CONTROLLING EXECUTION OF THREADS IN A MULTI-THREADED PROCESSOR

BACKGROUND

[0001] One or more aspects relate, in general, to multi-threaded processors, and in particular, to controlling execution of threads in such processors.

[0002] A processor may include multiple hardware threads that have instructions executing simultaneously. Such a processor is said to implement simultaneous multi-threading (SMT), which is a technique used to improve overall efficiency of a processor by permitting multiple independent threads of execution to better utilize resources provided by modern processor architectures.

[0003] By controlling execution of the threads of a multi-threaded processor, further efficiencies may be gained.

SUMMARY

[0004] Shortcomings of the prior art are overcome and advantages are provided through the provision of a method as claimed in claim 1, and corresponding system and computer program product.

[0005] Additional features and advantages are realized. Other embodiments and aspects are described in detail herein and are considered a part of the claimed invention.

BRIEF DESCRIPTION OF THE SEVERAL VIEWS OF THE DRAWINGS

[0006] One or more aspects are particularly pointed out and distinctly claimed as examples in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages are apparent from the following detailed description taken in conjunction with the accompanying drawings in which:

FIG. 1 depicts one example of a computing environment to incorporate and use one or more aspects of controlling execution of threads;

FIG. 2 depicts another example of a computing environment to incorporate and use one or more aspects of controlling execution of threads;

FIG. 3A depicts yet another example of a computing environment to incorporate and use one or more aspects of controlling execution of threads;

FIG. 3B depicts further details of a memory of the computing environment of FIG. 3A;

FIG. 3C depicts a further example of a computing environment to incorporate and use one or more aspects of controlling execution of threads;

FIG. 4A depicts one example of a control register used in accordance with one aspect of controlling execution of threads;

FIG. 4B depicts one example of an instruction address register used in accordance with one aspect of controlling execution of threads;

FIG. 5 depicts one example of stages of a pipeline;

FIG. 6 depicts one example of logic to control execution of threads in a multi-threaded processor;

FIG. 7A depicts one example of a format of a Drain instruction;

FIG. 7B depicts one embodiment of logic associated with the Drain instruction of FIG. 7A;

FIG. 8A depicts one example of a format of a Compare And Swap R-Unit Register instruction;

FIG. 8B depicts one embodiment of logic associated with the Compare And Swap R-Unit Register instruction of FIG. 8A;

FIG. 9A depicts one example of a format of a Load and OR R-Unit Register instruction;

FIG. 9B depicts one embodiment of logic associated with the Load and OR R-Unit Register instruction of FIG. 9A;

FIG. 10A depicts one example of a format of a Load and AND R-Unit Register instruction;

FIG. 10B depicts one embodiment of logic associated with the Load and AND R-Unit Register instruction of FIG. 10A;

FIGs. 11A-11B depict one example of logic associated with interlocking used by one or more instructions; and

FIG. 12 depicts one embodiment of a computer program product.

DETAILED DESCRIPTION

[0007] In accordance with one or more aspects, a capability is provided for controlling execution of threads (e.g., hardware threads) in a core (e.g., a physical hardware processor; also referred to herein as a processor or processor core) operating within a computing environment. The core supports, for instance, multi-threading, such as simultaneous multi-threading (SMT), which means there can be effectively multiple logical central processing units (CPUs) operating simultaneously on the same physical processor hardware. Each of these logical CPUs is considered a thread.

[0008] In such a multi-threading environment, it may be desirous for one thread to stop other threads on the processor core from executing. This may be in response to running a critical sequence or other sequence that needs the processor core resources or is manipulating processor core resources in a way that other threads would interfere with its execution. In one example, as part of the capability, it may be desirable to wait until some condition has been satisfied for all the threads on the processor core. For example, assume software or firmware running on a particular hardware thread wants to perform a system action that first requires no stores are in progress from the entire processor core, that is, no stores are in progress on all the threads on the processor core. To determine if the other threads are stopped, an instruction, referred to herein as a Drain instruction, is provided, in accordance with one aspect, that monitors the status of the threads on the processor core.

[0009] Further, in accordance with one or more aspects, in controlling execution of the threads, various atomic instructions may be used. These instructions operate on registers accessible to and shared by the threads of the SMT processor, rather than storage or memory. (Memory and storage are used interchangeably herein, unless otherwise noted implicitly or explicitly.) This allows multiple threads to communicate and share information using the shared registers, rather than storage. These instructions, referred to herein as Compare And

Swap R-Unit Register or Compare and Swap Register instruction, Load and OR R-Unit Register or Load and OR Register instruction, and Load and AND R-Unit Register or Load and AND Register instruction, control access to the shared registers using interlocking, as described herein.

[0010] One example of a computing environment to incorporate and use one or more aspects of controlling execution of threads is described with reference to FIG. 1. Referring to FIG. 1, in one example, a computing environment 100 is based on the z/Architecture, offered by International Business Machines (IBM®) Corporation, Armonk, New York. The z/Architecture is described in an IBM Publication entitled "z/Architecture – Principles of Operation," Publication No. SA22-7832-09, 10th Edition, September 2012.

[0011] Z/ARCHITECTURE, IBM, and Z/VM, Z/OS, POWER, and POWERPC (referenced herein) are registered trademarks of International Business Machines Corporation, Armonk, New York. Other names used herein may be registered trademarks, trademarks or product names of International Business Machines Corporation or other companies.

[0012] As one example, computing environment 100 includes a central processor complex (CPC) 102 coupled to one or more input/output (I/O) devices 106 via one or more control units 108. Central processor complex 102 includes, for instance, a processor memory 104 (a.k.a., main memory, main storage, central storage) coupled to one or more processor cores 110, and an input/output subsystem 111, each of which is described below.

[0013] Processor memory 104 includes, for example, one or more partitions 112 (e.g., logical partitions), and processor firmware 113, which includes, e.g., a logical partition hypervisor 114 and other processor firmware 115. One example of logical partition hypervisor 114 is the Processor Resource/System Manager (PR/SM), offered by International Business Machines Corporation, Armonk, New York.

[0014] A logical partition functions as a separate system and has one or more applications 120, and optionally, a resident operating system 122 therein, which may differ for each logical partition. In one embodiment, the operating system is the z/OS operating system, the z/VM

operating system, the z/Linux operating system, or the TPF operating system, offered by International Business Machines Corporation, Armonk, New York.

[0015] Logical partitions 112 are managed by logical partition hypervisor 114, which is implemented by firmware running on cores 110. As used herein, firmware includes, e.g., the microcode and/or millicode of the processor core. It includes, for instance, the hardware-level instructions and/or data structures used in implementation of higher level machine code. In one embodiment, it includes, for instance, proprietary code that is typically delivered as microcode that includes trusted software or microcode specific to the underlying hardware and controls operating system access to the system hardware.

[0016] Processor cores 110 are physical processor resources allocated to the logical partitions. In particular, each logical partition 112 has one or more logical processors, each of which represents all or a share of a core 110 allocated to the partition. The logical processors of a particular partition 112 may be either dedicated to the partition, so that the underlying core resource 110 is reserved for that partition; or shared with another partition, so that the underlying core resource is potentially available to another partition.

