example, memory consistency can be maintained by forwarding store data (buffered in the load-store queue) from a committed block to an executing block even while the stores from the committed
20 block is still being written to memory.

[0106] During the abort state, the pipeline of the core can be halted to reduce dynamic power dissipation. In some applications, the core can be power gated to reduce static power dissipation. Overlapping with or at the conclusion of the commit/abort states, the processor core can receive a new instruction block to be executed on the processor core,
25 the core can be refreshed, the core can be idled, or the core can be reset.

[0107] At state 660, it can be determined if the instruction block resident on the processor core can be refreshed. As used herein, an instruction block refresh or a processor core refresh means enabling the processor core to re-execute one or more instruction blocks that are resident on the processor core. In one embodiment, refreshing a core can include
30 resetting the active-ready state for one or more instruction blocks. It may be desirable to re-execute the instruction block on the same processor core when the instruction block is part of a loop or a repeated sub-routine or when a speculative block was terminated and is to be re-executed. The decision to refresh can be made by the processor core itself (contiguous reuse) or by outside of the processor core (non-contiguous reuse). For

example, the decision to refresh can come from another processor core or a control core performing instruction block scheduling. There can be a potential energy savings when an instruction block is refreshed on a core that already executed the instruction as opposed to executing the instruction block on a different core. Energy is used to fetch and decode the instructions of the instruction block, but a refreshed block can save most of the energy used in the fetch and decode states by bypassing these states. In particular, a refreshed block can re-start at the execute state (640) because the instructions have already been fetched and decoded by the core. When a block is refreshed, the decoded instructions and the decoded ready state can be maintained while the active ready state is cleared. The decision to refresh an instruction block can occur as part of the commit operations or at a later time. If an instruction block is not refreshed, the processor core can be idled.

[0108] At state 670, the state of the respective processor core can be idle. The performance and power consumption of the block-based processor can potentially be adjusted or traded off based on the number of processor cores that are active at a given time. For example, performing speculative work on concurrently running cores may increase the speed of a computation but increase the power if the speculative misprediction rate is high. As another example, immediately allocating new instruction blocks to processors after committing or aborting an earlier executed instruction block may increase the number of processors executing concurrently, but may reduce the opportunity to reuse instruction blocks that were resident on the processor cores. Reuse may be increased when a cache or pool of idle processor cores is maintained. For example, when a processor core commits a commonly used instruction block, the processor core can be placed in the idle pool so that the core can be refreshed the next time that the same instruction block is to be executed. As described above, refreshing the processor core can save the time and energy used to fetch and decode the resident instruction block. The instruction blocks/processor cores to place in an idle cache can be determined based on a static analysis performed by the compiler or a dynamic analysis performed by the instruction block scheduler. For example, a compiler hint indicating potential reuse of the instruction block can be placed in the header of the block and the instruction block scheduler can use the hint to determine if the block will be idled or reallocated to a different instruction block after committing the instruction block. When idling, the processor core can be placed in a low-power state to reduce dynamic power consumption, for example.

[0109] At state 680, it can be determined if the instruction block resident on the idle processor core can be refreshed. If the core is to be refreshed, the block refresh signal can

be asserted and the core can transition to the execute state (640). If the core is not going to be refreshed, the block reset signal can be asserted and the core can transition to the unmapped state (605). When the core is reset, the core can be put into a pool with other unmapped cores so that the instruction block scheduler can allocate a new instruction block to the core.

IX. Examples of Load-Store Queue Architectures

[0110] FIG. 7 illustrates an example snippet of instructions 700 of a program for a block-based processor. The program can include multiple blocks of instructions, such as instruction blocks 710-712. The execution order of the instruction blocks 710-712 is determined dynamically at run-time based on control decisions of the program. As illustrated, the block 710 is followed by block 711 which is followed by 712. An instruction block can include instructions that are to be executed as a group. For example, a given instruction block can include a single basic block, a portion of a basic block, or multiple basic blocks, so long as the instruction block can be executed within the constraints of the ISA and the hardware resources of the targeted computer. A basic block is a block of code where control can only enter the block at the first instruction of the block and control can only leave the block at the last instruction of the basic block. Thus, a basic block is a sequence of instructions that are executed together. Multiple basic blocks can be combined into a single instruction block using predicated instructions so that intra-instruction-block branches are converted to dataflow instructions

[0111] Each of the instruction blocks 710-712 can include load and store instructions for writing and reading to memory at various granularities. For example, the load instructions can include a load word (lw) (e.g., four bytes), a load half-word (lh) (e.g., two bytes), and a load byte (lb) (e.g., one byte) instruction. Similarly, the store instructions can include a store word (sw) (e.g., four bytes), a store half-word (sh) (e.g., two bytes), and a store byte (sb) (e.g., one byte) instruction. In other embodiments, wider data widths can be supported, such as 64-bit or 128-bit widths, using load double, load quad-word, store double, and store quad-word instructions. Each of the load and store instructions can include a memory address from which to load or store data. For example, the memory can be byte-addressable. As illustrated in FIG. 7, the addresses are indicated with the "@" symbol and the addresses are hexadecimal byte addresses. A store instruction can include data that is to be written to the memory. As illustrated in FIG. 7, the store data of an instruction is presented after the "=" symbol and is in hexadecimal. A load instruction can respond with or return data that is read or loaded from the memory. As illustrated in FIG.

7, the expected load response data of an instruction is presented after the “=>” symbol and is in hexadecimal.

[0112] The instruction blocks 710-712 can each be atomically executed on a block-based processor core of a processor. For example, the instructions of a given instruction block can be executed and the visible architectural state can be locally buffered and then updated in an atomic transaction. Specifically, the updated visible architectural state of an instruction block can be locally buffered while the instructions of the instruction block are being executed, and the visible architectural state can be updated in an atomic transaction during a commit of the instruction block. Updating the visible architectural state can include writing to registers and to memory. The local buffering can be performed by a load-store buffer or queue of the processor core or a shared load-store queue of the processor. The load-store queue can buffer the load and store instructions so that load response data can be returned to the processor core when it is ready and store data can be written to the memory when the instruction block commits. The writing of the committed store data can overlap in time with the execution of the next instruction block and the load-store queue can ensure that the memory is consistent across the instruction blocks. Specifically, the load-store queue can enable load instructions in later blocks to load committed store data from earlier blocks even if the committed store data is still being written to memory.

[0113] Additionally, an instruction block may be partially or fully executed and then aborted so that none of the visible architectural state is updated. The load-store queue can be used to prevent stores in an aborted instruction block from being written back to memory when the instruction block aborts. Specifically, the stores can be locally buffered in the load-store queue during execution of the block, and then dropped (e.g., not written to architecturally visible memory) after an abort condition is detected.

[0114] Within an instruction block, the load-store queue can ensure proper ordering of loads and stores even when the load and store instructions are issued out of program order. Specifically, an out-of-order processor core can execute instructions in an execution order that is different than a program order. The program order is the order that a programmer intended whereas the execution order is the order that the processor executes the instruction. For example, a compiler can specify the program order of the instructions (load, store, and non-load-store instructions) using an instruction identifier (IID) that is encoded within the instructions and/or inferred from the order of the instructions output by the compiler. The processor can execute the instructions in a data-flow order that is

different from the program order. In order to maintain correct operation of the program, the program order of loads from and stores to the memory is tracked so that the correct data can be returned from the load or stored in the memory. Specifically, each load and store instruction can include a load-store identifier (LSID or lsid) assigned by a compiler
5 of the program. The LSID indicates a relative program order of the load and store instructions within an instruction block. For example, an instruction with a lower LSID can indicate the instruction is earlier in the program order than an instruction with a higher LSID. As one example, an LSID can be encoded in a field of each load and store instruction. Thus, the load and store instructions can be placed within the instruction block
10 independent of the program order, such as to optimize decode logic of the processor or for other reasons. As another example, the compiler can encode the LSID based on a placement of the load or store instruction within the stored order of the instructions of the instruction block. For example, the LSID can be the same as the IID (such as when all of the load and store instructions are placed at the beginning of the instruction block); the
15 LSID can be the *i*th load or store in the block (e.g., the first load or store instruction is LSID = 0, the second load or store instruction is LSID = 1, and so forth); or the LSID can be some other function of the block and the instructions in the block.

[0115] Generally, load and store instructions can complete out-of-program order if the target addresses of the load and store instructions are non-overlapping. However, if the
20 load and store instructions have overlapping target addresses, then the program order of the load and store instructions is taken into account when completing the load and store instructions. Because addresses can be calculated at run-time, determining whether the loads and stores are to overlapping addresses is a dynamic run-time decision. The load-store queue can make a dynamic run-time decision to determine which loads and stores
25 have overlapping addresses and can ensure that the data affected by the overlapping loads and stores is correct. For example, the load-store queue can order operations of the loads and stores to overlapping addresses so that the data affected by the loads and stores is updated as if the loads and stores occur in program order.

[0116] Loads and stores to overlapping target addresses can create data dependencies such
30 as read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependencies. For the read-after-write dependency, a load instruction that occurs later in program order than a store instruction to the same target address should return the data written by the store instruction. For the write-after-read dependency, a load instruction that occurs earlier in program order than a store instruction to the same target address should

return the data stored at the target address before the store instruction updates the data at the target address. For the write-after-write dependency, data written by a first store instruction that occurs earlier in program order than a second store instruction to the same target address should be overwritten by the data of the second store instruction.

5 [0117] Data dependencies can occur within instruction blocks and between instruction blocks. The example 700 illustrates many different RAW, WAR, and WAW dependencies within and between instruction blocks and the expected results from the load and store instructions. One example of a read-after-write dependency within an instruction block is the load instruction 721 following the store instruction 720 in block 710. When the
10 program order (as indicated by their respective LSIDs) of the instructions 720 and 721 is taken into account, the store value “10” is buffered in the load-store queue for address 21 and the “10” is read out by the load instruction 721. Thus, the response data provided by the load 721 is dependent on the store 720 even though the store 720 can be committed to the memory hierarchy after the load 721 is completed. However, if the instructions 720
15 and 721 were completed out of program order (e.g., in reverse order), the read response data returned by instruction 721 could be the “BB” (stored by instruction 722) instead of the “10” stored by instruction 720. If instruction 721 returned a “BB” an error would be introduced in the program. One example of a read-after-write dependency between instruction blocks is the load instruction 730 in block 711 following the store instruction
20 720 in block 710. For correct operation, the load instruction 730 will return the data (e.g., “D11A”) written by the store instruction 720, even if execution of block 711 is overlapping in time with the commit of the stores from block 710.

[0118] FIG. 8 illustrates an example computing system 800 including a load-store queue 870. The computing system 800 can be used for executing a program on a block-based
25 processor core. For example, the program can include the instruction blocks A-E. The instruction blocks A-E can be stored in a memory 810 that can be accessed by the processor 805. The processor 805 can include a plurality of block-based processor cores (including block-based processor core 820), an optional memory controller and level-two (L2) cache 840, cache coherence logic 845, a control unit 850, and an input/output (I/O)
30 interface 860. The block-based processor core 820 can communicate with a memory hierarchy or memory sub-system used for storing and retrieving instructions and data of the program.

[0119] The memory hierarchy can be used to potentially increase the speed of accessing data stored in the memory 810. Generally, a memory hierarchy includes multiple levels of

memory having different speeds and sizes. Levels within or closer to the processor core are generally faster and smaller than levels farther from the processor core. For example, a memory hierarchy can include a level-one (L1) cache within a processor core, a level-two (L2) cache within a processor that is shared by multiple processor cores, main memory that is off-chip or external to the processor, and backing store that is located on a storage device, such as a hard-disk drive. When the memory hierarchy is accessed, the faster and closer levels of the memory hierarchy can be accessed before the slower and farther levels of the memory hierarchy. As one example, the memory hierarchy can include the level-one (L1) cache 828, the memory controller and level-two (L2) cache 840, and the memory 810. The memory controller and the level-two (L2) cache 840 can be used to generate the control signals for communicating with the memory 810 and to provide temporary storage for information coming from or going to the memory 810. As illustrated in FIG. 8, the memory 810 is off-chip or external to the processor 805. However, the memory 810 can be fully or partially integrated within the processor 805.

[0120] The control unit 850 can be used for implementing all or a portion of a run-time environment for the program. The runtime environment can be used for managing the usage of the block-based processor cores and the memory 810. For example, the memory 810 can be partitioned into a code segment 812 comprising the instruction blocks A-E and a data segment 815 comprising a static section, a heap section, and a stack section. As another example, the control unit 850 can be used for allocating processor cores to execute instruction blocks. The optional I/O interface 860 can be used for connecting the processor 805 to various input devices (such as an input device 866), various output devices (such as a display 864), and a storage device 862. In some examples, the components of the processor core 820, the memory controller and L2 cache 840, the cache coherence logic 845, the control unit 850, and the I/O interface 860 are implemented at least in part using one or more of: hardwired finite state machines, programmable microcode, programmable gate arrays, or other suitable control circuits. In some examples, the cache coherence logic 845, the control unit 850, and the I/O interface 860 are implemented at least in part using an external computer (e.g., an off-chip processor executing control code and communicating with the processor 805 via a communications interface (not shown)).

[0121] All or part of the program can be executed on the processor 805. Specifically, the control unit 850 can allocate one or more block-based processor cores, such as the processor core 820, to execute the program. The control unit 850 can communicate a starting address of an instruction block to the processor core 820 so that the instruction

block can be fetched from the code segment 812 of the memory 810. Specifically, the processor core 820 can issue a read request to the memory controller and L2 cache 840 for the block of memory containing the instruction block. The memory controller and L2 cache 840 can return the instruction block to the processor core 820. The instruction block
5 can include an instruction header and instructions. The instruction header can be decoded by the header decode logic 832 to determine information about the instruction block, such as a store mask of the instruction block. The individual instructions of the instruction block can be decoded by instruction decode logic 833 and the decoded instructions can be stored in the instruction windows 822 and 823. The instruction decode logic 833 can also
10 be used to determine additional information about the instruction block, such as when the information is not present in the instruction header. For example, the instruction decode logic 833 can be used to generate the store mask by identifying each store instruction within the instruction block. During execution, the instructions of the instruction block are issued or scheduled dynamically for execution by the instruction issue logic 834, based on
15 when the instruction operands become available. Thus, the issued or execution order of the instructions can be different from the program ordering of the instructions. The instructions can be fully or partially executed using execution logic 824 (such as arithmetic logic units). As the instructions execute, intermediate values of the instruction block (such as operand buffers of the instruction windows 822 and 823, and buffers of a
20 load-store queue 870) are calculated and stored locally within state of the processor core 820. The architecturally visible results of the instructions (such as writes to the memory 810) are committed atomically for the instruction block. Thus, the intermediate values generated by the processor core 820 are not visible outside of the processor core 820 and the final results (such as writes to the memory 810 or to a global register file (not shown))
25 are released as a single transaction.

[0122] As described above, the load and store instructions can include target addresses that are dynamically calculated at run-time, such as by the execution logic 824. For example, a load instruction can include a dynamically calculated target address that references one or more memory locations (e.g., an address of a byte, half-word, or word)
30 from which to read data. A store instruction can include a dynamically calculated target address that references one or more memory locations (e.g., an address of a byte, half-word, or word) at which to store data when the instruction block commits. The load-store queue 870 can act as an interface between the execution logic 824 and the memory hierarchy. For example, the load-store queue 870 can buffer store data until it can be

committed and build load response data using the locally buffered store data and/or data retrieved from the memory hierarchy.

[0123] Components of the load-store queue 870 can include load-store intake logic 872 for receiving requests corresponding to issued load and store instructions from the execution logic 824, queue memory 874 for storing information associated with the issued load and store instructions, hash data structures 876 for storing information associated with the issued store instructions, memory interface logic 878 for communicating with the memory hierarchy, load response logic 880 for generating load response data for the load instructions, and commit logic 882 for managing a commit phase of the instruction block and draining store data from the load-store queue 870.

[0124] The load-store intake logic 872 can receive load requests and store requests corresponding to individual load and store instructions. For example, the load requests can include an instruction number corresponding to a load instruction stored in an instruction window 822-823, a target address from which to read data, a byte mask indicating which bytes within the target address range are to be read. The store requests can include an instruction number corresponding to a store instruction stored in an instruction window 822-823, a target address to store data at, data to store, a byte mask indicating which bytes within the target address range are to be written.

