



(51) International Patent Classification:

G06F 9/38 (2006.01) G11C 11/40 (2006.01)
G06F 13/14 (2006.01)

(21) International Application Number:

PCT/US2013/047429

(22) International Filing Date:

24 June 2013 (24.06.2013)

(25) Filing Language:

English

(26) Publication Language:

English

(30) Priority Data:

13/725,934 21 December 2012 (21.12.2012) US

(71) Applicant: INTEL CORPORATION [US/US]; 2000 Mission College Boulevard, M/S: RNB-4-150, Santa Clara, California 95054 (US).

(72) Inventors: CALLISTER, James; 1904 Lookout Drive, Windsor, Colorado 80550 (US). SOLTIS, Don; 8854 Longs Peak Circle, Windsor, Colorado 80550 (US). BHATTIA, Rohit; 2936 Antelope Road, Fort Collins, Colorado 80525 (US). SRINIVASAN, Ramkumar; 4701 Technology Parkway, Fort Collins, Colorado 80528 (US). BOSTIAN, Steven; 4900 Boardwalk Drive, Apt. E104, Fort Collins, Colorado 80525 (US). BLUMBERG, Richard M.; 2990 East 17th Avenue, #1301, Denver, Colorado 80206 (US).

(74) Agents: MALLIE, Michael J. et al.; Blakely Sokoloff Taylor & Zafman LLP, 1279 Oakmead Parkway, Sunnyvale, California 94085 (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, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, 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, 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, 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).

Published:

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

(54) Title: MECHANISM TO PROVIDE HIGH PERFORMANCE AND FAIRNESS IN A MULTI-THREADING COMPUTER SYSTEM

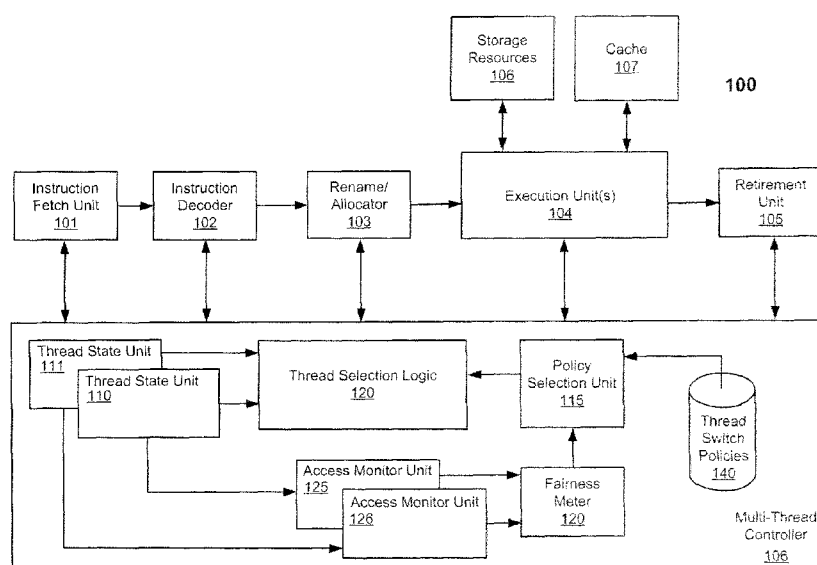


FIG. 1

(57) Abstract: According to one embodiment, a processor includes an execution pipeline for executing a plurality of threads, including a first thread and a second thread. The processor further includes a multi-thread controller (MTC) coupled to the execution pipeline to determine whether to switch threads between the first and second thread based on a thread switch policy that is selected from a list of thread switch policies based on unfairness levels of the first and second thread, and in response to determining to switch threads, to switch from executing the first thread to executing the second thread.

MECHANISM TO PROVIDE HIGH PERFORMANCE AND FAIRNESS IN A MULTI-THREADING COMPUTER SYSTEM

TECHNICAL FIELD

Embodiments of the present invention relate generally to processor architecture, and,
5 more specifically, to techniques for providing high performance and fairness in a multi-threading computer system.

BACKGROUND ART

Many modern computer systems are able to execute more than one distinct software program, or "thread", without having to explicitly save the state for one thread and restore the
10 state for another thread. For this reason they are referred to as "multi-threaded" computer systems. In one conventional approach, called sequential multi-threaded operation, the operating system or other control mechanism permits the several threads to share resources by permitting each thread that is a candidate for execution to operate in sequence on the processor. Changing between threads may be referred to as thread switching. In some of these conventional
15 approaches, threads are switched when the currently executing thread, i.e., the foreground thread, executes for a certain period or reaches a point when it cannot proceed, such as waiting for a memory access or an input/output (I/O) transfer to finish, or simply to ensure fairness amongst the tasks. The selection of the next thread to be switched in (permitted use of execution resources) may be made on the basis of strict priority. In other approaches, a round-robin
20 approach may be used in thread switching.

Multi-threading (MT) increases total system throughput by allowing two (or more) software processes to use shared system resources concurrently. System throughput is increased when the shared resources are not fully utilized by any single process and can be profitably used concurrently by another process. Maximizing system throughput is equivalent to maximizing
25 utilization of shared resources.

In temporal multi-threading, only one thread can utilize the main execution pipeline at any given time, so the system must explicitly switch the pipeline to the other thread in order to execute instructions from that thread. Each thread is assigned to a distinct hardware thread, each maintaining a separate architectural state. Thread switching policies should be guided by the goal
30 of maximizing system throughput (or equivalently, maximizing utilization). This includes policies such as switching threads when the current foreground thread cannot make as much progress as the background thread could, or minimizing the time a critical resource is used by

any thread. Note that such policies should take into consideration the overhead of switching threads (when neither the incoming nor outgoing process can make progress).

While increased total utilization of system resources is the primary motivation for multi-threading, to completely ignore the notion of fairness among the hardware threads on a system can lead to customer-visible performance issues, including denial of service and system crashes.

For example, if thread “A” is totally compute bound, and the multi-threading policy is only concerned with maximizing total utilization, there would be no reason to ever switch to thread “B”. A viable thread switch policy needs to back off from maximizing total utilization enough to guarantee forward progress for all threads and satisfy desired quality of service (QoS) metrics for all threads.

When two threads share a resource, what does it mean to give each thread fair access? Some notions of fairness are built on a foundation of giving all requestors (threads) the same amount access to a shared resource. An MT fairness policy corresponding to this philosophy might divide the resource Solomon-style and give each of two threads exactly half the resource. In the case of a main pipeline, this means giving each thread exclusive use of the pipeline for half the time.

While this approach may work well for homogeneous workloads, in general it is wasteful. Suppose thread “A” needs the main pipeline for 75% of the time, but thread “B” only needs the main pipeline for 20% of the time, giving a total resource demand of 95%. If each thread is given exactly 50% of the pipeline then the total utilization of the pipeline is only 70%: Thread “A” uses its entire allotment of 50% and Thread “B” only consumes the 20% it needs.

This “equality for all” notion of fairness may lead to an MT policy of equalizing the slowdown experienced by each thread due to multi-threading. This policy has similar problems when the resource demands of each thread are different. If resource demands for each thread vary during program execution, equalizing slowdown can lead to serious inefficiencies and hurt overall utilization, and thus limiting the performance gain that can be derived from multi-threading.

Continuing with the example above, notice that 30% of the time, the pipeline is devoted to thread “B” when “B” cannot make use of it. Ignoring secondary effects, it does not affect the performance of thread “B” to let thread “A” make use of that 30%, in addition to the 50% of the pipeline it already owns. In this case, pipeline utilization is maximized and neither thread experiences a slowdown of more than 2X compared to single-threaded performance (in this example, neither thread experiences a slowdown). Denying thread “A” the main pipeline when

thread “B” cannot use it does not help thread “B” execute any faster, it only slows down thread “A” and decreases overall throughput.

BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments of the invention are illustrated by way of example and not limitation in the figures of the accompanying drawings in which like references indicate similar elements.

Figure 1 is a block diagram of an execution pipeline of a processor or processor core according to one embodiment of the invention.

Figure 2 is a block diagram of an embodiment of multi-thread controller.

Figure 3 is a flow diagram illustrating a method for thread switching.

Figure 4 is a flow diagram illustrating a method for updating a fairness counter.

Figure 5 is a flow diagram illustrating a method for selecting a thread for execution based on a thread switch policy.

Figure 6A is a block diagram of a fairness meter.

Figure 6B is a block diagram of a policy selection unit.

Figure 7 is a block diagram of a thread state unit.

Figure 8 is a flow diagram illustrating a method for selecting a thread for execution based on thread priority level.

Figure 9A illustrates an exemplary advanced vector extensions (AVX) instruction format according to one embodiment of the invention.

Figure 9B illustrates an exemplary advanced vector extensions (AVX) instruction format according to another embodiment of the invention.

Figure 9C illustrates an exemplary advanced vector extensions (AVX) instruction format according to another embodiment of the invention.

Figure 10A is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention.

Figure 10B is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention.

Figure 11A is a block diagram illustrating an exemplary specific vector friendly instruction format according to one embodiment of the invention.

Figure 11B is a block diagram illustrating a generic vector friendly instruction format according to another embodiment of the invention.

Figure 11C is a block diagram illustrating a generic vector friendly instruction format according to another embodiment of the invention.

Figure 11D is a block diagram illustrating a generic vector friendly instruction format according to another embodiment of the invention.

Figure 12 is a block diagram of register architecture according to one embodiment of the invention.

5 Figure 13A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention.

10 Figure 13B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution architecture core to be included in a processor according to embodiments of the invention.

Figure 14A is a block diagram of a processor according to one embodiment of the invention.

Figure 14B is a block diagram of a processor according to another embodiment of the invention.

15 Figure 15 is a block diagram of a processor according to embodiments of the invention.

Figure 16 is a block diagram of a system in accordance with one embodiment of the invention.

Figure 17 is a block diagram of a more specific exemplary system in accordance with an embodiment of the invention.

20 Figure 18 is a block diagram of a more specific exemplary system in accordance with another embodiment of the invention.

Figure 19 is a block diagram of a SoC in accordance with an embodiment of the invention.

25 Figure 20 is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention.

DETAILED DESCRIPTION

Various embodiments and aspects of the inventions will be described with reference to details discussed below, and the accompanying drawings will illustrate the various embodiments.

30 The following description and drawings are illustrative of the invention and are not to be construed as limiting the invention. Numerous specific details are described to provide a thorough understanding of various embodiments of the present invention. However, in certain instances, well-known or conventional details are not described in order to provide a concise discussion of embodiments of the present inventions.

Reference in the specification to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in conjunction with the embodiment can be included in at least one embodiment of the invention. The appearances of the phrase “in one embodiment” in various places in the specification do not necessarily all refer to the same
5 embodiment.

According to some embodiments of the invention, an architecture and set of mechanisms is provided to enhance performance and fairness in a multi-threading system. When a first thread is initiated by the software, it is executed by the system, which uses one or more shared resources of the system, e.g., the main execution pipeline of a processor core or a memory
10 pipeline, which shall herein be generally referred to as pipeline. During the execution of the first thread, software (SW) initiates a second thread. For the remainder of this description, a thread which is currently being executed by the pipeline shall be referred to as the “foreground” thread, and the thread which is waiting to be executed shall be referred to as the “background” thread. Thus, under this nomenclature, a thread may switch from foreground to background, or vice
15 versa, as the system performs thread switching.

According to one embodiment of the invention, the system makes thread switching decisions according to several factors. In one embodiment, the system performs thread switching according to a currently selected thread switch policy. In one embodiment, the thread switch policy is selected from a set of policies, which can be understood as a sliding scale. In the middle
20 region of the scale, where fairness among threads is achieved, the policy is geared towards maximum system utilization. On each end of the scale, the policy is geared towards providing maximum fairness to a victim thread, i.e., a thread that has been unfairly denied of access to the pipeline.

In one embodiment, the system also makes thread switching decisions according to the
25 information of each thread. According to one aspect of the invention, this information includes the software-assigned priority level of each thread. In one embodiment, the system also takes into consideration the execution status of the thread, .e.g., whether it is able to fully utilize the pipeline during a given cycle. In one embodiment of the invention, the system makes thread switching decisions according to triggering events such as timer expiration, external interrupts,
30 etc.

It will be appreciated that the factors described above (thread switch policy, thread information, and external events) may be considered by the system individually, collectively, or any combination thereof, in determining whether thread switching should be performed. It will be further appreciated that factors discussed above are only intended for illustrative purpose, and

the system is not limited the above factors in determining whether to switch threads.

Furthermore, the thread switch techniques described throughout this application will be described between two threads. However, this is not so limited; the thread switching techniques can also be applied to switching amongst more than two threads.

5 Figure 1 is a block diagram of a processor or processor core according to one embodiment of the invention. Processor 100 may be an SMT or a Switch on Event Multi-Threading (SoEMT) capable processor available from Intel Corporation of Santa Clara, California. Referring to Figure 1, processor 100 may represent any kind of instruction processing apparatuses or processing elements. A processing element refers to a thread, a process, a
10 context, a logical processor, a hardware thread, a core, and/or any processing element, which shares access to other shared resources of the processor, such as reservation units, execution units, pipelines, and higher level caches/memory. A physical processor typically refers to an integrated circuit, which potentially includes any number of other processing elements, such as cores or hardware threads. A core often refers to logic located on an integrated circuit capable of
15 maintaining an independent architectural state, where each independently maintained architectural state is associated with at least some dedicated execution resources. In one embodiment, processor 100 may be a general-purpose processor. Processor 100 may be any of various complex instruction set computing (CISC) processors, various reduced instruction set computing (RISC) processors, various very long instruction word (VLIW) processors, various
20 hybrids thereof, or other types of processors entirely. Processor 100 may also represent one or more processor cores.

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general
25 purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a central processing unit (CPU) including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor
30 including one or more special purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a coprocessor is sometimes referred to as special purpose logic, such as

integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality. Exemplary core architectures are described next, followed by

5 descriptions of exemplary processors and computer architectures.

In one embodiment, processor 100 includes, but is not limited to, instruction fetch unit 101, instruction decoder 102, rename/allocator 103, one or more execution units 104, and retirement unit 105, forming a processor pipeline. A pipeline or portion of a pipeline, such as a front-end or instruction decode portion 102 of the pipeline, can be shared by multiple threads.

10 Architecture state registers (not shown) are replicated, so individual architecture states/contexts are capable of being stored for different logical processors. Other smaller resources, such as instruction pointers and renaming logic in rename allocator logic 103 may also be replicated for the threads. Some resources, such as re-order buffers in a reorder/retirement unit 105, load/store buffers, and queues may be shared through partitioning. While resources, such as general
15 purpose internal registers (e.g., registers 106), page-table base registers, a low-level data-cache (e.g., cache 107) and data translation buffer (TLB), execution unit(s) 104, and an out-of-order unit (not shown) may be potentially fully shared.

In one embodiment, instruction decoder 102 is to decode the instructions received from instruction fetch unit 101. The instructions may be macroinstructions fetched from cache
20 memory 107 that is integral within processor 100 or closely associated therewith, or may be retrieved from an external memory via a system bus. Instruction decoder 102 may decode the macroinstructions and generate or output one or more micro-operations, micro-code, entry points, microinstructions, other instructions, or other control signals, which reflect, or are derived from, the instructions. Instruction decoder 102 may be implemented using various different
25 mechanisms. Examples of suitable mechanisms include, but are not limited to, microcode read only memories (ROMs), look-up tables, hardware implementations, programmable logic arrays (PLAs), and the like.

In one embodiment, allocator and rename unit 103 includes an allocator to reserve resources, such as register files to store instruction processing results. However, a thread is
30 potentially capable of an out-of-order execution, where allocator and rename unit 103 also reserves other resources, such as reorder buffers to track instruction results. It may also include a register renamer to rename program/instruction reference registers to other registers internal to the processor. During such a renaming stage, references to external or logical registers are

converted into internal or physical register references to eliminate dependencies caused by register reuse.

Execution units 104, which may include an arithmetic logic unit, or another type of logic unit capable of performing operations based on instructions. As a result of instruction decoder
5 102 decoding the instructions, execution unit 104 may receive one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which reflect, or are derived from, the instructions. Execution unit 104 may be operable as a result of instructions indicating one or more source operands (SRC) and to store a result in one or more destination operands (DEST) of a register set indicated by the instructions. Execution unit 104
10 may include circuitry or other execution logic (e.g., software combined with hardware and/or firmware) operable to execute instructions or other control signals derived from the instructions and perform an operation accordingly. Execution unit 104 may represent any kinds of execution units such as logic units, arithmetic logic units (ALUs), arithmetic units, integer units, etc.

Processor 100 further includes a scheduler and dispatch unit (not shown) to schedule and
15 dispatch instructions to execution units 104 for execution. In fact, instructions/operations are potentially scheduled on execution units 104 according to their type availability. For example, a floating point instruction is scheduled on a port of an execution unit that has an available floating point execution unit. Examples of execution units include a floating point execution unit, an integer execution unit, a jump execution unit, a load execution unit, a store execution unit, and
20 other known execution units. In one embodiment, reorder/retirement unit 105 includes components, such as the reorder buffers mentioned above, load buffers, and store buffers, to support out-of-order execution and later in-order retirement of instructions executed out-of-order.

Some or all of the source and destination operands may be stored in storage resources
25 106 such as registers of a register set or memory. A register set may be part of a register file, along with potentially other registers, such as status registers, flag registers, etc. A register may be a storage location or device that may be used to store data. The register set may often be physically located on die with the execution unit(s). The registers may be visible from the outside of the processor or from a programmer's perspective. For example, instructions may
30 specify operands stored in the registers. Various different types of registers are suitable, as long as they are capable of storing and providing data as described herein. The registers may or may not be renamed. Examples of suitable registers include, but are not limited to, dedicated physical registers, dynamically allocated physical registers using register renaming, combinations of dedicated and dynamically allocated physical registers, etc. Alternatively, one or more of the

source and destination operands may be stored in a storage location other than a register, such as, for example, a location in system memory.

In one embodiment, cache 107 includes a variety of cache such as a high level and/or low level cache. Higher-level or further-out cache is to cache recently fetched and/or operated on elements. Note that higher-level or further-out refers to cache levels increasing or getting further way from the execution unit(s). In one embodiment, the higher-level cache is a second-level data cache. However, the higher level cache is not so limited, as it may be or include an instruction cache, which may also be referred to as a trace cache. A trace cache may instead be coupled after a decoder to store recently decoded instructions. It also potentially includes a branch target buffer to predict branches to be executed or taken, and an instruction-translation buffer (I-TLB) to store address translation entries for instructions.

Lower level data cache and data translation buffer (D-TLB) may be coupled to an execution unit(s). The data cache is to store recently used/operated on elements, such as data operands, which are potentially held in memory coherency states, such as modified, exclusive, shared, and invalid (MESI) states. The D-TLB is to store recent virtual/linear to physical address translations. Previously, a D-TLB entry includes a virtual address, a physical address, and other information, such as an offset, to provide inexpensive translations for recently used virtual memory addresses.

Processor 100 further includes a bus interface unit (not shown). A bus interface unit is to communicate with devices external to a processor, such as system memory, a chipset, a northbridge, or other integrated circuit. The memory may be dedicated to the processor or shared with other devices in a system. Examples of the memory includes dynamic random access memory (DRAM), static RAM (SRAM), non-volatile memory (NV memory), and long-term storage. Typically the bus interface unit includes input/output (I/O) buffers to transmit and receive bus signals on an interconnect. Examples of the interconnect include a Gunning Transceiver Logic (GTL) bus, a GTL+ bus, a double data rate (DDR) bus, a pumped bus, a differential bus, a cache coherent bus, a point-to-point bus, a multi-drop bus or other known interconnect implementing any known bus protocol. The bus interface unit may also communicate with a higher level cache.

In one embodiment, the various stages described above can be organized into three phases. The first phase can be referred to as an in-order front end including the fetch stage 101, decode stage 102, allocate rename stage 103. During the in-order front end phase, the instructions proceed through the pipeline 100 in their original program order. The second phase can be referred to as the out-of-order execution phase including the schedule/dispatch stage (not

shown) and the execute stage 104. During this phase, each instruction may be scheduled, dispatched and executed as soon as its data dependencies are resolved and the execution unit is available, regardless of its sequential position in the original program. The third phase, referred to as the in-order retirement phase which includes the retire stage 105 in which instructions are
5 retired in their original, sequential program order to preserve the integrity and semantics of the program, and to provide a precise interrupt model.

In one embodiment, processor 100 further includes multi-thread controller (MTC) 106 to determine whether to switch threads based on fairness and forward progress information. In one embodiment, MTC 106 switches the execution pipeline from executing the foreground thread to
10 executing the background thread. According to one embodiment, MTC 106 may switch the pipeline at various thread selection points. For example, the pipeline can be switched at instruction fetch unit 101, between instruction fetch unit 101 and rename/allocator 103, between rename/allocator 103 and execution unit 104, between execution unit 104 and retirement unit 105, etc.

According to one embodiment, MTC 106 includes thread selection logic (TSL) 120
15 which decides whether or not to switch threads based on information provided by thread state units (TSUs) 110-111. According to one embodiment, when a thread is initiated, SW (e.g., the OS scheduler) associates/assigns it to a hardware thread, i.e., a set of hardware resources, including for instance, a TSU. For instance, TSU 110 may be associated with a first thread and
20 TSU 111 may be associated with a second thread. In one embodiment, TSUs 110-111 provide information of each respective thread to TSL 120, such as the thread priority level. In one embodiment, TSU 110 and 111 may also provide execution status of each respective thread, e.g., its ability to use the pipeline in the current cycle and the cycles in the near future.

As illustrated in Figure 1, MTC 106 includes TSUs 110-111, thus, MTC 106 is capable
25 of supporting up to two threads. It will be appreciated, however, that TSUs 110-111 are provided only for illustrative purposes, and MTC 106 is not limited to include only two TSUs. MTC 106 may include more TSUs in order to support multi-threading of more threads.

In one embodiment, TSL 120 is configured to decide whether or not to switch threads based on a thread switch policy provided by policy selection unit (PSU) 115. According to one
30 embodiment, PSU 115 selects a default thread switch policy, from a set of thread switch policies 140, which is optimized for maximum utilization of hardware resources. However, as threads are executed over time, one thread may be unfairly denied access to the pipeline, and PSU 115 responds by selecting another thread switch policy which is more geared toward “fairness” to the threads rather than maximizing system utilization.

In one embodiment, each thread's fair access to the pipeline is monitored by a corresponding access monitor unit (AMU). For instance, AMU 125-126 may monitor how often a first and second thread, respectively, has been denied access to the pipeline. In one embodiment, AMUs 125-126 determine if a thread has been unfairly denied access to the pipeline according to the thread states provided by the corresponding TSU 110-111. For example, AMU 125 may indicate to FM 120 that the corresponding thread is unfairly denied access to the pipeline if corresponding TSU 110 indicates the thread is in a state where it is able to use the pipeline. On the other hand, if TSU 110 indicates the thread is not ready to use the pipeline, e.g., because it is blocked, waiting for data from another hardware resource, AMU 125 may not indicate to FM 120 that the thread is being unfairly denied access to the pipeline. In other words, according to one embodiment, a thread is considered to be unfairly denied access to the pipeline only if it is ready to use the pipeline.

In one embodiment, information of unfairness is provided to fairness meter (FM) 120 which utilizes the information to determine which, if any, thread is a "victim" thread, i.e., the thread which has been unfairly denied access to the pipeline. In one embodiment, in addition to providing PSU 115 with information of which is the victim thread, FM 120 also provides PSU 115 information on the level of unfairness. In one embodiment, the level of unfairness is used as a factor by PSU 115 in deciding a thread switch policy from the set of thread switch policies 140.