[0017] In one example, at least one of the cores is a multi-threading processor, such as a simultaneous multi-threading processor, that includes multiple threads (i.e., multiple logical CPUs operating simultaneously). In one example, the core includes two threads, but in other embodiments, there may be more than two threads. Two threads, referred to herein as T0 (126) and T1 (128), are only one example.

[0018] In support of simultaneous multi-threading, the processor core hardware contains the full architected state (e.g., z/Architecture and micro-architected state) for each thread. Thus, processor-wide registers 130, which are common to all threads (referred to herein as common registers), as well as thread-specific registers 132, which are unique to a thread (referred to herein as unique registers) are provided. Use of these registers is described further below.

[0019] To control execution of the multiple threads, core 110 includes hardware and/or logic to provide such control as described herein. This hardware and/or logic is referred to herein as a thread control facility 134 for convenience.

[0020] Input/output subsystem 111 directs the flow of information between input/output devices 106 and main storage 104. It is coupled to the central processing complex, in that it can be a part of the central processing complex or separate therefrom. The I/O subsystem relieves the processor cores of the task of communicating directly with the input/output devices and permits data processing to proceed concurrently with input/output processing. To provide communications, the I/O subsystem employs I/O communications adapters. There are various types of communications adapters including, for instance, channels, I/O adapters, PCI cards, Ethernet cards, Small Computer Storage Interface (SCSI) cards, etc. In the particular example described herein, the I/O communications adapters are channels, and therefore, the I/O subsystem is referred to herein as a channel subsystem. However, this is only one example. Other types of I/O subsystems can be used.

[0021] The I/O subsystem uses one or more input/output paths as communication links in managing the flow of information to or from input/output devices 106. In this particular example, these paths are called channel paths, since the communication adapters are channels.

[0022] Another example of a computing environment to incorporate and use one or more aspects of controlling execution of threads is described with reference to FIG. 2. In this example, a computing environment 200 includes a non-partitioned environment implemented based on the z/Architecture (or another architecture in another embodiment). It includes a core 202 that includes, for instance, one or more caches 204; at least two threads, T0 (206), T1 (208); a common set of registers 210 for the threads; and a unique set of registers 212 for each thread, as well as a thread control facility 214.

[0023] Core 202 is communicatively coupled to a memory 216 having one or more caches 218 and at least one control utility 220, such as an operating system; and to an input/output (I/O) subsystem 222. I/O subsystem 222 is communicatively coupled to external I/O devices

224 that may include, for example, data input devices, sensors and/or output devices, such as displays.

[0024] Another embodiment of a computing environment to incorporate and use one or more aspects of controlling execution of threads is described with reference to FIG. 3A. In this example, a computing environment 300a includes, for instance, a native core 302, a memory 304, and one or more input/output devices and/or interfaces 306 coupled to one another via, for example, one or more buses 308 and/or other connections. As examples, computing environment 300a may include a PowerPC processor or a Power Systems server offered by International Business Machines Corporation, Armonk, New York; an HP Superdome with Intel Itanium II processors offered by Hewlett Packard Co., Palo Alto, California; and/or other machines based on architectures offered by International Business Machines Corporation, Hewlett Packard, Intel, Oracle, or others.

[0025] Native core 302 includes one or more native registers 310, such as one or more general purpose registers and/or one or more special purpose registers used during processing within the environment that include information that represents the state of the environment at any particular point in time. Further, native core may include, for instance, at least two threads, T0 (311), T1 (313); a set of common registers 315 for the threads; a set of thread-specific registers 317 for each thread; and a thread control facility 319.

[0026] Moreover, native core 302 executes instructions and code that are stored in memory 304. In one particular example, the processor core executes emulator code 312 stored in memory 304. This code enables the computing environment configured in one architecture to emulate one or more other architectures. For instance, emulator code 312 allows machines based on architectures other than the z/Architecture, such as PowerPC processors, Power Systems servers, HP Superdome servers or others, to emulate the z/Architecture and to execute software and instructions developed based on the z/Architecture.

[0027] In a further embodiment, as shown in FIG. 3C, core 302 is a single-threaded core, but a multi-threaded core is being emulated and included within emulator code 312. For instance, emulator code 312 includes an emulated thread control facility 320; emulated threads

322, 324; emulated common registers 326 and emulated unique registers 328, each of which is based on an architecture different from the architecture of native core 302, such as the z/Architecture.

[0028] Further details relating to emulator code 312 are described with reference to FIG. 3B. Guest instructions 350 stored in memory 304 comprise software instructions (e.g., correlating to machine instructions) that were developed to be executed in an architecture other than that of native core 302. For example, guest instructions 350 may have been designed to execute on a z/Architecture core 202, but instead, are being emulated on native core 302, which may be, for example, an Intel Itanium II processor. In one example, emulator code 312 includes an instruction fetching routine 352 to obtain one or more guest instructions 350 from memory 304, and to optionally provide local buffering for the instructions obtained. It also includes an instruction translation routine 354 to determine the type of guest instruction that has been obtained and to translate the guest instruction into one or more corresponding native instructions 356. This translation includes, for instance, identifying the function to be performed by the guest instruction and choosing the native instruction(s) to perform that function.

[0029] Further, emulator code 312 includes an emulation control routine 360 to cause the native instructions to be executed. Emulation control routine 360 may cause native core 302 to execute a routine of native instructions that emulate one or more previously obtained guest instructions and, at the conclusion of such execution, return control to the instruction fetch routine to emulate the obtaining of the next guest instruction or a group of guest instructions. The guest instructions may be instructions of the thread control facility described herein. Execution of the native instructions 356 may include loading data into a register from memory 304; storing data back to memory from a register; or performing some type of arithmetic or logic operation, as determined by the translation routine.

[0030] Each routine is, for instance, implemented in software, which is stored in memory and executed by native core 302. In other examples, one or more of the routines or operations are implemented in firmware, hardware, software or some combination thereof. The registers of the emulated processor may be emulated using registers 310 of the native core or by using

locations in memory 304. In embodiments, guest instructions 350, native instructions 356 and emulator code 312 may reside in the same memory or may be disbursed among different memory devices.

[0031] The computing environments described above are only examples of computing environments that can be used. Other environments, including but not limited to, other non-partitioned environments, other partitioned environments, and/or other emulated environments, may be used; embodiments are not limited to any one environment.

[0032] As indicated above, associated with each thread is a plurality of registers. One shared register common to the threads is a control register, such as a millicode control register (MCR), MCR002, an example of which is depicted in FIG. 4A. MCR002 (400) includes various controls for SMT that determines how the threads behave. In one embodiment, MCR002 (400) includes a plurality of fields 402, and those fields used in accordance with one or more aspects include, for instance:

[0033] (a) A transient stop I-fetch field 404: The two bits of this field correspond one-to-one with threads 0 and 1 (if there were more than two threads, then there may be more than two bits). When a bit is '1'b, this becomes an effective transient master override to block I-fetching regardless of the state of other control bits; and

[0034] (b) A No I-fetch stopping allowed field 406: The two bits of this field correspond one-to-one with threads 0 and 1 (if there were more than two threads, then there may be more than two bits). When a bit is '1'b, it indicates this thread is entering a section of code (e.g., critical section) in which the other thread is not allowed to turn on the stop I-fetch bit for this thread.