[0125] The load-store intake logic 872 can update data structures of the load-store queue 870 which can trigger additional operations of the load-store queue 870. For example, the information associated with each load and store instruction can be stored in the queue memory 874 and the issued loads and stores can be tracked (such as by using an issued-load bitmask and an issued-store bitmask, where each bit of the bitmask can correspond to an LSID). As discussed further below, knowing which loads and stores of the instruction block have issued can be used to determine whether an instruction block can be committed and whether load response data can be generated for a load instruction. The addresses of the individual load and store instructions can be hashed using one or more hash functions to generate one or more hash values for each of the load and store instructions. Information associated with store instructions can be stored in the hash data structures 876 and indexed by the hash values of the store instructions. A hash data structure is a data structure that is arranged based on a hash value of a quantity (such as a target address of load or store instruction. As one example, the hash data structures 876 can include a hash table memory, and the address and the LSID of the store instruction can be stored in the hash table memory. As another example, the hash data structures 876 can include a Bloom

filter, and the Bloom filter can be used to store an indication that a store instruction with a target address having a given hash value was received. A store mask can also be received which indicates all of the LSIDs of the instruction block that are store instructions.

[0126] The load response logic 880 can generate load response data for the load
5 instructions and forward the generated load response data to the execution logic 824 and/or the instruction windows 822 and 823. For example, the load response data can be stored to one or more operand buffers of the instruction windows 822 and 823. The load response data for a given load instruction can be generated after all stores with an earlier LSID have issued. This condition can be tested by comparing a pending-store bitmask to
10 the issued-load bitmask, for example. When the instruction block begins execution, the pending-store bitmask can be initialized as a copy of the store mask (e.g., from the store mask field of the block header). When a store request for a particular LSID is received, the corresponding bit in the pending-store bitmask can be deasserted. Specifically, all stores are pending when an instruction block begins, and the stores transition from being pending
15 to issued as the individual stores are issued. Because stores earlier in program order may affect data of a later load, the load response data can be built (non-speculatively) for the load only after earlier stores have been issued. It can be determined that all earlier stores have issued when there are no pending stores with LSIDs less than the LSID of a given issued load instruction. For example, a pair of priority encoders can be used to efficiently
20 find the lowest issued load and pending store, using the issued-load and pending-store bitmasks, respectively, as inputs to the priority encoders. The load response data can be generated using data retrieved from the memory hierarchy and/or data retrieved from one or more earlier stores. The most recently written data (on a byte granularity) to the load address is the data to return in the load response. Thus, the load response data can be
25 obtained from up to four different sources (e.g., memory or different store instructions) when a word is four bytes and byte-enabled stores are supported by the ISA.

[0127] A high-performance, but area-intensive and high-power solution for searching for earlier store instructions can use a CAM to match queued entries with the same address. For example, a CAM is a resource-intensive approach that may be used to perform the
30 search in a single cycle. However, a solution using fewer resources or using less power may be desirable for lower-power integrated circuits (ICs) and/or FPGAs. A simple but slow solution for finding the response data can include storing the store instruction information in a RAM (e.g., a RAM with one read port and one write port) and serially searching through all of the previously issued and non-committed store instructions to find

any stores with overlapping addresses of the load instruction. As one example, one issued and non-committed store instruction can be searched in a given cycle. However, using the hash data structures 876 can potentially reduce the number of stores to search by approximately $1 - 1/N$, where N is a number of buckets of the hash data structures 876. By
5 reducing the number of stores to search, a serial search can be performed faster (e.g., in fewer cycles) and with less energy than if all of the issued and non-committed stores were to be searched.

[0128] A hash function can be used to generate a set of hash values that are smaller than the set of input values to the hash function. Specifically, the addresses of the load and
10 store instructions can be the input to the hash function. For example, a 32-bit ISA can include 32-bit addresses or 2^{32} different addresses. An example hash function can include truncating and/or shifting an address so that the hash value of the hash function is the low-order bits of the address. For example, a hexadecimal 32-bit address of 32FC0030 can be
15 hashed to an 8-bit hexadecimal value of 30 when only the low-order eight bits of the address are kept by the hash function. Other hash functions are possible (such as by performing logical operations (e.g., exclusive-or'ing) on different bits of the address) and different inputs to the hash function are possible (such as by inputting a word address rather than a byte address).

[0129] One or more hash functions can be used to generate hash values for addresses of
20 respective load and store instructions to help correlate loads and stores so that queued stores occurring earlier in program order may forward their store data to loads occurring relatively later in program order. Specifically, the hash values can be used as indexes for accessing the hash data structures 876. For example, the hash data structures 876 can include a hash table and a first hash function can be used to generate an index for the hash
25 table. The hash table can store information about the enqueued store instructions, and the hash table can be searched to find store data to forward to later load requests. As another example, the hash data structures 876 can include a Bloom filter data structure and a second hash function can be used to generate an index for the Bloom filter. The Bloom filter data structure can be analyzed to determine whether enqueued store instructions may
30 match the target address of a load request. The hash table and the Bloom filter can be used alone or as complements to each other. Specifically, the hash data structures 876 can have only the hash table, only the Bloom filter, or both the hash table and the Bloom filter. By using both the hash table and the Bloom filter, a load response may be generated faster than using only one hash data structure since the algorithms for searching the structures

are different and one algorithm may be faster than the other for a given set of inputs. By using only one structure, the area of the processor may be reduced. The hash data structures 876 can include a hash table memory and a hash-bucket head data structure. The hash-bucket head data structure can be used to track entries in the hash table memory. For example, the hash-bucket head data structure can include an entry for every possible value of the hash function, and the entry can be a pointer to a location or entry within the hash table memory. Each location within the hash table memory can include a pointer to a next entry having the same hash value. Thus, the entries within the hash table memory can be maintained as a linked list where the head of the linked list is stored in the hash-bucket head data structure. Each location within the hash table memory can also include a field for the address of the store, the LSID of the store, byte enables of the store, store data, an instruction number of the store, and/or other information related to the store instruction or for maintaining the hash table data structure. In another embodiment, an entry in the hash bucket headers can include the LSID of a first store instruction whose address hashes to that bucket, or a NULL if none do, and an entry (indexed by LSID) in the hash table memory includes the LSID of the next store instruction whose address hashes to that bucket, or NULL if there are no more such entries. Thus, the load response logic 880 can use the hash data structures 876 (e.g., the hash bucket headers and the hash table memory) to enumerate the LSIDs (and hence addresses) of the store instructions queued in the queue memory 874.

[0130] The load response logic 880 can search the hash data structures 876 for stores having addresses matching the address of the load instruction. For example, the hash value of the load address can be used as an index into the hash table memory. Specifically, the entry of the hash-bucket head data structure (the head entry) corresponding to the hash value of the load address can be accessed. If the head entry is empty (e.g., a sentinel value is stored at the head entry), there are no stores with an overlapping address of the load address, and the load response data can be generated by using only data received from the memory hierarchy. If the head entry points to a location in the hash table memory, the entries of the linked list can be traced or walked to find all of the stores having the same hash value as the load instruction. It should be noted that having the same hash value is not necessarily the same as having the same address. For example, addresses 32FC0030 and FFFFFFF30 can both have the hash value of 30 when the hash function is a simple truncation. Thus, the addresses of the stores in the linked list can be compared to the load address to find stores to the same location as the load. The LSID of each of the matching

stores can be compared so that the response data can include the most recently (latest in program order) written store data. When byte enables are used, the response data will include the most recently written store data for each byte and can include data from the memory hierarchy when a particular byte has not been written by an issued store instruction.

[0131] Additionally or alternatively, the hash data structures 876 can include a Bloom filter data structure. For example, the Bloom filter data structure can be a bit vector where each bit of the vector corresponds to a different value of the hash function used to index the Bloom filter so that all of the possible values of the hash function have a corresponding bit in the Bloom filter. The Bloom filter can be initialized so that all bits are deasserted (e.g., all zeroes) when a new instruction block is fetched. The Bloom filter can provide an indication of whether a target address of an enqueued store instruction may match a target address of an enqueued load instruction. For example, when a store instruction is issued, a hash value of the target address of the store instruction can be generated, and the corresponding bit can be asserted or set in the Bloom filter. For each load instruction, a hash value of the target address of the load instruction can be generated and the hash value can be used to index the Bloom filter. If the bit corresponding to the hash value of the load instruction is not asserted, then there is no store instruction enqueued at the same target address. The load response data can be generated by accessing the memory hierarchy and a search of the queue memory 874 is not performed. However, if the bit corresponding to the hash value of the load instruction is asserted, then a store instruction to the same target address may be enqueued. Specifically, the load instruction and at least one of the store instructions have the same hash value, but may still have different addresses. The load response logic 880 can search the queue memory 874 for store instructions with LSIDs less than (earlier in program order) the load instruction to determine if there are store instructions with matching addresses. For example, the store mask can be used to identify all of the store instructions with LSIDs less than the load instruction. By searching from the most recent store instruction (later in program order) relative to the load instruction, to the least recent store instruction, the search may end before searching all store instructions. For example, if the most recent store instruction stored all bytes at an overlapping address of a load instruction, the search can end since the most recent store contains the data for the load response.

[0132] The memory interface logic 878 can be used to issue a request to the memory hierarchy for load data at a particular address. Because a response time of the memory

hierarchy can be relatively slow compared to the execution speed of instructions, the request for load data to the memory hierarchy can be sent when the load instruction is issued to potentially hide the latency of obtaining the load data. The memory interface can include local buffering for storing data returned from memory before the load response is built.

[0133] The commit logic 882 can be used for managing a commit phase and/or a post-commit phase of the instruction block and draining store data from the load-store queue 870. One condition for committing an instruction block can be that all stores identified in the store mask have issued. The commit logic 882 can test for this condition, such as by monitoring the pending-store bitmask (e.g., when all bits in the pending-store bitmask are deasserted, all stores have issued). Other conditions may also be used to determine if the instruction block can commit. When all conditions for committing the block are satisfied, the commit logic 882 can begin the commit phase of the instruction block. In one embodiment, the commit phase of one instruction block can overlap with the execution phase of one or more instruction blocks that follow the committing instruction block.

[0134] The post-commit phase can include draining issued store instructions from the queue memory 874 by writing the store data to the memory hierarchy. For example, the memory interface 878 can be used to issue a store request for each of the store instructions to be committed. Alternatively, stores can be merged so that fewer store requests are sent to the memory hierarchy. For example, stores that are overwritten by a later store can be omitted, and stores to different bytes within a word can be merged into a single store for the word. The progress of the post-commit phase can be tracked using a committed-store bitmask which can be initialized with the store mask of the committing block. As stores are written to the memory hierarchy, the corresponding bits in the committed-store bitmask can be deasserted. When all bits in the committed-store bitmask are deasserted, the post-commit phase can complete. The hash data structures 876 can also be updated as the stores are written to the memory hierarchy. Specifically, the entries of the linked list corresponding to the store being written can be removed from the linked list and the hash table memory can be updated as the stores are written to the memory hierarchy. Alternatively, the entries of the linked list corresponding to the stores of the instruction block can be removed from the linked list and the hash table memory at the end of the post-commit phase after the stores are written to the memory hierarchy.

X. Example Methods of Load-Store Queues

[0135] FIG. 9 is a flowchart illustrating an example method 900 of processing instructions

received by a load-store queue, such as can be performed by the load-store intake logic 872 of FIG. 8.

[0136] At process block 910, a load/store request is received. For example, the load/store request can correspond to a load or store instruction issued by issue logic of a block-based processor core. The request can include information such as whether the instruction is a load or store, an address to load data from or store data at, an LSID indicating a relative program order of the instruction within an instruction block, an instruction number within an instruction window, byte enables, store data (if the instruction is a store), and/or other various information related to the instruction, instruction block, and/or processor core.

[0137] At process block 920, a hash value can be generated for the address of the load/store request. For example, the hash value can be generated using one or more bits of the address. As one example, the hash value can be generated by truncating the lower bits of a word address and using the lower bits of the word address as the hash value. The word address is the byte address without the lower bits corresponding to the individual bytes of the word. As another example, the hash function can perform logical operations on all or a portion of the address bits. For example, the hash function can include an exclusive-or tree that pairs different address bits together and performs an exclusive-or operation on the bits to generate a hash value with fewer bits than the input value.

[0138] At process block 930, the load/store request can be enqueued or stored in local memory of the load-store queue. All or a portion of the information received in the load/store request can be stored in the local memory. As a specific example, the load-store queue can store load/store requests for two instruction blocks. For example, one of the instruction blocks can be in the commit phase and one of the instruction blocks can be in the execution phase. The ISA can specify a maximum number of load and store instructions per instruction block (and hence a maximum LSID). In one embodiment, the local memory can be sized to hold a maximum number of load and store instructions that can be executed in a single instruction block. In another embodiment, the local memory can be sized to hold twice the number of load and store instructions that can be executed in a single instruction block. The LSID of the load/store request can be mapped to the lower address bits of the local memory and the upper bit of the load/store request can indicate the source instruction block of the load/store instruction.

[0139] At process block 940, it can be determined whether the load/store request is for a store instruction or a load instruction and different operations can occur based on whether the request is for a load or a store. If the request is for a load instruction, the method 900

can continue at process block 950. However, if the request is for a store instruction, the method 900 can continue at process block 960.

[0140] At process block 950, an issued-load data structure can be updated. The issued-load data structure can indicate which load instructions of the instruction block have been
5 issued but not completed. For example, the issued-load data structure can be a bitmask where each bit of the bitmask corresponds to an LSID. When a load request is received, the bit corresponding to the LSID of the load instruction can be asserted or set in the bitmask. When load response data is generated and forwarded to the processor core, the bit corresponding to the LSID of the completing load instruction can be deasserted or cleared
10 in the bitmask.

[0141] At process block 960, a hash data structure can be updated. For example, the hash data structure can include a hash table and the hash table can be updated. Specifically, the LSID of the store instruction can be added to a hash table bucket corresponding to the hash value of the address of the store instruction. The hash table buckets can be used to
15 find all of the store requests stored in the local memory that have the same hash value. The hash table buckets can be organized in various ways and can include an entry for each store request in the hash table bucket. For example, the hash table buckets can be organized as a singly linked-list, a doubly linked-list, or a list. A singly linked-list can include a header that points to first entry of the list, and each entry can include a pointer
20 that points to the next entry of the list. A pointer to a null or sentinel value (such as a value greater than the maximum LSID) can indicate there are no more entries in the list. For example, a header pointing to a null value can indicate that the hash bucket is empty. The pointer can indicate the LSID of the entry that is pointed to. The hash table entries can include all or a portion of the information received in the store request. As another
25 example, the hash data structure can include a Bloom filter data structure and the Bloom filter can be updated. Specifically, the Bloom filter can include a bit vector, and the bit corresponding to the hash value of the address of the store instruction can be asserted.

[0142] At process block 970, a pending-store data structure can be updated. The pending-store data structure can indicate which store instructions of the instruction block have not
30 yet issued. For example, the pending-store data structure can be a bitmask where each bit of the bitmask corresponds to an LSID. When an instruction block begins execution, the pending-store bitmask can be initialized a copy of the store mask, where the store mask identifies all of the store instructions (by LSID) of an instruction block. Thus, after initialization the pending-store bitmask can have a bit asserted or set for each LSID of

every store instruction of the instruction block. When a store request is received, the bit corresponding to the LSID of the store instruction can be deasserted or cleared in the bitmask.

[0143] FIG. 10 is a flowchart illustrating an example method 1000 of load response logic of a load-store queue, such as can be performed by the load response logic 880 of FIG. 8.

[0144] At process block 1010, it can be determined if there is a load request stored in the local memory of the load-store queue. The load requests stored in the local memory correspond to issued load instructions. If there is a load request stored in the local memory, the method 1000 can continue at process block 1020. However, if there is not a load request stored in the local memory, the method 1000 can end.

[0145] At process block 1020, it can be determined if there is an issued load request having an LSID less than the LSID of the lowest pending store request. In other words, it can be determined if there are any issued loads that are earlier in program order than the earliest pending store request. If there are no issued load requests having LSIDs less than the LSID of the lowest pending store request, then none of the load requests can be completed non-speculatively because a store request with a lower LSID (earlier in program order) may write to the address of the load request and change the value of the data at the load address. However, if there is an issued load request having an LSID less than the LSID of the lowest pending store request, then the load request can be completed non-speculatively because no store requests earlier in program order will overwrite the load data at the load address. If there is an issued load request having an LSID less than the LSID of the lowest pending store request, the method 1000 can continue at process block 1030.