Figure 2 is a block diagram illustrating one embodiment of MTC 106 of Figure 1. Referring now to Figure 2, MTC 106 includes, but is not limited to, TSL 120 coupled to PSU 115, and TSUs 110 and 111. According to one embodiment, TSL 120 decides whether to switch threads by sampling/evaluating, on every clock or execution cycle or every predetermined numbers of clock or execution cycles, various information of each thread, either directly or indirectly. In one embodiment, TSL 120 may be implemented as a look-up table that returns a Boolean value on whether to switch threads according to the information provided by PSU 115, TSUs 110 and 111, and/or external events 230. TSUs 110 and 111 may provide various information of the threads, such as, for example, thread execution statuses, thread priorities, and/or timeout values provided by one or more timeout counters. The thread statuses of a thread may be set by a monitor based on the corresponding execution of the thread. The thread status of a thread may include, but it not limited to, an unstalled state, blocked state, and stalled state. A thread may be assigned to or associated with a priority in view other threads currently executed or pending within the processor. In one embodiment, a thread may be in one of a high priority, nominal priority, and low priority. However, it will be appreciated that these priorities are listed only for illustrative purposes, and a thread is not limited to three priorities. In another

embodiment, a thread may be associated with fewer than three priorities. Yet in another embodiment, a thread may be associated with more than three priorities. Such a priority may be set by software and/or hardware. In one embodiment, a software program associated with a thread may instruct the hardware to assign a particular priority to the thread, for example, via an instruction (e.g., a hint instruction). Based on the priority and/or the execution status of the thread, TSL 120 can make an intelligent decision on whether to switch from a first thread currently executed to a second thread pending to be executed. In addition, TSL 120 further selects one of the thread switching policies (not shown) which is determined by policy selection unit 115 that is most appropriate at the given point in time. The list of available thread switching policies may be preconfigured based a variety of factors, such as, for example, thread fairness values amongst the threads provided by fairness meter 120. The fairness of a thread is determined based on whether the corresponding thread receives a fair share of usage of the execution resources compared to other threads. The fairness of a thread may be determined based on a number of cycles that the thread has been denied or granted for requesting execution resources, which may be monitored by AMUs 125-126. In one embodiment, there is one monitoring unit for each of the threads and alternatively, there may be a single or shared monitoring unit for multiple threads. The fairness of a thread may further be determined based on the execution states, priorities, and/or timeout values provided by thread state units 110-111.

Figure 3 is a flow diagram illustrating a method 300 for deciding whether to switch threads. Method 300 may be performed by MTC 106 of Figure 2, for example, TSL 120 of MTC 106. Referring now to Figure 3, at block 305, TSL determines whether to switch threads based on a thread switch policy that is selected from a list of thread policies based on unfairness levels of the first and second threads. In one embodiment, the thread switch policy is provided by PSU 115, which selects the policy according to the unfairness levels of the threads as provided by fairness meter (FM) 120 of Figure 2.

At block 310, TSL switches, in response to determining to switch threads, from executing the first thread to executing the second thread. In one embodiment, TSL switches from executing the first thread to executing the second thread by providing one or more switch threads command(s) to various thread switch selection points of the pipeline as discussed above.

Referring back to Figure 2, MTC 106 includes PSU 115 for selecting a thread switch policy which is used by TSL 120 as discussed above. In one embodiment, PSU 115 selects a thread switch policy according to information of unfairness provided by FM 120. In one embodiment, this includes information indicating which thread is the victim thread, and the corresponding unfairness level of that thread. In one embodiment, FM 120 determines the

unfairness level of the victim thread by comparing the unfairness counter of each corresponding thread.

In one embodiment, when a thread's unfairness counter increments, it reflects that in that cycle, the thread wanted to use the pipeline but was unable to use it because it was assigned to another thread. Thus, the value of each unfairness counter reflects a running total of the cycles where the thread was denied access to the pipeline. However, when the thread does get access to the pipeline for a cycle, the hardware (e.g., FM 120) compensates for the earlier denial of the pipeline cycle, by decrementing its corresponding unfairness counter. Thus, in one embodiment, an unfairness counter having a count of zero means that the thread has made up all the cycles it had earlier been denied. In one embodiment, the unfairness counters saturate at zero, i.e., they do not rollover to a negative value. In other words, there is no notion of negative unfairness – each thread only tracks cycles where it was denied access to the pipeline.

Referring still to Figure 2, in one embodiment, the unfairness counter for each thread is updated according to information provided by access monitor unit (AMU) 125 and 126, which monitors accessibility of each corresponding thread to the pipeline. In one embodiment, the information provided by each AMU includes an instruction set comprising of $\{+1, 0, -1\}$, where “+1” indicates the corresponding thread needed and was denied access to the pipeline; “-1” indicates the corresponding thread needed and was granted access to the pipeline; and “0” indicates the corresponding thread did not need or was not able to use the pipeline because it was waiting for other resources such as data return from a cache.

Although Figure 2 illustrates FM 120 updating its unfairness counters according to information provided by two AMUs, it will be appreciated that the number of AMUs shown in Figure 2 is only for illustrative purposes. It will be appreciated that more AMUs may be implemented in order to provide FM 120 with information to track more than two unfairness counters, which in turn enables MTC 106 to support more threads.

Figure 4 is a flow diagram illustrating a method 400 for maintaining an unfairness counter. Method 400 may be performed by a combination of AMU 125, 126 and FM 120 of Figure 2. Thus, all references made in the text discussing method 400 are made with respect to Figure 2. Moreover, the following discussion assumes that AMU 125 is associated with thread A and AMU 126 is associated with thread B.

At block 405, it is determined that a thread requires access to the pipeline. For instance, FM 120 may determine that thread A requires access to the pipeline because the corresponding AMU 125 issued a “+1” or “-1” instruction. Likewise, FM 120 may determine that thread B

requires access to the pipeline because the corresponding AMU 126 issued a “+1” or “-1” instruction.

At block 410, it is determined whether the requesting thread was granted access to the pipeline. If so, at block 415, the unfairness counter corresponding to the requesting thread is decremented. According to one embodiment, this is implemented as FM 120 decrementing the unfairness counter corresponding to thread A when AMU 125 issues a “-1” instruction. Likewise, FM 120 may decrement the unfairness counter corresponding to thread B when AMU 126 issues a “-1” instruction.

At block 420, it is determined whether the thread which was denied access to the pipeline was in the Unstalled state. In one embodiment, this state information is provided by the corresponding thread state unit (TSU) 110 and 111. Discussed in details below.

At block 425, after determining that the denied thread was in the Unstalled state, the unfairness counter corresponding to the denied thread is incremented. According to one embodiment, this is implemented as FM 120 incrementing the unfairness counter corresponding to thread A when AMU 125 issues a “+1” instruction. Likewise, FM 120 may increment the unfairness counter corresponding to thread B when AMU 126 issues a “+1” instruction.

At block 430, after determining that the denied thread was not in the Unstalled state, the unfairness counter corresponding to the denied thread is left unchanged. According to one embodiment, this is implemented as FM 120 receiving a “0” instruction from AMU 125 or 126.

According to one embodiment of the invention, method 400 is evaluated on every cycle. Thus, an unfairness counter may be updated on every cycle. In one embodiment, method 400 is implemented such that each unfairness counter may be updated on every cycle. Thus, for instance, method 400 may be duplicated multiple times in a system, each corresponding to a thread.

Figure 5 is a flow diagram illustrating a method 500 for selecting a thread based on a thread switch policy. Method 500 may be implemented by a combination of TSL 120, PSU 115, and FM 120 of Figure 2. Accordingly, all references made within the discussion of method 500 below are made with respect to Figure 2.

At block 505, a value of a first and second unfairness counter corresponding to the first and second thread, respectively, is received. In one embodiment, the first and second unfairness counters are implemented as part of FM 120.

At block 510, the value of the first unfairness counter is compared with the value of the second unfairness counter. In one embodiment, the comparison is performed by FM 120.

At block 515, a thread switch policy is identified based on the comparison. In one embodiment, the thread switch policy is identified by PSU 115 according to the comparison result of block 510.

At block 520, a thread is selected for execution based on the thread switch policy identified at block 515. In one embodiment, the thread selection is performed by TSL 120, based on the thread switch policy selected by PSU 115.

Figure 6A is a block diagram illustrating one embodiment of FM 120 of Figure 2. In one embodiment, FM 120 includes, but is not limited to, two unfairness counters 605 and 610, each corresponding to a thread. For example, unfairness counter 605 may be associated with thread A and unfairness counter 610 may be associated with thread B. In one embodiment, unfairness counters 605 and 610 are updated according to information from corresponding AMU 125 and AMU 126 of Figure 2, as discussed above.

In one embodiment, FM 120 determines the victim thread and its corresponding unfairness level by comparing the values of the unfairness counters. In one embodiment, the comparison is performed by subtracting the value of one unfairness counter from the other unfairness counter. By way of example, by comparing (e.g., subtracting) the value of unfairness counter 610 (corresponding to thread B) from the value of unfairness counter 605 (corresponding to thread A), FM 120 determines that thread A is the victim thread if the resulting difference is a positive number. Alternatively, FM 120 determines that thread B is the victim thread if the resulting difference is a negative number. In one embodiment, the magnitude of the difference is the unfairness level of the victim thread, which is used to influence the thread switch policy selection process performed by PSU 115 of Figure 2.

Although Figure 6A illustrates the unfairness level being implemented as a subtraction of two unfairness counters, each corresponding to a different thread, it will be appreciated that the unfairness level may be implemented in other ways. For instance, the unfairness level may be implemented using a single counter. In such an embodiment, the counter may be updated (e.g., incremented, decremented, by one or by some factor) depending on which thread is granted or denied access to the pipeline. By way of example, the single counter may be incremented whenever thread A is denied access to the pipeline, and decremented whenever thread B is denied access to the pipeline. Thus, a positive count value indicates that thread A is the victim thread, and a negative count value indicates that thread B is the victim thread. In such an embodiment, the magnitude of the single counter indicates the level of unfairness of the corresponding victim thread.

Figure 6B is a block diagram illustrating an embodiment of PSU 115 of Figure 2. In one embodiment, PSU 115 is implemented as a graded unfairness response system. As illustrated in Figure 6B, PSU 115 includes four regions which represent four different thread switch policies.

In one embodiment, PSU 115 defaults to a thread switch policy that is optimized for maximum utilization (i.e., region 0). However, when the unfairness level from FM 120 falls into one of the four regions, the corresponding thread switch policy is activated in order to provide fairness to the victim thread. In one embodiment, the polarity of the unfairness level (i.e., positive or negative), indicates which thread is the victim. For example, positive unfairness level may indicate thread A is the victim, while negative unfairness level indicates thread B is the victim. In one embodiment, since an increasing magnitude in the difference between the unfairness counters signifies an increasing degree of unfairness between the threads, the associated policies will increasingly favor a quick restoration of fairness over utilization and throughput. In other words, the unfairness policy in region 4 is the strongest fairness policy while the policy in region 0 is the weakest. This graded response scheme ensures that the minimal necessary performance is sacrificed in exchange for restoring fairness to the threads. In one embodiment, the regions and their respective thread switch policies are defined as follows:

Region	Unfairness Level Boundary	Thread Switch Policy
0	-L1 to +L1	Default thread switch policy which is optimized for maximizing pipeline utilization and performance.
1	-L2 to -L1 +L2 to +L1	The maximum duration for which a thread can continuously use the pipeline (called Timeout; described below in the text relating to Figure 7) is modulated to favor the victim thread.
2	-L3 to -L2 +L3 to +L2	In addition to region -1 policies, the pipeline switches to the victim thread whenever it is in the Unblocked state (describe below). This occurs irrespective of whether or not the victim's peer thread is in the Blocked state.
3	-L4 to -L3 +L4 to +L3	The pipeline switches to the victim thread and does not switch to the peer thread until software indicates that it has no useful work to do (using hint@pause instruction; described below).
4	-Max to -L4 +Max to +L4	The pipeline switches to the victim thread and does not switch to the peer thread until fairness is restored.

A fairness policy triggers when the unfairness level crosses into a particular region's trigger threshold (i.e., L1, L2, L3, or L4). In one embodiment, when a particular thread switch policy engages, the policy is sustained until the fairness is restored which, in one embodiment, occurs when the unfairness level reaches zero. This hysteresis makes the unfairness response policies robust. For example, it ensures that a thread never remains a victim perpetually by

oscillating between successive unfairness regions wherein the stronger policy restores enough fairness for the weaker policy to engage which is weak enough to cause another unfairness escalation.

In one embodiment, a mechanism is provided for deliberate biasing of resource allocation to a particular thread. Consider a case where it is desirable to have an N:M bias in allocating the pipeline to thread A vs. thread B. N and M shall herein be referred to as the FairTick of the corresponding thread. For example, by setting N=4 and M=1, the hardware is informed that thread A should have 4X more pipeline time than thread B. In one embodiment, this biasing is implemented as part of FM 120 of Figure 6A. According to one embodiment, each cycle of pipeline denial or access is multiplied by the FairTick before it is applied to the unfairness counter. For example, each cycle of pipeline denial that a thread suffers is multiplied by the FairTick of the thread and the result is added to the unfairness counter of the thread. In one embodiment, each cycle a thread is granted access to the pipeline is multiplied by the FairTick of a peer thread before it is decremented from the unfairness counter of the thread that has been given access to the pipeline.

Referring now back to Figure 2, which illustrates TSL 120 receiving thread state information of each thread from the corresponding TSU in order to make a more informed decision on whether to switch threads.

Figure 7 is a block diagram illustrating an embodiment of a thread state unit such as TSU 110 and 111 of Figure 2. When the SW initiates a thread, it assigns it to a TSU, which in one embodiment, maintains the information of the respective thread. In one embodiment, the information of each respective thread is used to influence the decision making process implemented as part of TSL 120 of Figure 2.

Referring now to Figure 7, according to one embodiment, in order to maximize utilization, each thread carries state information about its ability to use the pipeline. In one embodiment, this state information can be implemented as a finite state machine (FSM) 710 which includes the following states: Unstalled, Stalled, and Blocked.

According to one embodiment, when a thread is in the Unstalled state, it is able to make use of the pipeline; it is not waiting on anything. In one embodiment, the Stalled state indicates to TSL 120 that the thread cannot use the pipeline in the current cycle, but will likely be able to use of the pipeline shortly (i.e., the utilization benefit of switching to the other thread is probably less than the overhead of the thread switch itself). According to one embodiment, the Blocked state indicates to TSL 120 that the thread cannot use the resource in the current cycle, and the benefit of switching is probably greater than the overhead of the switch. By way of example, a

thread may start out in the Unstalled state, indicating that it is ready to utilize the pipeline. Assuming the thread is granted access to the pipeline, at one point during the execution, the thread may be required to wait (such as for data return from a cache). In such a scenario, the thread may enter the Stalled state. Depending on the time waited in the Stalled state and other
5 information such as hit/miss indications from various caches, a decision may be made by the thread to transition from the Stalled state to the Blocked state, which reflects the probability that more utilization will result from switching to an Unstalled background thread.

In one embodiment, to provide further flexibility and control over the state transitions of FSM 710, TSU 110 and 111 may be associated with software-configurable registers which may
10 serve as thresholds that must be reached or exceeded by the occurrence of an event before a state transition may or must occur. For example, a register may be provided which allows SW to indicate the minimum number of cycles that a thread must remain in the Unstalled state before it is permitted to transition to the Stalled state. This allows SW to ensure that a particular thread does not relinquish its priority unless the expected waiting time exceeds a certain programmable
15 period. In one embodiment, a register may also be provided which allows SW to indicate the maximum number of cycles that a thread may remain in the Blocked state. This allows SW to ensure that a particular thread is not stuck in the low priority for too long. These registers are discussed for illustrative purposes only, and FSM 710 is not limited to the use of these configurable registers. It will be appreciated that other configurable register(s) may be
20 implemented which may influence the behavior of FSM 710.

According to one embodiment of the invention, TSU 110 and 111 monitor the status of various units in the pipeline to determine the appropriate state transitions, including, for example, the status of instruction fetch unit 101, instruction decoder 102, rename/allocator 103, execution unit 104, and/or retirement unit 105 of processor pipeline 100 of Figure 1. For
25 instance, based on information provided by execution unit 104, TSU 110 or 111 may determine that the execution of the corresponding thread is temporarily halted because execution unit 104 is waiting for data from cache 107. In such a scenario, the corresponding TSU may elect to transition from Unstalled to Stalled state, allowing TSL 120 the freedom to switch to a peer background thread.

The hardware-based switch policies discussed above are effective in most situations. However, the effectiveness of the policies can be increased when SW can communicate to the hardware that certain situations apply. For example, SW may, in one embodiment, communicate the priority level of a thread in order to assure that it can be timely executed, or to inform the hardware that the thread is not time critical, and the pipeline may be dedicated to higher priority
30

threads. In one embodiment of the invention, TSU 110 and 111 include priority manager (PM) 720 to maintain information of thread priority level, which is provided to TSL 120 in order to help it make a more informed decision on whether or not to switch threads. In one embodiment, the thread priority level is configurable by SW. In one embodiment, the priority levels include:

5 High, Nominal, and Low. In one embodiment, PM 720 may also include timeout counter 721, which is configurable by SW to ensure that a thread is not stuck in a particular priority level for too long. In one embodiment, the expiration of timeout counter 721 may be used to influence the policy selection process performed by PSU 115 as discussed above.

Figure 8 is a flow diagram illustrating a method 800 for selecting a thread for execution.

10 Method 800 may be performed by the combination of TSU 110, 111 and TSL 120 of Figure 2. Thus, unless otherwise specified, the references made in the text relating to method 800 are with respect to Figure 2.

At block 805, the priority of a thread is set to Nominal. For example, TSU 110 may set its PM 720 of Figure 7 to indicate that the thread has Nominal priority.

15 At block 810, a hint instruction from SW is received. At block 815, it is determined whether the hint instruction is a hint@pause instruction, which in one embodiment, informs the hardware that the thread has nothing to do for some time to come. By way of example, this instruction may be issued inside the idle loop in the operating system (OS) since instructions executed in the loop do not accomplish any real work. During the time interval initiated by

20 hint@pause the hardware will allow the other thread to be executed without accruing any unfairness to the thread which issued the hint@pause instruction. If the instruction is a hint@pause instruction, the thread priority is set to Low at block 820. For example, TSU 110 may set its PM 720 of Figure 7 to indicate the thread has Low priority.

At block 825, it is determined whether the hint instruction is a hint@fair instruction,

25 which informs the hardware that a context switch is in progress and to change thread switch policies as needed to minimize unfair counts of both threads. By way of example, this instruction would typically be issued when a decision has been made by the OS to switch processes on a thread. The process switch routine, though optimized, necessarily stores a lot of state and may encounter many cache misses. This provides the hardware some freedom in selecting a thread

30 switch policy in order to minimize the unfair counts of both threads by the time the outgoing processes' context has been saved. If the instruction is a hint@fair instruction, the thread priority is set to Nominal at block 830. For example, TSU 110 may set its PM 720 of Figure 7 to indicate the thread has Low priority.

At block 835, it is determined whether the hint instruction is a hint@priority instruction, which informs that hardware not to switch threads while this situation is in force. This is especially important in cooperating processes where the critical code sections are important to system performance. By way of example, a hint@priority instruction may be issued when a lock is acquired in the critical code section. This permits the thread to execute the code as quickly as possible, denying the other thread access to the pipeline. If the instruction is a hint@priority instruction, the thread priority is set to High at block 840. For example, TSU 110 may set its PM 720 of Figure 7 to indicate the thread has High priority.

The setting of the thread priority is discussed above with respect to TSU 110. However, it will be appreciated that each of the above operations are to be performed for each thread. Thus, for example, the operations are also applicable to TSU 111, or any other additional TSUs that the system may include in order to support more threads.

At block 845, a thread is selected for execution based on the thread priority in view of the execution status of the thread. In one embodiment, the thread is selected by TSL 120 based on the thread priority provided by TSU 110 and 111. In one embodiment, the thread is also selected according to the execution status of the thread. In one embodiment, the execution status is indicated by FSM 710 of each corresponding TSU as shown in Figure 7.

Method 800 is described above as a sequence of operations. However, it will be appreciated that method 800 is not necessarily intended to be performed by a single unit or process sequentially. In fact, some operations of method 800 may be performed by one unit/process while other operations of method 800 may be performed by another unit/process. In addition, the various units/processes which perform the operations may perform them in parallel or in different sequence. Moreover, some operations of method 800 may be performed for each thread, while others are not. By way of example, as discussed above, the operations of blocks 805 through 840 may be performed by both TSU 110 and TSU 111; these operations may be performed by TSU 110 and TSU 111 in parallel or sequentially. Moreover, the operation of block 845 may be performed by a different unit/process such as TSL 120. Again, this operation may be performed in parallel with the operations of blocks 805 through 840. In fact, block 845 may be performed even though blocks 805 through 840 are not performed. For example, TSL 120 may continually, on each cycle, select a thread for execution based on the priority levels of the threads, even though the priority levels are not being updated, i.e. blocks 805 through 840 are not performed.

Referring again back to Figure 2, which illustrates that TSL 120 makes thread switching decisions based in part on external events 230. In some instances, even if a thread is totally

compute bound and can use the pipeline every cycle, a thread switch is forced after the expiration of a timeout counter, which may be configurable by SW. This is conceptually the same as timeout counter 721 of each TSU as illustrated in Figure 7, but configured with a different time out value.

5 External interrupts are an important way to notify the processor of a condition that needs timely handling. Thus, according to one embodiment, TSL 120 is also configured to make thread switching decisions according to such external interrupts.

In complex pipelined SMT processors, there could be situations where a thread is not able to retire an instruction before being switched out. And it could be possible that the execution
10 of the other thread could perpetuate this situation so that although a thread has access to the pipeline, it can never make forward progress. Thus, according to one embodiment, MTC 106 includes a forward progress mechanism to detect such conditions and inform TSL 120 to switch threads as needed to guarantee forward progress.

The thread switching mechanism discussed above may be implemented to switch the
15 main execution pipeline, such as processor pipeline 100 of Figure 1. However, it will be appreciated that the mechanism can be extended to cover more than just the main execution pipeline. In one embodiment, the above thread switching mechanism may also be applied to a memory pipeline which maintains queues of memory transactions and can issue memory operations for any thread at any time. For example, there can be situations where thread A has
20 access to the main pipeline, but cannot issue a memory transaction because the queue of memory transactions is filled with transactions from thread B. In this case, thread A is wasting a main pipeline cycle due to prior ownership of a resource from thread B and thus meets the definition of unfair cycle. Thus, in one embodiment, the unfairness counters may be extended to count an unfair cycle in these situations in order to prevent another form of unfairness at a very reasonable
25 cost.

An instruction set, or instruction set architecture (ISA), is the part of the computer architecture related to programming, and may include the native data types, instructions, register architecture, addressing modes, memory architecture, interrupt and exception handling, and external input and output (I/O). The term instruction generally refers herein to macro-
30 instructions – that is instructions that are provided to the processor (or instruction converter that translates (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morphs, emulates, or otherwise converts an instruction to one or more other instructions to be processed by the processor) for execution – as opposed to micro-instructions or

micro-operations (micro-ops) – that is the result of a processor’s decoder decoding macro-instructions.

The ISA is distinguished from the microarchitecture, which is the internal design of the processor implementing the instruction set. Processors with different microarchitectures can share a common instruction set. For example, Intel® Pentium 4 processors, Intel® Core™ processors, and processors from Advanced Micro Devices, Inc. of Sunnyvale CA implement nearly identical versions of the x86 instruction set (with some extensions that have been added with newer versions), but have different internal designs. For example, the same register architecture of the ISA may be implemented in different ways in different microarchitectures using well-known techniques, including dedicated physical registers, one or more dynamically allocated physical registers using a register renaming mechanism (e.g., the use of a Register Alias Table (RAT), a Reorder Buffer (ROB), and a retirement register file; the use of multiple maps and a pool of registers), etc. Unless otherwise specified, the phrases register architecture, register file, and register are used herein to refer to that which is visible to the software/programmer and the manner in which instructions specify registers. Where a specificity is desired, the adjective logical, architectural, or software visible will be used to indicate registers/files in the register architecture, while different adjectives will be used to designation registers in a given microarchitecture (e.g., physical register, reorder buffer, retirement register, register pool).