[0035] Another register used is an instruction address register, which is unique for each thread. This register, referred to as IAREGFA, includes information about a program interruption detected by hardware. An example of IAREGFA is depicted in FIG. 4B. As shown, IAREGFA 450 includes a plurality of fields 452. One field used in accordance with one or more aspects is field 454 that indicates the thread is in process of taking an exception.

[0036] Each of the above registers may include additional, less and/or different fields. Further, there may be other registers that are used. The registers and fields described herein are examples of registers and/or fields that may be used. Further, MCR and IAREGFA are just examples of names of the registers. Many variations are possible.

[0037] To increase instruction throughput, each thread uses an instruction pipeline for processing allowing multiple operations to be performed at the same time. An instruction pipeline includes a plurality of stages, and one example of such a pipeline is described with reference to FIG. 5. Referring to FIG. 5, a pipeline 500 which supports out-of-order processing, includes, for instance, an instruction fetch stage 502 in which instructions are fetched from memory; an instruction decode/dispatch stage 504 which forms dispatch/completion groups and puts instructions into the issue queue; an issue stage 506 in which the instructions are issued (out-of-order); an execute stage 508 in which the instructions are executed (out-of-order); a finish stage 510 in which instructions are finished (out-of-order); a completion stage 512 which refers to an architectural checkpoint; and a recovery checkpoint stage 514. Other pipelines may include additional, less and/or different stages. The stages described herein are only examples.

[0038] In one example, up to three instructions (in particular, micro-operations) can be placed into a group. However, certain instructions, such as branch instructions, end a group even if it is not full. A full group of instructions is steered to the same issue queue, and then, the next group goes into another issue queue.

[0039] In accordance with an aspect of the present invention, a capability is provided for one thread running in a core to stop one or more other threads executing within the core in order to perform one or more operations. In the examples described herein, the core is an SMT-2 design indicating that there are two threads. However, in other embodiments, there may be more than two threads.

[0040] One embodiment of the logic used to control execution of one or more threads is described with reference to FIG. 6. In this example, Thread 0 (T0) executing on a core is attempting to stop Thread 1 (T1) executing on the core, and therefore, the description refers to

T0 and T1; however, in other embodiments, T1 may be attempting to stop T0; and/or there may be more than one thread executing on the core being stopped. For instance, T0 may be stopping T1, T2, T3, etc. Further, in the examples described herein, the logic is performed by firmware of the core; however, in one or more other embodiments, it may be performed by general purpose software. Many other variations are possible.

[0041] As described with reference to FIG. 6, in one embodiment, one thread stops execution of another thread, and the stopping uses one or more controls (e.g., indicators, bits, etc.) in one or more registers (e.g., hardware registers) shared by the threads.

[0042] Referring to FIG. 6, in one embodiment, Thread 0 checks whether T1 (or in other embodiments, one or more threads of the core) is prohibiting being stopped, STEP 600. In one example, this is determined by checking a selected bit (e.g., bit 25) of MCR002, as well as a selected bit (e.g., bit 4) of IAREGFA. This is accomplished, in one example, by T0 testing a branchpoint, referred to a STPIFALW. STPIFALW tests the selected bits of MCR002 and IAREGFA. For instance, if MCR0002.25 (i.e., bit 25 of MCR002) is set to zero and IAREGIFA.4 (i.e., bit 4 of IAREGIFA) is set to zero, then the stopping of I-fetching of T1 is allowed.

[0043] If STPIFALW indicates that T1 is prohibiting being stopped, INQUIRY 602, then processing continues to STEP 600. However, if T1 is not prohibiting being stopped, as indicated by STPIFALW, and in particular MCR002.25=0 and IAREGFA.4=0, then processing continues by T0 stopping instruction fetching and execution on T1, STEP 604. In one example, this includes T0 setting the transient stop I-fetch bit for T1 (e.g., MCR002.9), which stops instruction fetching and execution on T1. This bit is set using, for instance, a Compare and Swap R-Unit Register (CSGRU) instruction or a Load and OR R-Unit register instruction, each of which is described below.

[0044] Thereafter, T0 performs a drain operation for all the threads (DRAIN ALLTIDS), STEP 606, which holds instruction dispatch for T0 until all instructions on T1 are drained or flushed from the pipeline, and queries T1 for status. In one example, a Drain instruction is used to perform the drain operation, an example of which is described below.

[0045] While the pipeline is being drained of the instructions on the other threads, INQUIRY 608, processing continues with STEP 606. However, responsive to the instructions on T1 being drained, T0 continues instruction dispatch and execution for T0, STEP 610.

[0046] Thereafter, T0 checks again whether T1 (and other threads, if any) is prohibiting being stopped, to ensure T1 did not change its status after being tested but before being stopped, STEP 612. This check is performed, as described above, using STPIFALW. If T1 is now prohibiting being stopped, INQUIRY 614, then T0 allows T1 to continue to execute by turning off bit 9 of MCR002 (i.e., set it to zero), STEP 616. Processing proceeds to STEP 600.

[0047] Otherwise, if T1 is not prohibiting being stopped, INQUIRY 614, then T0 performs the instruction sequence (e.g., one or more operations) that caused the stopping of T1, STEP 618. After that instruction sequence is complete, T1 is allowed to continue, STEP 620. Thus, T0 resets bit 9 in MCR002 by using, for instance, a Load and AND R-Unit (LNRU) instruction or CSGRU, as described below. Thereafter, both threads execute normally, STEP 622.

[0048] As described above, a number of instructions are used to control execution of one or more threads of a multi-threading processor. Each of these instructions is described below.

[0049] Referring to FIGs. 7A-7B, one embodiment of a Drain instruction is described. In particular, FIG. 7A depicts one embodiment of a format of the Drain instruction, and FIG. 7B depicts one embodiment of the logic associated with the Drain instruction.

[0050] With reference to FIG. 7A, a Drain instruction 700 includes an opcode field 702 that includes an operation code identifying a drain operation; a mask (M3) field 704 that includes a value indicating a stall count, which specifies how many cycles processing is stalled; and an instruction field 706 (I2) that indicates the type of drain, which, in this example, is a drain all TIDS (Thread IDs) specifying that all threads are to be drained.

[0051] In operation and with reference to FIG. 7B, thread T0 halts instruction processing for T0, at the instruction decode or dispatch stage of the pipeline until specified conditions are met, STEP 750. Specified bits of the I2 field of the instruction (e.g., bits 0:31 of I2, which are,

e.g., bits 16:47 of the I-text field, which includes all of the fields of the instruction) specify which one or more hardware conditions are to be met prior to continuing instruction processing. In one embodiment, the specified conditions include a cross-thread control (e.g., bit 0 of the I2 field; bit 16 of the I-text field), which checks the status of T1 (or other threads) to determine whether processing has been halted on T1. When bit 0 of the I2 field is '1'b, it specifies that all the other drain conditions are to be met on both threads in order to continue processing on this thread (the other thread(s) are not blocked by a DRAIN on this thread). When using this function, care is to be taken to avoid hangs.