[0146] At process block 1030, the method 1000 can begin to generate response data for one of the load instructions (referred to as the load instruction being completed or the completing load instruction) stored in the local memory. The completing load instruction can be any load instruction having an LSID that is less than the lowest pending store's LSID. In other words, the completing load instruction can be any load request that is earlier in program order than the earliest pending store in program order. Generating the response data for the completing load instruction can begin by generating a hash value of the address of the completing load instruction. The hash value is generated by the same hash function that generates hash values in the load/store intake logic. The hash value can be used as an index into the hash table. For example, the hash value can be used to identify the hash bucket that may potentially include stores to the same hashed address of the

completing load instruction.

[0147] At process block 1040, a load request can be sent to the memory hierarchy to retrieve data stored at the address of the completing load instruction. The data returned from the memory hierarchy may be used in the response data for the completing load instruction. For example, if no issued or committed (and not written back to memory) stores have written to the address, the data from the memory hierarchy is used for the load response data of the completing load instruction. The memory hierarchy can be relatively slow to respond relative to the execution speed of the instructions. Thus, the sooner the load request is sent to the memory hierarchy, the earlier that the load response data can potentially be completed. For example, the load request to the memory hierarchy can occur as soon as the load instruction is issued. As illustrated, the load request to the memory hierarchy occurs after the completing load instruction is identified.

[0148] At process block 1050, it can be determined if there is a hit in the hash data structure. For example, a hit can indicate that the local memory contains a store request with a target address matching the target address of the load request. A miss can indicate that the local memory does not contain any store requests with target addresses matching the target address of the load request. As one example, a miss can occur when the hash bucket corresponding to the address of the completing load instruction is empty. If the hash bucket is empty, there are no issued or committed stores to the same hashed address, and the method 1000 can continue at process block 1080. If the hash bucket is not empty (a hit), a search of the entries of the hash bucket can be performed at process block 1060. As another example, a miss can occur when the Bloom filter bit corresponding to the address of the completing load instruction is deasserted. In contrast, a hit can occur when the Bloom filter bit corresponding to the address of the completing load instruction is asserted. When both the Bloom filter and the hash table are used, the method 1000 can continue at process block 1080 when a miss occurs in either structure. The method 1000 can continue at process block 1060 when a hit occurs in both structures.

[0149] At process block 1060, load response data can be generated using forwarded data from the issued stores. When a hash table is used, the hash bucket corresponding to the address of the completing load instruction can be searched. Each entry within the hash bucket corresponds to an issued or committed store instruction that has not yet been written to the memory hierarchy. The hash bucket can include stores to different addresses that have the same hash value. Thus, the addresses of the hash bucket entries can be compared to the address of the completing load instruction to determine if the store

instruction was to the same address. The hash bucket can include issued stores that are later in program order than the completing load instruction. These issued stores later in program order can be ignored when generating the response data because they do not affect the value of the response data. The hash bucket can include issued stores that are

5 earlier in program order than the completing load instruction. The data written by the issued stores earlier in program order to the same address as the completing load instruction may be used to generate the response data. In particular, the most recent (latest in program order) stores take precedence over the less recent stores when generating the response data. The response data can be generated on a byte-by-byte basis, based on the

10 byte enables of the store instructions. A data structure and an example of building the response data are described in further detail with reference to FIG. 11 below. Any data from the store instructions not yet written to the memory hierarchy and used to generate the response data is referred to as being forwarded from the store instructions.

[0150] When a Bloom filter is used, the local memory can be searched for issued stores

15 with a target address matching the target address of the completing load instruction. For example, the store mask can be used to identify all store instructions earlier in program order than the completing load instruction. Specifically, the store instructions earlier in program order at the store instructions having LSIDs less than the LSID of the completing load instruction. The local memory can be searched, starting with the store instruction

20 having the LSID closest to and earlier than the LSID of the completing load instruction. If the target addresses of the load and store instructions match, the response data can be generated on a byte-by-byte basis until all bytes are present. When all bytes are present, the search can end. Alternatively, the search can end when all store instructions having LSIDs less than the LSID of the completing load instruction have been analyzed.

25 **[0151]** When both the hash table and the Bloom filter are used, the search can end when the first search completes. The load response data generated using the hash table and the Bloom filter will yield the same result, so the method 1000 can potentially be accelerated by continuing at process block 1070 as soon as either of the search methods yields load response data.

30 **[0152]** At process block 1070, it can be determined if all of the bytes of the load response for the completing load instruction have been filled with forwarded store data. If all of the bytes have been filled with forwarded store data, the data read from the memory hierarchy is not used to generate the response data since the data stored in the memory hierarchy will be overwritten with the store data. The method 1000 can continue at process block 1090 if

the data from the memory hierarchy is not used for the response data. However, if at least one of the bytes have not been filled with forwarded store data, the data read from the memory hierarchy will be used to generate the response data. The method 1000 can continue at process block 1080 if the data from the memory hierarchy is used for the response data.

[0153] At process block 1070, the data at the address of the completing load instruction can be received from the memory hierarchy. Any bytes that are not filled with forwarded store data can be filled with the data received from the memory hierarchy so that the load response data can be completed.

[0154] At process block 1080, the generated load response data can be returned to the processor core. For example, the generated load response data can be returned to the execution logic and/or one or more operand buffers of the instruction windows. The generated load response data can also be returned to the out-of-order dataflow instruction scheduler to cause an instruction which consumes the load response data to become ready to issue. The issued load data structure can be updated when the load response data is returned. For example, the bit corresponding to the LSID of the completing load instruction can be deasserted or cleared in the issued-load bitmask.

[0155] FIG. 11 illustrates example data structures 1100 that can be used by load response logic of a load-store queue. The data structures 1100 can include a data response register 1110, a source data register 1120, and a hash table 1130. In some examples, the data structures 1100 are implemented at least in part using one or more of: hardwired finite state machines, programmable microcode, programmable gate arrays, programmable processors, random access memory, or other suitable control circuits. As one example, the hash bucket headers 1132 can include a RAM (such as a RAM with one read port and one write port) storing an array of hash table pointers, one for every possible hash value. As another example, the hash memory 1140 can include a RAM (such as a RAM with one read port and one write port) storing an array of pointers, one entry for every possible non-NULL pointer value, or equivalently one for every entry in the queue memory 874. Each pointer can be a one-bit block index concatenated with an LSID, or can be a distinguished NULL value, for example, a reserved block index and/or a reserved LSID value. When implemented on an FPGA, the hash table can be implemented using one or more block RAMs or LUT RAMs. However, the hash table may be more compact when a block RAM is used. The hash bucket headers can be implemented using one or more block RAMs or LUT RAMs.

- [0156] It should be noted that the addresses of both load and store instructions can be stored in hash tables to potentially increase the performance of the processor. As one example, a load-store queue can include functionality for performing dependency prediction speculation. In particular, the load-store queue can predict that an issued load
- 5 does not have a RAW dependency on a yet-to-be-issued store (e.g., when the store issues, it will be to a different, disjoint address) and thereby issue a speculative load response even before the store with the prior (e.g., lower) LSID has been issued. In some embodiments, speculated loads can be tracked by adding their addresses to a set of hash table buckets similar to the hash buckets 1132, but tracking speculated loads to certain
- 10 addresses. In some embodiments, when a later-issued store instruction issues to the load-store queue, its address can be hashed and compared with speculated loads in the corresponding hash bucket in the speculated loads hash table, in order to detect a speculated-RAW misspeculation. In some embodiments, a load-store queue misspeculation can be handled by aborting or reexecuting the block.
- 15 [0157] The example 700 of FIG. 7 can be used in conjunction with FIG. 11 to illustrate how the data structures 1100 operate. Table 1 summarizes the relevant instructions from the instruction block 710:

Instruction	LSID	Byte Address	Word Address	Byte Enables	Write/Read (Word) Data
Load Word	0	00	00	1111	1111
Store Half-Word	1	20	08	0011	00BB
Store Byte	2	00	00	0001	000A
Store Byte	3	03	00	1000	D000
Store Word	4	20	08	1111	3210
Load Word	5	00	00	1111	D11A

- [0158] The LSID can identify the individual load/store instructions within the instruction
- 20 block and can indicate the program order of the instructions. The program order can be different from the execution order. For example, both loads to word address 00 can be issued before the stores, and the store instruction with LSID = 3 (store 3) can be issued before the store instruction with LSID = 2 (store 2). Because the load word instruction with LSID = 0 (load 0) is earlier in program order than the store instructions, the load 0

can be completed before the store instructions are issued. The “1111” data retrieved by the load 0 reflects the state of the memory hierarchy before the instruction block 710 executes. The load word instruction with LSID = 5 (load 5) is later in program order than the store instructions and so the response data for the load 5 will include forwarded store data from the store instructions. For correct program operation, the response for load 5 cannot be generated until the store 2 and the store 3 have issued. However, the load and store addresses can be dynamically generated at run-time so the addresses may not be known until run-time. Therefore, the response for load 5 will not be generated until the stores 1-4 have issued.

10 **[0159]** The hash table 1130 can be updated as each store is issued. The hash table 1130 can include hash bucket headers 1132 and hash memory 1140. The hash bucket headers 1132 can include a header bucket for each hash value that can be generated from the hash function. In this example, the hash values can be generated by truncating the word address. Thus, word address 0 (byte addresses 0-3) has a hash value of 0, word address 1 (byte addresses 4-7) has a hash value of 1, word address 8 (byte addresses 20-23) has a hash value of 8, and so forth. The header buckets are initialized to point to null values. When a new store request is issued, a hash value of the address of the store is calculated, and a reference to the store is added to the hash table 1130 using the hash value as an index to the hash value. For example, when store 3 is issued, the hash value of the store 3 address is 0. Information associated with the store 3 can be stored at a hash memory entry 1143 and the header bucket 0 can be updated with a pointer to the hash memory entry 1143. Since store 3 is the first entry in the bucket 0, a next pointer of the hash memory entry 1143 can point to a null value. It should be noted that FIG. 11 is illustrated to show the state of the hash table 1130 after the next operation of the hash table 1130. When store 2 is issued, the hash value of the store 2 address is 0. Information associated with the store 2 can be stored at a hash memory entry 1142 and the header bucket 0 can be updated with a pointer to the hash memory entry 1143. The hash memory entry 1142 can include a next pointer to the hash memory entry 1143. Thus, the hash memory entries form a linked list data structure for the bucket within the hash table. The ISA can specify a maximum number of load and store instructions per instruction block (and hence a maximum LSID). In one embodiment, the hash memory 1140 can include a number of hash memory entries that is twice the number of store instructions that can be executed in a single instruction block. The LSID of the store request of the hash memory entry can be mapped to the lower address bits of the hash memory 1140 and the upper bit of the store request can indicate the source

instruction block of the store instruction. Stores 1 and 4 have a word address of 08 and so they will map to a different hash bucket than stores 2 and 3 when there are more than eight hash buckets in the hash table 1130 or when a different hash function is used.

[0160] Load response data can be generated based on a search of the hash table 1130. In
5 some embodiments the search may proceed serially, reading one entry from the hash table 1130 at a time. In some embodiments, possibly with different hash table organizations, the search may proceed with a greater degree of parallelism, such as by reading more than one entry from the hash table 1130 in a given cycle. The data response register 1110 can be used for storing intermediate results when generating the load response data. The source
10 data register 1120 can be used to indicate the source of the data bytes stored in the data response register 1110. When new response data is to be generated, the source data register 1120 can be initialized by storing null or sentinel values in each of the source byte registers. For example, the load 5 response can be generated after all of the stores 1-4 have issued. The source data register 1120 can be initialized with null values. The hash table
15 1130 can be searched for stores that have the same the hash value (0) as the load 5. In particular, the header bucket 0 can be accessed and the hash memory entry 1142 can be identified as the head of the linked list. Accessing the hash memory entry 1142 can provide information about store 2, such as the word address is 0 (a match to address of the load 5), the LSID is 2, the byte enables are "0001," and the store data is "000A." Based on
20 this information, data byte 0 of the data response register 1110 can be updated with an "A" and source byte 0 of the source data register 1120 can be updated with a 2 to indicate the relative program order of the store 2. Only data byte 0 of the data response register 1110 and source byte 0 of the source data register 1120 are updated because only byte 0 is enabled for the store 2. The next entry in the linked list for hash bucket 0 is the hash
25 memory entry 1143 which can provide information about store 3, such as the word address is 0 (a match to address of the load 5), the LSID is 3, the byte enables are "1000," and the store data is "D000." Based on this information, data byte 3 of the data response register 1110 can be updated with a "D" and source byte 3 of the source data register 1120 can be updated with a 3 to indicate the relative program order of the store 3. Only data byte 3 of
30 the data response register 1110 is updated because it is the only byte enabled by the store 3. It should be noted that if byte 0 were enabled for the store 3, the information corresponding to store 3 would overwrite the information from the store 2 because the store 3 is later in program order (and still earlier than the load 5). The search of the hash table 1130 can end when the null is encountered at the end of the linked list of hash bucket

0.

[0161] At the end of the search of the hash table 1130, the source bytes 1 and 2 of the source data register 1120 have not been updated with a value different from the null value. When a source byte of the source data register 1120 is null after the search of the hash
 5 table 1130, the corresponding data bytes of the data response register 1110 are updated with values that are retrieved from the memory hierarchy. In this example, each of the data bytes 1 and 2 can be updated with a “1” so that the value of the data response register 1110 is “D11A.” This value can be returned to the execution logic and/or instruction window(s) of the processor core for the load 5.

10 **[0162]** FIG. 12 is a flowchart illustrating an example method 1200 of commit logic of a load-store queue, such as can be performed by the commit logic 882 of FIG. 8. During execution of an instruction block, updates to the visible architectural state (such as the memory hierarchy) can be buffered locally (such as in a load-store queue). If the instruction block is aborted, the locally buffered updates can be dropped before any
 15 updates are made to the visible architectural state. However, if the instruction block completes successfully, the locally buffered updates can be written to the visible architectural state during a commit phase of the instruction block. The instruction block is committed at the end of the commit phase.

[0163] At process block 1210, it can be determined if all store instructions of the
 20 instruction block are received. For example, a store mask can be used to identify all of the store instructions of the instruction block. When store instructions have issued for all of the store instructions identified by the store mask, the method can continue at process block 1220.

[0164] At process block 1220, it can be determined if the instruction block preceding the
 25 current instruction block is committed. For example, the instruction blocks can be ordered with respect to each other and the instruction blocks can be committed in order. As another example, the local memory and hash table memory of the load-store queue can be sized to hold instructions for only two in-flight instruction blocks. As a specific example, one instruction block can be executing while the preceding instruction block is in the
 30 commit phase. Thus, local buffer space in the load-store queue may not be available until the preceding instruction block is committed. The local memory can be divided into two areas, where one area is stores load-store information for the executing block and one area stores load-store information for the committed block. As the instructions of the committed block are drained and the next instruction completes execution, the areas of the

local memory can be swapped or ping-pong'ed between the activities of storing load-store information of the executing block and storing load-store information of the committed block. In an alternative embodiment, the local memory can be sized to hold instructions for a single in-flight instruction block. On the one hand, the processor core can be smaller
5 when a smaller local memory is used, but on the other hand, the performance may be reduced. For example, the stores of a committed instruction block must be drained from the local memory (by writing the data to the memory hierarchy) before the next instruction block can reuse the local memory. If the preceding instruction block is committed, the method 1200 can continue at process block 1230.

10 **[0165]** At process block 1230, it can be determined if other commit conditions of the instruction block are met. For example, an instruction block can be committed after all updates to the visible architectural state are calculated. Specifically, the commit conditions can include that all store instructions have issued, all register writes have issued, and a target address for the next instruction block is determined. When all commit conditions are
15 satisfied, the instruction block can begin the commit phase of the instruction block at process block 1240. At the beginning of the commit phase, the issued-store data structure or the store mask can be copied to a committed-store data structure. Thus, the committed-store data structure can indicate which of the store instructions are committed but are waiting to be written to the memory hierarchy. As one example, the committed-store data
20 structure can be a bitmask having a bit for each LSID and where a bit is set or asserted for each store instruction to commit.

[0166] At process block 1240, the commit phase of the instruction block is active. In particular, the committed store with the lowest LSID can be issued to the memory hierarchy. For example, the committed-store data structure can be used to determine which
25 committed store has the lowest LSID. Thus, the committed store instruction that is earliest in program order can be written to the memory hierarchy before store instructions that are later in program order. By following this order, write-after-write dependencies can be satisfied.

[0167] At process block 1250, the hash memory entries corresponding to the committed
30 store with the lowest LSID can be removed from the hash table (such as by updating the linked list). The committed store data structure can be updated to reflect that the committed store has been written to memory. For example the bit corresponding to the committed store with the lowest LSID can be cleared.