An instruction set includes one or more instruction formats. A given instruction format defines various fields (number of bits, location of bits) to specify, among other things, the operation to be performed (opcode) and the operand(s) on which that operation is to be performed. Some instruction formats are further broken down though the definition of instruction templates (or subformats). For example, the instruction templates of a given instruction format may be defined to have different subsets of the instruction format’s fields (the included fields are typically in the same order, but at least some have different bit positions because there are less fields included) and/or defined to have a given field interpreted differently. Thus, each instruction of an ISA is expressed using a given instruction format (and, if defined, in a given one of the instruction templates of that instruction format) and includes fields for specifying the operation and the operands. For example, an exemplary ADD instruction has a specific opcode and an instruction format that includes an opcode field to specify that opcode and operand fields to select operands (source1/destination and source2); and an occurrence of this ADD instruction in an instruction stream will have specific contents in the operand fields that select specific operands.

Scientific, financial, auto-vectorized general purpose, RMS (recognition, mining, and synthesis), and visual and multimedia applications (e.g., 2D/3D graphics, image processing, video compression/decompression, voice recognition algorithms and audio manipulation) often require the same operation to be performed on a large number of data items (referred to as "data parallelism"). Single Instruction Multiple Data (SIMD) refers to a type of instruction that causes a processor to perform an operation on multiple data items. SIMD technology is especially suited to processors that can logically divide the bits in a register into a number of fixed-sized data elements, each of which represents a separate value. For example, the bits in a 256-bit register may be specified as a source operand to be operated on as four separate 64-bit packed data elements (quad-word (Q) size data elements), eight separate 32-bit packed data elements (double word (D) size data elements), sixteen separate 16-bit packed data elements (word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). This type of data is referred to as packed data type or vector data type, and operands of this data type are referred to as packed data operands or vector operands. In other words, a packed data item or vector refers to a sequence of packed data elements, and a packed data operand or a vector operand is a source or destination operand of a SIMD instruction (also known as a packed data instruction or a vector instruction).

By way of example, one type of SIMD instruction specifies a single vector operation to be performed on two source vector operands in a vertical fashion to generate a destination vector operand (also referred to as a result vector operand) of the same size, with the same number of data elements, and in the same data element order. The data elements in the source vector operands are referred to as source data elements, while the data elements in the destination vector operand are referred to as destination or result data elements. These source vector operands are of the same size and contain data elements of the same width, and thus they contain the same number of data elements. The source data elements in the same bit positions in the two source vector operands form pairs of data elements (also referred to as corresponding data elements; that is, the data element in data element position 0 of each source operand correspond, the data element in data element position 1 of each source operand correspond, and so on). The operation specified by that SIMD instruction is performed separately on each of these pairs of source data elements to generate a matching number of result data elements, and thus each pair of source data elements has a corresponding result data element. Since the operation is vertical and since the result vector operand is the same size, has the same number of data elements, and the result data elements are stored in the same data element order as the source vector operands, the result data elements are in the same bit positions of the result vector operand as their

corresponding pair of source data elements in the source vector operands. In addition to this exemplary type of SIMD instruction, there are a variety of other types of SIMD instructions (e.g., that has only one or has more than two source vector operands, that operate in a horizontal fashion, that generates a result vector operand that is of a different size, that has a different size data elements, and/or that has a different data element order). It should be understood that the term destination vector operand (or destination operand) is defined as the direct result of performing the operation specified by an instruction, including the storage of that destination operand at a location (be it a register or at a memory address specified by that instruction) so that it may be accessed as a source operand by another instruction (by specification of that same location by the another instruction).

The SIMD technology, such as that employed by the Intel® Core™ processors having an instruction set including x86, MMX™, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, and SSE4.2 instructions, has enabled a significant improvement in application performance. An additional set of SIMD extensions, referred to the Advanced Vector Extensions (AVX) (AVX1 and AVX2) and using the Vector Extensions (VEX) coding scheme, has been , has been released and/or published (e.g., see Intel® 64 and IA-32 Architectures Software Developers Manual, October 2011; and see Intel® Advanced Vector Extensions Programming Reference, June 2011).

Embodiments of the instruction(s) described herein may be embodied in different formats. Additionally, exemplary systems, architectures, and pipelines are detailed below. Embodiments of the instruction(s) may be executed on such systems, architectures, and pipelines, but are not limited to those detailed.

VEX encoding allows instructions to have more than two operands, and allows SIMD vector registers to be longer than 128 bits. The use of a VEX prefix provides for three-operand (or more) syntax. For example, previous two-operand instructions performed operations such as $A = A + B$, which overwrites a source operand. The use of a VEX prefix enables operands to perform nondestructive operations such as $A = B + C$.

Figure 9A illustrates an exemplary AVX instruction format including a VEX prefix 2102, real opcode field 2130, Mod R/M byte 2140, SIB byte 2150, displacement field 2162, and IMM8 2172. Figure 9B illustrates which fields from Figure 9A make up a full opcode field 2174 and a base operation field 2142. Figure 9C illustrates which fields from Figure 9A make up a register index field 2144.

VEX Prefix (Bytes 0-2) 2102 is encoded in a three-byte form. The first byte is the Format Field 2140 (VEX Byte 0, bits [7:0]), which contains an explicit C4 byte value (the unique

value used for distinguishing the C4 instruction format). The second-third bytes (VEX Bytes 1-2) include a number of bit fields providing specific capability. Specifically, REX field 2105 (VEX Byte 1, bits [7-5]) consists of a VEX.R bit field (VEX Byte 1, bit [7] – R), VEX.X bit field (VEX byte 1, bit [6] – X), and VEX.B bit field (VEX byte 1, bit[5] – B). Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding VEX.R, VEX.X, and VEX.B. Opcode map field 2115 (VEX byte 1, bits [4:0] – mmmmm) includes content to encode an implied leading opcode byte. W Field 2164 (VEX byte 2, bit [7] – W) - is represented by the notation VEX.W, and provides different functions depending on the instruction. The role of VEX.vvvv 2120 (VEX Byte 2, bits [6:3]-vvvv) may include the following: 1) VEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) VEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3) VEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. If VEX.L 2168 Size field (VEX byte 2, bit [2]-L) = 0, it indicates 128 bit vector; if VEX.L = 1, it indicates 256 bit vector. Prefix encoding field 2125 (VEX byte 2, bits [1:0]-pp) provides additional bits for the base operation field.

Real Opcode Field 2130 (Byte 3) is also known as the opcode byte. Part of the opcode is specified in this field. MOD R/M Field 2140 (Byte 4) includes MOD field 2142 (bits [7-6]), Reg field 2144 (bits [5-3]), and R/M field 2146 (bits [2-0]). The role of Reg field 2144 may include the following: encoding either the destination register operand or a source register operand (the rrr of Rrrr), or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 2146 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) - The content of Scale field 2150 (Byte 5) includes SS 2152 (bits [7-6]), which is used for memory address generation. The contents of SIB.xxx 2154 (bits [5-3]) and SIB.bbb 2156 (bits [2-0]) have been previously referred to with regard to the register indexes Xxxx and Bbbb. The Displacement Field 2162 and the immediate field (IMM8) 2172 contain address data.

A vector friendly instruction format is an instruction format that is suited for vector instructions (e.g., there are certain fields specific to vector operations). While embodiments are described in which both vector and scalar operations are supported through the vector friendly

instruction format, alternative embodiments use only vector operations the vector friendly instruction format.

Figure 10A, Figure 10B, and Figure 10C are block diagrams illustrating a generic vector friendly instruction format and instruction templates thereof according to embodiments of the invention. Figure 10A is a block diagram illustrating a generic vector friendly instruction format and class A instruction templates thereof according to embodiments of the invention; while Figure 10B is a block diagram illustrating the generic vector friendly instruction format and class B instruction templates thereof according to embodiments of the invention. Specifically, a generic vector friendly instruction format 2200 for which are defined class A and class B instruction templates, both of which include no memory access 2205 instruction templates and memory access 2220 instruction templates. The term generic in the context of the vector friendly instruction format refers to the instruction format not being tied to any specific instruction set.

While embodiments of the invention will be described in which the vector friendly instruction format supports the following: a 64 byte vector operand length (or size) with 32 bit (4 byte) or 64 bit (8 byte) data element widths (or sizes) (and thus, a 64 byte vector consists of either 16 doubleword-size elements or alternatively, 8 quadword-size elements); a 64 byte vector operand length (or size) with 16 bit (2 byte) or 8 bit (1 byte) data element widths (or sizes); a 32 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); and a 16 byte vector operand length (or size) with 32 bit (4 byte), 64 bit (8 byte), 16 bit (2 byte), or 8 bit (1 byte) data element widths (or sizes); alternative embodiments may support more, less and/or different vector operand sizes (e.g., 256 byte vector operands) with more, less, or different data element widths (e.g., 128 bit (16 byte) data element widths).

The class A instruction templates in Figure 10A include: 1) within the no memory access 2205 instruction templates there is shown a no memory access, full round control type operation 2210 instruction template and a no memory access, data transform type operation 2215 instruction template; and 2) within the memory access 2220 instruction templates there is shown a memory access, temporal 2225 instruction template and a memory access, non-temporal 2230 instruction template. The class B instruction templates in Figure 10B include: 1) within the no memory access 2205 instruction templates there is shown a no memory access, write mask control, partial round control type operation 2212 instruction template and a no memory access, write mask control, vsize type operation 2217 instruction template; and 2) within the memory

access 2220 instruction templates there is shown a memory access, write mask control 2227 instruction template.

The generic vector friendly instruction format 2200 includes the following fields listed below in the order illustrated in Figure 10A and Figure 10B. Format field 2240 – a specific value (an instruction format identifier value) in this field uniquely identifies the vector friendly instruction format, and thus occurrences of instructions in the vector friendly instruction format in instruction streams. As such, this field is optional in the sense that it is not needed for an instruction set that has only the generic vector friendly instruction format. Base operation field 2242 – its content distinguishes different base operations.

Register index field 2244 – its content, directly or through address generation, specifies the locations of the source and destination operands, be they in registers or in memory. These include a sufficient number of bits to select N registers from a PxQ (e.g. 32x512, 16x128, 32x1024, 64x1024) register file. While in one embodiment N may be up to three sources and one destination register, alternative embodiments may support more or less sources and destination registers (e.g., may support up to two sources where one of these sources also acts as the destination, may support up to three sources where one of these sources also acts as the destination, may support up to two sources and one destination).

Modifier field 2246 – its content distinguishes occurrences of instructions in the generic vector instruction format that specify memory access from those that do not; that is, between no memory access 2205 instruction templates and memory access 2220 instruction templates. Memory access operations read and/or write to the memory hierarchy (in some cases specifying the source and/or destination addresses using values in registers), while non-memory access operations do not (e.g., the source and destinations are registers). While in one embodiment this field also selects between three different ways to perform memory address calculations, alternative embodiments may support more, less, or different ways to perform memory address calculations.

Augmentation operation field 2250 – its content distinguishes which one of a variety of different operations to be performed in addition to the base operation. This field is context specific. In one embodiment of the invention, this field is divided into a class field 2268, an alpha field 2252, and a beta field 2254. The augmentation operation field 2250 allows common groups of operations to be performed in a single instruction rather than 2, 3, or 4 instructions. Scale field 2260 – its content allows for the scaling of the index field's content for memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base}$).

Displacement Field 2262A– its content is used as part of memory address generation (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{displacement}$). Displacement Factor Field 2262B (note that the juxtaposition of displacement field 2262A directly over displacement factor field 2262B indicates one or the other is used) – its content is used as part of address generation; it specifies a displacement factor that is to be scaled by the size of a memory access (N) – where N is the number of bytes in the memory access (e.g., for address generation that uses $2^{\text{scale}} * \text{index} + \text{base} + \text{scaled displacement}$). Redundant low-order bits are ignored and hence, the displacement factor field's content is multiplied by the memory operands total size (N) in order to generate the final displacement to be used in calculating an effective address. The value of N is determined by the processor hardware at runtime based on the full opcode field 2274 (described later herein) and the data manipulation field 2254C. The displacement field 2262A and the displacement factor field 2262B are optional in the sense that they are not used for the no memory access 2205 instruction templates and/or different embodiments may implement only one or none of the two.

Data element width field 2264 – its content distinguishes which one of a number of data element widths is to be used (in some embodiments for all instructions; in other embodiments for only some of the instructions). This field is optional in the sense that it is not needed if only one data element width is supported and/or data element widths are supported using some aspect of the opcodes.

Write mask field 2270 – its content controls, on a per data element position basis, whether that data element position in the destination vector operand reflects the result of the base operation and augmentation operation. Class A instruction templates support merging-writemasking, while class B instruction templates support both merging- and zeroing-writemasking. When merging, vector masks allow any set of elements in the destination to be protected from updates during the execution of any operation (specified by the base operation and the augmentation operation); in other one embodiment, preserving the old value of each element of the destination where the corresponding mask bit has a 0. In contrast, when zeroing vector masks allow any set of elements in the destination to be zeroed during the execution of any operation (specified by the base operation and the augmentation operation); in one embodiment, an element of the destination is set to 0 when the corresponding mask bit has a 0 value. A subset of this functionality is the ability to control the vector length of the operation being performed (that is, the span of elements being modified, from the first to the last one); however, it is not necessary that the elements that are modified be consecutive. Thus, the write mask field 2270 allows for partial vector operations, including loads, stores, arithmetic, logical,

etc. While embodiments of the invention are described in which the write mask field's 2270 content selects one of a number of write mask registers that contains the write mask to be used (and thus the write mask field's 2270 content indirectly identifies that masking to be performed), alternative embodiments instead or additionally allow the mask write field's 2270 content to

5 directly specify the masking to be performed.

Immediate field 2272 – its content allows for the specification of an immediate. This field is optional in the sense that it is not present in an implementation of the generic vector friendly format that does not support immediate and it is not present in instructions that do not use an immediate. Class field 2268 – its content distinguishes between different classes of
10 instructions. With reference to Figure 10A and Figure 10B, the contents of this field select between class A and class B instructions. In Figure 10A and Figure 10B, rounded corner squares are used to indicate a specific value is present in a field (e.g., class A 2268A and class B 2268B for the class field 2268 respectively in Figure 10A and Figure 10B).

In the case of the non-memory access 2205 instruction templates of class A, the alpha
15 field 2252 is interpreted as an RS field 2252A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 2252A.1 and data transform 2252A.2 are respectively specified for the no memory access, round type operation 2210 and the no memory access, data transform type operation 2215 instruction templates), while the beta field 2254 distinguishes which of the operations of the specified type is to be
20 performed. In the no memory access 2205 instruction templates, the scale field 2260, the displacement field 2262A, and the displacement scale field 2262B are not present.

In the no memory access full round control type operation 2210 instruction template, the beta field 2254 is interpreted as a round control field 2254A, whose content(s) provide static rounding. While in the described embodiments of the invention the round control field 2254A
25 includes a suppress all floating point exceptions (SAE) field 2256 and a round operation control field 2258, alternative embodiments may support may encode both these concepts into the same field or only have one or the other of these concepts/fields (e.g., may have only the round operation control field 2258).

SAE field 2256 – its content distinguishes whether or not to disable the exception event
30 reporting; when the SAE field's 2256 content indicates suppression is enabled, a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler.

Round operation control field 2258 – its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-

to-nearest). Thus, the round operation control field 2258 allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 2250 content overrides that register value.

5 In the no memory access data transform type operation 2215 instruction template, the beta field 2254 is interpreted as a data transform field 2254B, whose content distinguishes which one of a number of data transforms is to be performed (e.g., no data transform, swizzle, broadcast).

10 In the case of a memory access 2220 instruction template of class A, the alpha field 2252 is interpreted as an eviction hint field 2252B, whose content distinguishes which one of the eviction hints is to be used (in Figure 10A, temporal 2252B.1 and non-temporal 2252B.2 are respectively specified for the memory access, temporal 2225 instruction template and the memory access, non-temporal 2230 instruction template), while the beta field 2254 is interpreted as a data manipulation field 2254C, whose content distinguishes which one of a number of data
15 manipulation operations (also known as primitives) is to be performed (e.g., no manipulation; broadcast; up conversion of a source; and down conversion of a destination). The memory access 2220 instruction templates include the scale field 2260, and optionally the displacement field 2262A or the displacement scale field 2262B.

20 Vector memory instructions perform vector loads from and vector stores to memory, with conversion support. As with regular vector instructions, vector memory instructions transfer data from/to memory in a data element-wise fashion, with the elements that are actually transferred is dictated by the contents of the vector mask that is selected as the write mask.

25 Temporal data is data likely to be reused soon enough to benefit from caching. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely. Non-temporal data is data unlikely to be reused soon enough to benefit from caching in the 1st-level cache and should be given priority for eviction. This is, however, a hint, and different processors may implement it in different ways, including ignoring the hint entirely.

30 In the case of the instruction templates of class B, the alpha field 2252 is interpreted as a write mask control (Z) field 2252C, whose content distinguishes whether the write masking controlled by the write mask field 2270 should be a merging or a zeroing.

In the case of the non-memory access 2205 instruction templates of class B, part of the beta field 2254 is interpreted as an RL field 2257A, whose content distinguishes which one of the different augmentation operation types are to be performed (e.g., round 2257A.1 and vector length (VSIZE) 2257A.2 are respectively specified for the no memory access, write mask

control, partial round control type operation 2212 instruction template and the no memory access, write mask control, VSIZE type operation 2217 instruction template), while the rest of the beta field 2254 distinguishes which of the operations of the specified type is to be performed. In the no memory access 2205 instruction templates, the scale field 2260, the displacement field 2262A, and the displacement scale field 2262B are not present.

In the no memory access, write mask control, partial round control type operation 2210 instruction template, the rest of the beta field 2254 is interpreted as a round operation field 2259A and exception event reporting is disabled (a given instruction does not report any kind of floating-point exception flag and does not raise any floating point exception handler).

Round operation control field 2259A – just as round operation control field 2258, its content distinguishes which one of a group of rounding operations to perform (e.g., Round-up, Round-down, Round-towards-zero and Round-to-nearest). Thus, the round operation control field 2259A allows for the changing of the rounding mode on a per instruction basis. In one embodiment of the invention where a processor includes a control register for specifying rounding modes, the round operation control field's 2250 content overrides that register value.

In the no memory access, write mask control, VSIZE type operation 2217 instruction template, the rest of the beta field 2254 is interpreted as a vector length field 2259B, whose content distinguishes which one of a number of data vector lengths is to be performed on (e.g., 128, 256, or 512 byte).

In the case of a memory access 2220 instruction template of class B, part of the beta field 2254 is interpreted as a broadcast field 2257B, whose content distinguishes whether or not the broadcast type data manipulation operation is to be performed, while the rest of the beta field 2254 is interpreted the vector length field 2259B. The memory access 2220 instruction templates include the scale field 2260, and optionally the displacement field 2262A or the displacement scale field 2262B.

With regard to the generic vector friendly instruction format 2200, a full opcode field 2274 is shown including the format field 2240, the base operation field 2242, and the data element width field 2264. While one embodiment is shown where the full opcode field 2274 includes all of these fields, the full opcode field 2274 includes less than all of these fields in embodiments that do not support all of them. The full opcode field 2274 provides the operation code (opcode).

The augmentation operation field 2250, the data element width field 2264, and the write mask field 2270 allow these features to be specified on a per instruction basis in the generic vector friendly instruction format. The combination of write mask field and data element width

field create typed instructions in that they allow the mask to be applied based on different data element widths.

The various instruction templates found within class A and class B are beneficial in different situations. In some embodiments of the invention, different processors or different
5 cores within a processor may support only class A, only class B, or both classes. For instance, a high performance general purpose out-of-order core intended for general-purpose computing may support only class B, a core intended primarily for graphics and/or scientific (throughput) computing may support only class A, and a core intended for both may support both (of course, a core that has some mix of templates and instructions from both classes but not all templates and
10 instructions from both classes is within the purview of the invention). Also, a single processor may include multiple cores, all of which support the same class or in which different cores support different class. For instance, in a processor with separate graphics and general purpose cores, one of the graphics cores intended primarily for graphics and/or scientific computing may support only class A, while one or more of the general purpose cores may be high performance
15 general purpose cores with out of order execution and register renaming intended for general-purpose computing that support only class B. Another processor that does not have a separate graphics core, may include one more general purpose in-order or out-of-order cores that support both class A and class B. Of course, features from one class may also be implemented in the other class in different embodiments of the invention. Programs written in a high level language
20 would be put (e.g., just in time compiled or statically compiled) into an variety of different executable forms, including: 1) a form having only instructions of the class(es) supported by the target processor for execution; or 2) a form having alternative routines written using different combinations of the instructions of all classes and having control flow code that selects the routines to execute based on the instructions supported by the processor which is currently
25 executing the code.

Figure 11 is a block diagram illustrating an exemplary specific vector friendly instruction format according to embodiments of the invention. Figure 11 shows a specific vector friendly instruction format 2300 that is specific in the sense that it specifies the location, size, interpretation, and order of the fields, as well as values for some of those fields. The specific
30 vector friendly instruction format 2300 may be used to extend the x86 instruction set, and thus some of the fields are similar or the same as those used in the existing x86 instruction set and extension thereof (e.g., AVX). This format remains consistent with the prefix encoding field, real opcode byte field, MOD R/M field, SIB field, displacement field, and immediate fields of

the existing x86 instruction set with extensions. The fields from Figure 10 into which the fields from Figure 11 map are illustrated.

It should be understood that, although embodiments of the invention are described with reference to the specific vector friendly instruction format 2300 in the context of the generic vector friendly instruction format 2200 for illustrative purposes, the invention is not limited to the specific vector friendly instruction format 2300 except where claimed. For example, the generic vector friendly instruction format 2200 contemplates a variety of possible sizes for the various fields, while the specific vector friendly instruction format 2300 is shown as having fields of specific sizes. By way of specific example, while the data element width field 2264 is illustrated as a one bit field in the specific vector friendly instruction format 2300, the invention is not so limited (that is, the generic vector friendly instruction format 2200 contemplates other sizes of the data element width field 2264).

The generic vector friendly instruction format 2200 includes the following fields listed below in the order illustrated in Figure 11A. EVEX Prefix (Bytes 0-3) 2302 - is encoded in a four-byte form. Format Field 2240 (EVEX Byte 0, bits [7:0]) - the first byte (EVEX Byte 0) is the format field 2240 and it contains 0x62 (the unique value used for distinguishing the vector friendly instruction format in one embodiment of the invention). The second-fourth bytes (EVEX Bytes 1-3) include a number of bit fields providing specific capability.

REX field 2305 (EVEX Byte 1, bits [7-5]) – consists of a EVEX.R bit field (EVEX Byte 1, bit [7] – R), EVEX.X bit field (EVEX byte 1, bit [6] – X), and EVEX.B bit field (EVEX byte 1, bit [5] – B). The EVEX.R, EVEX.X, and EVEX.B bit fields provide the same functionality as the corresponding VEX bit fields, and are encoded using 1s complement form, i.e. ZMM0 is encoded as 1111B, ZMM15 is encoded as 0000B. Other fields of the instructions encode the lower three bits of the register indexes as is known in the art (rrr, xxx, and bbb), so that Rrrr, Xxxx, and Bbbb may be formed by adding EVEX.R, EVEX.X, and EVEX.B.