[0052] In one or more embodiments, other conditions may be specified in the I2 field. A one in a given bit position indicates that condition is to be met prior to resuming instruction processing; if more than one bit is on, all selected conditions are to be met. In implementation, in one embodiment, when I-text bit 16 (i.e., bit 0 of the I2 field) is 1, the logical OR of both (or all) hardware threads' status functions are performed, on a bit-by-bit basis, before ORing together all functions that are selected to determine the final value of whether the DRAIN conditions are satisfied.

[0053] A determination is made as to whether the specified conditions have been met, INQUIRY 752. If not, then the halting continues, STEP 750. Otherwise, if the conditions have been met, processing is stalled an additional number of cycles, STEP 754. This additional number may be zero or more, and is specified in the M3 field of the Drain instruction. For instance, the M3 field specifies an additional number of cycles between 0 and 15, as examples, to stall after the conditions specified in the I2 field are satisfied. Subsequent to stalling the additional number of cycles, instruction processing resumes, STEP 756.

[0054] In one embodiment, if a prior instruction and the drain are being dispatched simultaneously, the prior instruction is allowed to complete dispatch and continue through the pipeline normally, but the Drain instruction and all subsequent instructions will be blocked at dispatch until the conditions are satisfied. Note that the Drain instruction only operates on this thread in delaying processing. To stop another thread, the technique described herein is used. However, a specified bit (e.g., bit 0 of I2), when 1, indicates that all specified conditions on all threads are to be met in order to continue processing after the drain instruction on this thread.

[0055] As indicated, the M3 field of the instruction specifies the number of additional cycles to stall at the pipeline. This may be used in conjunction with any of the allowed conditions in the I2 field. It may also be specified with the I2 field all zeros which gives an immediate cycle count delay at dispatch. There is a stall of one cycle in dispatching of the Drain instruction even when the M3 field is zero. Therefore, this count specifies the number of cycles to delay plus one cycle. The hardware can issue the Drain along with other instructions and it can issue out-of-order since it only affects the front-end stages of the pipeline.

[0056] This instruction is intended for use where the necessary interlocks to guarantee correct operation are not built into the hardware. In most cases, the hardware automatically covers windows from prior instructions in the pipeline.

[0057] The condition code is not changed by this instruction.

[0058] Another instruction used is the Compare and Swap R-Unit Register instruction, which is described with reference to FIGs. 8A-8B. In particular, FIG. 8A depicts one embodiment of a format of the Compare and Swap R-Unit Register instruction, and FIG. 8B depicts one embodiment of the logic associated with the Compare and Swap R-Unit Register instruction. It should be noted that R-Unit in the instructions discussed herein refers to a particular unit within the core that performs the instruction. However, the use of a particular unit is not necessary. It may be performed by other units or simply by the core.

[0059] With reference to FIG. 8A, a CSGRU instruction 800 includes at least one opcode field 802a, 802b including an operation code specifying a compare and swap register operation; a first register field (R1) 804; a second register field (R3) 806; and an instruction field (I2) 808, each of which is described below.

[0060] In operation and with reference to FIG. 8B, the contents of the R-Unit register (referred to herein as MCR) specified by the 10-bit absolute register number indicated in selected bits (e.g., bits 22:31 of the I-text (e.g., bits 6:15 of the I2 field (808)) are compared with the contents of a general register (GR) specified in R1, STEP 850. If they are equal, INQUIRY 852, then the contents of MCR is written to the general register specified in R1,

STEP 854, and the contents of the general register specified in R3 is written to MCR, STEP 856. Additionally, the condition code is set to zero, STEP 858, and processing of CSGRU is complete.

[0061] Returning to INQUIRY 852, if the contents of MCR and the register specified in R1 are not equal, then the contents of MCR are written to the register specified in R1, STEP 860, and the condition code is set to one, STEP 858. This concludes processing of CSGRU.

[0062] The read-compare-replace function of CSGRU is an atomic operation as observed by this thread, T0, and the other threads of this processor (e.g., T1). In one embodiment, CSGRU is executed with the SLOW option on in order to avoid cross-thread hangs. The SLOW option is indicated by setting a selected bit (e.g., bit 17) of I2 (808) to one, and is used to request slow-mode, which means there is only one instruction in the entire pipeline at a time. Further, interlocking is performed with this instruction, as described below, and therefore, a selected bit (e.g., bit 16) of I2 (808), referred to herein as ILOCK, is set to one.

[0063] In one embodiment, this instruction is rejected and reissued, if another selected instruction, such as RSR (Read Special Register), WSR (Write Special Register), NSR (AND Special Register), OSR (OR Special Register), XSR (Exclusive OR Special Register), TRBIT (Test Register Bit), RASR (Read Absolute Special Register), WASR (Write Absolute Special Register), TARBIT (Test Absolute Register Bit), NASR (AND Absolute Special Register), OASR (OR Absolute Special Register), XASR (Exclusive OR Absolute Special Register), LORU (Load and OR R-Unit Register), LNRU (Load and AND R-Unit Register) or CSGRU (Compare and Swap R-Unit Register), is in the pipeline for this thread (T0) or any other thread and the ILOCK bit (e.g., I2 bit 16) is on for the other instruction. This instruction is issued, for instance, only after all prior instructions from this thread have issued and also forces all future instructions from this thread to be dependent on it.

[0064] The condition code settings include, for instance: CC0—comparison equal, R-unit register replaced by GR R1; CC1—comparison unequal, R-unit register is unchanged.

[0065] Another instruction used is the Load and OR R-Unit Register (LORU) instruction, which is described with reference to FIGs. 9A-9B. In particular, FIG. 9A depicts one embodiment of a format of the Load and OR R-Unit Register instruction, and FIG. 9B depicts one embodiment of the logic associated with the Load and OR R-Unit Register instruction.

[0066] With reference to FIG. 9A, a LORU instruction 900 includes at least one opcode field 902a, 902b including an operation code specifying a load and OR register operation; a first register field (R1) 904; a second register field (R3) 906; and an instruction field (I2) 908, each of which is described below.

[0067] In operation and with reference to FIG. 9B, the contents of the R-Unit register (referred to herein as MCR) specified by the 10-bit absolute register number indicated in selected bits (e.g., bits 22:31 of the I-text (e.g., bits 6:15) of the I2 field (908)) are loaded into the general register specified in R1, STEP 950. Further, the contents of the general register specified in R3 are logically ORed with the contents of MCR, STEP 952, and the result is written into MCR, STEP 954.

[0068] The read-OR-replace function of LORU is an atomic operation as observed by this thread, T0, and the other threads of this processor (e.g., T1). In one embodiment, LORU is executed with the SLOW option on in order to avoid cross-thread hangs. The SLOW option is indicated by setting a selected bit (e.g., bit 17) of I2 (908) to one. Further, interlocking is performed with this instruction, as described below, and therefore, a selected bit (e.g., bit 16) of I2 (908), referred to herein as ILOCK, is set to one.

[0069] In one embodiment, this instruction is rejected and reissued, if another selected instruction, such as RSR (Read Special Register), WSR (Write Special Register), NSR (AND Special Register), OSR (OR Special Register), XSR (Exclusive OR Special Register), TRBIT (Test Register Bit), RASR (Read Absolute Special Register), WASR (Write Absolute Special Register), TARBIT (Test Absolute Register Bit), NASR (AND Absolute Special Register), OASR (OR Absolute Special Register), XASR (Exclusive OR Absolute Special Register), LORU (Load and OR R-Unit Register), LNRU (Load and AND R-Unit Register) or CSGRU (Compare and Swap R-Unit Register), is in the pipeline for this thread (T0) or any other thread

and the ILOCK bit (I2 bit 16) is on for the other instruction. This instruction is issued, for instance, only after all prior instructions from this thread have issued and also forces all future instructions from this thread to be dependent on it.