[0168] At process block 1260, it can be determined if there are more store instructions to

commit. For example, the committed-store data structure can be analyzed to determine if there are more store instructions to commit. As a specific example, a committed-store bitmask with all bits deasserted can indicate that all of the committed store instructions have been written to the memory hierarchy. If there are still committed store instructions
5 to write to the memory hierarchy, the method 1200 can continue at process block 1240. If all committed store instructions for the instruction block have been written to the memory hierarchy, the method 1200 can continue at process block 1270.

[0169] At process block 1270, a commit signal can be issued for the instruction block indicating the instruction block is finished committing. The commit signal can trigger
10 additional operations, such as fetching or executing another instruction block.

[0170] FIG. 13 is a flowchart illustrating an example method 1300 of executing an instruction block on a block-based processor core. At process block 1310, load and store instructions issued by a processor can be enqueued. For example, the load and store instructions can be enqueued in a load-store buffer. One or more of the load and store
15 instructions can be issued out of program order. Each of the load and store instructions can include a load-store identifier specifying a relative program order of the respective instructions. For example, the relative program order can indicate the relative program order of the load and store instructions with respect to each other.

[0171] At process block 1320, hash values can be generated for addresses of the load and
20 store instructions. The addresses of the load and store instructions can be generated dynamically at run-time based on operands of the load and store instructions, for example. The hash values can be generated by a hash function. One example of the hash function is to truncate a word address of the load and store instructions.

[0172] At process block 1330, a hash data structure can be updated based on the generated
25 hash values of the enqueued store instructions. As one example, the hash data structure can include a hash table, and references to the enqueued store instructions can be stored in the hash table using the generated hash values as an index of the hash table. For example, hash buckets of the hash table can correspond to the hash values generated by the hash function. References to the store instructions can be stored in the hash table at the buckets
30 corresponding to the hash value of the address of the store instruction. When multiple store instructions map to the same hash bucket, references to the store instructions can be stored using a linked list corresponding to the hash bucket. As another example, the hash data structure can include a Bloom filter data structure, and the generated hash values of the enqueued store instructions can be used to assert bits within the Bloom filter. The

asserted bits can indicate the presence of enqueued store instructions having target addresses corresponding to the generated hash values.

[0173] At process block 1340, for each of the load instructions, the hash data structure can be searched and load response data can be generated for the load instructions based on the search. As one example, the hash data structure can include a hash table, and the hash table can be searched for enqueued store instructions having a matching hash value and address. As another example, the hash data structure can include a Bloom filter, and it can be determined if there is a hit in the Bloom filter for the hash value corresponding to the load instruction. If there is a hit, the store instructions enqueued in the local memory can be searched for enqueued store instructions having a target address matching the target address of the load instruction. The store mask can be used to identify the store instructions. Byte enables and the relative program order of the matching enqueued stores can be used to build load response data for a given load instruction. When the enqueued stores have data corresponding to all enabled bytes of the given load instruction, the load response data can be completed without retrieving data from the memory hierarchy. However, when the enqueued stores do not have data corresponding to all enabled bytes of the given load instruction, the load response data can be completed after retrieving data from the memory hierarchy.

[0174] At process block 1350, the generated load response data can be forwarded to an execution unit (such as an ALU) of the processor. Additionally or alternatively, the generated load response data can be forwarded to an instruction window of the processor.

[0175] FIG. 14 is a flowchart illustrating an example method 1400 of executing an instruction block on a block-based processor core. At process block 1410, a plurality of issued load and store instructions for an instruction block can be received. At least one of the issued instructions can be issued out of program order. A relative program order of the issued instructions can be determined by decoding a load-store identifier of the instructions.

[0176] At process block 1420, an address of at least one of the issued store instructions can be stored in a local memory indexed with a hash value based on the address. Byte enables, store data, and the load-store identifier can also be stored in the local memory.

[0177] At process block 1430, load response data can be generated by accessing the local memory using an index based on an address of a load instruction. For example, all of the store instructions having the same hash value can be located by using an index. The store instructions can be analyzed to determine if the data written by the store instructions can

be used to build the load response data. For example, generating the load response data can include merging byte-enabled data from the issued store instructions. If the write data of the issued store instructions does not overlap all of the bytes of the data to load, generating the load response data can include merging byte-enabled data from the issued store instructions and data retrieved from a memory hierarchy of the processor.

[0178] At process block 1440, the load response data can be forwarded to execution logic of the processor. For example, the load instruction can be completed by the execution logic.

XI. Example Computing Environments

[0179] FIG. 15 illustrates a generalized example of a suitable computing environment 1500 in which the described embodiments, techniques, and technologies can be implemented.

[0180] The computing environment 1500 is not intended to suggest any limitation as to scope of use or functionality of the technology, as the technology may be implemented in diverse general-purpose or special-purpose computing environments. For example, the disclosed technology may be implemented with other computer system configurations, including hand held devices, multi-processor systems, programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The disclosed technology may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules (including executable instructions for block-based instruction blocks) may be located in both local and remote memory storage devices.

[0181] With reference to FIG. 15, the computing environment 1500 includes at least one block-based processing unit 1510 and memory 1520. In FIG. 15, this most basic configuration 1530 is included within a dashed line. The block-based processing unit 1510 executes computer-executable instructions and may be a real or a virtual processor. In a multi-processing system, multiple processing units execute computer-executable instructions to increase processing power and as such, multiple processors can be running simultaneously. The memory 1520 may be volatile memory (*e.g.*, registers, cache, RAM), non-volatile memory (*e.g.*, ROM, EEPROM, flash memory, *etc.*), or some combination of the two. The memory 1520 stores software 1580, images, and video that can, for example, implement the technologies described herein. A computing environment may have additional features. For example, the computing environment 1500 includes storage 1540,

one or more input devices 1550, one or more output devices 1560, and one or more communication connections 1570. An interconnection mechanism (not shown) such as a bus, a controller, or a network, interconnects the components of the computing environment 1500. Typically, operating system software (not shown) provides an
5 operating environment for other software executing in the computing environment 1500, and coordinates activities of the components of the computing environment 1500.

[0182] The storage 1540 may be removable or non-removable, and includes magnetic disks, magnetic tapes or cassettes, CD-ROMs, CD-RWs, DVDs, or any other medium which can be used to store information and that can be accessed within the computing
10 environment 1500. The storage 1540 stores instructions for the software 1580, plugin data, and messages, which can be used to implement technologies described herein.

[0183] The input device(s) 1550 may be a touch input device, such as a keyboard, keypad, mouse, touch screen display, pen, or trackball, a voice input device, a scanning device, or another device, that provides input to the computing environment 1500. For audio, the
15 input device(s) 1550 may be a sound card or similar device that accepts audio input in analog or digital form, or a CD-ROM reader that provides audio samples to the computing environment 1500. The output device(s) 1560 may be a display, printer, speaker, CD-writer, or another device that provides output from the computing environment 1500.

[0184] The communication connection(s) 1570 enable communication over a
20 communication medium (*e.g.*, a connecting network) to another computing entity. The communication medium conveys information such as computer-executable instructions, compressed graphics information, video, or other data in a modulated data signal. The communication connection(s) 1570 are not limited to wired connections (*e.g.*, megabit or gigabit Ethernet, Infiniband, Fibre Channel over electrical or fiber optic connections) but
25 also include wireless technologies (*e.g.*, RF connections via Bluetooth, WiFi (IEEE 802.11a/b/n), WiMax, cellular, satellite, laser, infrared) and other suitable communication connections for providing a network connection for the disclosed agents, bridges, and agent data consumers. In a virtual host environment, the communication(s) connections can be a virtualized network connection provided by the virtual host.

[0185] Some embodiments of the disclosed methods can be performed using computer-executable instructions implementing all or a portion of the disclosed technology in a
30 computing cloud 1590. For example, disclosed compilers and/or block-based-processor servers are located in the computing environment 1530, or the disclosed compilers can be executed on servers located in the computing cloud 1590. In some examples, the disclosed

compilers execute on traditional central processing units (*e.g.*, RISC or CISC processors).

[0186] Computer-readable media are any available media that can be accessed within a computing environment 1500. By way of example, and not limitation, with the computing environment 1500, computer-readable media include memory 1520 and/or storage 1540.

5 As should be readily understood, the term computer-readable storage media includes the media for data storage such as memory 1520 and storage 1540, and not transmission media such as modulated data signals.

[0187] FIG. 16 is a block diagram 1600 that depicts an example field programmable gate array (FPGA) architecture that is configured to implement certain examples of the disclosed technology. For example, the block-based processor 100 discussed above
10 regarding FIG. 1, including those examples that used the microarchitecture 200 depicted in FIG. 2 can be mapped to the FPGA architecture of FIG. 16. As another example, the block-based processor 800 discussed above regarding FIG. 8 can be mapped to the FPGA architecture of FIG. 16

15 **[0188]** The FPGA includes an array of reconfigurable logic blocks arranged in an array. For example, the FPGA includes a first row of logic blocks, including logic blocks 1610, 1611, and 1619, and a second row of logic blocks including logic blocks 1620, 1621, and 1629. Each of the logic blocks includes logic that can be reconfigured to implement arbitrary logic functions and can also include sequential logic elements such as latches,
20 flip-flops, and memories. The logic blocks are interconnected to each other using a routing fabric that includes a number of interconnect switches that can also be programmable. For example, there is a first row of switch blocks 1630, 1631, 1632, *etc.*, positioned between the first row of reconfigurable logic blocks and the second row of reconfigurable logic blocks. The switches can be configured in order to change wire connections that carry
25 signals between the reconfigurable logic blocks. For example, fetch logic, decode logic, instruction schedulers, load-store queues, functional or execution units, pipeline buffers, operand buffers, and other components of a processor can be mapped to the logic blocks connected using the switch blocks of FIG. 16.

[0189] The FPGA also includes a number of more complex components. For example, the
30 logic block includes a number of block RAMs, for example, block RAM 1640 and block RAM 1649. The block RAMs typically contain a larger number of memory bits, for example, a few thousand memory bits that are accessed by applying an address to the memory, and reading from one or more read ports. In some examples, the block RAMs can include two or more write ports and two or more read ports. In other examples, the

block RAMs may only have a single read and/or a single write port. While the block RAMs are typically accessed by applying an address and reading corresponding data, in some examples, the block RAMs can be configured with additional circuitry that allows for implementation of more complex functions including shift registers and First-In First-Out (FIFO) buffers.

[0190] The illustrated FPGA also includes a number of hard macro blocks including hard macro block 1650 and hard macro block 1659. These macro blocks can include more complex functionality such as processor functionality, digital signal processing functionality, accelerators, or other functions deemed to be desirable. The FPGA is further surrounded by an I/O ring 1660 that can be coupled to the logic blocks, the block rams, and/or the hard macro blocks in order to receive and send signals to components away from the FPGA. In some examples, the I/O signals are full rail voltage signals, while in other examples, differential signals are used. In some examples, the I/O ports can be multiplexed (*e.g.* time-multiplexed) in order to support input and output of more signals than the number of pins available on the FPGA.

[0191] While many examples of FPGAs are typically reconfigurable an arbitrary number of times through the use of electrically erasable memories, in other examples, one-time programmable logic elements can be used. For example, the logic blocks and switches can be programmed with the use of fuses, anti-fuses, or with a ROM mask to program a logic function once that is not easily reversible.

[0192] In the reconfigurable case, the FPGA typically has a configuration port that receives data according to a file dubbed a bitstream, or a configuration bitstream. The bitstream data is read into the device and used to program and configure the logic blocks, the switches, the block rams, and/or the hard macros. When a new design is desired, the configuration can be erased and a new design configured into the device. In some examples, the FPGA can be partially reconfigured in order to save on programming time. For example, a subset of the logic blocks, the switches, or block rams can be dynamically reconfigured in the field without reprogramming the entire device.

[0193] One challenge for block-based processor implementations mapped onto reconfigurable logic is determining micro-architectural structures that can be efficiently implemented using the available blocks of a custom or off-the-shelf device. However, using the disclosed technologies, higher performance, and/or more efficient structures can be implemented. Further, it should be readily understood that while some examples of the FPGAs are a stand-alone integrated circuit, in other examples, the FPGA may be packaged

differently, for example, in a multi-chip module (MCM), or on the same circuit die as a custom or basic system-on-chip (SoC).

[0194] FIG. 17 is a block diagram 1700 illustrating four reconfigurable logic blocks 1710, 1711, 1712, and 1713 that can be configured to form part of the logic fabric of an example
5 FPGA-integrated circuit. The components inside the reconfigurable logic blocks shown are identical, or homogenous, but it should be readily understood, in other examples, more than one type of reconfigurable logic block may be present on a single FPGA.

[0195] A first reconfigurable logic block 1710 includes a six-input Look Up Table (LUT) 1720 that is coupled to carry logic 1730, a number of multiplexers 1740 and 1745, and a
10 storage element (here, a D flip-flop) 1750. The LUT 1720 can be implemented using a small memory (for example, a memory having six address bits and two output bits as shown). Thus, any six-input Boolean function can be implemented by using a single LUT. In some examples, outputs of LUTs can be combined, or a reconfigurable logic block can have multiple LUTs that can be connected together in order to perform more complex
15 logic functions. In some examples, common logic functions can be providing in addition to the LUT. For example, the carry logic 1730 can be configured to perform the carry propagation logic for an adder. The multiplexers are used to select various output from other components. For example, the multiplexer 1740 can be used to select the output of either the LUT 1720 or the carry logic 1730, while the multiplexer 1745 can be used to
20 select another output of the LUT 1720 or the multiplexer 1740. In some examples, the multiplexer is used to either select a sequential output of a state element (*e.g.* flip-flop 1750), or a combinational output of a Look Up Table. It should be readily understood to one of ordinary skill in the art that different logic functions, LUT sizes, and sequential elements can be employed in a reconfigurable logic element. Thus, techniques for
25 mapping block-based processors to such reconfigurable logic can vary depending on the specific target FPGA architecture. The configuration of the logic inside the reconfigurable logic block can be programmed using the configuration port of the FPGA. In some examples, the LUTs are not programmed once, but can be configured to act as small memories that store certain data used in the block-based processor.

[0196] In some examples of the disclosed technology, a logic synthesis tool (logic
30 compiler) is used to transform a specification for a block-processor into a configuration bitstream that can be applied to a configuration port of an FPGA to configure logic to implement a block-based processor. In some examples, the designer can use an RPM (relationally placed macro) methodology to improve area and interconnect delays and

achieve a repeatable layout for easy routing and timing closure under module composition and massive replication. For example, by including structural RTL instantiating modules and tiling them into a processor core, logic for the processor core can be locked to a set of single LUTs, allow for a compact clustering and placement of logic within the FPGA.

5 **XII. Additional Examples of the Disclosed Technology**

[0197] Additional examples of the disclosed subject matter are discussed herein in accordance with the examples discussed above.

[0198] In one embodiment, a processor includes issue logic and a load-store buffer. The issue logic is configured to issue load and store instructions out of program order. Each of
 10 the load and store instructions include an identifier specifying a relative program order of the respective instructions. The load-store buffer is configured to enqueue the issued load and store instructions, and generate hash values for addresses of the load and store instructions. The load-store buffer is further configured to update a hash data structure using the generated hash values as an index of the hash data structure. As one example, the
 15 hash data structure can include a hash table, and updating the hash data structure can include storing references to the enqueued store instructions in the hash table using the generated hash values as an index of the hash table. As another example, the hash data structure can include a Bloom filter data structure, and updating the hash data structure can include asserting bits corresponding to the generated hash values of the enqueued store
 20 instructions in the Bloom filter data structure using the generated hash values as an index of the Bloom filter data structure. The load-store buffer is further configured to search the hash data structure and generate load response data for the load instructions, based on the search. The load-store buffer is further configured to forward the generated load response data to an execution unit of the processor.

25 [0199] When the hash data structure includes a hash table, generating load response data for the load instructions can include searching the hash table for store instructions with a matching hash value of the respective load instructions and matching the address of the respective load instructions. When the hash data structure includes a Bloom filter data structure, searching the hash data structure can include determining whether there is a hit
 30 in the Bloom filter data structure corresponding to the generated hash value for a respective load instruction. The load and store instructions can include byte enables, and the byte enables and the relative program order of the store instructions can be used to generate the load response data. The load response data for a respective load instruction can be forwarded to the execution unit when data corresponding to all enabled byte

enables of the respective load instruction is found during the search of the hash table and before a response from a memory hierarchy for the respective load instruction is received.