REX' field 2210 – this is the first part of the REX' field 2210 and is the EVEX.R' bit field (EVEX Byte 1, bit [4] - R') that is used to encode either the upper 16 or lower 16 of the extended 32 register set. In one embodiment of the invention, this bit, along with others as indicated below, is stored in bit inverted format to distinguish (in the well-known x86 32-bit mode) from the BOUND instruction, whose real opcode byte is 62, but does not accept in the MOD R/M field (described below) the value of 11 in the MOD field; alternative embodiments of the invention do not store this and the other indicated bits below in the inverted format. A value of 1 is used to encode the lower 16 registers. In other words, R'Rrrr is formed by combining EVEX.R', EVEX.R, and the other RRR from other fields.

Opcode map field 2315 (EVEX byte 1, bits [3:0] – mmmm) – its content encodes an implied leading opcode byte (0F, 0F 38, or 0F 3). Data element width field 2264 (EVEX byte 2, bit [7] – W) – is represented by the notation EVEX.W. EVEX.W is used to define the granularity (size) of the data type (either 32-bit data elements or 64-bit data elements). EVEX.vvvv2320

(EVEX Byte 2, bits [6:3]-vvvv)- the role of EVEX.vvvv may include the following: 1)

EVEX.vvvv encodes the first source register operand, specified in inverted (1s complement) form and is valid for instructions with 2 or more source operands; 2) EVEX.vvvv encodes the destination register operand, specified in 1s complement form for certain vector shifts; or 3)

EVEX.vvvv does not encode any operand, the field is reserved and should contain 1111b. Thus,

EVEX.vvvv field 2320 encodes the 4 low-order bits of the first source register specifier stored in inverted (1s complement) form. Depending on the instruction, an extra different EVEX bit field is used to extend the specifier size to 32 registers. EVEX.U 2268 Class field (EVEX byte 2, bit [2]-U) – If EVEX.U = 0, it indicates class A or EVEX.U0; if EVEX.U = 1, it indicates class B or EVEX.U1.

Prefix encoding field 2325 (EVEX byte 2, bits [1:0]-pp) – provides additional bits for the base operation field. In addition to providing support for the legacy SSE instructions in the EVEX prefix format, this also has the benefit of compacting the SIMD prefix (rather than requiring a byte to express the SIMD prefix, the EVEX prefix requires only 2 bits). In one embodiment, to support legacy SSE instructions that use a SIMD prefix (66H, F2H, F3H) in both the legacy format and in the EVEX prefix format, these legacy SIMD prefixes are encoded into the SIMD prefix encoding field; and at runtime are expanded into the legacy SIMD prefix prior to being provided to the decoder's PLA (so the PLA can execute both the legacy and EVEX format of these legacy instructions without modification). Although newer instructions could use the EVEX prefix encoding field's content directly as an opcode extension, certain embodiments expand in a similar fashion for consistency but allow for different meanings to be specified by these legacy SIMD prefixes. An alternative embodiment may redesign the PLA to support the 2 bit SIMD prefix encodings, and thus not require the expansion.

Alpha field 2252 (EVEX byte 3, bit [7] – EH; also known as EVEX.EH, EVEX.rs, EVEX.RL, EVEX.write mask control, and EVEX.N; also illustrated with α) – as previously described, this field is context specific. Beta field 2254 (EVEX byte 3, bits [6:4]-SSS, also known as EVEX.s2-0, EVEX.r2-0, EVEX.rr1, EVEX.LL0, EVEX.LLB; also illustrated with $\beta\beta\beta$) – as previously described, this field is context specific.

REX' field 2210 – this is the remainder of the REX' field and is the EVEX.V' bit field (EVEX Byte 3, bit [3] - V') that may be used to encode either the upper 16 or lower 16 of the

extended 32 register set. This bit is stored in bit inverted format. A value of 1 is used to encode the lower 16 registers. In other words, V'VVVV is formed by combining EVEX.V', EVEX.vvvv.

Write mask field 2270 (EVEX byte 3, bits [2:0]-kkk) – its content specifies the index of a register in the write mask registers as previously described. In one embodiment of the invention, the specific value EVEX.kkk=000 has a special behavior implying no write mask is used for the particular instruction (this may be implemented in a variety of ways including the use of a write mask hardwired to all ones or hardware that bypasses the masking hardware).

Real Opcode Field 2330 (Byte 4) is also known as the opcode byte. Part of the opcode is specified in this field. MOD R/M Field 2340 (Byte 5) includes MOD field 2342, Reg field 2344, and R/M field 2346. As previously described, the MOD field's 2342 content distinguishes between memory access and non-memory access operations. The role of Reg field 2344 can be summarized to two situations: encoding either the destination register operand or a source register operand, or be treated as an opcode extension and not used to encode any instruction operand. The role of R/M field 2346 may include the following: encoding the instruction operand that references a memory address, or encoding either the destination register operand or a source register operand.

Scale, Index, Base (SIB) Byte (Byte 6) - As previously described, the scale field's 2250 content is used for memory address generation. SIB.xxx 2354 and SIB.bbb 2356 – the contents of these fields have been previously referred to with regard to the register indexes Xxxx and Bbbb. Displacement field 2262A (Bytes 7-10) – when MOD field 2342 contains 10, bytes 7-10 are the displacement field 2262A, and it works the same as the legacy 32-bit displacement (disp32) and works at byte granularity.

Displacement factor field 2262B (Byte 7) – when MOD field 2342 contains 01, byte 7 is the displacement factor field 2262B. The location of this field is that same as that of the legacy x86 instruction set 8-bit displacement (disp8), which works at byte granularity. Since disp8 is sign extended, it can only address between -128 and 127 bytes offsets; in terms of 64 byte cache lines, disp8 uses 8 bits that can be set to only four really useful values -128, -64, 0, and 64; since a greater range is often needed, disp32 is used; however, disp32 requires 4 bytes. In contrast to disp8 and disp32, the displacement factor field 2262B is a reinterpretation of disp8; when using displacement factor field 2262B, the actual displacement is determined by the content of the displacement factor field multiplied by the size of the memory operand access (N). This type of displacement is referred to as disp8*N. This reduces the average instruction length (a single byte of used for the displacement but with a much greater range). Such compressed displacement is

based on the assumption that the effective displacement is multiple of the granularity of the memory access, and hence, the redundant low-order bits of the address offset do not need to be encoded. In other words, the displacement factor field 2262B substitutes the legacy x86 instruction set 8-bit displacement. Thus, the displacement factor field 2262B is encoded the same way as an x86 instruction set 8-bit displacement (so no changes in the ModRM/SIB encoding rules) with the only exception that disp8 is overloaded to disp8*N. In other words, there are no changes in the encoding rules or encoding lengths but only in the interpretation of the displacement value by hardware (which needs to scale the displacement by the size of the memory operand to obtain a byte-wise address offset). Immediate field 2272 operates as previously described.

Figure 11B is a block diagram illustrating the fields of the specific vector friendly instruction format 2300 that make up the full opcode field 2274 according to one embodiment of the invention. Specifically, the full opcode field 2274 includes the format field 2240, the base operation field 2242, and the data element width (W) field 2264. The base operation field 2242 includes the prefix encoding field 2325, the opcode map field 2315, and the real opcode field 2330.

Figure 11C is a block diagram illustrating the fields of the specific vector friendly instruction format 2300 that make up the register index field 2244 according to one embodiment of the invention. Specifically, the register index field 2244 includes the REX field 2305, the REX' field 2310, the MODR/M.reg field 2344, the MODR/M.r/m field 2346, the VVVV field 2320, xxx field 2354, and the bbb field 2356.

Figure 11D is a block diagram illustrating the fields of the specific vector friendly instruction format 2300 that make up the augmentation operation field 2250 according to one embodiment of the invention. When the class (U) field 2268 contains 0, it signifies EVEX.U0 (class A 2268A); when it contains 1, it signifies EVEX.U1 (class B 2268B). When U=0 and the MOD field 2342 contains 11 (signifying a no memory access operation), the alpha field 2252 (EVEX byte 3, bit [7] – EH) is interpreted as the rs field 2252A. When the rs field 2252A contains a 1 (round 2252A.1), the beta field 2254 (EVEX byte 3, bits [6:4]– SSS) is interpreted as the round control field 2254A. The round control field 2254A includes a one bit SAE field 2256 and a two bit round operation field 2258. When the rs field 2252A contains a 0 (data transform 2252A.2), the beta field 2254 (EVEX byte 3, bits [6:4]– SSS) is interpreted as a three bit data transform field 2254B. When U=0 and the MOD field 2342 contains 00, 01, or 10 (signifying a memory access operation), the alpha field 2252 (EVEX byte 3, bit [7] – EH) is

interpreted as the eviction hint (EH) field 2252B and the beta field 2254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as a three bit data manipulation field 2254C.

When U=1, the alpha field 2252 (EVEX byte 3, bit [7] – EH) is interpreted as the write mask control (Z) field 2252C. When U=1 and the MOD field 2342 contains 11 (signifying a no memory access operation), part of the beta field 2254 (EVEX byte 3, bit [4]- S0) is interpreted as the RL field 2257A; when it contains a 1 (round 2257A.1) the rest of the beta field 2254 (EVEX byte 3, bit [6-5]- S2-1) is interpreted as the round operation field 2259A, while when the RL field 2257A contains a 0 (VSIZE 2257.A2) the rest of the beta field 2254 (EVEX byte 3, bit [6-5]- S2-1) is interpreted as the vector length field 2259B (EVEX byte 3, bit [6-5]- L1-0). When U=1 and the MOD field 2342 contains 00, 01, or 10 (signifying a memory access operation), the beta field 2254 (EVEX byte 3, bits [6:4]- SSS) is interpreted as the vector length field 2259B (EVEX byte 3, bit [6-5]- L1-0) and the broadcast field 2257B (EVEX byte 3, bit [4]- B).

Figure 12 is a block diagram of a register architecture 2400 according to one embodiment of the invention. In the embodiment illustrated, there are 32 vector registers 2410 that are 512 bits wide; these registers are referenced as zmm0 through zmm31. The lower order 256 bits of the lower 16 zmm registers are overlaid on registers ymm0-16. The lower order 128 bits of the lower 16 zmm registers (the lower order 128 bits of the ymm registers) are overlaid on registers xmm0-15. The specific vector friendly instruction format 2300 operates on these overlaid register file as illustrated in the below tables.

Adjustable Vector Length	Class	Operations	Registers
Instruction Templates that do not include the vector length field 2259B	A (Figure 10A; U=0)	2210, 2215, 2225, 2230	zmm registers (the vector length is 64 byte)
	B (Figure 10B; U=1)	2212	zmm registers (the vector length is 64 byte)
Instruction Templates that do include the vector length field 2259B	B (Figure 10B; U=1)	2217, 2227	zmm, ymm, or xmm registers (the vector length is 64 byte, 32 byte, or 16 byte) depending on the vector length field 2259B

In other words, the vector length field 2259B selects between a maximum length and one or more other shorter lengths, where each such shorter length is half the length of the preceding

length; and instructions templates without the vector length field 2259B operate on the maximum vector length. Further, in one embodiment, the class B instruction templates of the specific vector friendly instruction format 2300 operate on packed or scalar single/double-precision floating point data and packed or scalar integer data. Scalar operations are operations performed on the lowest order data element position in an zmm/ymm/xmm register; the higher order data element positions are either left the same as they were prior to the instruction or zeroed depending on the embodiment.

Write mask registers 2415 - in the embodiment illustrated, there are 8 write mask registers (k0 through k7), each 64 bits in size. In an alternate embodiment, the write mask registers 2415 are 16 bits in size. As previously described, in one embodiment of the invention, the vector mask register k0 cannot be used as a write mask; when the encoding that would normally indicate k0 is used for a write mask, it selects a hardwired write mask of 0xFFFF, effectively disabling write masking for that instruction.

General-purpose registers 2425 - in the embodiment illustrated, there are sixteen 64-bit general-purpose registers that are used along with the existing x86 addressing modes to address memory operands. These registers are referenced by the names RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, and R8 through R15.

Scalar floating point stack register file (x87 stack) 2445, on which is aliased the MMX packed integer flat register file 2450 - in the embodiment illustrated, the x87 stack is an eight-element stack used to perform scalar floating-point operations on 32/64/80-bit floating point data using the x87 instruction set extension; while the MMX registers are used to perform operations on 64-bit packed integer data, as well as to hold operands for some operations performed between the MMX and XMM registers.

Alternative embodiments of the invention may use wider or narrower registers. Additionally, alternative embodiments of the invention may use more, less, or different register files and registers.

Processor cores may be implemented in different ways, for different purposes, and in different processors. For instance, implementations of such cores may include: 1) a general purpose in-order core intended for general-purpose computing; 2) a high performance general purpose out-of-order core intended for general-purpose computing; 3) a special purpose core intended primarily for graphics and/or scientific (throughput) computing. Implementations of different processors may include: 1) a CPU including one or more general purpose in-order cores intended for general-purpose computing and/or one or more general purpose out-of-order cores intended for general-purpose computing; and 2) a coprocessor including one or more special

purpose cores intended primarily for graphics and/or scientific (throughput). Such different processors lead to different computer system architectures, which may include: 1) the coprocessor on a separate chip from the CPU; 2) the coprocessor on a separate die in the same package as a CPU; 3) the coprocessor on the same die as a CPU (in which case, such a

5 coprocessor is sometimes referred to as special purpose logic, such as integrated graphics and/or scientific (throughput) logic, or as special purpose cores); and 4) a system on a chip that may include on the same die the described CPU (sometimes referred to as the application core(s) or application processor(s)), the above described coprocessor, and additional functionality.

Exemplary core architectures are described next, followed by descriptions of exemplary
10 processors and computer architectures.

Figure 13A is a block diagram illustrating both an exemplary in-order pipeline and an exemplary register renaming, out-of-order issue/execution pipeline according to embodiments of the invention. Figure 13B is a block diagram illustrating both an exemplary embodiment of an in-order architecture core and an exemplary register renaming, out-of-order issue/execution
15 architecture core to be included in a processor according to embodiments of the invention. The solid lined boxes illustrate the in-order pipeline and in-order core, while the optional addition of the dashed lined boxes illustrates the register renaming, out-of-order issue/execution pipeline and core. Given that the in-order aspect is a subset of the out-of-order aspect, the out-of-order aspect will be described.

20 In Figure 13A, a processor pipeline 2500 includes a fetch stage 2502, a length decode stage 2504, a decode stage 2506, an allocation stage 2508, a renaming stage 2510, a scheduling (also known as a dispatch or issue) stage 2512, a register read/memory read stage 2514, an execute stage 2516, a write back/memory write stage 2518, an exception handling stage 2522, and a commit stage 2524.

25 Figure 13B shows processor core 2590 including a front end unit 2530 coupled to an execution engine unit 2550, and both are coupled to a memory unit 2570. The core 2590 may be a reduced instruction set computing (RISC) core, a complex instruction set computing (CISC) core, a very long instruction word (VLIW) core, or a hybrid or alternative core type. As yet another option, the core 2590 may be a special-purpose core, such as, for example, a network or
30 communication core, compression engine, coprocessor core, general purpose computing graphics processing unit (GPGPU) core, graphics core, or the like.

The front end unit 2530 includes a branch prediction unit 2532 coupled to an instruction cache unit 2534, which is coupled to an instruction translation lookaside buffer (TLB) 2536, which is coupled to an instruction fetch unit 2538, which is coupled to a decode unit 2540. The

decode unit 2540 (or decoder) may decode instructions, and generate as an output one or more micro-operations, micro-code entry points, microinstructions, other instructions, or other control signals, which are decoded from, or which otherwise reflect, or are derived from, the original instructions. The decode unit 2540 may be implemented using various different mechanisms.

5 Examples of suitable mechanisms include, but are not limited to, look-up tables, hardware implementations, programmable logic arrays (PLAs), microcode read only memories (ROMs), etc. In one embodiment, the core 2590 includes a microcode ROM or other medium that stores microcode for certain macroinstructions (e.g., in decode unit 2540 or otherwise within the front end unit 2530). The decode unit 2540 is coupled to a rename/allocator unit 2552 in the
10 execution engine unit 2550.

The execution engine unit 2550 includes the rename/allocator unit 2552 coupled to a retirement unit 2554 and a set of one or more scheduler unit(s) 2556. The scheduler unit(s) 2556 represents any number of different schedulers, including reservations stations, central instruction window, etc. The scheduler unit(s) 2556 is coupled to the physical register file(s) unit(s) 2558.

15 Each of the physical register file(s) units 2558 represents one or more physical register files, different ones of which store one or more different data types, such as scalar integer, scalar floating point, packed integer, packed floating point, vector integer, vector floating point, status (e.g., an instruction pointer that is the address of the next instruction to be executed), etc.

In one embodiment, the physical register file(s) unit 2558 comprises a vector registers
20 unit, a write mask registers unit, and a scalar registers unit. These register units may provide architectural vector registers, vector mask registers, and general purpose registers. The physical register file(s) unit(s) 2558 is overlapped by the retirement unit 2554 to illustrate various ways in which register renaming and out-of-order execution may be implemented (e.g., using a reorder buffer(s) and a retirement register file(s); using a future file(s), a history buffer(s), and a
25 retirement register file(s); using a register maps and a pool of registers; etc.). The retirement unit 2554 and the physical register file(s) unit(s) 2558 are coupled to the execution cluster(s) 2560.

The execution cluster(s) 2560 includes a set of one or more execution units 2562 and a set of one or more memory access units 2564. The execution units 2562 may perform various operations (e.g., shifts, addition, subtraction, multiplication) and on various types of data (e.g.,
30 scalar floating point, packed integer, packed floating point, vector integer, vector floating point). While some embodiments may include a number of execution units dedicated to specific functions or sets of functions, other embodiments may include only one execution unit or multiple execution units that all perform all functions.

The scheduler unit(s) 2556, physical register file(s) unit(s) 2558, and execution cluster(s) 2560 are shown as being possibly plural because certain embodiments create separate pipelines for certain types of data/operations (e.g., a scalar integer pipeline, a scalar floating point/packed integer/packed floating point/vector integer/vector floating point pipeline, and/or a memory access pipeline that each have their own scheduler unit, physical register file(s) unit, and/or execution cluster – and in the case of a separate memory access pipeline, certain embodiments are implemented in which only the execution cluster of this pipeline has the memory access unit(s) 2564). It should also be understood that where separate pipelines are used, one or more of these pipelines may be out-of-order issue/execution and the rest in-order.

The set of memory access units 2564 is coupled to the memory unit 2570, which includes a data TLB unit 2572 coupled to a data cache unit 2574 coupled to a level 2 (L2) cache unit 2576. In one exemplary embodiment, the memory access units 2564 may include a load unit, a store address unit, and a store data unit, each of which is coupled to the data TLB unit 2572 in the memory unit 2570. The instruction cache unit 2534 is further coupled to a level 2 (L2) cache unit 2576 in the memory unit 2570. The L2 cache unit 2576 is coupled to one or more other levels of cache and eventually to a main memory.

By way of example, the exemplary register renaming, out-of-order issue/execution core architecture may implement the pipeline 2500 as follows: 1) the instruction fetch 2538 performs the fetch and length decoding stages 2502 and 2504; 2) the decode unit 2540 performs the decode stage 2506; 3) the rename/allocator unit 2552 performs the allocation stage 2508 and renaming stage 2510; 4) the scheduler unit(s) 2556 performs the schedule stage 2512; 5) the physical register file(s) unit(s) 2558 and the memory unit 2570 perform the register read/memory read stage 2514; the execution cluster 2560 perform the execute stage 2516; 6) the memory unit 2570 and the physical register file(s) unit(s) 2558 perform the write back/memory write stage 2518; 7) various units may be involved in the exception handling stage 2522; and 8) the retirement unit 2554 and the physical register file(s) unit(s) 2558 perform the commit stage 2524.

The core 2590 may support one or more instructions sets (e.g., the x86 instruction set (with some extensions that have been added with newer versions); the MIPS instruction set of MIPS Technologies of Sunnyvale, CA; the ARM instruction set (with optional additional extensions such as NEON) of ARM Holdings of Sunnyvale, CA), including the instruction(s) described herein. In one embodiment, the core 2590 includes logic to support a packed data instruction set extension (e.g., AVX1, AVX2, and/or some form of the generic vector friendly instruction format (U=0 and/or U=1) previously described), thereby allowing the operations used by many multimedia applications to be performed using packed data.

It should be understood that the core may support multithreading (executing two or more parallel sets of operations or threads), and may do so in a variety of ways including time sliced multithreading, simultaneous multithreading (where a single physical core provides a logical core for each of the threads that physical core is simultaneously multithreading), or a combination thereof (e.g., time sliced fetching and decoding and simultaneous multithreading thereafter such as in the Intel® Hyper threading technology).

While register renaming is described in the context of out-of-order execution, it should be understood that register renaming may be used in an in-order architecture. While the illustrated embodiment of the processor also includes separate instruction and data cache units 2534/2574 and a shared L2 cache unit 2576, alternative embodiments may have a single internal cache for both instructions and data, such as, for example, a Level 1 (L1) internal cache, or multiple levels of internal cache. In some embodiments, the system may include a combination of an internal cache and an external cache that is external to the core and/or the processor. Alternatively, all of the cache may be external to the core and/or the processor.

Figure 14A and Figure 14B illustrate a block diagram of a more specific exemplary in-order core architecture, which core would be one of several logic blocks (including other cores of the same type and/or different types) in a chip. The logic blocks communicate through a high-bandwidth interconnect network (e.g., a ring network) with some fixed function logic, memory I/O interfaces, and other necessary I/O logic, depending on the application.

Figure 14A is a block diagram of a single processor core, along with its connection to the on-die interconnect network 2602 and with its local subset of the Level 2 (L2) cache 2604, according to embodiments of the invention. In one embodiment, an instruction decoder 2600 supports the x86 instruction set with a packed data instruction set extension. An L1 cache 2606 allows low-latency accesses to cache memory into the scalar and vector units. While in one embodiment (to simplify the design), a scalar unit 2608 and a vector unit 2610 use separate register sets (respectively, scalar registers 2612 and vector registers 2614) and data transferred between them is written to memory and then read back in from a level 1 (L1) cache 2606, alternative embodiments of the invention may use a different approach (e.g., use a single register set or include a communication path that allow data to be transferred between the two register files without being written and read back).

The local subset of the L2 cache 2604 is part of a global L2 cache that is divided into separate local subsets, one per processor core. Each processor core has a direct access path to its own local subset of the L2 cache 2604. Data read by a processor core is stored in its L2 cache subset 2604 and can be accessed quickly, in parallel with other processor cores accessing their

own local L2 cache subsets. Data written by a processor core is stored in its own L2 cache subset 2604 and is flushed from other subsets, if necessary. The ring network ensures coherency for shared data. The ring network is bi-directional to allow agents such as processor cores, L2 caches and other logic blocks to communicate with each other within the chip. Each ring data-path is 1012-bits wide per direction.

Figure 14B is an expanded view of part of the processor core in Figure 14A according to embodiments of the invention. Figure 14B includes an L1 data cache 2606A part of the L1 cache 2604, as well as more detail regarding the vector unit 2610 and the vector registers 2614. Specifically, the vector unit 2610 is a 16-wide vector processing unit (VPU) (see the 16-wide ALU 2628), which executes one or more of integer, single-precision float, and double-precision float instructions. The VPU supports swizzling the register inputs with swizzle unit 2620, numeric conversion with numeric convert units 2622A-B, and replication with replication unit 2624 on the memory input. Write mask registers 2626 allow predicating resulting vector writes.

Figure 15 is a block diagram of a processor 2700 that may have more than one core, may have an integrated memory controller, and may have integrated graphics according to embodiments of the invention. The solid lined boxes in Figure 15 illustrate a processor 2700 with a single core 2702A, a system agent 2710, a set of one or more bus controller units 2716, while the optional addition of the dashed lined boxes illustrates an alternative processor 2700 with multiple cores 2702A-N, a set of one or more integrated memory controller unit(s) 2714 in the system agent unit 2710, and special purpose logic 2708.