[0070] The condition code is unchanged.

[0071] Another instruction used is the Load and AND R-Unit Register (LNRU) instruction, which is described with reference to FIGs. 10A-10B. In particular, FIG. 10A depicts one embodiment of a format of the Load and AND R-Unit Register instruction, and FIG. 10B depicts one embodiment of the logic associated with the Load and AND R-Unit Register instruction.

[0072] With reference to FIG. 10A, a LNRU instruction 1000 includes at least one opcode field 1002a, 1002b including an operation code specifying a load and AND register operation; a first register field (R1) 1004; a second register field (R3) 1006; and an instruction field (I2) 1008, each of which is described below.

[0073] In operation and with reference to FIG. 10B, the contents of the R-Unit register (referred to herein as MCR) specified by its 10-bit absolute register number indicated in selected bits (e.g., bits 22:31 of the I-text (e.g., bits 6:15 of the I2 field (1008)) are loaded into the general register specified in R1, STEP 1050. Further, the contents of the general register specified in R3 are logically ANDed with the contents of MCR, STEP 1052, and the result is written into MCR, STEP 1054.

[0074] The read-AND-replace function of LNRU is an atomic operation as observed by this thread, T0, and the other threads of this processor (e.g., T1). In one embodiment, LNRU is executed with the SLOW option on in order to avoid cross-thread hangs. The SLOW option is indicated by setting a selected bit (e.g., bit 17) of I2 (1008) to one. Further, interlocking is performed with this instruction, as described below, and therefore, a selected bit (e.g., bit 16) of I2 1008, referred to herein as ILOCK, is set to one.

[0075] In one embodiment, this instruction is rejected and reissued, if another selected instruction, such as RSR (Read Special Register), WSR (Write Special Register), NSR (AND Special Register), OSR (OR Special Register), XSR (Exclusive OR Special Register), TRBIT (Test Register Bit), RASR (Read Absolute Special Register), WASR (Write Absolute Special Register), TARBIT (Test Absolute Register Bit), NASR (AND Absolute Special Register), OASR (OR Absolute Special Register), XASR (Exclusive OR Absolute Special Register), LORU (Load and OR R-Unit Register), LNRU (Load and AND R-Unit Register) or CSGRU (Compare and Swap R-Unit Register), is in the pipeline for this thread (T0) or any other thread and the ILOCK bit (e.g., I2 bit 16) is on for the other instruction. This instruction is issued, for instance, only after all prior instructions from this thread have issued and also forces all future instructions from this thread to be dependent on it.

[0076] The condition code is unchanged.

[0077] LNRU, as well as LORU and CSGRU, use registers that are accessible to all threads in the SMT core, rather than storage as a means of shared communication. These registers are, for instance, hardware registers separate from memory or storage of the processor. For example, in one core design, there are approximately 64 registers that are shared (common) to all threads on the core; threads can freely read and write these shared registers. In some cases of control registers, if both threads would attempt to write them without special interlocks, an update by one of the threads could be lost. In other cases, only one of the threads is permitted to "own" a resource controlled by bits in the register. Therefore, these atomic instructions that operate on shared registers are used to control and order access to these shared registers.

[0078] LNRU, LORU and CSGRU each allows an atomic operation between general registers and MCR across threads by using interlocking to control inter-thread operations and execution. As indicated, each of the instructions has an ILOCK bit, and when that bit is on for an instruction executing in the pipeline, if a second instruction enters the pipeline with its ILOCK bit also set, the second instruction is rejected (and re-executed later when the first instruction completes). This guarantees atomicity with accesses to these registers between threads.

[0079] There are, for instance, two types of interlocking instructions: a single micro-operation μop instruction, such as LNRU and LORU; and a two μop instruction, such as CSGRU. With a single μop instruction, the interlock is set at the μop issue (RSR- and WSR-type instruction) and cleared at the μop completion for an RSR-type and on checkpoint for a WSR-type.. In a two μop instruction, the interlock is set at the first μop (RSR-type) issue and is cleared at checkpoint of the second μop (WSR-type).

[0080] Further details regarding using interlocking and interlocking are described with reference to FIGs. 11A-11B. This logic is performed by the core, and in particular, by a pipe on which the instruction is issued.

[0081] Referring initially to FIG. 11A, an instruction to be executed (e.g., LNRU, LORU, CSGRU) is obtained by a multi-threaded processor, STEP 1100. Execution of the instruction is initiated by the multi-threaded processor to perform an operation, STEP 1102. The operation includes multiple sub-operations to be performed atomically. A determination is made as to whether the instruction is to continue to execute, INQUIRY 1104. The determining uses, for instance, interlocking to determine whether the instruction has atomic access to one or more registers shared by the thread and one or more other threads.

[0082] If the instruction is to continue to execute, execution continues, which includes performing the operation using at least one shared register, STEP 1106. Otherwise, if the instruction is not to continue, it is rejected, STEP 1108.

[0083] Further details relating to the interlocking are described with reference to FIG. 11B. Initially, when an instruction enters the R-unit, in one example, a check is made as to whether a lock indicator, such as the ILOCK bit (e.g., bit 32 of I-text – a.k.a., bit 16 of I2) of the incoming instruction, is set (e.g., set to 1), INQUIRY 1150. If the ILOCK bit is not set, then interlocking processing is complete; however, if the ILOCK bit in the incoming instruction is set, then a further determination is made as to whether a lock, referred to as an interlock, is set, INQUIRY 1152. The interlock is placed in a hardware register accessible to multiple threads.

[0084] If the interlock is set (e.g., a bit is set to one) indicating that another instruction is processing that has its ILOCK bit set, then the incoming instruction is rejected, STEP 1153.

[0085] However, if the interlock is not set, then it is set, STEP 1154, and processing of the instruction proceeds (e.g., in the pipe), STEP 1156. When the instruction completes (or is checkpointed), the interlock is reset (e.g., set to zero), STEP 1158.

[0086] Further details regarding interlocking include:

[0087] (A) Interlock can be set by pipe0 when, for instance:

[0088] - there is an instruction in pipe0 which needs to set the interlock and it is issued alone

[0089] - there is an instruction in pipe0 which needs to set the interlock and there is another instruction in pipe1 which does not want to set the lock - both instructions from same thread.

[0090] - there is an instruction in pipe0 which needs to set the interlock and there is another instruction in pipe1 which needs to set the lock but the instruction in pipe0 is older - both instructions from same thread.

[0091] - there is an instruction in pipe0 which needs to set the interlock and there is another instruction in pipe1 which does not want to set the lock - both instructions from different threads.

[0092] - there is an instruction in pipe0 which needs to set the interlock and there is another instruction in pipe1 which needs to set the lock - both instructions from different threads – and the LFSR (Linear Feedback Shift Register) points to pipe0. The LFSR is used to produce a pseudo-random number and by taking the most significant bit of the number, a pseudo-random selection is provided between the two pipes (i.e., randomly choose which pipe would set the interlock).