[0200] The load-store buffer can be further configured to receive a store mask indicating a relative position in program order of the store instructions within an instruction block. The

5 store mask can be used to determine when to search the hash data structure. For example, the store mask can be used to determine when to search the hash table for respective load instructions. In one example, the search of the hash table for a respective load instruction can occur only after all store instructions earlier in program order have been enqueued. The load-store buffer can be further configured to commit the instruction block only after
10 all store instructions indicated in the store mask have been enqueued, and committing the instruction block can include issuing a store request to a memory subsystem for at least one of the enqueued store instructions.

[0201] Load data can be requested from a memory subsystem when the respective issued load instructions are enqueued. The references to the enqueued store instructions can be
15 stored in the hash table using a linked-list. The load and store instructions can be arranged in instruction blocks, and references to the enqueued store instructions can be removed from the linked list of the hash table after a commit of the instruction block. The commit of the instruction block can include issuing at least one store request to a memory subsystem for the enqueued store instructions.

20 **[0202]** The processor can be used in a variety of different computing systems. For example, a server computer can include non-volatile memory and/or storage devices; a network connection; memory storing one or more instruction blocks; and the processor including the block-based processor core for executing the instruction blocks. As another example, a device can include a user-interface component; non-volatile memory and/or
25 storage devices; a cellular and/or network connection; memory storing one or more of the instruction blocks; and the processor including the block-based processor core for executing the instruction blocks. The user-interface component can include at least one or more of the following: a display, a touchscreen display, a haptic input/output device, a motion sensing input device, and/or a voice input device.

30 **[0203]** In one embodiment, a method includes, with a processor, receiving a plurality of issued load and store instructions for an instruction block. At least one of the issued instructions is issued out of program order. The method further includes updating a hash data structure indexed with a hash value based on an address of at least one of the issued store instructions. For example, the hash data structure can include a hash memory, and an

address of at least one of the issued store instructions can be stored in the hash memory indexed with a hash value based on the address. As another example, the hash data structure can include a Bloom filter, and updating the hash data structure can include asserting a bit of the Bloom filter corresponding to the hash value based on the address of
5 the at least one of the issued store instructions. The method further includes generating load response data by accessing the hash data structure using an index based on an address of a load instruction. The method further includes forwarding the load response data to execution logic of the processor for completing the load instruction.

[0204] As one example, the load response data of the load instruction can be generated
10 only after all store instructions earlier in program order of the instruction block have been issued. The method can further include receiving a store mask identifying the store instructions within the instruction block. For example, the store mask can be decoded from an instruction header of the instruction block. As another example, the store mask can be determined by decoding individual instructions of the instruction block so that all LSIDs
15 of the store instructions can be identified. The store mask can be used for determining when all store instructions earlier in program order of the instruction block have been issued. The method can further include committing the instruction block only after all store instructions identified by the store mask have issued.

[0205] Generating the load response data can include merging byte-enabled data from the
20 issued store instructions. Additionally or alternatively, generating the load response data can include merging byte-enabled data from the issued store instructions and data retrieved from a memory hierarchy of the processor.

[0206] One or more computer-readable storage media can store computer-readable instructions that, when executed by a computer, cause the computer to perform the
25 method.

[0207] In one embodiment, a block-based processor includes issue logic and a load-store queue. The issue logic is configured to issue load and store instructions of an instruction block out of program order. Each of the load and store instructions include an address and an identifier specifying a relative program order of the instruction within the instruction
30 block. The load-store queue includes a computer-readable memory configured to store the issued load and store instructions in relative program order using the identifiers specifying the relative program order. The load-store queue includes hash function logic configured to generate hash values for the addresses of the issued load and store instructions. The load-store queue includes a hash data structure configured to store information associated

with the respective issued store instructions. The information in the hash data structure is indexed by the hash values of the addresses of the load and store instructions. The load-store queue includes load response logic configured to generate load response data for the load instructions. The load response data for a respective load instruction is generated after
5 all issued store instructions earlier in the program order than the respective load instruction have been stored in the computer-readable memory. The load response data is generated by searching the hash data structure. As one example, the hash data structure can include a hash table, and the hash table can be searched for store instructions having addresses matching the address of the respective load instruction. As another example, the hash data
10 structure can include a Bloom filter, and the Bloom filter can be searched for a presence of store instructions having addresses matching the address of the respective load instruction.

[0208] The block-based processor can further include header decode logic in communication with the load-store queue. The header decode logic can be configured to decode a store mask from a header of the instruction block. The store mask can identify all
15 of the store instructions of the instruction block, and can be used by the load response logic to determine when to generate load response data for the load instructions. The load store queue can further include commit logic configured to commit the instruction block only after all of the store instructions identified by the store mask have issued.

[0209] In view of the many possible embodiments to which the principles of the disclosed
20 subject matter may be applied, it should be recognized that the illustrated embodiments are only preferred examples and should not be taken as limiting the scope of the claims to those preferred examples. Rather, the scope of the claimed subject matter is defined by the following claims. We therefore claim as our invention all that comes within the scope of these claims.

CLAIMS

1. A processor comprising:
issue logic configured to issue load and store instructions out of program order, each of the load and store instructions including an identifier specifying a relative program order of the respective instructions; and
a load-store buffer configured to:
enqueue the issued load and store instructions;
generate hash values for addresses of the enqueued load and store instructions;
update a hash data structure using the generated hash values for the issued store instructions as an index of the hash data structure;
for the load instructions, search the hash data structure and generate load response data for the enqueued, issued load instructions based on the search; and
forward the generated load response data to an execution unit of the processor.
2. The processor of claim 1, wherein the hash data structure includes a hash table, and the load response data for the load instructions is generated by searching the hash table for store instructions with a matching hash value of the respective load instructions and matching the address of the respective load instructions.
3. The processor of any one of claims 1 or 2, wherein the load-store buffer is further configured to receive a store mask indicating a relative position in program order of the store instructions within an instruction block, and the store mask is used to determine when to search the hash data structure.
4. The processor of claim 3, wherein the hash data structure includes a hash table, and the search of the hash table for a respective load instruction occurs only after all store instructions earlier in program order have been enqueued.
5. The processor of claim 4, wherein the load-store buffer is further configured to commit the instruction block only after all store instructions indicated in the store mask have been enqueued, and committing the instruction block comprises issuing a store request to a memory subsystem for at least one of the enqueued store instructions.
6. The processor of any one of claims 1-5, wherein the hash data structure includes a Bloom filter data structure and the search of the hash data structure comprises determining whether there is a hit in the Bloom filter data structure corresponding to the generated hash value for a respective load instruction.

7. The processor of any one of claims 1-6, wherein the references to the enqueued store instructions are stored in the hash data structure using a linked-list.

8. The processor of claim 7, wherein the load and store instructions are arranged in instruction blocks, and references to the enqueued store instructions are removed from the linked list of the hash data structure after a commit of the instruction block, the commit of the instruction block comprising issuing at least one store request to a memory subsystem for the enqueued store instructions.

9. A method comprising:

with a processor, receiving a plurality of issued load and store instructions for an instruction block, at least one of the issued instructions being issued out of program order;

updating a hash data structure indexed with a hash value based on an address of at least one of the issued store instructions;

generating load response data by accessing the hash data structure using an index based on an address of a load instruction; and

forwarding the load response data to execution logic of the processor for completing the load instruction.

10. The method of claim 9, wherein the load response data of the load instruction is generated only after all store instructions earlier in program order of the instruction block have been issued.

11. The method of claim 10, further comprising:

receiving a store mask identifying the store instructions within the instruction block, and the store mask is used for determining when all store instructions earlier in program order of the instruction block have been issued.

12. The method of claim 11, further comprising:

committing the instruction block only after all store instructions identified by the store mask have issued.

13. The method of claim 11, wherein the store mask is decoded from an instruction header of the instruction block.

14. The method of any one of claims 9-13, wherein generating the load response data comprises merging byte-enabled data from the issued store instructions.

15. A block-based processor comprising:

issue logic configured to issue load and store instructions of an instruction block out of program order, each of the load and store instructions including an address and an identifier specifying a relative program order of the instruction within the instruction

block; and

a load-store queue comprising:

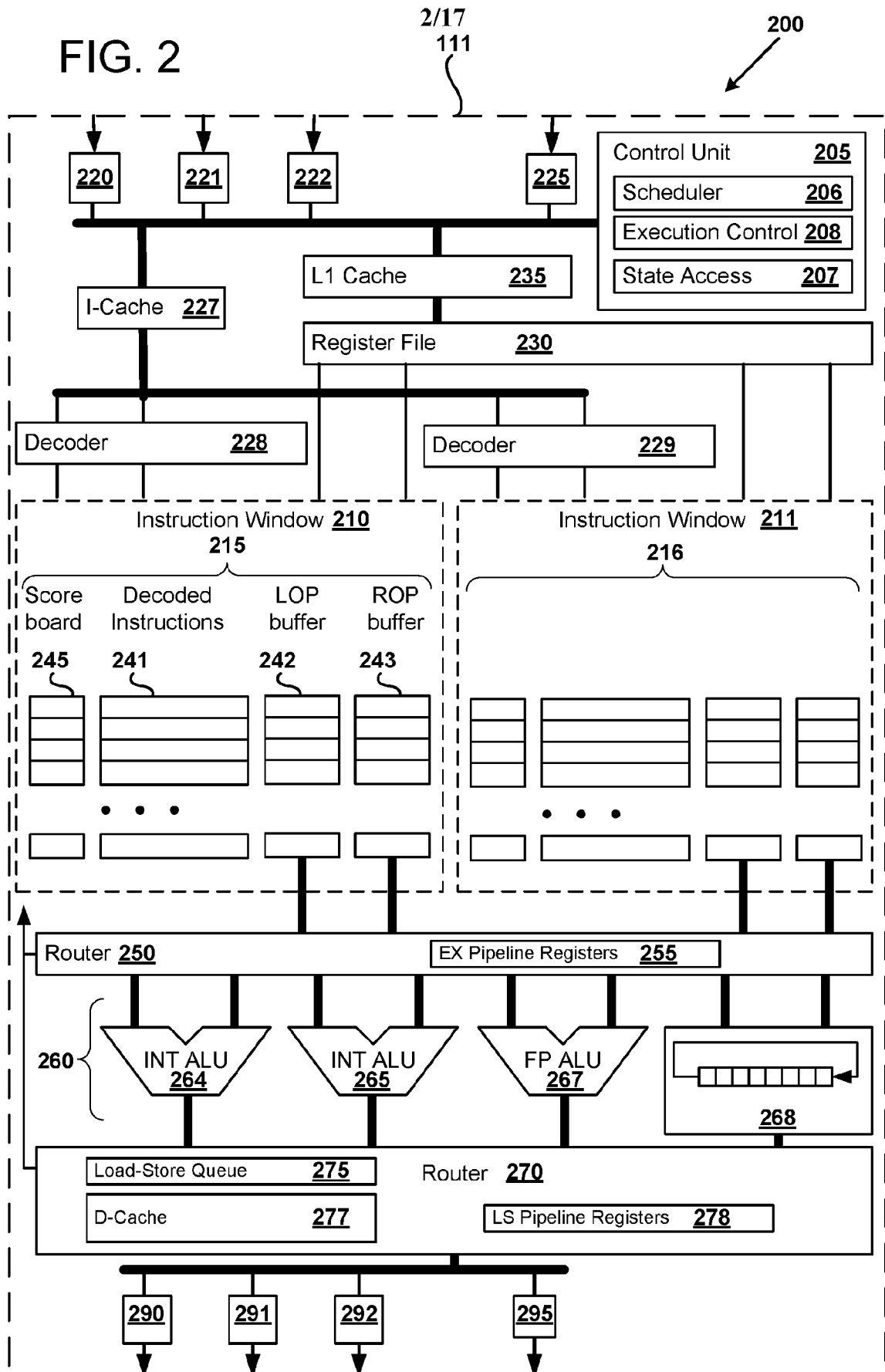
a computer-readable memory configured to store the issued load and store instructions in relative program order using the identifiers specifying the relative program order;

hash function logic configured to generate hash values for the addresses of the issued load and store instructions;

a hash data structure configured to store information associated with the respective issued store instructions, the information in the hash data structure indexed by the hash values of the addresses of the load and store instructions; and

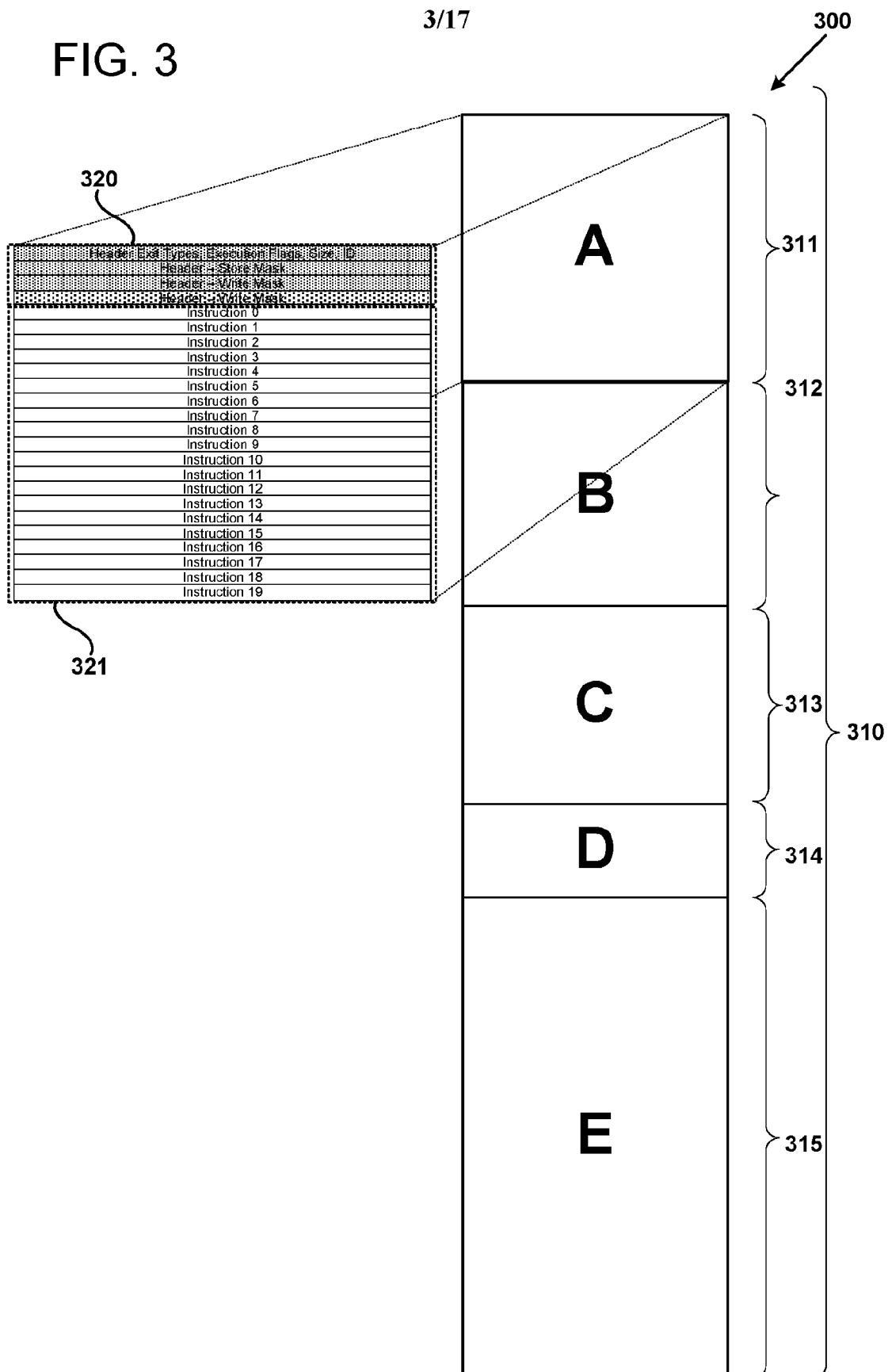
load response logic configured to generate load response data for the load instructions, the load response data for a respective load instruction generated after all issued store instructions earlier in the program order than the respective load instruction have been stored in the computer-readable memory, the load response data generated by searching the hash data structure.