Thus, different implementations of the processor 2700 may include: 1) a CPU with the special purpose logic 2708 being integrated graphics and/or scientific (throughput) logic (which may include one or more cores), and the cores 2702A-N being one or more general purpose cores (e.g., general purpose in-order cores, general purpose out-of-order cores, a combination of the two); 2) a coprocessor with the cores 2702A-N being a large number of special purpose cores intended primarily for graphics and/or scientific (throughput); and 3) a coprocessor with the cores 2702A-N being a large number of general purpose in-order cores. Thus, the processor 2700 may be a general-purpose processor, coprocessor or special-purpose processor, such as, for example, a network or communication processor, compression engine, graphics processor, GPGPU (general purpose graphics processing unit), a high-throughput many integrated core (MIC) coprocessor (including 30 or more cores), embedded processor, or the like. The processor may be implemented on one or more chips. The processor 2700 may be a part of and/or may be implemented on one or more substrates using any of a number of process technologies, such as, for example, BiCMOS, CMOS, or NMOS.

The memory hierarchy includes one or more levels of cache within the cores, a set or one or more shared cache units 2706, and external memory (not shown) coupled to the set of integrated memory controller units 2714. The set of shared cache units 2706 may include one or more mid-level caches, such as level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, a last level cache (LLC), and/or combinations thereof. While in one embodiment a ring based interconnect unit 2712 interconnects the integrated graphics logic 2708, the set of shared cache units 2706, and the system agent unit 2710/integrated memory controller unit(s) 2714, alternative embodiments may use any number of well-known techniques for interconnecting such units. In one embodiment, coherency is maintained between one or more cache units 2706 and cores 2702-A-N.

In some embodiments, one or more of the cores 2702A-N are capable of multi-threading. The system agent 2710 includes those components coordinating and operating cores 2702A-N. The system agent unit 2710 may include for example a power control unit (PCU) and a display unit. The PCU may be or include logic and components needed for regulating the power state of the cores 2702A-N and the integrated graphics logic 2708. The display unit is for driving one or more externally connected displays.

The cores 2702A-N may be homogenous or heterogeneous in terms of architecture instruction set; that is, two or more of the cores 2702A-N may be capable of execution the same instruction set, while others may be capable of executing only a subset of that instruction set or a different instruction set.

Figure 16 to Figure 20 are block diagrams of exemplary computer architectures. Other system designs and configurations known in the arts for laptops, desktops, handheld PCs, personal digital assistants, engineering workstations, servers, network devices, network hubs, switches, embedded processors, digital signal processors (DSPs), graphics devices, video game devices, set-top boxes, micro controllers, cell phones, portable media players, hand held devices, and various other electronic devices, are also suitable. In general, a huge variety of systems or electronic devices capable of incorporating a processor and/or other execution logic as disclosed herein are generally suitable.

Referring now to Figure 16, shown is a block diagram of a system 2800 in accordance with one embodiment of the present invention. The system 2800 may include one or more processors 2810, 2815, which are coupled to a controller hub 2820. In one embodiment the controller hub 2820 includes a graphics memory controller hub (GMCH) 2890 and an Input/Output Hub (IOH) 2850 (which may be on separate chips); the GMCH 2890 includes memory and graphics controllers to which are coupled memory 2840 and a coprocessor 2845;

the IOH 2850 is couples input/output (I/O) devices 2860 to the GMCH 2890. Alternatively, one or both of the memory and graphics controllers are integrated within the processor (as described herein), the memory 2840 and the coprocessor 2845 are coupled directly to the processor 2810, and the controller hub 2820 in a single chip with the IOH 2850.

5 The optional nature of additional processors 2815 is denoted in Figure 16 with broken lines. Each processor 2810, 2815 may include one or more of the processing cores described herein and may be some version of the processor 2700.

 The memory 2840 may be, for example, dynamic random access memory (DRAM), phase change memory (PCM), or a combination of the two. For at least one embodiment, the
10 controller hub 2820 communicates with the processor(s) 2810, 2815 via a multi-drop bus, such as a front side bus (FSB), point-to-point interface such as QuickPath Interconnect (QPI), or similar connection 2895.

 In one embodiment, the coprocessor 2845 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression
15 engine, graphics processor, GPGPU, embedded processor, or the like. In one embodiment, controller hub 2820 may include an integrated graphics accelerator.

 There can be a variety of differences between the physical resources 2810, 2815 in terms of a spectrum of metrics of merit including architectural, micro architectural, thermal, power consumption characteristics, and the like.

20 In one embodiment, the processor 2810 executes instructions that control data processing operations of a general type. Embedded within the instructions may be coprocessor instructions. The processor 2810 recognizes these coprocessor instructions as being of a type that should be executed by the attached coprocessor 2845. Accordingly, the processor 2810 issues these
25 coprocessor instructions (or control signals representing coprocessor instructions) on a coprocessor bus or other interconnect, to coprocessor 2845. Coprocessor(s) 2845 accept and execute the received coprocessor instructions.

 Referring now to Figure 17, shown is a block diagram of a first more specific exemplary system 2900 in accordance with an embodiment of the present invention. As shown in Figure
30 17, multiprocessor system 2900 is a point-to-point interconnect system, and includes a first processor 2970 and a second processor 2980 coupled via a point-to-point interconnect 2950. Each of processors 2970 and 2980 may be some version of the processor 2700. In one embodiment of the invention, processors 2970 and 2980 are respectively processors 2810 and 2815, while coprocessor 2938 is coprocessor 2845. In another embodiment, processors 2970 and 2980 are respectively processor 2810 coprocessor 2845.

Processors 2970 and 2980 are shown including integrated memory controller (IMC) units 2972 and 2982, respectively. Processor 2970 also includes as part of its bus controller units point-to-point (P-P) interfaces 2976 and 2978; similarly, second processor 2980 includes P-P interfaces 2986 and 2988. Processors 2970, 2980 may exchange information via a point-to-point (P-P) interface 2950 using P-P interface circuits 2978, 2988. As shown in Figure 17, IMCs 2972 and 2982 couple the processors to respective memories, namely a memory 2932 and a memory 2934, which may be portions of main memory locally attached to the respective processors.

Processors 2970, 2980 may each exchange information with a chipset 2990 via individual P-P interfaces 2952, 2954 using point to point interface circuits 2976, 2994, 2986, 2998. Chipset 2990 may optionally exchange information with the coprocessor 2938 via a high-performance interface 2939. In one embodiment, the coprocessor 2938 is a special-purpose processor, such as, for example, a high-throughput MIC processor, a network or communication processor, compression engine, graphics processor, GPGPU, embedded processor, or the like.

A shared cache (not shown) may be included in either processor or outside of both processors, yet connected with the processors via P-P interconnect, such that either or both processors' local cache information may be stored in the shared cache if a processor is placed into a low power mode. Chipset 2990 may be coupled to a first bus 2916 via an interface 2996. In one embodiment, first bus 2916 may be a Peripheral Component Interconnect (PCI) bus, or a bus such as a PCI Express bus or another third generation I/O interconnect bus, although the scope of the present invention is not so limited.

As shown in Figure 17, various I/O devices 2914 may be coupled to first bus 2916, along with a bus bridge 2918 which couples first bus 2916 to a second bus 2920. In one embodiment, one or more additional processor(s) 2915, such as coprocessors, high-throughput MIC processors, GPGPU's, accelerators (such as, e.g., graphics accelerators or digital signal processing (DSP) units), field programmable gate arrays, or any other processor, are coupled to first bus 2916. In one embodiment, second bus 2920 may be a low pin count (LPC) bus. Various devices may be coupled to a second bus 2920 including, for example, a keyboard and/or mouse 2922, communication devices 2927 and a storage unit 2928 such as a disk drive or other mass storage device which may include instructions/code and data 2930, in one embodiment. Further, an audio I/O 2924 may be coupled to the second bus 2920. Note that other architectures are possible. For example, instead of the point-to-point architecture of Figure 17, a system may implement a multi-drop bus or other such architecture.

Referring now to Figure 18, shown is a block diagram of a second more specific exemplary system 3000 in accordance with an embodiment of the present invention. Like

elements in Figure 18 and Figure 19 bear like reference numerals, and certain aspects of Figure 17 have been omitted from Figure 18 in order to avoid obscuring other aspects of Figure 18. Figure 18 illustrates that the processors 2970, 2980 may include integrated memory and I/O control logic ("CL") 2972 and 2982, respectively. Thus, the CL 2972, 2982 include integrated memory controller units and include I/O control logic. Figure 18 illustrates that not only are the memories 2932, 2934 coupled to the CL 2972, 2982, but also that I/O devices 3014 are also coupled to the control logic 2972, 2982. Legacy I/O devices 3015 are coupled to the chipset 2990.

Referring now to Figure 19, shown is a block diagram of a SoC 3100 in accordance with an embodiment of the present invention. Similar elements in Figure 15 bear like reference numerals. Also, dashed lined boxes are optional features on more advanced SoCs. In Figure 19, an interconnect unit(s) 3102 is coupled to: an application processor 3110 which includes a set of one or more cores 202A-N and shared cache unit(s) 2706; a system agent unit 2710; a bus controller unit(s) 2716; an integrated memory controller unit(s) 2714; a set or one or more coprocessors 3120 which may include integrated graphics logic, an image processor, an audio processor, and a video processor; an static random access memory (SRAM) unit 3130; a direct memory access (DMA) unit 3132; and a display unit 3140 for coupling to one or more external displays. In one embodiment, the coprocessor(s) 3120 include a special-purpose processor, such as, for example, a network or communication processor, compression engine, GPGPU, a high-throughput MIC processor, embedded processor, or the like.

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

Program code, such as code 2930 illustrated in Figure 17, may be applied to input instructions to perform the functions described herein and generate output information. The output information may be applied to one or more output devices, in known fashion. For purposes of this application, a processing system includes any system that has a processor, such as, for example; a digital signal processor (DSP), a microcontroller, an application specific integrated circuit (ASIC), or a microprocessor.

The program code may be implemented in a high level procedural or object oriented programming language to communicate with a processing system. The program code may also

be implemented in assembly or machine language, if desired. In fact, the mechanisms described herein are not limited in scope to any particular programming language. In any case, the language may be a compiled or interpreted language.

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

Such machine-readable storage media may include, without limitation, non-transitory, tangible arrangements of articles manufactured or formed by a machine or device, including storage media such as hard disks, any other type of disk including floppy disks, optical disks, compact disk read-only memories (CD-ROMs), compact disk rewritable’s (CD-RWs), and magneto-optical disks, semiconductor devices such as read-only memories (ROMs), random access memories (RAMs) such as dynamic random access memories (DRAMs), static random access memories (SRAMs), erasable programmable read-only memories (EPROMs), flash memories, electrically erasable programmable read-only memories (EEPROMs), phase change memory (PCM), magnetic or optical cards, or any other type of media suitable for storing electronic instructions.

Accordingly, embodiments of the invention also include non-transitory, tangible machine-readable media containing instructions or containing design data, such as Hardware Description Language (HDL), which defines structures, circuits, apparatuses, processors and/or system features described herein. Such embodiments may also be referred to as program products.

In some cases, an instruction converter may be used to convert an instruction from a source instruction set to a target instruction set. For example, the instruction converter may translate (e.g., using static binary translation, dynamic binary translation including dynamic compilation), morph, emulate, or otherwise convert an instruction to one or more other instructions to be processed by the core. The instruction converter may be implemented in software, hardware, firmware, or a combination thereof. The instruction converter may be on processor, off processor, or part on and part off processor.

Figure 20 is a block diagram contrasting the use of a software instruction converter to convert binary instructions in a source instruction set to binary instructions in a target instruction set according to embodiments of the invention. In the illustrated embodiment, the instruction

converter is a software instruction converter, although alternatively the instruction converter may be implemented in software, firmware, hardware, or various combinations thereof. Figure 20 shows a program in a high level language 3202 may be compiled using an x86 compiler 3204 to generate x86 binary code 3206 that may be natively executed by a processor with at least one x86 instruction set core 3216. The processor with at least one x86 instruction set core 3216 represents any processor that can perform substantially the same functions as an Intel processor with at least one x86 instruction set core by compatibly executing or otherwise processing (1) a substantial portion of the instruction set of the Intel x86 instruction set core or (2) object code versions of applications or other software targeted to run on an Intel processor with at least one x86 instruction set core, in order to achieve substantially the same result as an Intel processor with at least one x86 instruction set core. The x86 compiler 3204 represents a compiler that is operable to generate x86 binary code 3206 (e.g., object code) that can, with or without additional linkage processing, be executed on the processor with at least one x86 instruction set core 3216. Similarly, Figure 20 shows the program in the high level language 3202 may be compiled using an alternative instruction set compiler 3208 to generate alternative instruction set binary code 3210 that may be natively executed by a processor without at least one x86 instruction set core 3214 (e.g., a processor with cores that execute the MIPS instruction set of MIPS Technologies of Sunnyvale, CA and/or that execute the ARM instruction set of ARM Holdings of Sunnyvale, CA). The instruction converter 3212 is used to convert the x86 binary code 3206 into code that may be natively executed by the processor without an x86 instruction set core 3214. This converted code is not likely to be the same as the alternative instruction set binary code 3210 because an instruction converter capable of this is difficult to make; however, the converted code will accomplish the general operation and be made up of instructions from the alternative instruction set. Thus, the instruction converter 3212 represents software, firmware, hardware, or a combination thereof that, through emulation, simulation or any other process, allows a processor or other electronic device that does not have an x86 instruction set processor or core to execute the x86 binary code 3206.

According to one embodiment, a processor includes an execution pipeline for executing a plurality of threads, including a first thread and a second thread. The processor further includes a multi-thread controller (MTC) coupled to the execution pipeline to determine whether to switch threads between the first and second thread based on a thread switch policy that is selected from a list of thread switch policies based on unfairness levels of the first and second thread, and in response to determining to switch threads, to switch from executing the first thread to executing the second thread. According to one embodiment, the unfairness levels are determined based on

a first counter and a second counter associated with the first and second thread, respectively. In one embodiment, the first counter contains a number of cycles that the first thread has been denied access to the execution pipeline, and the second counter contains a number of cycles that the second thread has been denied access to the execution pipeline. In one embodiment, the first counter is incremented by a first predetermined value for each cycle that a first thread is denied access to the execution pipeline, and the second counter is incremented by a second predetermined value for each cycle that the second thread is denied access to the execution pipeline. According to one aspect of the invention, the first counter is decremented when a number of cycles that the first thread has been granted access to the execution pipeline matches a third predetermined threshold, and wherein the second counter is decremented when a number of cycles that the second thread has been granted access to the execution pipeline matches a fourth predetermined threshold. In one embodiment, switching between the first and second thread is determined based on a priority level and execution status of the first and second thread, respectively. In one embodiment, the priority of a thread is set via an instruction issued from a software program associated with the thread.

Some portions of the preceding detailed descriptions have been presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are the ways used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities.

It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the above discussion, it is appreciated that throughout the description, discussions utilizing terms such as those set forth in the claims below, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

The techniques shown in the figures can be implemented using code and data stored and executed on one or more electronic devices. Such electronic devices store and communicate (internally and/or with other electronic devices over a network) code and data using computer-

readable media, such as non-transitory computer-readable storage media (e.g., magnetic disks; optical disks; random access memory; read only memory; flash memory devices; phase-change memory) and transitory computer-readable transmission media (e.g., electrical, optical, acoustical or other form of propagated signals – such as carrier waves, infrared signals, digital signals).

The processes or methods depicted in the preceding figures may be performed by processing logic that comprises hardware (e.g. circuitry, dedicated logic, etc.), firmware, software (e.g., embodied on a non-transitory computer readable medium), or a combination of both. Although the processes or methods are described above in terms of some sequential operations, it should be appreciated that some of the operations described may be performed in a different order. Moreover, some operations may be performed in parallel rather than sequentially.

In the foregoing specification, embodiments of the invention have been described with reference to specific exemplary embodiments thereof. It will be evident that various modifications may be made thereto without departing from the broader spirit and scope of the invention as set forth in the following claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense.

CLAIMS

What is claimed is:

1. A processor, comprising:
5 an execution pipeline for executing a plurality of threads, including a first thread and a second thread;
a multi-thread controller (MTC) coupled to the execution pipeline to determine whether to switch threads between the first and second thread based on a thread switch policy that is selected from a list of thread switch policies based on
10 unfairness levels of the first and second thread, and in response to determining to switch threads, to switch from executing the first thread to executing the second thread.
2. The processor of claim 1, wherein the unfairness levels are determined based on a first
15 counter and a second counter associated with the first and second thread, respectively.
3. The processor of claim 2, wherein the first counter contains a number of cycles that the first thread has been denied access to the execution pipeline, and the second counter contains a number of cycles that the second thread has been denied access to the
20 execution pipeline.
4. The processor of claim 2, wherein the first counter is incremented by a first predetermined value for each cycle that a first thread is denied access to the execution pipeline, and wherein the second counter is incremented by a second predetermined value
25 for each cycle that the second thread is denied access to the execution pipeline.
5. The processor of claim 2, wherein the first counter is decremented when a number of cycles that the first thread has been granted access to the execution pipeline matches a third predetermined threshold, and wherein the second counter is decremented when a
30 number of cycles that the second thread has been granted access to the execution pipeline matches a fourth predetermined threshold.

6. The processor of claim 1, wherein switching between the first and second thread is determined based on a priority level and execution status of the first and second thread, respectively.
- 5 7. The processor of claim 6, wherein the priority of a thread is set via an instruction issued from a software program associated with the thread.
8. A method, comprising:
executing a plurality of threads, including a first thread and a second thread;
10 determining whether to switch threads between the first and second thread based on a thread switch policy that is selected from a list of thread switch policies based on unfairness levels of the first and second thread; and
switching, in response to determining to switch threads, from executing the first thread to executing the second thread.
- 15 9. The method of claim 8, wherein the unfairness levels are determined based on a first counter and a second counter associated with the first and second thread, respectively.
10. The method of claim 9, wherein the first counter contains a number of cycles that the first
20 thread has been denied access to the execution pipeline, and the second counter contains a number of cycles that the second thread has been denied access to the execution pipeline.
11. The method of claim 9, wherein the first counter is incremented by a first predetermined
25 value for each cycle that a first thread is denied access to the execution pipeline, and wherein the second counter is incremented by a second predetermined value for each cycle that the second thread is denied access to the execution pipeline.
12. The method of claim 9, wherein the first counter is decremented when a number
30 of cycles that the first thread has been granted access to the execution pipeline matches a third predetermined threshold, and wherein the second counter is decremented when a number of cycles that the second thread has been granted access to the execution pipeline matches a fourth predetermined threshold.

13. The method of claim 8, wherein switching between the first and second thread is determined based on a priority level and execution status of the first and second thread, respectively.
- 5 14. The method of claim 13, wherein the priority of a thread is set via an instruction issued from a software program associated with the thread.
15. A system comprising:
an interconnect;
10 a dynamic random access memory (DRAM) coupled to the interconnect; and
a processor coupled the interconnect, including
an execution pipeline for executing a plurality of threads, including a first
thread and a second thread;
a multi-thread controller (MTC) coupled to the execution pipeline to determine
15 whether to switch threads between the first and second thread based on a thread
switch policy that is selected from a list of thread switch policies based on
unfairness levels of the first and second thread, and in response to determining to
switch threads, to switch from executing the first thread to executing the second
thread.
20
16. The system of claim 15, wherein the unfairness levels are determined based on a first counter and a second counter associated with the first and second thread, respectively.
17. The system of claim 16, wherein the first counter contains a number of cycles that
25 the first thread has been denied access to the execution pipeline, and the second counter
a number of cycles that the second thread has been denied access to the execution
pipeline.
18. The system of claim 16, wherein the first counter is incremented by a first predetermined
value for each cycle that a first thread is denied access to the execution pipeline, and
30 wherein the second counter is incremented by a second predetermined value for each
cycle that the second thread is denied access to the execution pipeline.

19. The system of claim 16, wherein the first counter is decremented when a number of cycles that the first thread has been granted access to the execution pipeline matches a third predetermined threshold, and wherein the second counter is decremented when a number of cycles that the second thread has been granted access to the execution pipeline matches a fourth predetermined threshold.
20. The system of claim 15, wherein switching between the first and second thread is determined based on a priority level and execution status of the first and second thread, respectively.
21. The system of claim 20, wherein the priority of a thread is set via an instruction issued from a software program associated with the thread.