[0093] In one example, the interlock is a vector having a bit for each possible instruction in a dispatch group. For instance, in one example, there may be up to three instructions in a dispatch group, and therefore, the interlock includes three bits, one for each instruction. When a bit is set, e.g., to 1, this indicates that the instruction associated with that bit has the interlock.

[0094] The interlock can also be set by pipe1, as described above, however pipe0 is replaced with pipe1, and pipe1 with pipe0.

[0095] (B) Set of the interlock is performed when, for instance:

[0096] - there is a valid instruction in the pipe AND

[0097] - the ILOCK is set AND

[0098] - the predec_rd (i.e., an early indication of a Read (RSR)-type instruction) OR predec_wr (i.e., early indication of a write (WSR)-type instruction) is set AND

[0099] - the instruction in the pipe isn't flushed/xconded AND

[00100] - the interlock can be set by that pipe (according to (A)) AND

[00101] - the interlock is not yet set

[00102] (C) Interlock is updated when, for instance:

[00103] - there is a valid instruction in the pipe AND

[00104] - the ILOCK is set AND

[00105] - the predec_rd OR predec_wr is set AND

[00106] - the instruction in the pipe is not flushed/xconded AND

[00107] - the interlock is already set AND

[00108] - the instruction.GTAG (the identifier of a dispatch group that includes the instruction) = interlock.GTAG (i.e., is the identifier associated with the instruction = to the identifier that set the interlock) AND

[00109] - the instruction.th_id (thread id) = interlock.th_id

[00110] In one embodiment, reset of an interlock is performed on group completion if there is no write-type micro-operation (μop) in the group that grabbed the lock. If there is a write-type μop in the group but it did not grab the lock, then the lock is released also on that completion (did not grab the lock = ILOCK bit is 0 - this is why the ILOCK bit for CSGRU is also set in the WSR part so it is not released on the completion of the RSR). If the write-type instruction also grabbed the lock, then the lock will be released only on checkpoint. In that way atomicity will be seen. An exception is for CSGRU where the WSR is in the second group - hence the RSR of the first group sets the lock and the WSR in the second group releases the lock. In that case, the first group is to come before the second group (which has a GTAG that is greater by 1 from the GTAG of the first group).

[00111] A reject of a μop in the group might not reset the lock if that μop does not hold it. The reject will release the lock only, in one example, if there are no other μops in the group which also holds the lock.

[00112] A rescind of a μop in the group might not reset the lock if that μop does not hold it. The rescind will release the lock only, in one example, if there are no other μops in the group which also holds the lock.

[00113] When xcond is coming, a check is made as to whether the interlock can be released. The problem is that the xcond should release the lock only if the instruction that grabbed it was not completed yet. If the instruction that grabbed the lock is already completed, then the xcond should have no affect on the lock (this is true for the write-type instruction which grabbed the lock since that instruction will release it on checkpoint. For a read-type instruction, the release was already done on completion). One exception is CSGRU which its read-type part can already be completed but if there would be an xcond before the write-type completes the lock is to be released (if the write type completes then xcond which will come later should have no effect on the interlock).

[00114] A reset in case the instruction which set the lock needs to be flushed: the actual reset will be done only, e.g., if the lock is not held any more by any instructions of that group. For example, if the flush hits the first μop in the group and this μop holds the lock, then the lock is free (of course, the two other μops can hold it too but they are flushed). If the flush is coming on the second μop in the group and this μop holds the lock, then the lock is free only, e.g., if the first μop does not hold it too (the third will be flushed anyway so no need to check it).

[00115] (D) Interlock is reset when, for instance:

[00116] - Interlock is already set AND

[00117] - No updates from (C) AND

[00118] (

[00119] The instruction which grabbed the lock is completing:

[00120] - the read instruction which locked it is completing AND

[00121] - this is not the first group completion of CSGRU

[00122] OR

[00123] (

[00124] The instruction which grabbed the lock is checkpointing:

[00125] - the write instruction which locked it is checkpointing AND

[00126] - Interlock.GTAG = Instruction.GTAG

[00127] OR

[00128] - if this is the second group completion of the CSGRU, then wait for it to be checkpointed AND

[00129] - Interlock.GTAG+1 = Instruction.GTAG

[00130])

[00131] OR

[00132] The μop which grabbed the lock is rejected and no other holders in that group

[00133] OR

[00134] The μop which grabbed the lock is rescinded and no other holders in that group

[00135] OR

[00136] The μop which grabbed the lock is flushed/xconded and no other holders in that group

[00137] OR

[00138] recovery ongoing

[00139])

[00140] (E) Reject when, for instance:

[00141] 1) the interlock is locked AND

[00142] the instruction.th_id != (not equal) interlock.th_id AND

[00143] the instruction.GTAG != interlock.GTAG

[00144] For a CSGRU opcode, this eliminates the reject of the WSR μop when the interlock was locked by the RSR μop (they have the same GTAG and same thread ID).

[00145] It is also true for groups like (RSR, x, WSR) where the issue is in order but the RSR for some reason gets rejected and hence the WSR locks the lock. In such case if the reject will be on an individual instruction id basis, the RSR would not be able to enter since the lock is

locked and the whole group would not be able to complete ==> a deadlock since the WSR cannot release the lock. The solution is to use the GTAG so the RSR will be able to enter and when it completes, the WSR would be able to complete too and would release the lock.

[00146] 2) same thread on both pipes AND

[00147] the ILOCK is on in both pipes AND

[00148] current pipe holds the younger instruction

[00149] ==> the current younger instruction should be rejected (also if the interlock bit is not turned on yet by the older instruction).

[00150] In case the interlock is turned on, the older should be rejected too by the (1) condition (unless this is the WSR of the CSGRU instruction).

[00151] 3) different threads on both pipes AND

[00152] the ILOCK is on in both pipes AND

[00153] current pipe number does not equal the LFSR's value (which is 0 for pipe0 and 1 for pipe1) ==> the current pipe_x instruction should be rejected (also if the interlock bit is not turned on yet by the older instruction).

[00154] In case the interlock is turned on, both should be rejected by the (1) condition (unless one of them is the WSR of the CSGRU instruction).

[00155] Described herein is one embodiment of a technique for one thread to stop execution of one or more other threads of a multi-threaded processor. The technique is implemented to avoid hangs and to ensure all instructions associated with the other threads are completed before they are stopped. This technique includes, in one aspect, a pipeline DRAIN instruction that looks at status information from all hardware threads of the processor (or selected threads in another embodiment) to see if the conditions are satisfied before continuing operation at this thread.

[00156] Further, one embodiment of this technique uses atomic instructions, such as CSGRU, LORU and LNRU, to operate on shared registers. For instance, when two or more threads share a common core, in a multi-threaded core design (e.g., SMT), they often need to communicate and share information; this could include semaphores, locks, etc. This could involve firmware, millicode, or it could involve software. The threads could use existing ISA

instructions that communicate through storage. However, these may be slow and involve store-hit-load or load-hit-store conflicts (commonly known as Operand Store Compare (OSC)). In addition, if the communication is done by firmware, it may be undesirable or impossible to communicate via storage; a firmware routine could be in the middle of a critical sequence where operand loads and stores are prohibited. Thus, these instructions operate on registers, instead of storage.