FIG. 2



3/17

FIG. 3



4/17

FIG. 4

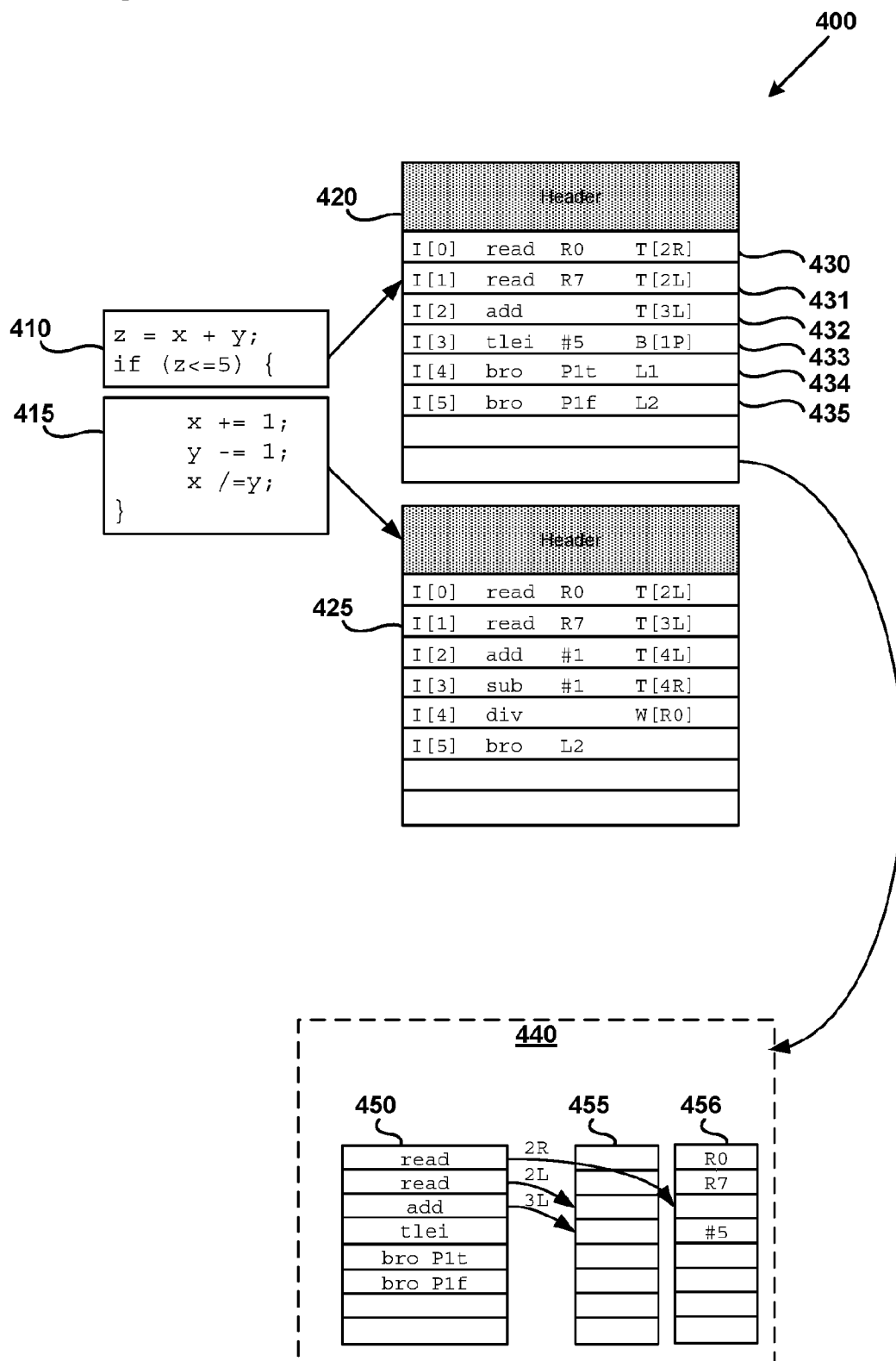


FIG. 5

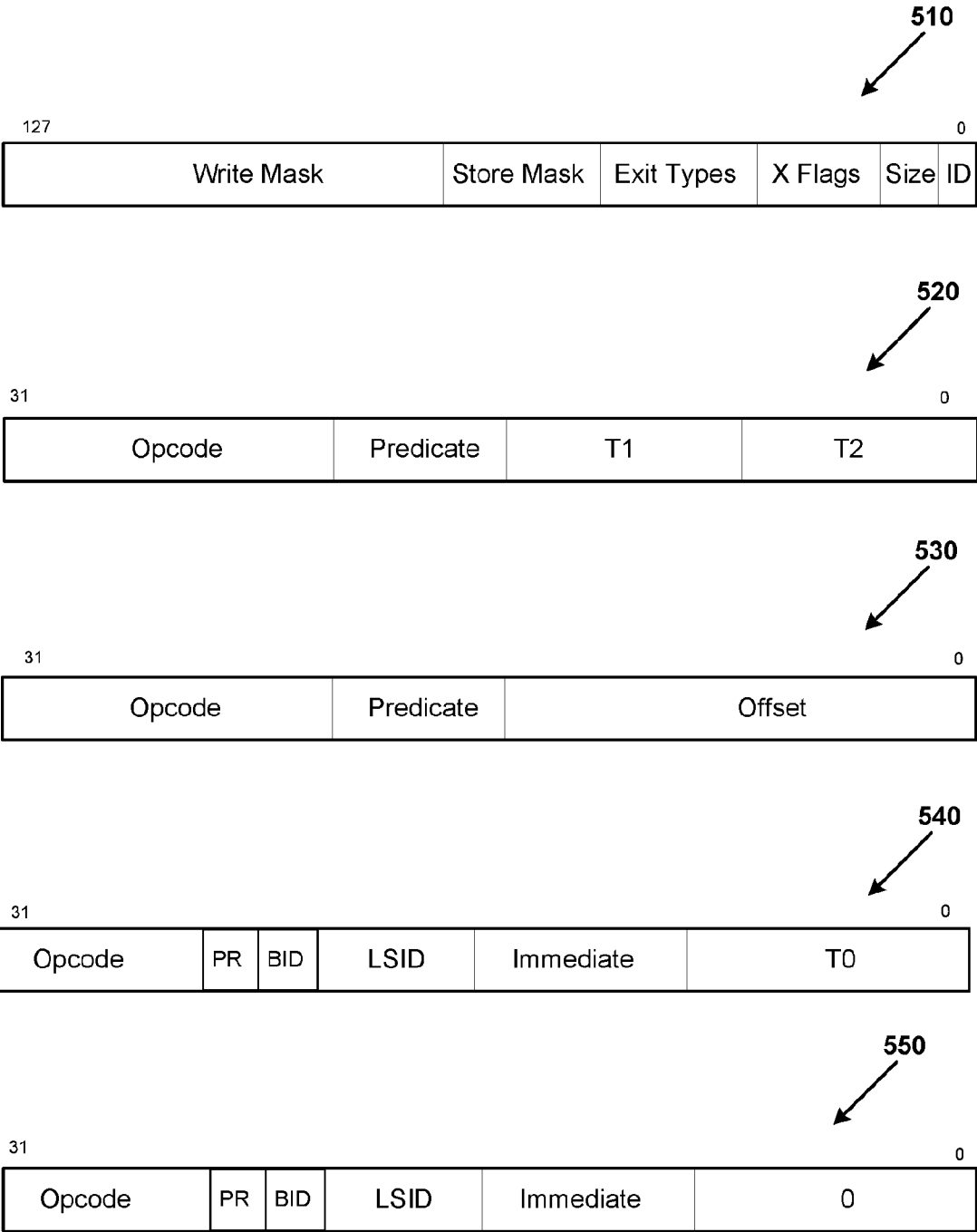
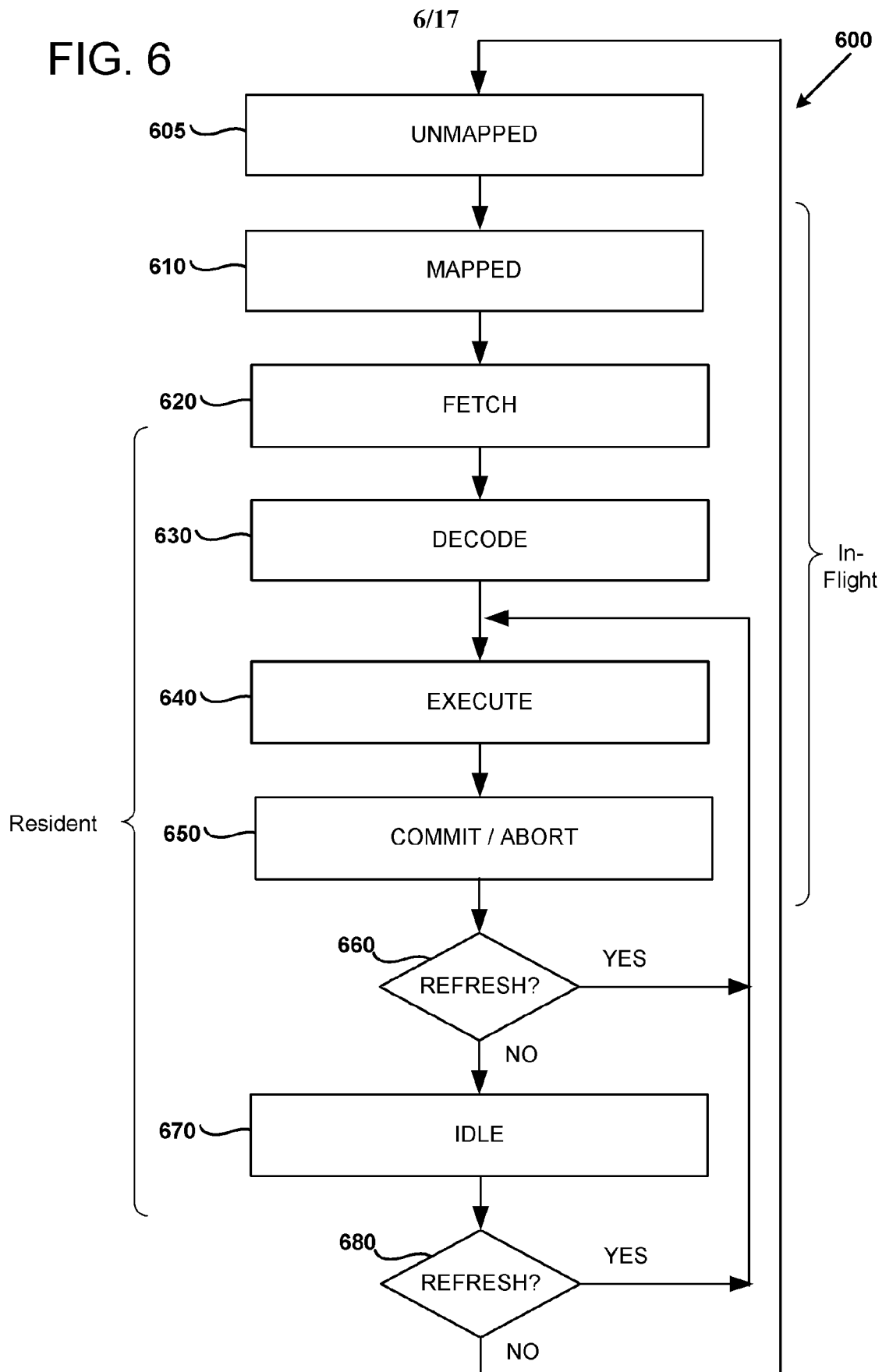