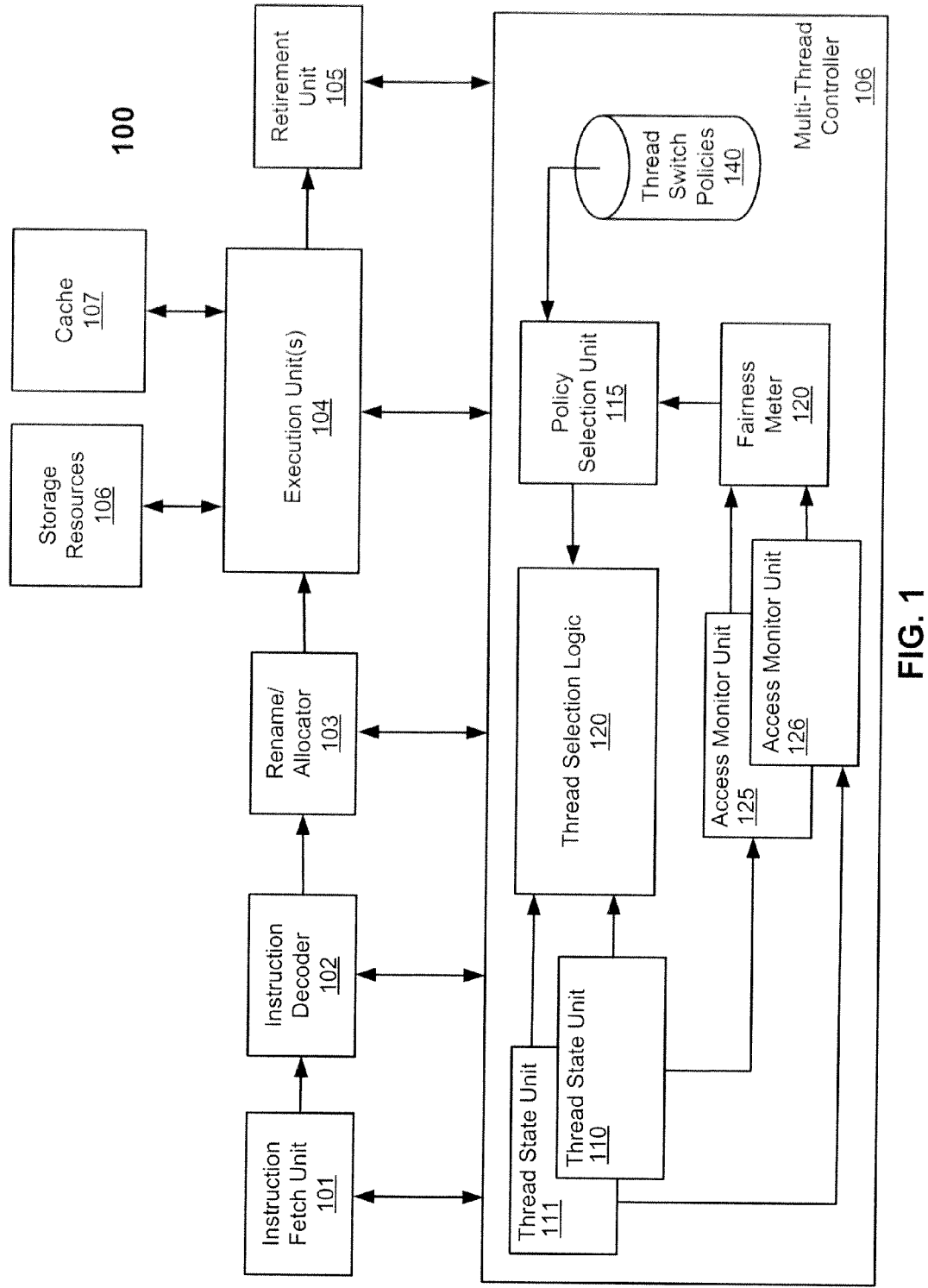


FIG. 1

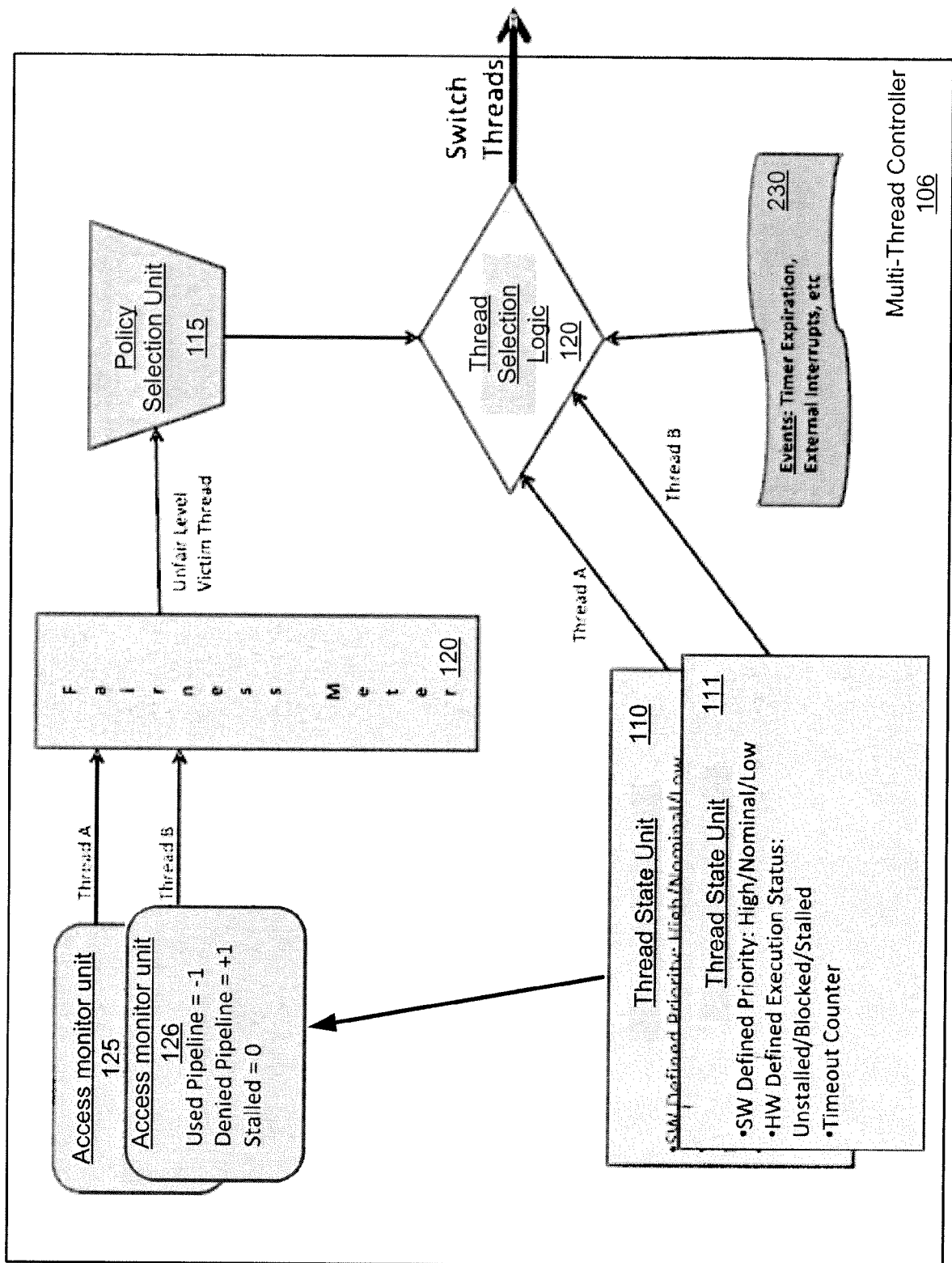


FIG. 2

300

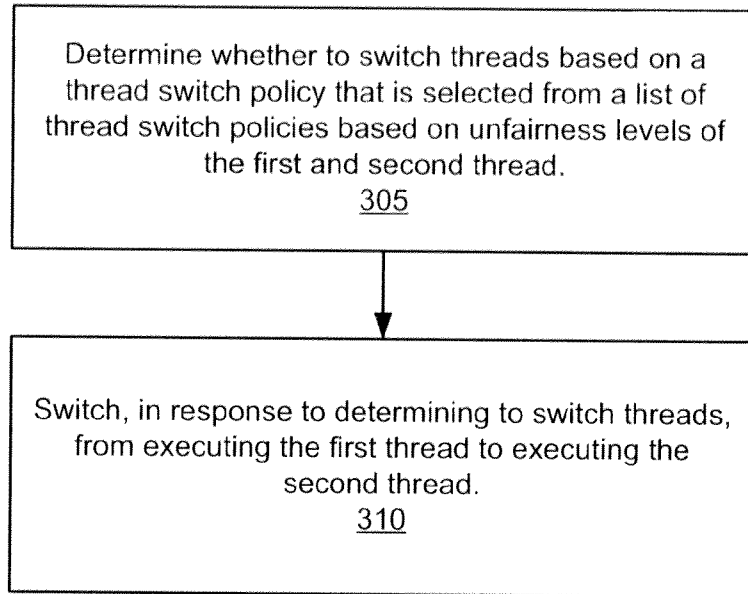


FIG. 3

400

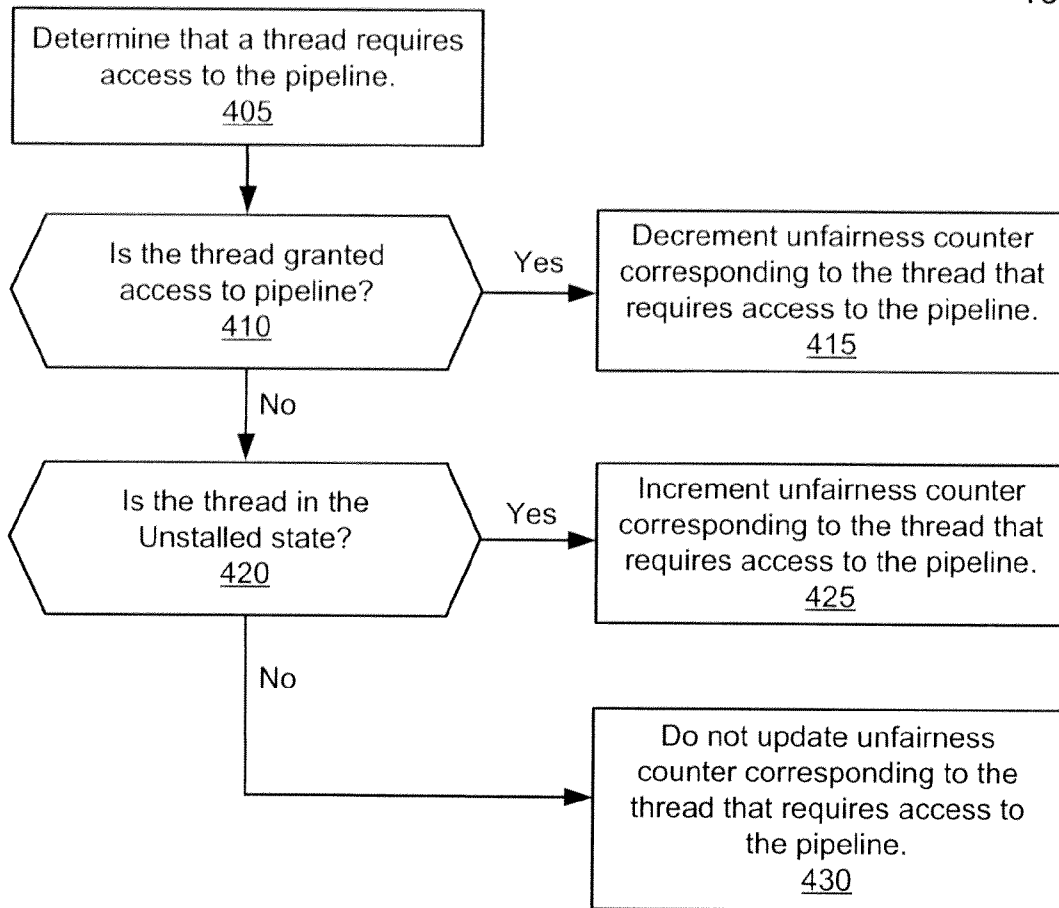


FIG. 4

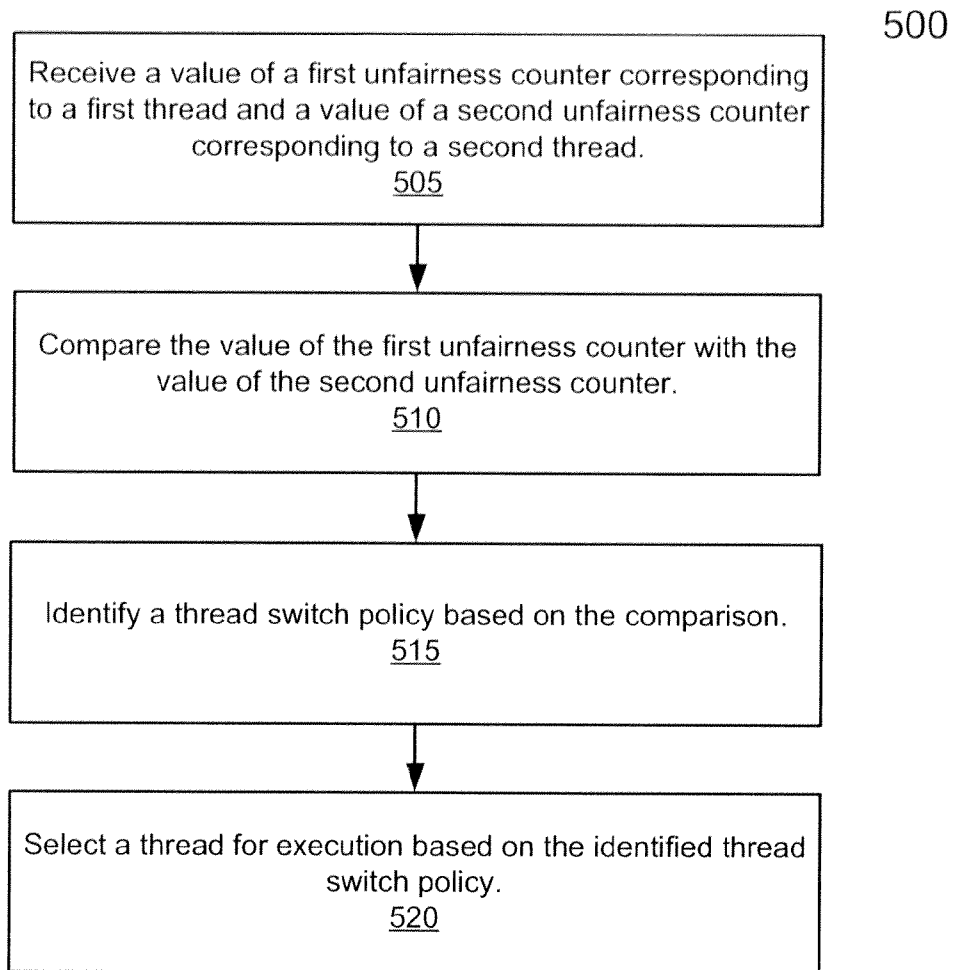


FIG. 5

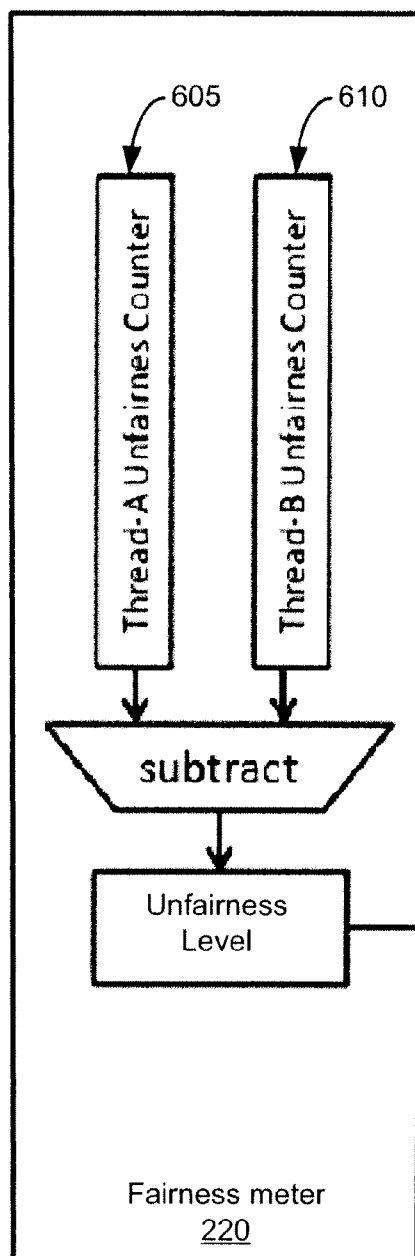


FIG. 6A

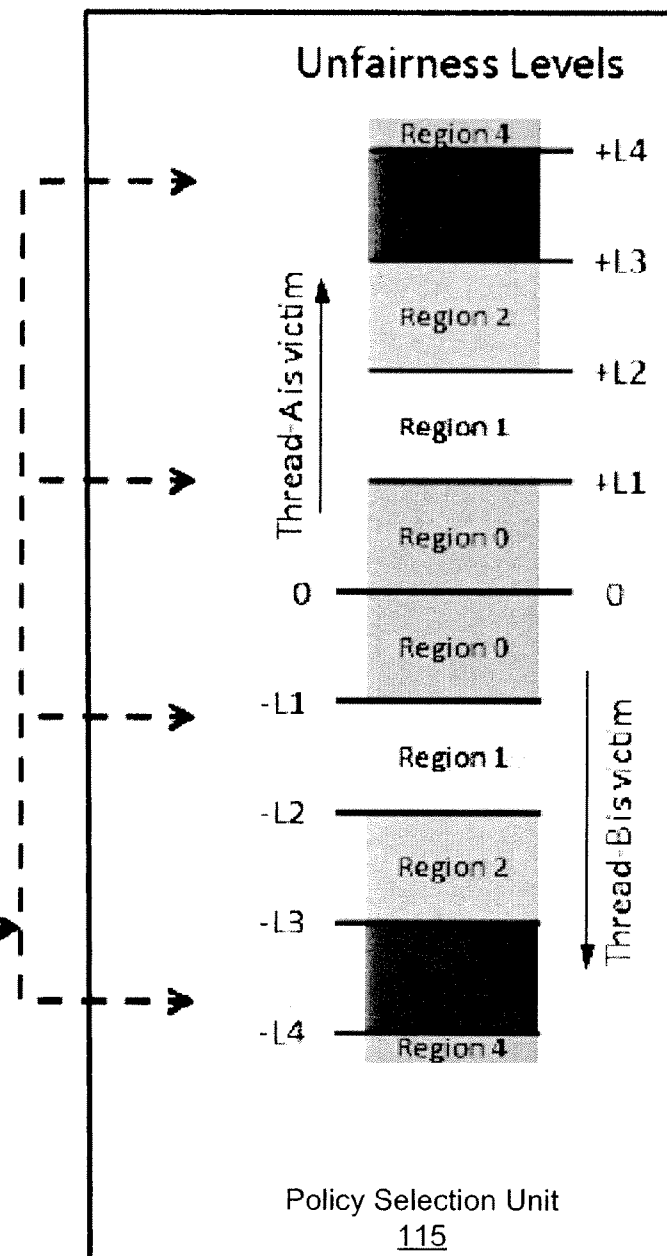


FIG. 6B

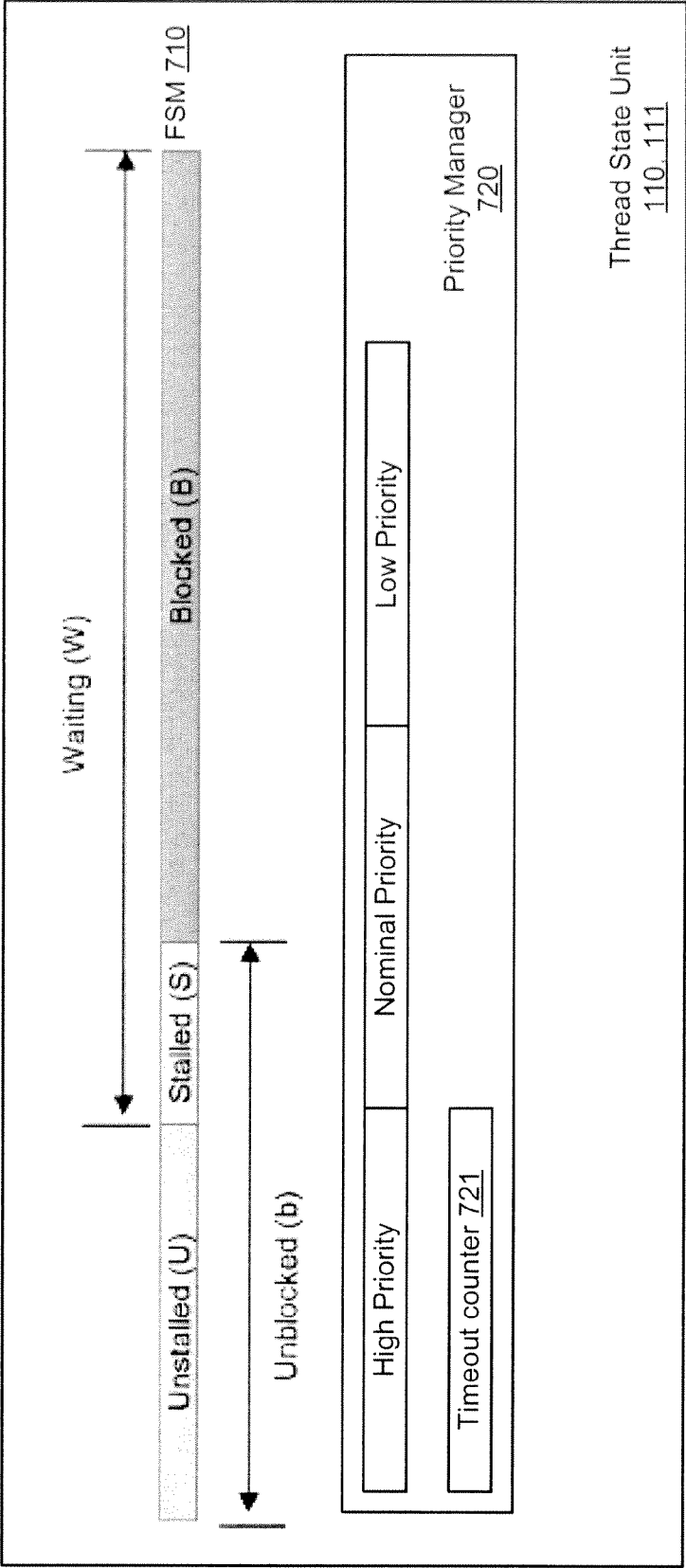


FIG. 7

8/22