[00157] Although the atomic instructions are described in relation to controlling execution of threads, they may be used for other purposes. Each instruction stands apart from the use described herein and may be utilized in other situations.

[00158] Referring to FIG. 12, in one example, a computer program product 1200 includes, for instance, one or more non-transitory computer readable storage media 1202 to store computer readable program code means, logic and/or instructions 1204 thereon to provide and facilitate one or more embodiments.

[00159] The present invention may be a system, a method, and/or a computer program product. The computer program product may include a computer readable storage medium (or media) having computer readable program instructions thereon for causing a processor to carry out aspects of the present invention.

[00160] The computer readable storage medium can be a tangible device that can retain and store instructions for use by an instruction execution device. The computer readable storage medium may be, for example, but is not limited to, an electronic storage device, a magnetic storage device, an optical storage device, an electromagnetic storage device, a semiconductor storage device, or any suitable combination of the foregoing. A non-exhaustive list of more specific examples of the computer readable storage medium includes the following: a portable computer diskette, a hard disk, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM or Flash memory), a static random access memory (SRAM), a portable compact disc read-only memory (CD-ROM), a digital versatile disk (DVD), a memory stick, a floppy disk, a mechanically encoded device such as punch-cards or raised structures in a groove having instructions recorded thereon, and any

suitable combination of the foregoing. A computer readable storage medium, as used herein, is not to be construed as being transitory signals *per se*, such as radio waves or other freely propagating electromagnetic waves, electromagnetic waves propagating through a waveguide or other transmission media (e.g., light pulses passing through a fiber-optic cable), or electrical signals transmitted through a wire.

[00161] Computer readable program instructions described herein can be downloaded to respective computing/processing devices from a computer readable storage medium or to an external computer or external storage device via a network, for example, the Internet, a local area network, a wide area network and/or a wireless network. The network may comprise copper transmission cables, optical transmission fibers, wireless transmission, routers, firewalls, switches, gateway computers and/or edge servers. A network adapter card or network interface in each computing/processing device receives computer readable program instructions from the network and forwards the computer readable program instructions for storage in a computer readable storage medium within the respective computing/processing device.

[00162] Computer readable program instructions for carrying out operations of the present invention may be assembler instructions, instruction-set-architecture (ISA) instructions, machine instructions, machine dependent instructions, microcode, firmware instructions, state-setting data, or either source code or object code written in any combination of one or more programming languages, including an object oriented programming language such as Smalltalk, C++ or the like, and conventional procedural programming languages, such as the "C" programming language or similar programming languages. The computer readable program instructions may execute entirely on the user's computer, partly on the user's computer, as a stand-alone software package, partly on the user's computer and partly on a remote computer or entirely on the remote computer or server. In the latter scenario, the remote computer may be connected to the user's computer through any type of network, including a local area network (LAN) or a wide area network (WAN), or the connection may be made to an external computer (for example, through the Internet using an Internet Service Provider). In some embodiments, electronic circuitry including, for example, programmable logic circuitry, field-programmable gate arrays (FPGA), or programmable logic arrays (PLA) may execute the computer readable program instructions by utilizing state information of the computer readable program

instructions to personalize the electronic circuitry, in order to perform aspects of the present invention.

[00163] Aspects of the present invention are described herein with reference to flowchart illustrations and/or block diagrams of methods, apparatus (systems), and computer program products according to embodiments of the invention. It will be understood that each block of the flowchart illustrations and/or block diagrams, and combinations of blocks in the flowchart illustrations and/or block diagrams, can be implemented by computer readable program instructions.

[00164] These computer readable program instructions may be provided to a processor of a general purpose computer, special purpose computer, or other programmable data processing apparatus to produce a machine, such that the instructions, which execute via the processor of the computer or other programmable data processing apparatus, create means for implementing the functions/acts specified in the flowchart and/or block diagram block or blocks. These computer readable program instructions may also be stored in a computer readable storage medium that can direct a computer, a programmable data processing apparatus, and/or other devices to function in a particular manner, such that the computer readable storage medium having instructions stored therein comprises an article of manufacture including instructions which implement aspects of the function/act specified in the flowchart and/or block diagram block or blocks.

[00165] The computer readable program instructions may also be loaded onto a computer, other programmable data processing apparatus, or other device to cause a series of operational steps to be performed on the computer, other programmable apparatus or other device to produce a computer implemented process, such that the instructions which execute on the computer, other programmable apparatus, or other device implement the functions/acts specified in the flowchart and/or block diagram block or blocks.

[00166] The flowchart and block diagrams in the Figures illustrate the architecture, functionality, and operation of possible implementations of systems, methods, and computer program products according to various embodiments of the present invention. In this regard,

each block in the flowchart or block diagrams may represent a module, segment, or portion of instructions, which comprises one or more executable instructions for implementing the specified logical function(s). In some alternative implementations, the functions noted in the block may occur out of the order noted in the figures. For example, two blocks shown in succession may, in fact, be executed substantially concurrently, or the blocks may sometimes be executed in the reverse order, depending upon the functionality involved. It will also be noted that each block of the block diagrams and/or flowchart illustration, and combinations of blocks in the block diagrams and/or flowchart illustration, can be implemented by special purpose hardware-based systems that perform the specified functions or acts or carry out combinations of special purpose hardware and computer instructions.

[00167] Although various embodiments are described above, these are only examples. For example, computing environments of other architectures can be used to incorporate and use one or more embodiments. Further, one or more aspects of the invention are applicable to forms of multi-threading, other than SMT. Yet further, different instructions, instruction formats, instruction fields and/or instruction values may be used. Many variations are possible.

[00168] Further, other types of computing environments can benefit and be used. As an example, a data processing system suitable for storing and/or executing program code is usable that includes at least two processors coupled directly or indirectly to memory elements through a system bus. The memory elements include, for instance, local memory employed during actual execution of the program code, bulk storage, and cache memory which provide temporary storage of at least some program code in order to reduce the number of times code must be retrieved from bulk storage during execution.

[00169] Input/Output or I/O devices (including, but not limited to, keyboards, displays, pointing devices, DASD, tape, CDs, DVDs, thumb drives and other memory media, etc.) can be coupled to the system either directly or through intervening I/O controllers. Network adapters may also be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices through intervening private or public networks. Modems, cable modems, and Ethernet cards are just a few of the available types of network adapters.

[00170] The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting. As used herein, the singular forms "a", "an" and "the" are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will be further understood that the terms "comprises" and/or "comprising", when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components and/or groups thereof.

[00171] The corresponding structures, materials, acts, and equivalents of all means or step plus function elements in the claims below, if any, are intended to include any structure, material, or act for performing the function in combination with other claimed elements as specifically claimed. The description of one or more embodiments has been presented for purposes of illustration and description, but is not intended to be exhaustive or limited to in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. The embodiment was chosen and described in order to best explain various aspects and the practical application, and to enable others of ordinary skill in the art to understand various embodiments with various modifications as are suited to the particular use contemplated.