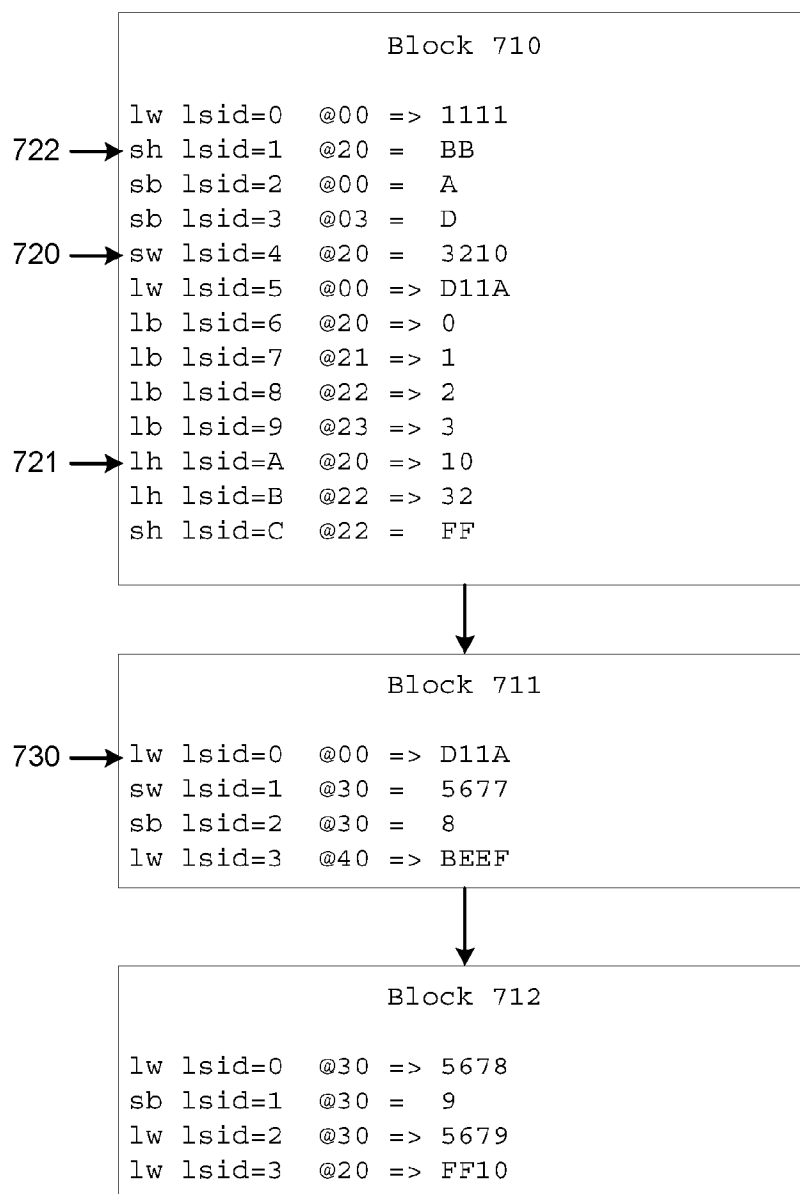
FIG. 6

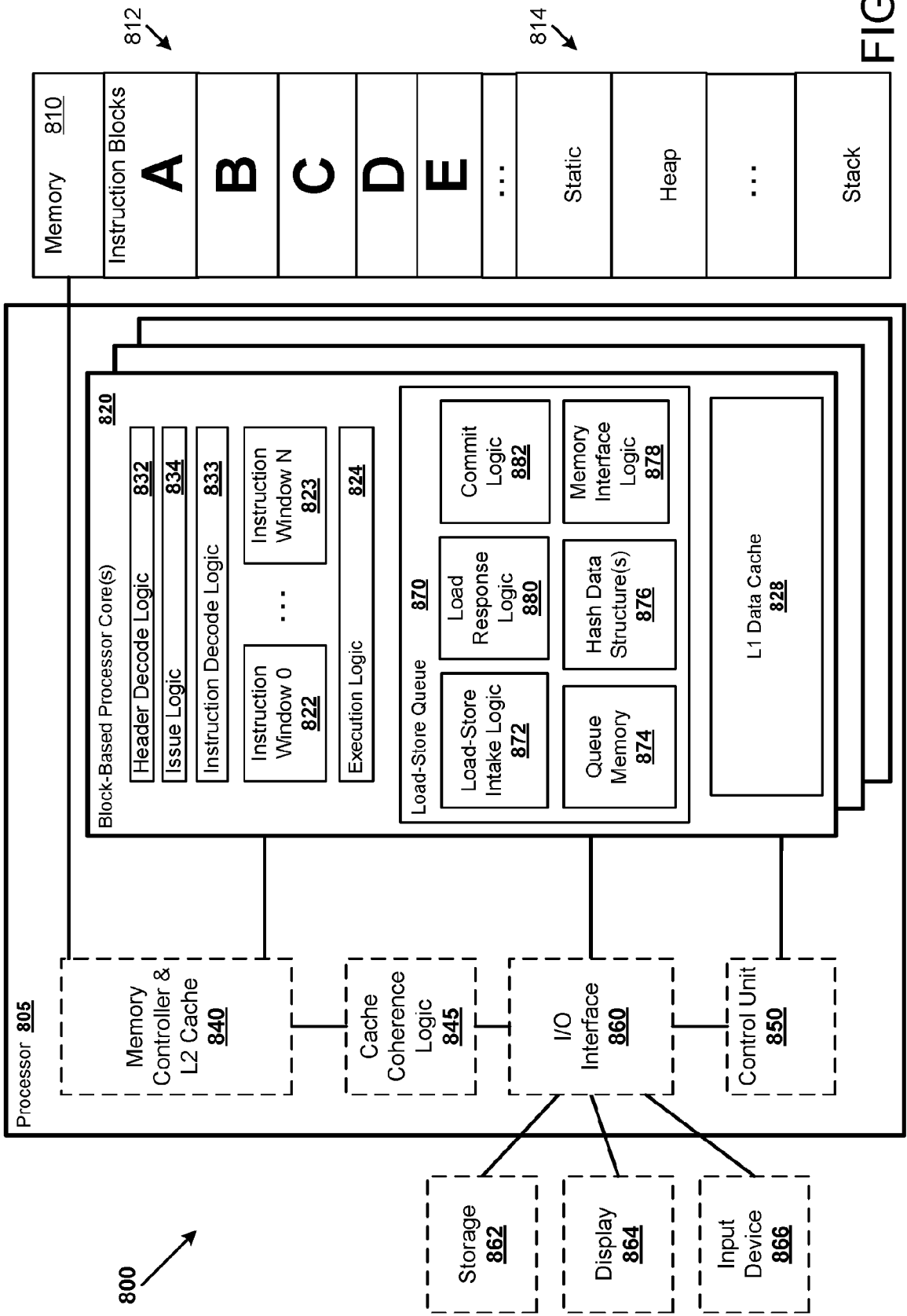


7/17

FIG. 7

700





9/17

FIG. 9

900

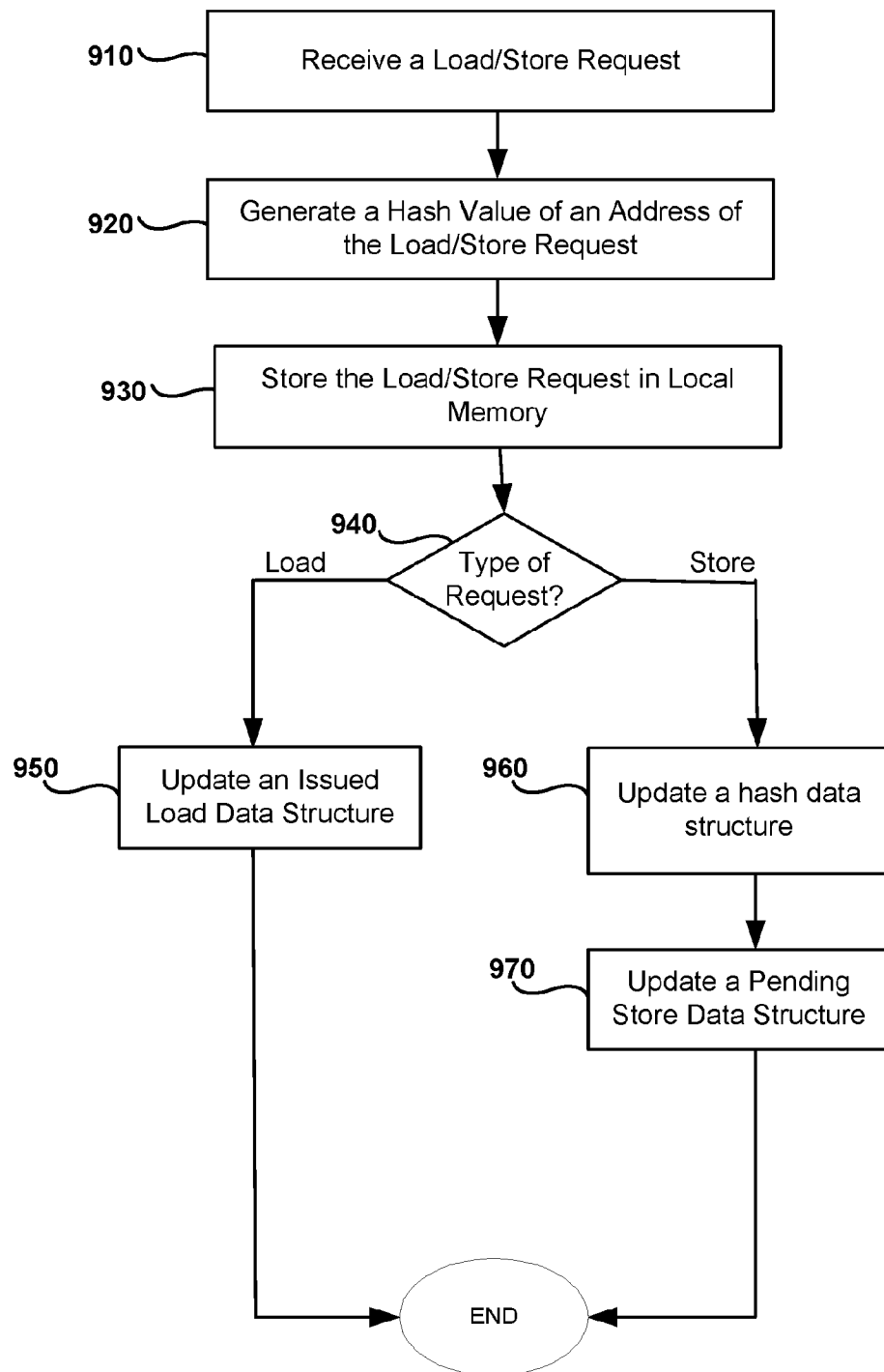


FIG. 10

10/17

1000

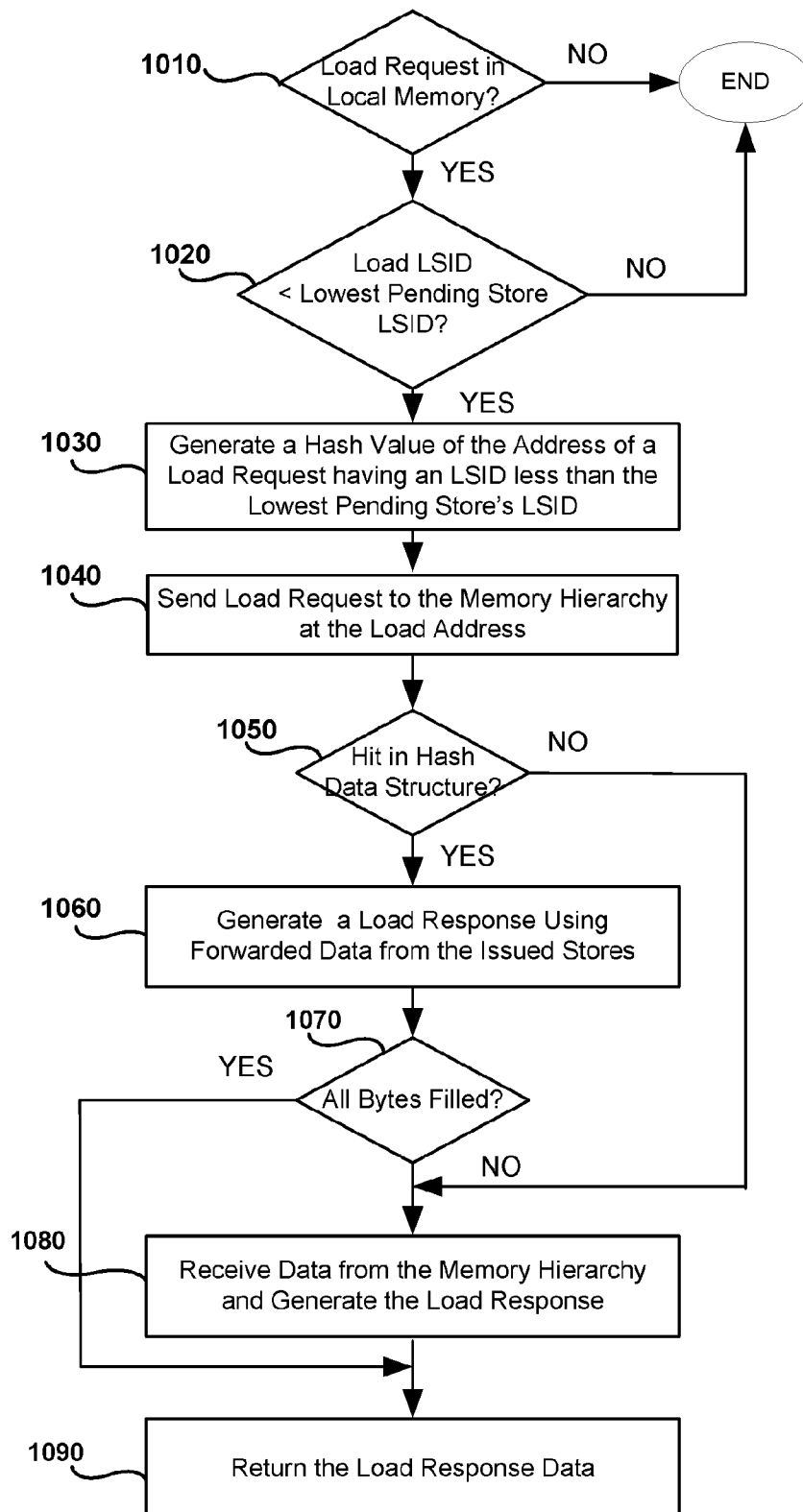
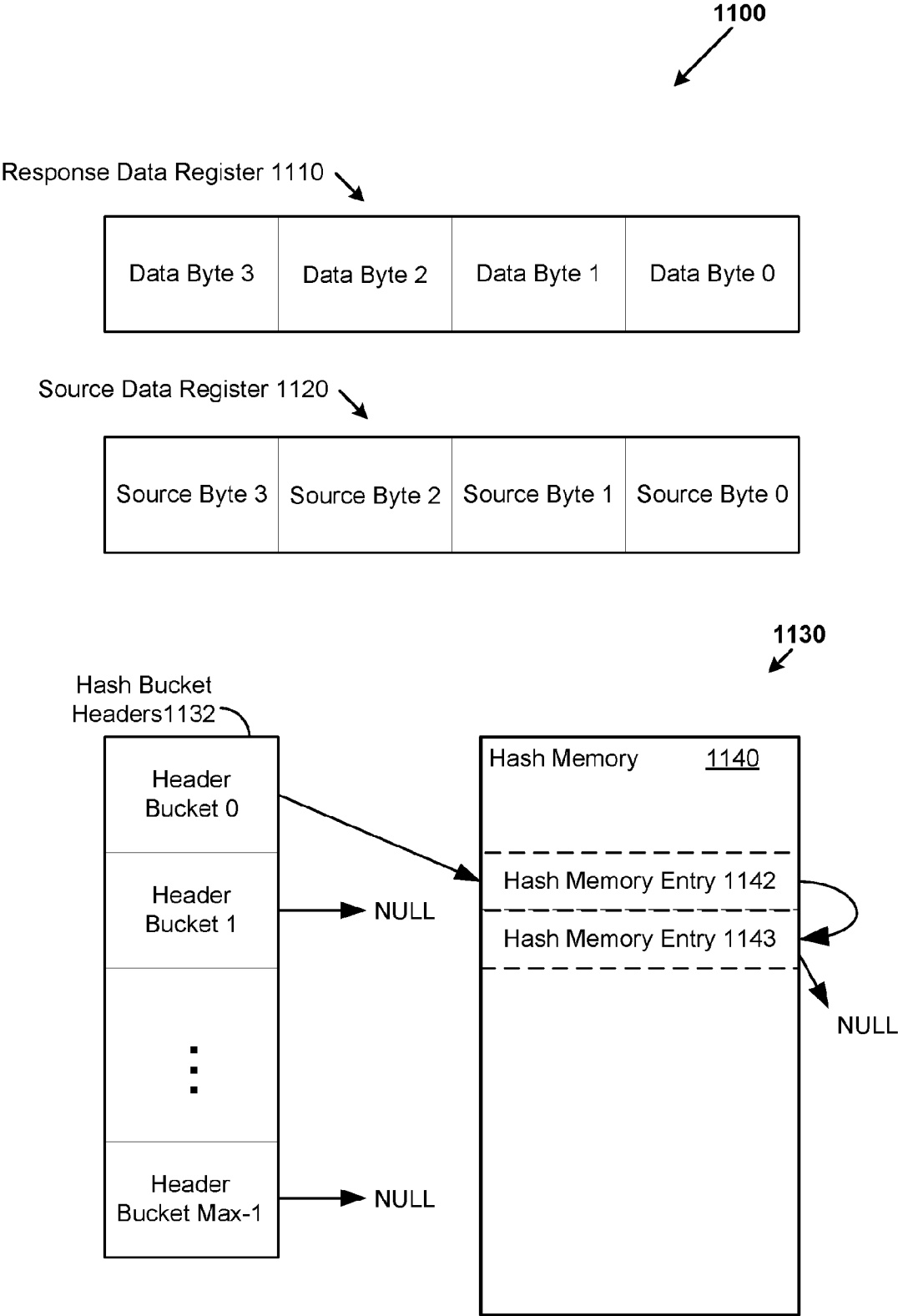