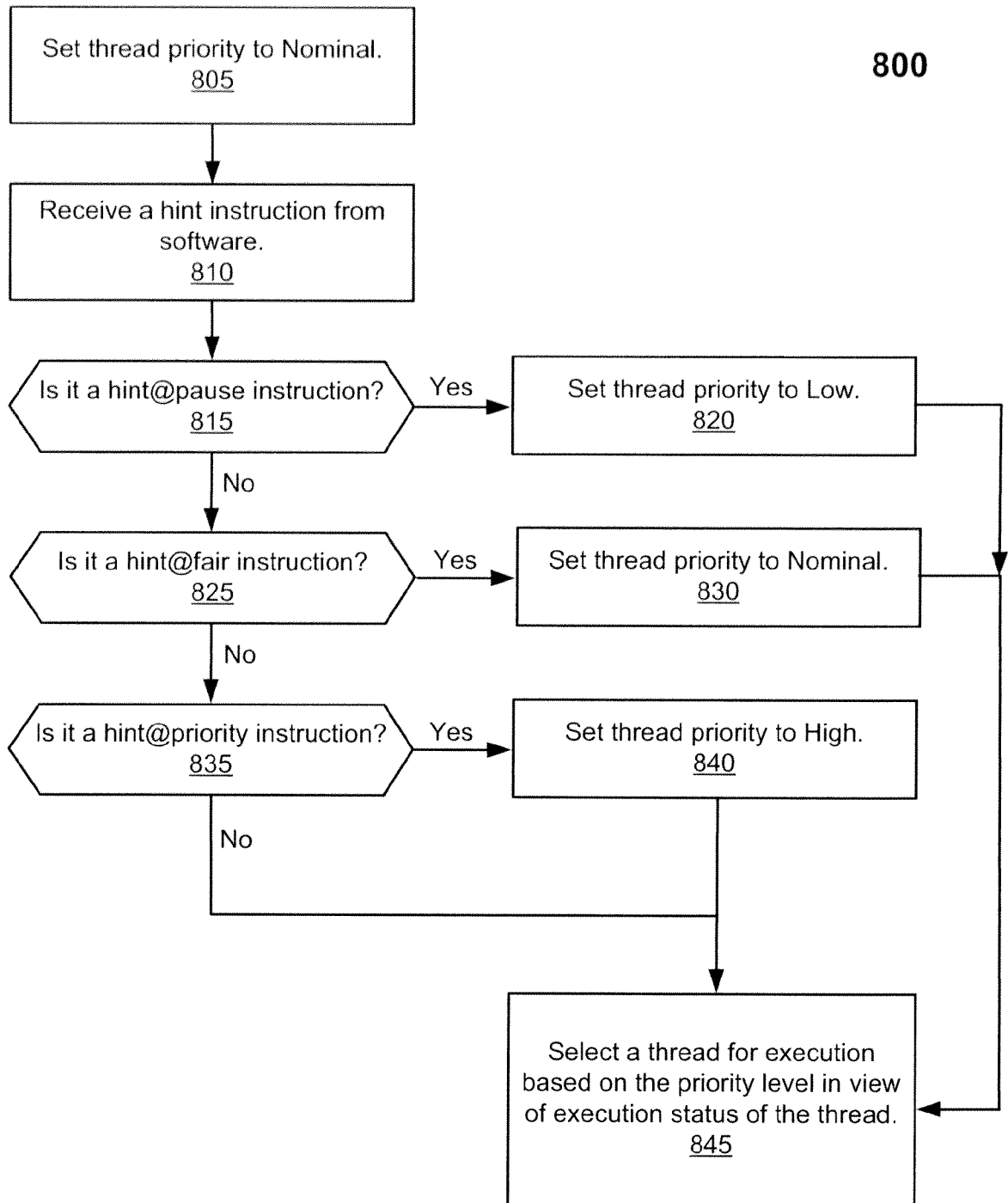


FIG. 8

FIG. 9A

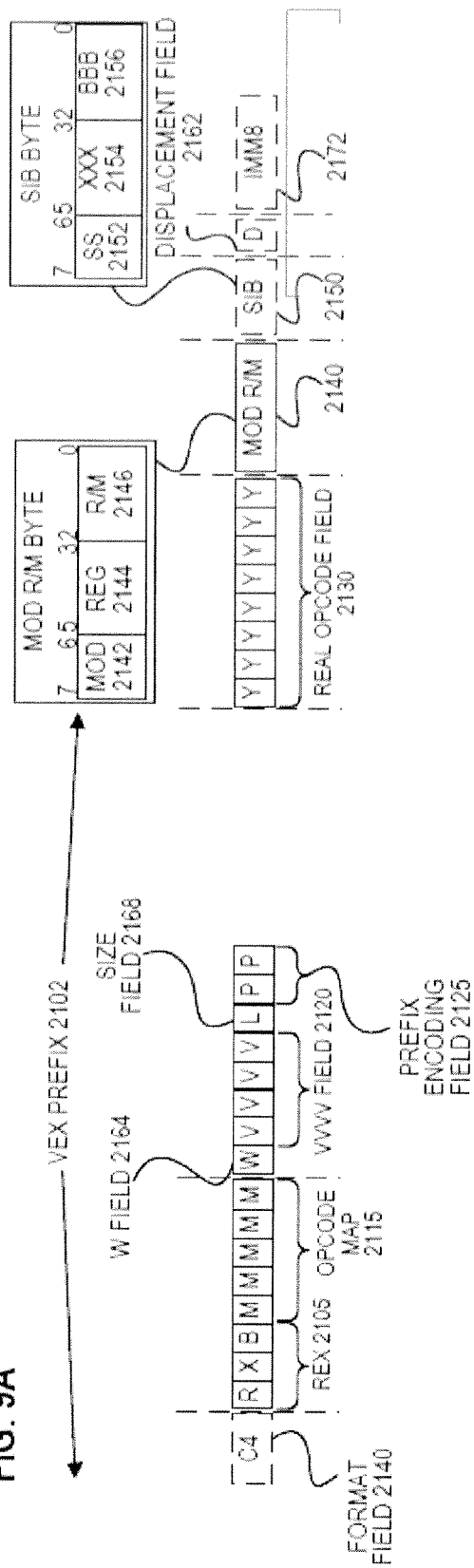


FIG. 9B

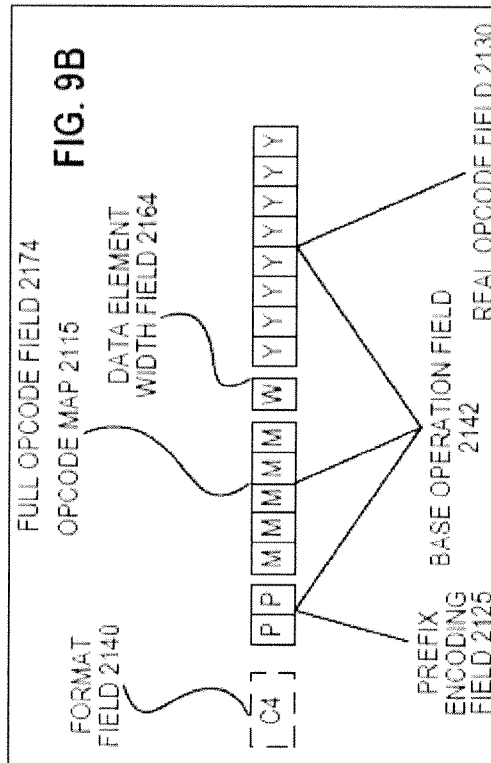
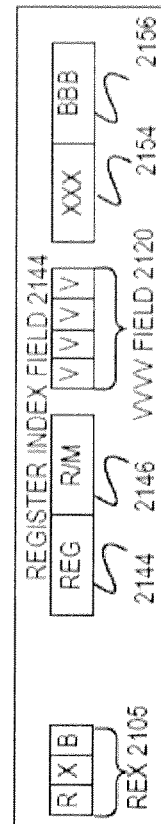
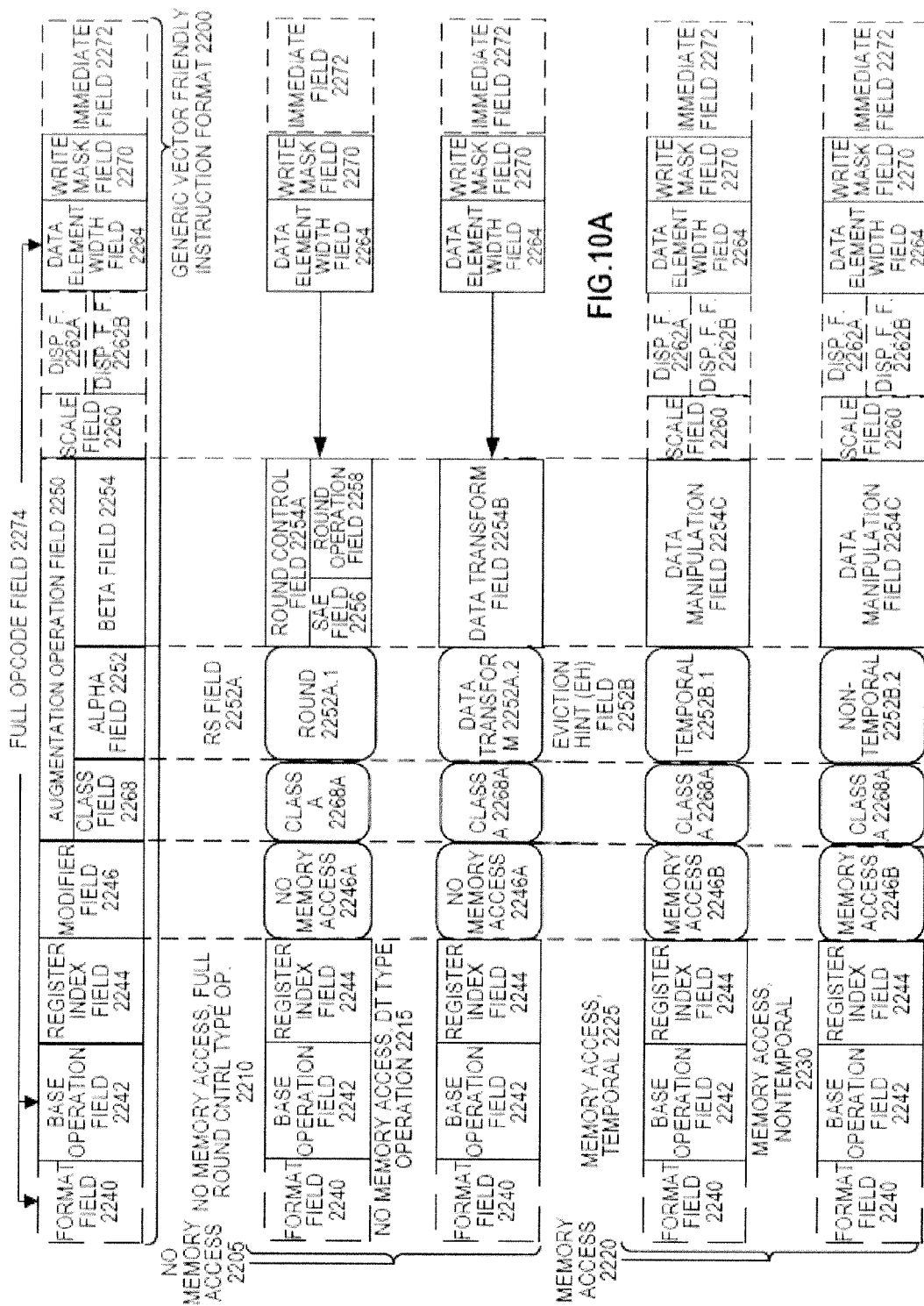


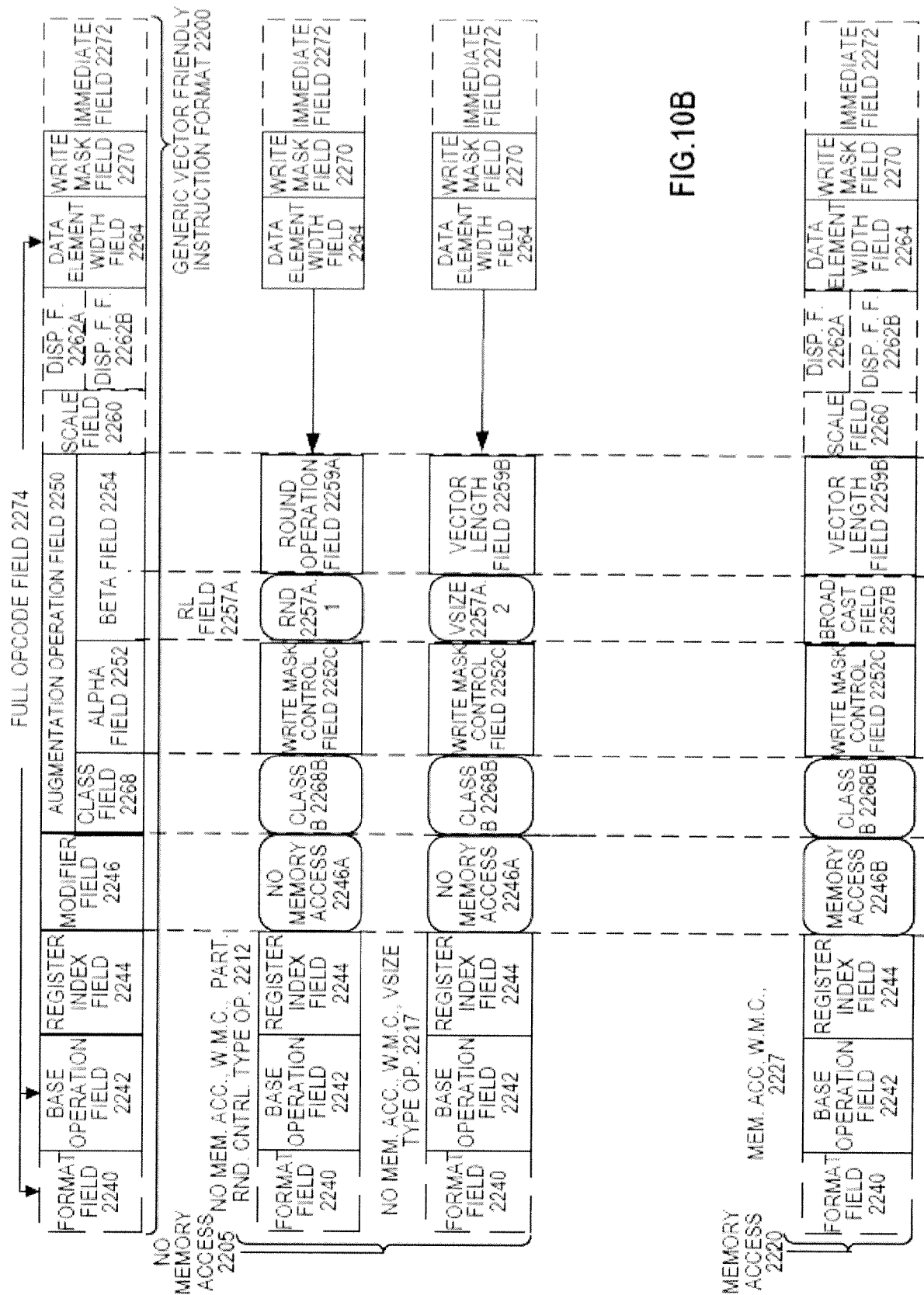
FIG. 9C



10/22



11/22



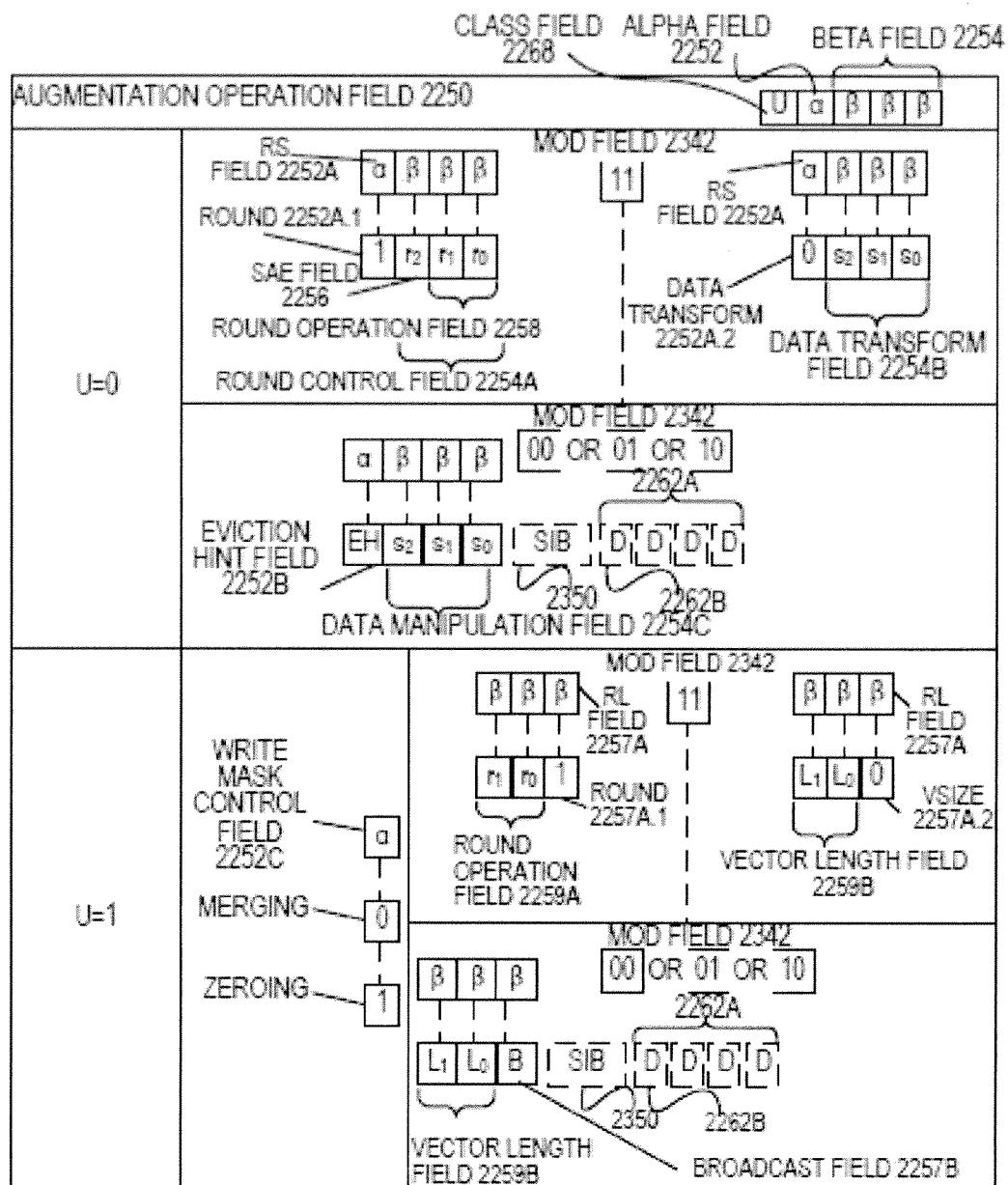
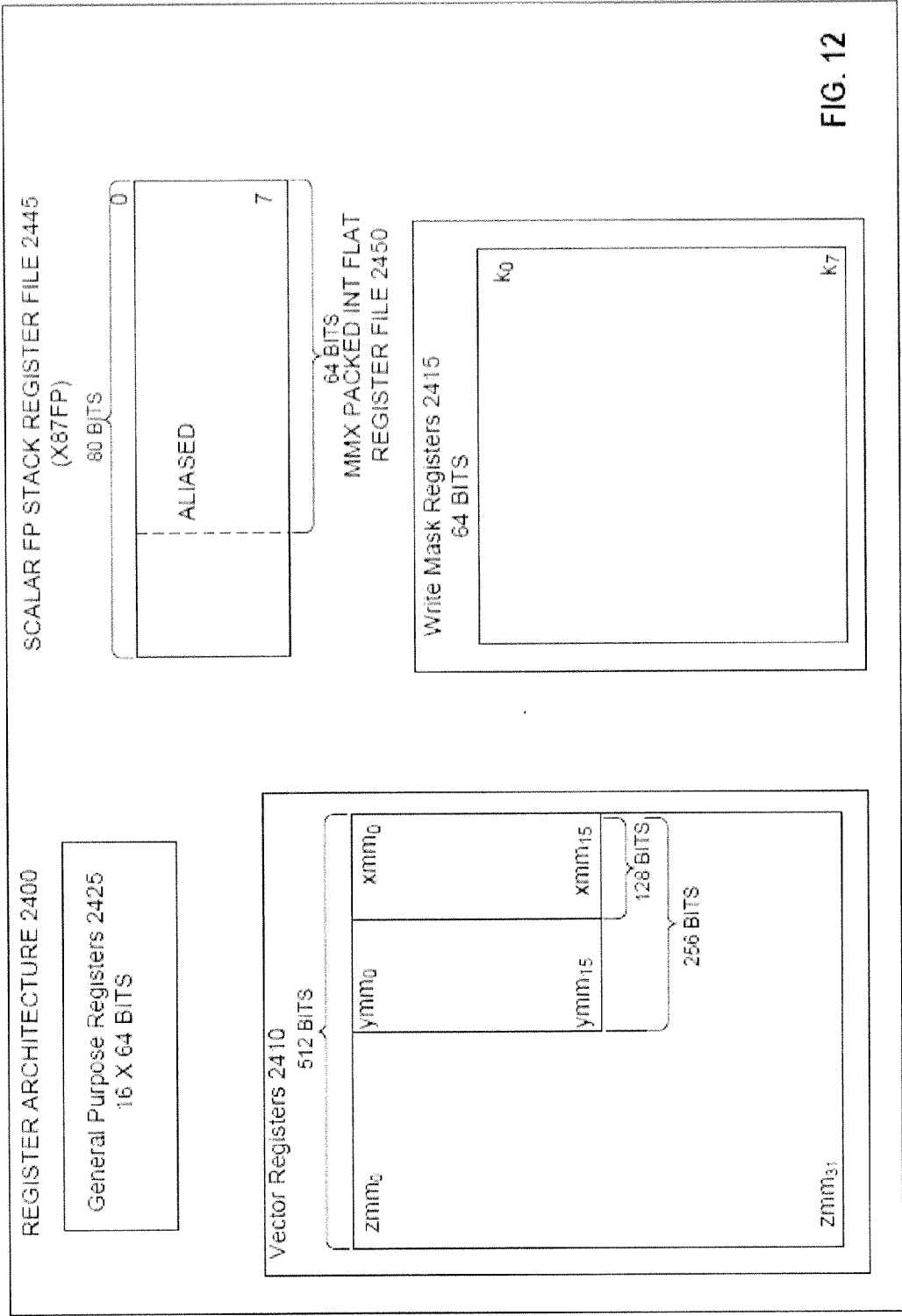
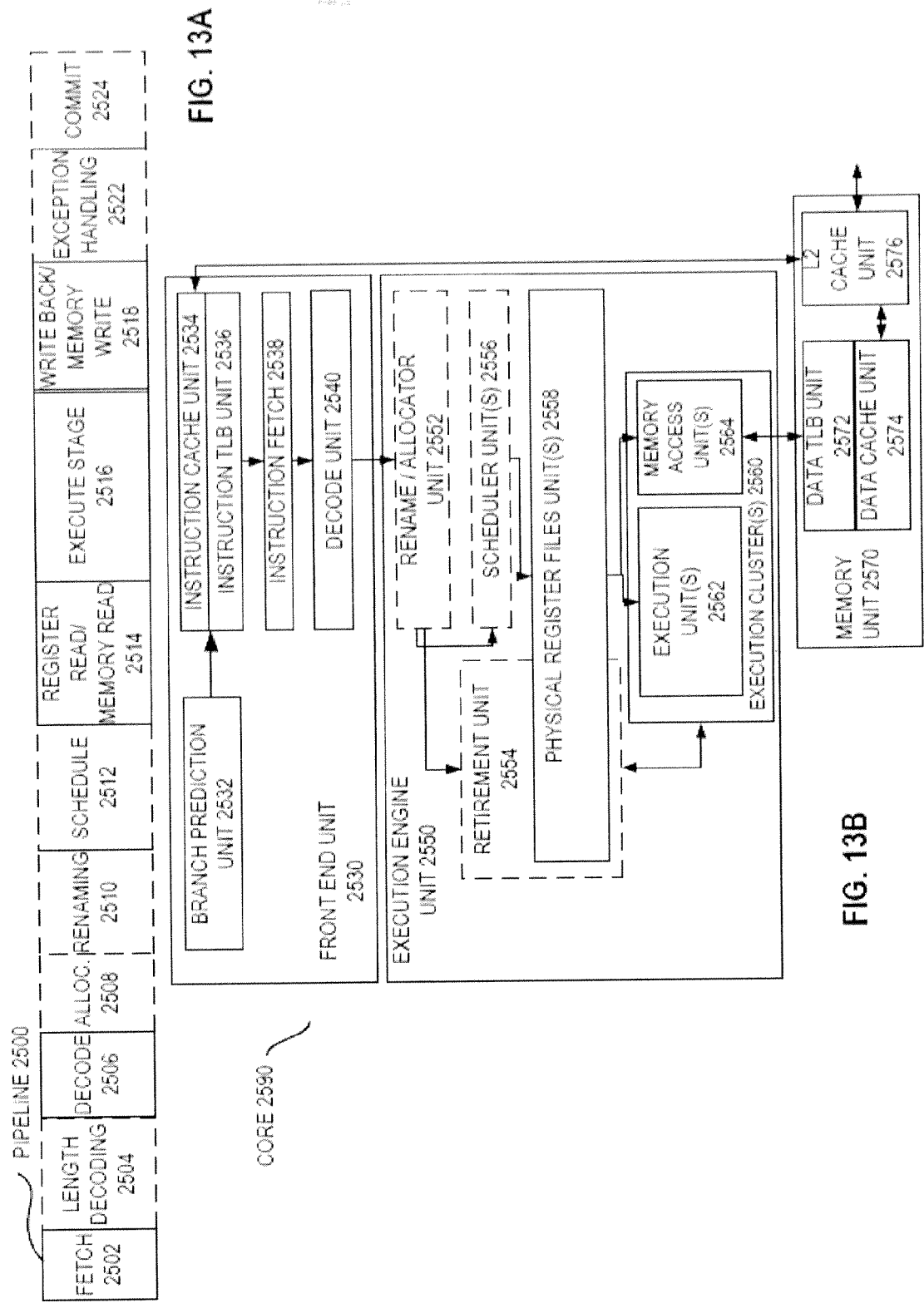