CLAIMS:

1. A computer-implemented method of controlling execution of threads in a computing environment, said computer-implemented method comprising:

stopping, by a thread running in a core of the processor of the computing environment, execution of another thread executing within the core of the processor, the stopping using one or more controls in one or more shared registers of the processor, the one or more shared registers being shared by the thread and the other thread, the stopping comprising:

determining, by the thread, whether the other thread is prohibiting being stopped;

stopping, by the thread, instruction fetching and execution on the other thread, based on the determining indicating the other thread is not prohibiting being stopped; and

checking status of the other thread to determine whether execution of the other thread has stopped, wherein the performing the one or more operations is based on the checking indicating execution of the other thread has stopped and that the other thread did not change from not prohibiting being stopped to prohibiting being stopped after the determining and before the stopping;

performing by the thread, one or more operations within the processor after the other thread was stopped from executing within the processor by the thread; and

based on completing the one or more operations, allowing, by the thread, the other thread to continue executing within the processor.

2. The computer-implemented method of claim 1, wherein the stopping execution of the other thread comprises:

obtaining status information for the other thread; and

determining, based on the status information, whether execution of the other thread is stopped, wherein the one or more operations are performed based on the determining indicating execution of the other thread is stopped.

3. The computer-implemented method of claim 2, wherein the determining comprises using a drain instruction configured to obtain status of the other thread.

4. The computer-implemented method of claim 1, wherein the one or more controls includes one or more indicators in the one or more shared registers, and wherein the determining whether the other thread is prohibiting being stopped comprises checking at least one indicator of the one or more indicators.

5. The computer-implemented method of claim 1, wherein the one or more controls includes one or more indicators in the one or more shared registers, and wherein the stopping instruction fetching and execution on the other thread includes setting an indicator of the one or more indicators to stop instruction fetching and execution on the other thread.

6. The computer-implemented method of claim 1, wherein the checking status of the other thread comprises using a drain instruction, the drain instruction configured to hold instruction dispatch on the thread and query status for the other thread.

7. The computer-implemented method of claim 6, wherein the drain instruction specifies one or more conditions to be satisfied prior to the performing the one or more operations, and wherein the performing the one or more operations is executed based on a result of the drain instruction indicating satisfaction of the one or more conditions.

8. The computer-implemented method of claim 1, wherein the method further comprises:

re-determining, based on the checking indicating execution of the other thread has stopped, whether the other thread is prohibiting being stopped;

allowing execution on the other thread, based on the re-determining indicating the other thread is prohibiting being stopped; and

performing the one or more operations, based on the re-determining indicating the other thread is not prohibiting being stopped.

9. The computer-implemented method of claim 1, wherein the stopping comprises stopping execution of a plurality of threads.

10. A computer system for controlling execution of threads in a computing environment, said computer system comprising:

a memory; and

a processor in communications with the memory, wherein the computer system is configured to perform a method, said method comprising:

stopping, by a thread running in a core of the processor of the computing environment, execution of another thread executing within the core of the processor, the stopping using one or more controls in one or more shared registers of the processor, the one or more shared registers being shared by the thread and the other thread, the stopping comprising:

determining, by the thread, whether the other thread is prohibiting being stopped;

stopping, by the thread, instruction fetching and execution on the other thread, based on the determining indicating the other thread is not prohibiting being stopped; and

checking status of the other thread to determine whether execution of the other thread has stopped, wherein the performing the one or more operations is based on the checking indicating execution of the other thread has stopped and that the other thread did not change from not prohibiting being stopped to prohibiting being stopped after the determining and before the stopping;

performing by the thread, one or more operations within the processor after the other thread was stopped from executing within the processor by the thread; and

based on completing the one or more operations, allowing, by the thread, the other thread to continue executing within the processor.

11. The computer system of claim 10, wherein the stopping execution of the other thread comprises:

obtaining status information for the other thread; and

determining, based on the status information, whether execution of the other thread is stopped, wherein the one or more operations are performed based on the determining indicating execution of the other thread is stopped.

12. The computer system of claim 11, wherein the determining comprises using a drain instruction configured to obtain status of the other thread.

13. The computer system of claim 10, wherein the one or more controls includes one or more indicators in the one or more shared registers, and wherein the determining whether the other thread is prohibiting being stopped comprises checking at least one indicator of the one or more indicators.

14. The computer system of claim 10, wherein the one or more controls includes one or more indicators in the one or more shared registers, and wherein the stopping instruction fetching and execution on the other thread includes setting an indicator of the one or more indicators to stop instruction fetching and execution on the other thread.

15. The computer system of claim 10, wherein the checking status of the other thread comprises using a drain instruction, the drain instruction configured to hold instruction dispatch on the thread and query status for the other thread.

16. The computer system of claim 15, wherein the drain instruction specifies one or more conditions to be satisfied prior to the performing the one or more operations, and wherein the performing the one or more operations is executed based on a result of the drain instruction indicating satisfaction of the one or more conditions.

17. The computer system of claim 10, wherein the method further comprises:

re-determining, based on the checking indicating execution of the other thread has stopped, whether the other thread is prohibiting being stopped;

allowing execution on the other thread, based on the re-determining indicating the other thread is prohibiting being stopped; and

performing the one or more operations, based on the re-determining indicating the other thread is not prohibiting being stopped.

18. The computer system of claim 10, wherein the stopping comprises stopping execution of a plurality of threads.

19. A computer program product for controlling execution of threads in a computing environment, said computer program product comprising:

a computer readable storage medium readable by a processing circuit and storing instructions for execution by the processing circuit for performing a method comprising:

stopping, by a thread running in a core of the processor of the computing environment, execution of another thread executing within the core of the processor, the stopping using one or more controls in one or more shared registers of the processor, the one or more shared registers being shared by the thread and the other thread, the stopping comprising:

determining, by the thread, whether the other thread is prohibiting being stopped;

stopping, by the thread, instruction fetching and execution on the other thread, based on the determining indicating the other thread is not prohibiting being stopped; and

checking status of the other thread to determine whether execution of the other thread has stopped, wherein the performing the one or more operations is based on the checking indicating execution of the other thread has stopped and that the other thread did not change from not prohibiting being stopped to prohibiting being stopped after the determining and before the stopping;

performing by the thread, one or more operations within the processor after the other thread was stopped from executing within the processor by the thread; and

based on completing the one or more operations, allowing, by the thread, the other thread to continue executing within the processor.

20. The computer program product of claim 19, wherein the stopping execution of the other thread comprises:

obtaining status information for the other thread; and

determining, based on the status information, whether execution of the other thread is stopped, wherein the one or more operations are performed based on the determining indicating execution of the other thread is stopped.

21. The computer program product of claim 19, wherein the checking status of the other thread comprises using a drain instruction, the drain instruction configured to hold instruction dispatch on the thread and query status for the other thread, and wherein the drain instruction specifies one or more conditions to be satisfied prior to the performing the one or more operations, and wherein the performing the one or more operations is executed based on a result of the drain instruction indicating satisfaction of the one or more conditions.

22. The computer program product of claim 19, wherein the method further comprises:

re-determining, based on the checking indicating execution of the other thread has stopped, whether the other thread is prohibiting being stopped;

allowing execution on the other thread, based on the re-determining indicating the other thread is prohibiting being stopped; and

performing the one or more operations, based on the re-determining indicating the other thread is not prohibiting being stopped.