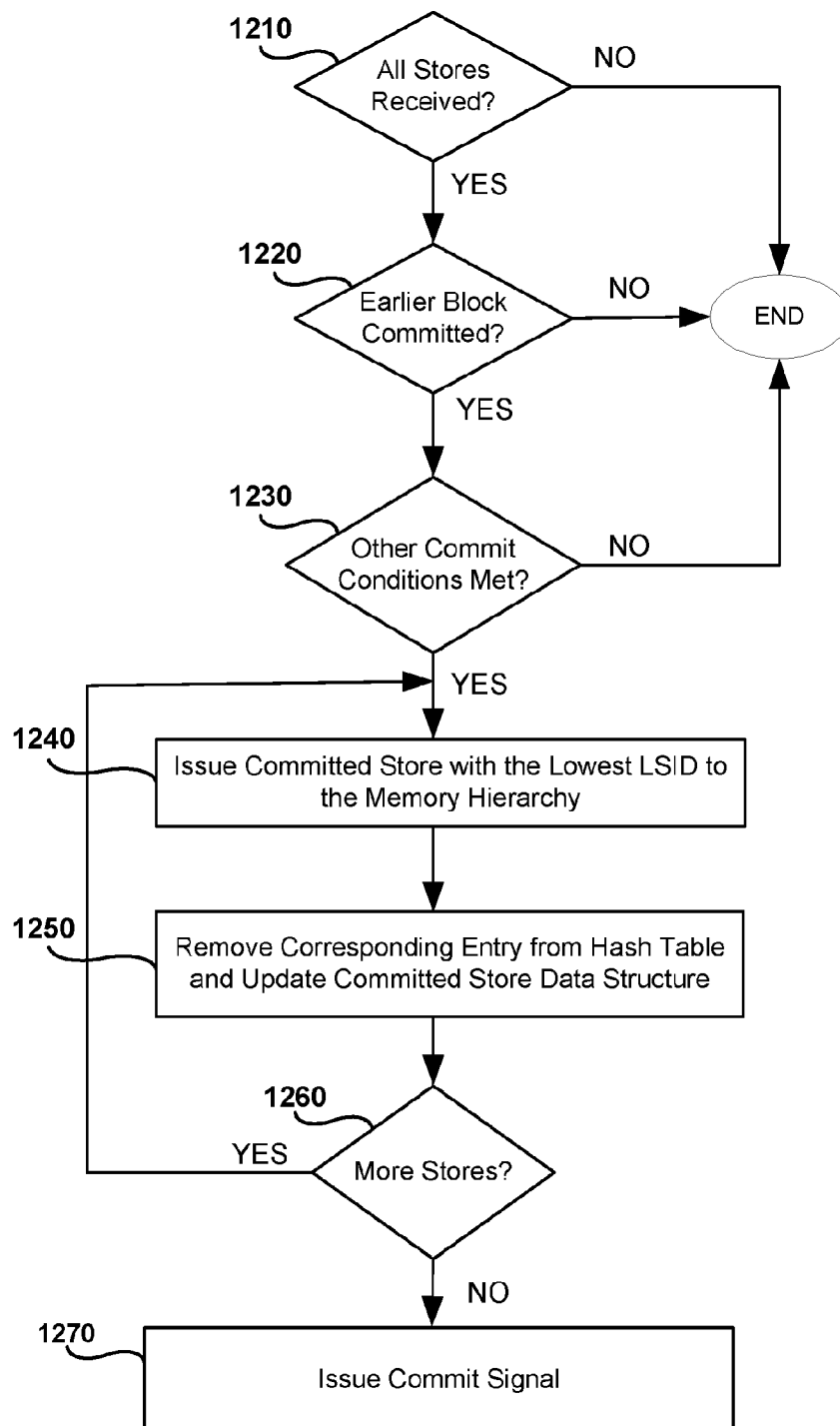
FIG. 11



12/17

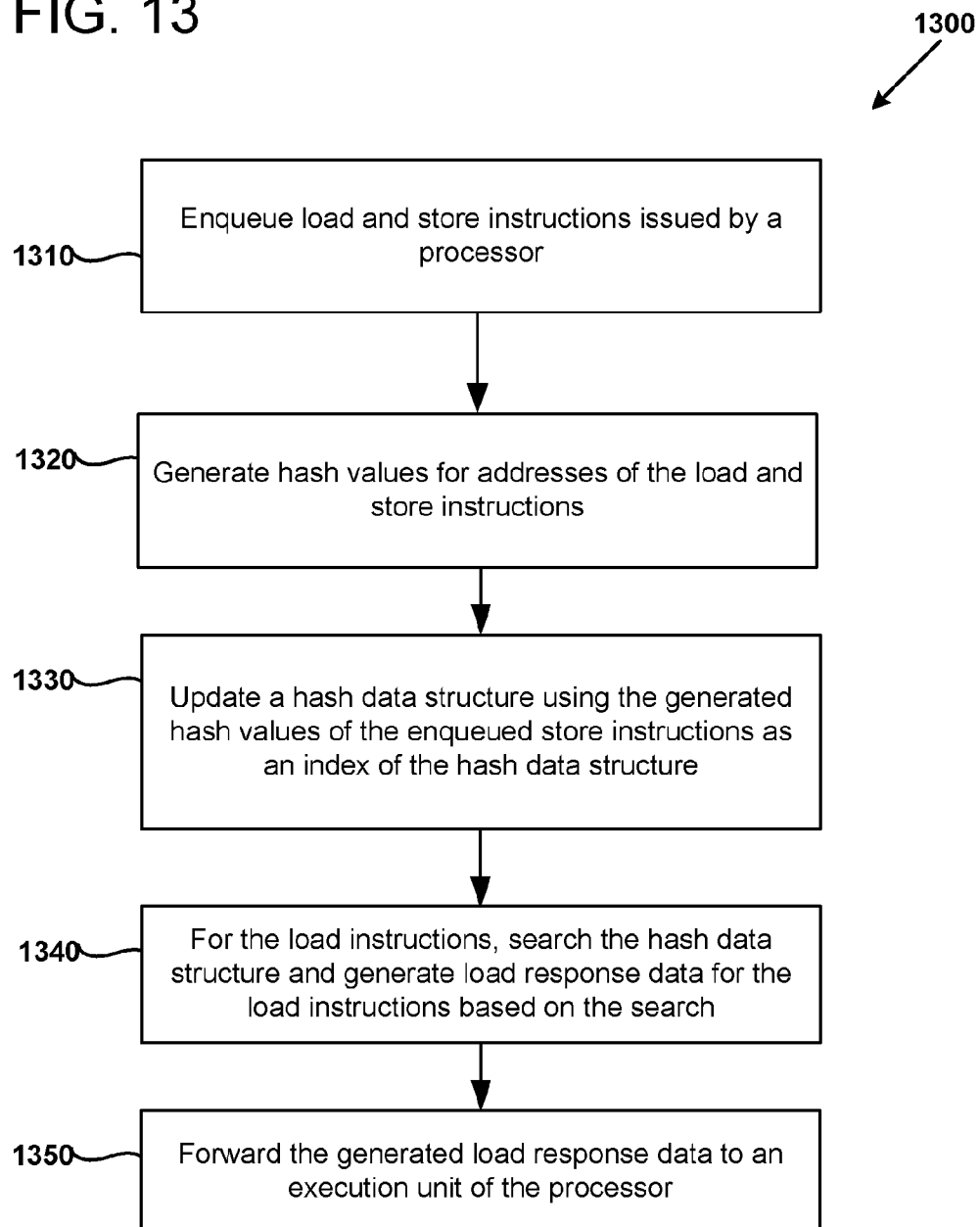
FIG. 12

1200



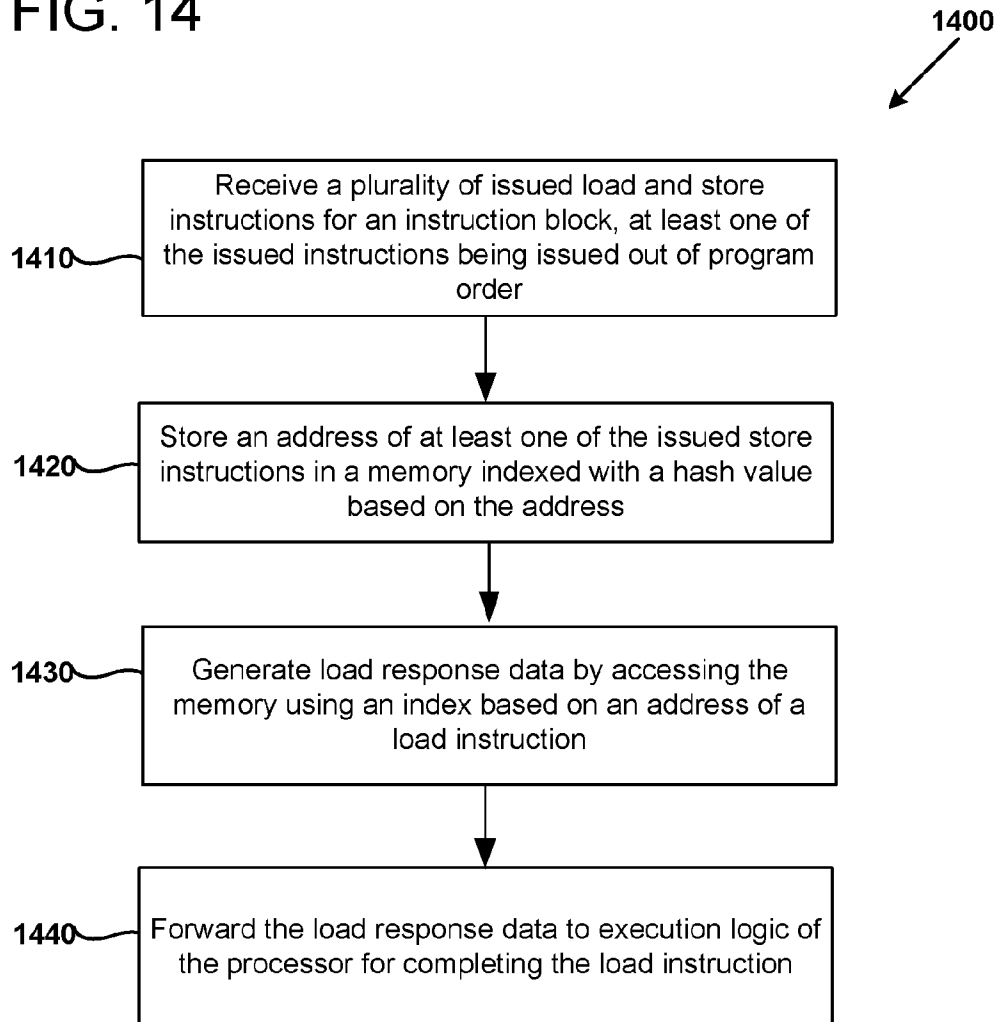
13/17

FIG. 13



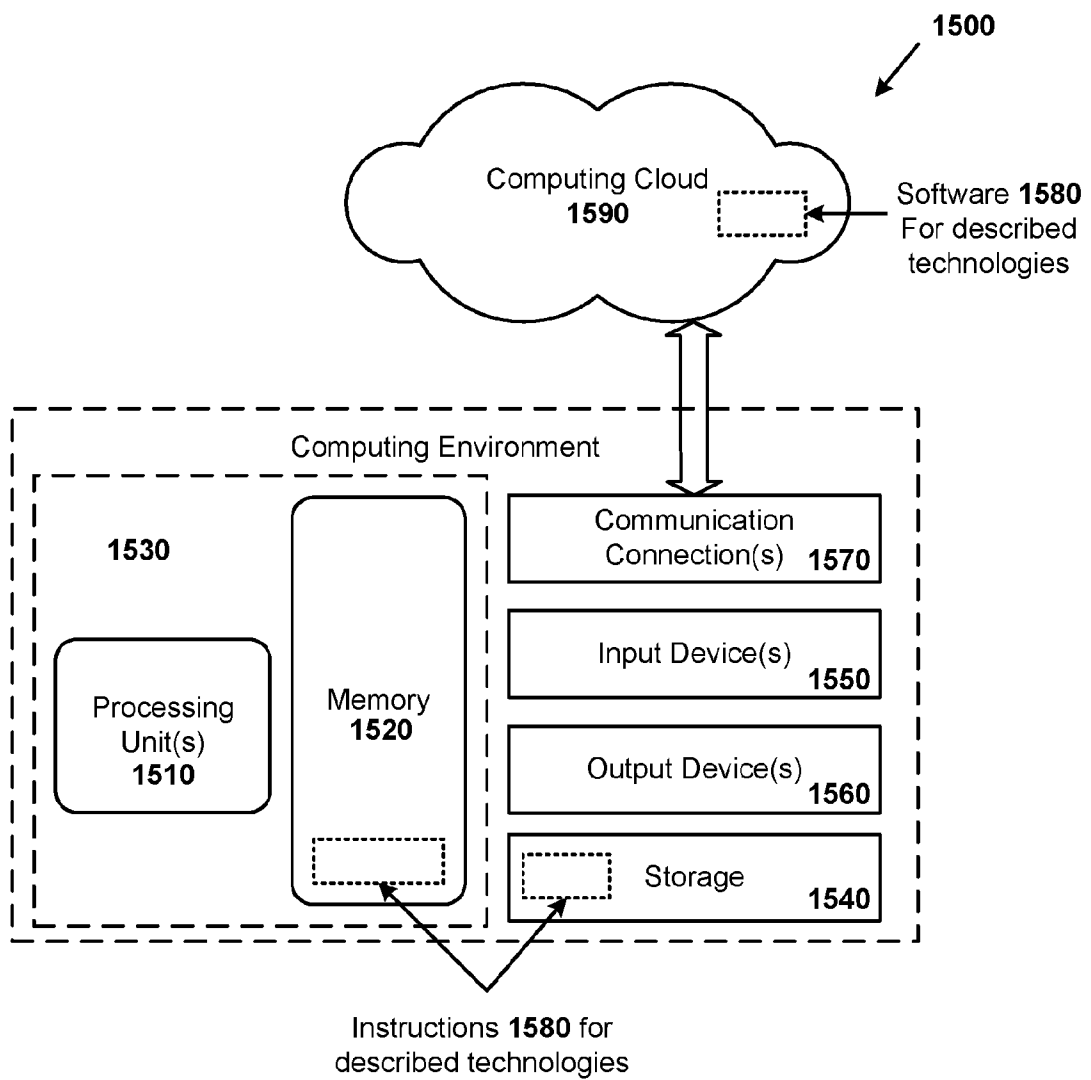
14/17

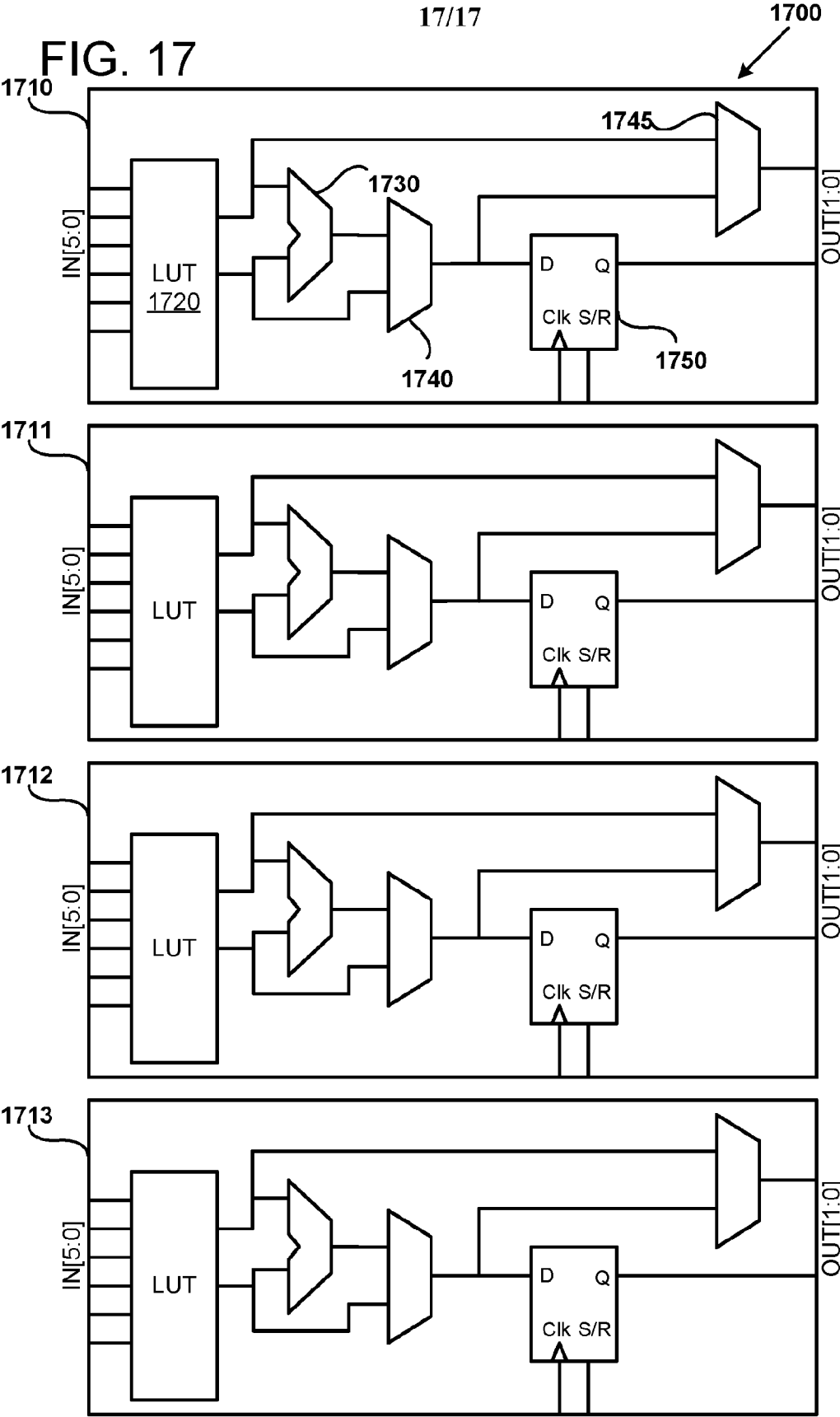
FIG. 14



15/17

FIG. 15





INTERNATIONAL SEARCH REPORT

International application No

PCT/US2017/038204

A. CLASSIFICATION OF SUBJECT MATTER

INV. G06F9/38

ADD.

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

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

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

EPO-Internal, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	KARTHIKEYAN SANKARALINGAM ET AL: "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor", 2014 47TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE; [PROCEEDINGS OF THE ANNUAL ACM/IEEE INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE], IEEE COMPUTER SOCIETY, 1730 MASSACHUSETTS AVE., NW WASHINGTON, DC 20036-1992 USA, 9 December 2006 (2006-12-09), pages 480-491, XP058106955, ISSN: 1072-4451, DOI: 10.1109/MICRO.2006.19 ISBN: 978-0-7695-3047-5 the whole document ----- -/-	1-15



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

20 September 2017

Date of mailing of the international search report

29/09/2017

Name and mailing address of the ISA/

European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040,
Fax: (+31-70) 340-3016

Authorized officer

Brandt, Jan

INTERNATIONAL SEARCH REPORT

International application No

PCT/US2017/038204

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	AARON SMITH ET AL: "Compiling for EDGE Architectures", FOURTH INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION (CGO '06). MARCH 26-29, 2006, IEEE, US, 26 March 2006 (2006-03-26), pages 185-195, XP058144521, DOI: 10.1109/CGO.2006.10 ISBN: 978-0-7695-2499-3 the whole document -----	1-15