FIG. 11D





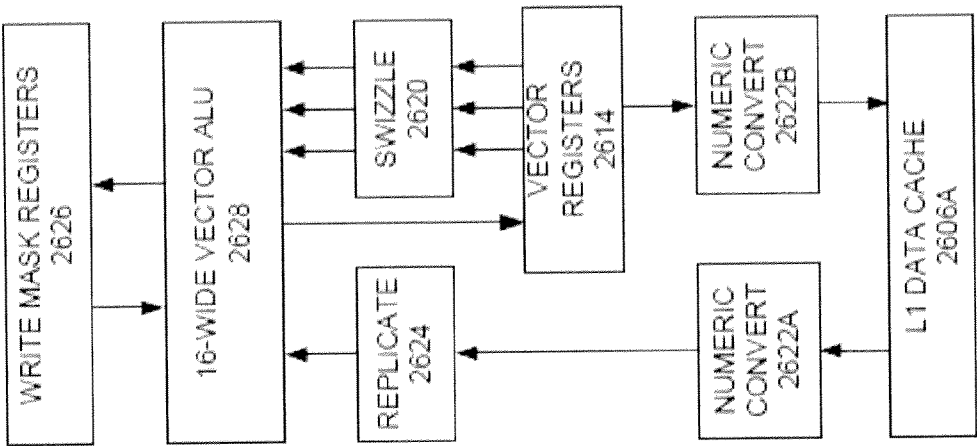


FIG. 14B

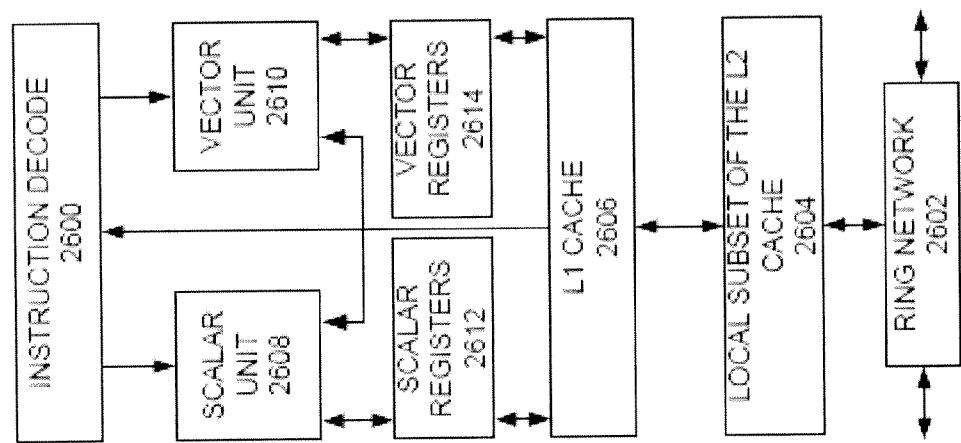


FIG. 14A

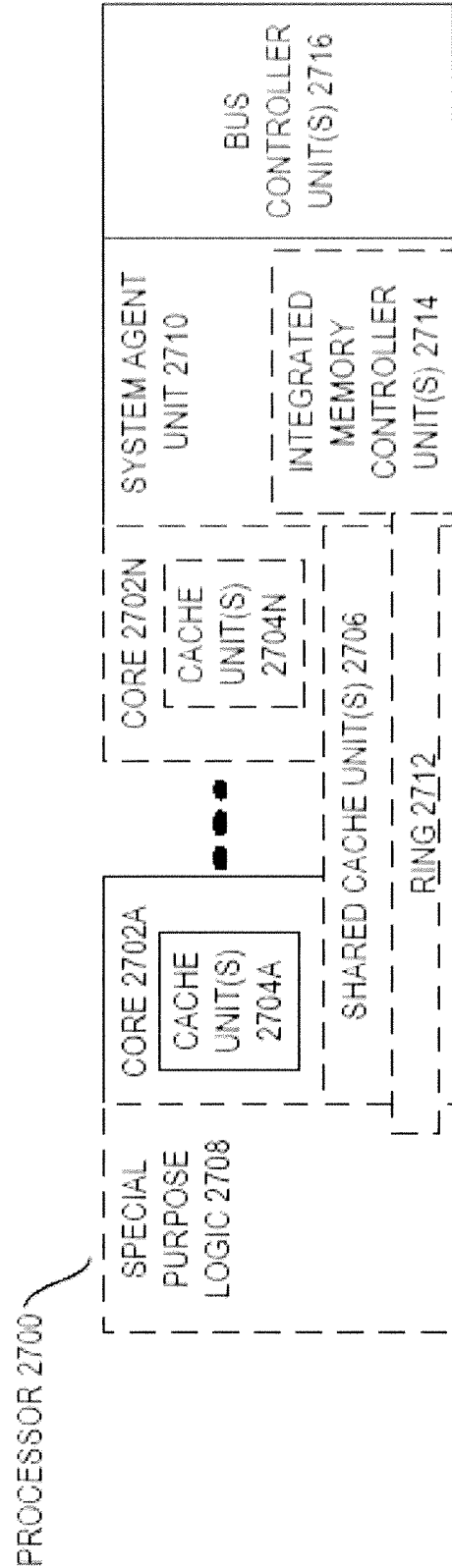


FIG. 15

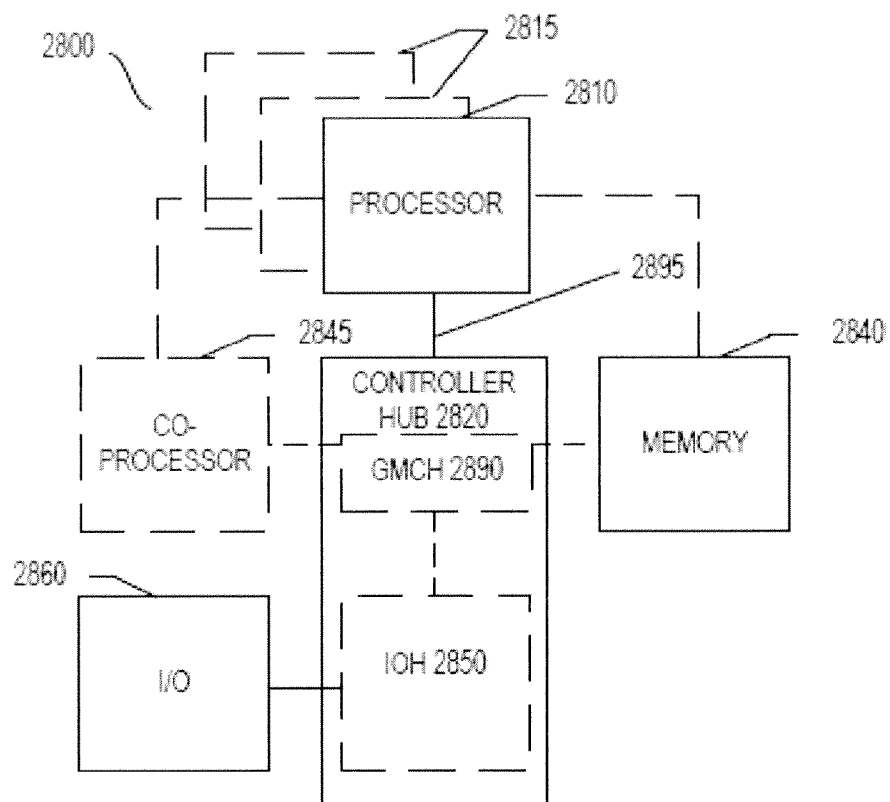


FIG. 16

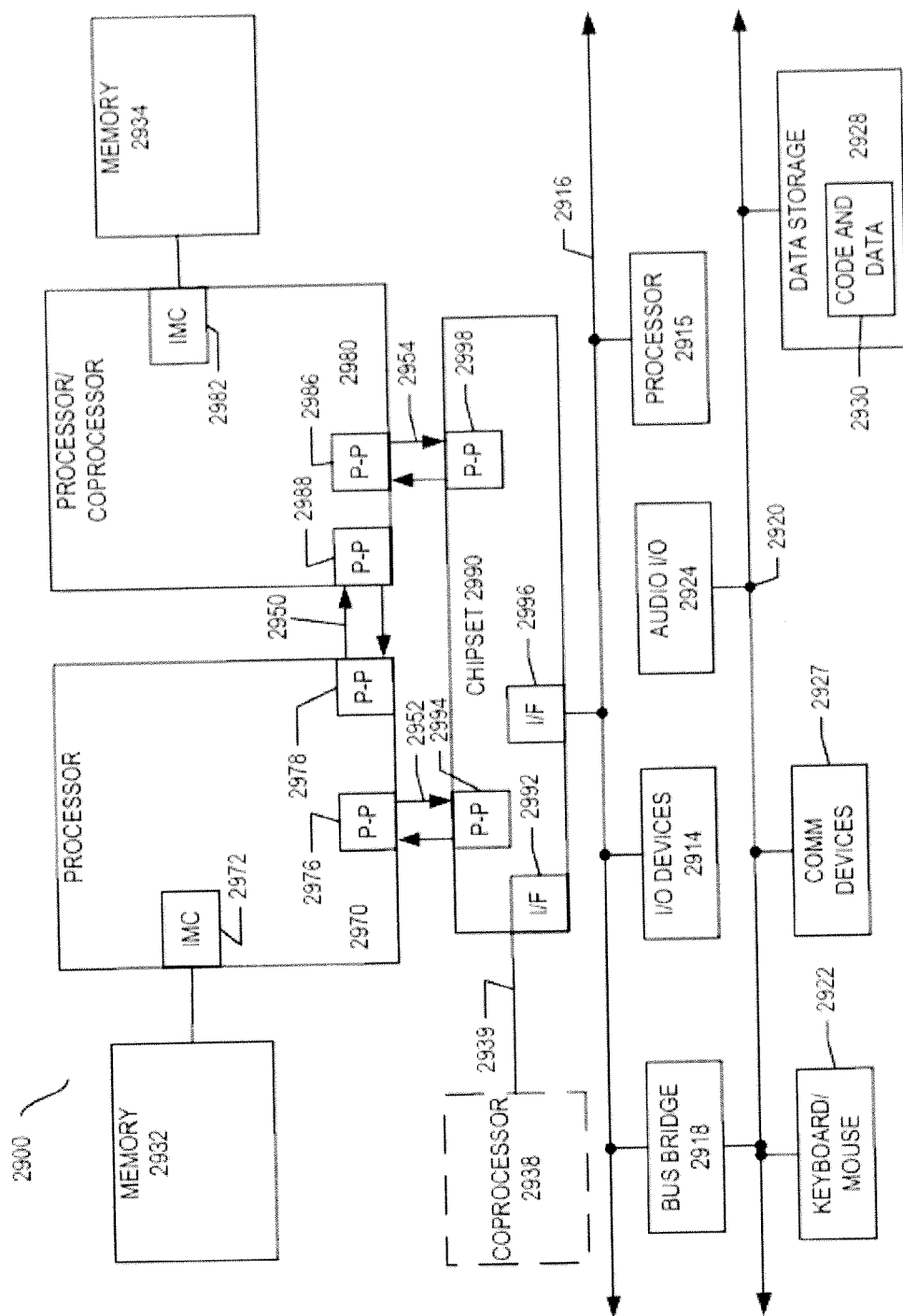


FIG. 17

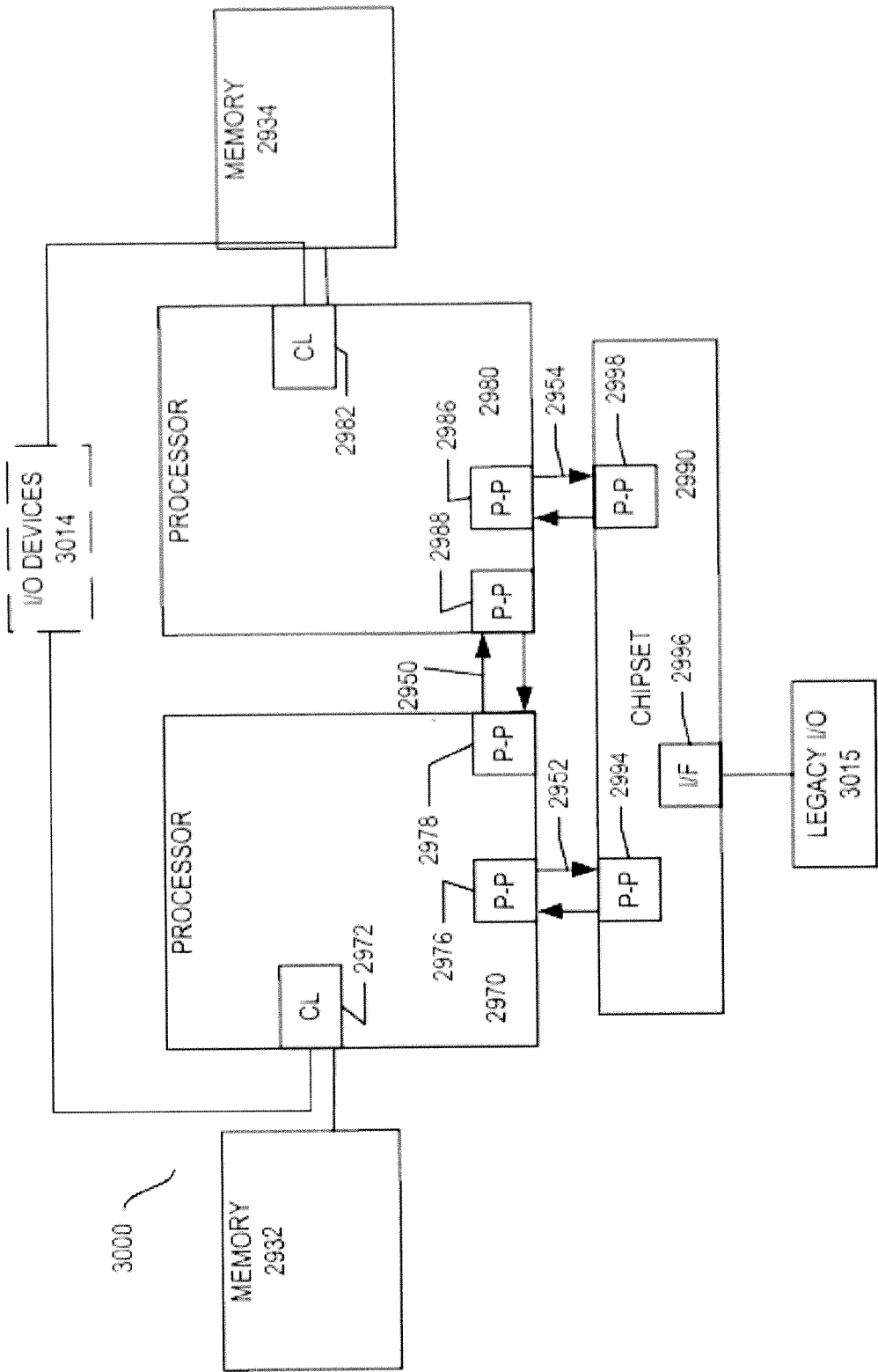


FIG. 18

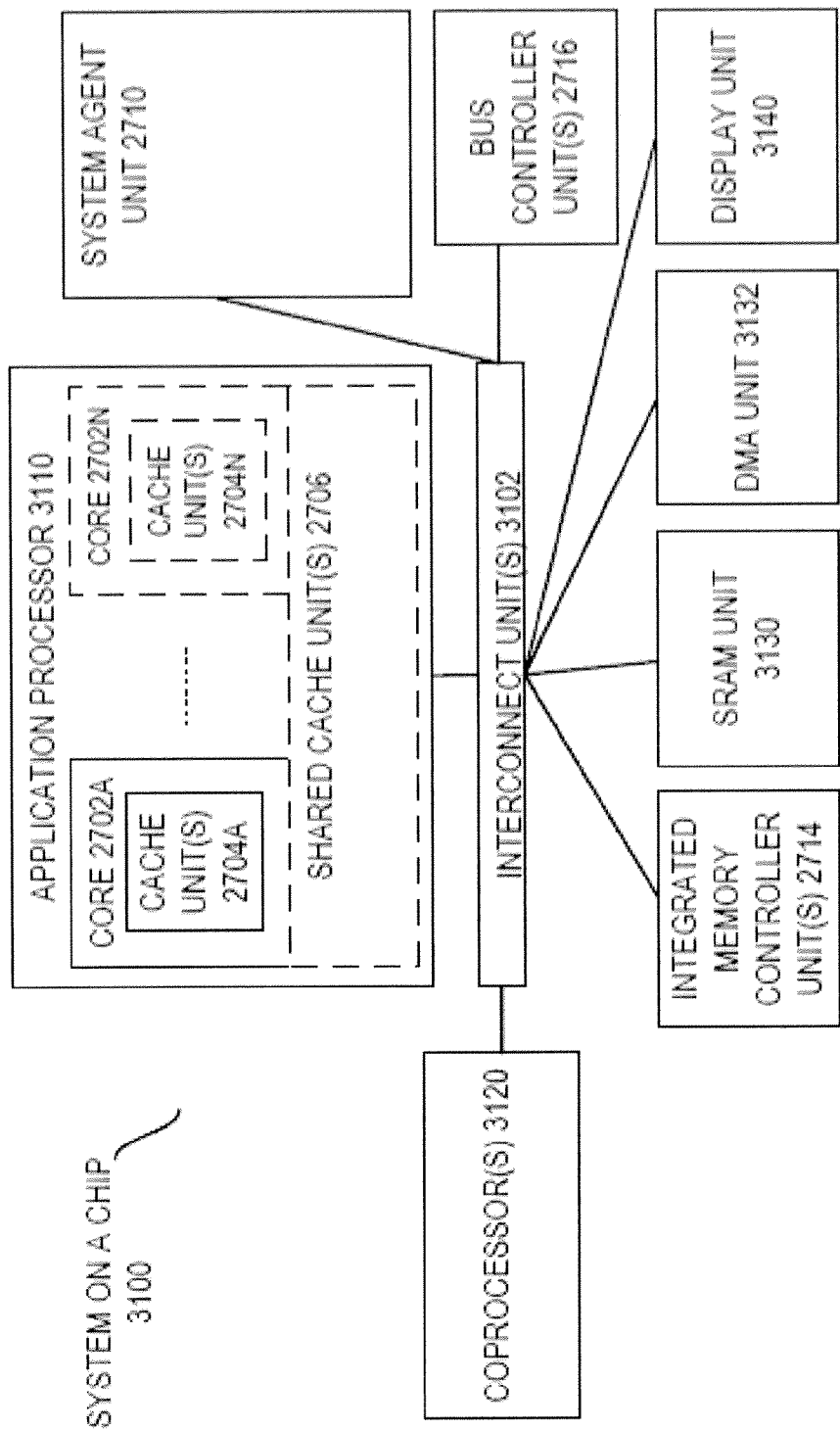


FIG. 19

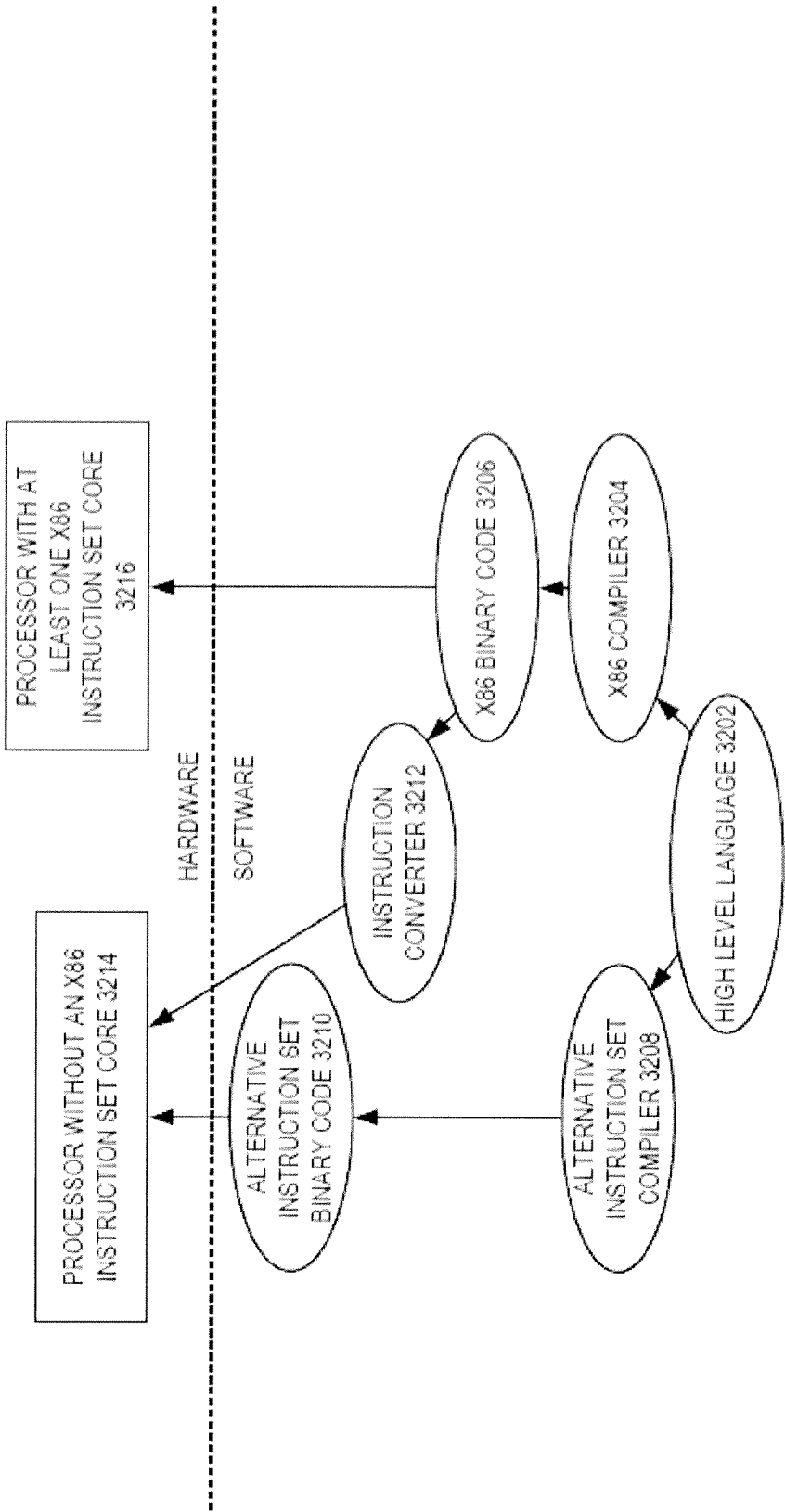


FIG. 20

A. CLASSIFICATION OF SUBJECT MATTER**G06F 9/38(2006.01)i, G06F 13/14(2006.01)i, G11C 11/40(2006.01)i**

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

G06F 9/38; G06F 9/30; G06F 9/46; G06F 9/00; G06F 13/14; G11C 11/40

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

Korean utility models and applications for utility models

Japanese utility models and applications for utility models

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

eKOMPASS(KIPO internal) & Keywords: thread, pipeline, switch, priority, fair

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2003-0233394 A1 (RUDD, KEVIN W. et al.) 18 December 2003 See paragraphs [0017], [0021]-[0025], [0029]-[0034], and figures 2-7.	1,2-5,6,7,8,9-12 ,13,14,15,16,17-19 ,20,21
X	US 6567839 B1 (BORKENHAGEN, JOHN MICHAEL et al.) 20 May 2003 See column 6, lines 28-42; column 17, line 54 - column 18, line 56, and figures 4A, 6.	1,6,7,8,13,14,15 ,20,21
A		2-5,9-12,16-19
A	US 2006-0136915 A1 (AINGARAN KATHIRGAMAR et al.) 22 June 2006 See paragraphs [0040]-[0048] and figure 9.	1-21
A	US 2012-0166777 A1 (MCLELLAN, EDWARD J. et al.) 28 June 2012 See paragraphs [0018]-[0022].	1-21



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 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

05 September 2013 (05.09.2013)

Date of mailing of the international search report

05 September 2013 (05.09.2013)

Name and mailing address of the ISA/KR

Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City,
302-701, Republic of Korea

Facsimile No. +82-42-472-7140

Authorized officer

JANG Ho Keun

Telephone No. +82-42-481-8187



INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.

PCT/US2013/047429

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2003-0233394 A1	18/12/2003	US 8024735 B2	20/09/2011
US 6567839 B1	20/05/2003	CN 1127017 C0	05/11/2003
		CN 1276887 A0	13/12/2000
		DE 69807729 D1	10/10/2002
		DE 69807729 T2	05/06/2003
		EP 1027645 A1	16/08/2000
		EP 1027645 B1	04/09/2002
		IL 135459 A	19/02/2004
		IL 135459 D0	20/05/2001
		JP 03595504 B2	02/12/2004
		JP 2001-521216 A	06/11/2001
		JP 2001-521216 T	06/11/2001
		TW 406241 A	21/09/2000
		TW 406241 B	21/09/2000
		WO 99-21083 A1	29/04/1999
US 2006-0136915 A1	22/06/2006	GB 0525139 D0	18/01/2006
		GB 2421325 A	21/06/2006
		GB 2421328 A	21/06/2006
		GB 2421328 B	18/07/2007
		US 2006-0136919 A1	22/06/2006
US 2012-0166777 A1	28/06/2012	None	