(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
2 August 2012 (02.08.2012)

WIPO | PCT

(10) International Publication Number
WO 2012/103359 A2

(51) International Patent Classification:
G06F 1/00 (2006.01) *G06F 9/38* (2006.01)
G06F 9/30 (2006.01) *G06F 15/76* (2006.01)

(21) International Application Number:
PCT/US2012/022760

(22) International Filing Date:
26 January 2012 (26.01.2012)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/436,966 27 January 2011 (27.01.2011) US

(71) Applicant *(for all designated States except US)*: **SOFT MACHINES, INC.** [US/US]; 3211 Scott Boulevard, Suite 202, Santa Clara, CA 95054 (US).

(72) Inventor; and

(75) Inventor/Applicant *(for US only)*: **ABDALLAH, Mohammad** [US/US]; 3868 Suncrest Avenue, San Jose, CA 95132 (US).

(74) Agent: **BARNES, Glenn, D.**; Murabito Hao & Barnes LLP, Two North Market Street, Third Floor, San Jose, CA 95113 (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, 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, KM, 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, 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, MD, 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, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report (Rule 48.2(g))*

(54) **Title**: HARDWARE ACCELERATION COMPONENTS FOR TRANSLATING GUEST INSTRUCTIONS TO NATIVE INSTRUCTIONS

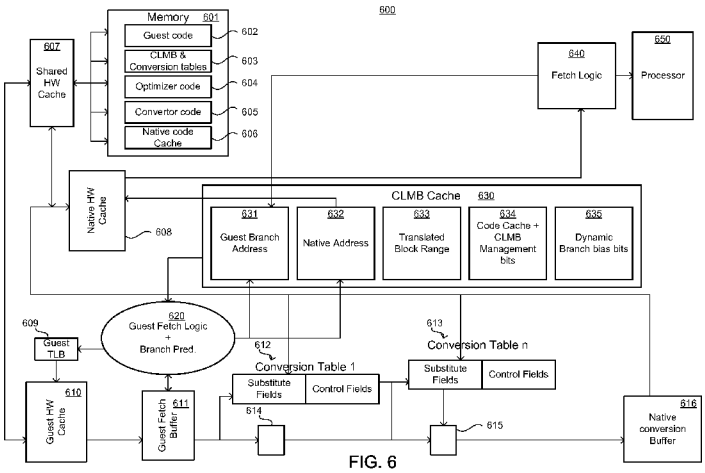

FIG. 6

(57) **Abstract**: A hardware based translation accelerator. The hardware includes a guest fetch logic component for accessing guest instructions; a guest fetch buffer coupled to the guest fetch logic component and a branch prediction component for assembling guest instructions into a guest instruction block; and conversion tables coupled to the guest fetch buffer for translating the guest instruction block into a corresponding native conversion block. The hardware further includes a native cache coupled to the conversion tables for storing the corresponding native conversion block, and a conversion look aside buffer coupled to the native cache for storing a mapping of the guest instruction block to corresponding native conversion block, wherein upon a subsequent request for a guest instruction, the conversion look aside buffer is indexed to determine whether a hit occurred, wherein the mapping indicates the guest instruction has a corresponding converted native instruction in the native cache.

WO 2012/103359 A2

HARDWARE ACCELERATION COMPONENTS FOR TRANSLATING GUEST INSTRUCTIONS TO NATIVE INSTRUCTIONS

This application claims the benefit co-pending commonly assigned US Provisional Patent Application serial number 61/436966, titled "HARDWARE ACCELERATION COMPONENTS FOR TRANSLATING GUEST INSTRUCTIONS TO NATIVE INSTRUCTIONS" by Mohammad A. Abdallah, filed on January 27, 2011, and which is incorporated herein in its entirety.

FIELD OF THE INVENTION

[001] The present invention is generally related to digital computer systems, more particularly, to a system and method for translating instructions comprising an instruction sequence.

BACKGROUND OF THE INVENTION

[001] Many types of digital computer systems utilize code transformation/translation or emulation to implement software-based functionality. Generally, translation and emulation both involve examining a program of software instructions and performing the functions and actions dictated by the software instructions, even though the instructions are not "native" to the computer system. In the case of translation, the non-native instructions are translated into a form of native instructions which are designed to execute on the hardware of the computer system. Examples include prior art translation software and/or hardware that operates with industry standard x86 applications to enable the applications to execute on non-x86 or alternative computer architectures. Generally, a translation process utilizes a large number of processor cycles, and thus, imposes a substantial amount of overhead. The performance penalty imposed by the overhead can substantially erode any benefits provided by the translation process.

[002] One attempt at solving this problem involves the use of just-in-time compilation. Just-in-time compilation (JIT), also known as dynamic translation, is a

method to improve the runtime performance of computer programs. Traditionally, computer programs had two modes of runtime transformation, either interpretation mode or JIT (Just-In-Time) compilation/translation mode. Interpretation is a decoding process that involves decoding instruction by instruction to transform the code from guest to native with lower overhead than JIT compilation, but it produces a transformed code that is less performing. Additionally, the interpretation is invoked with every instruction. JIT compilers or translators represent a contrasting approach to interpretation. With JIT conversion, it usually has a higher overhead than interpreters, but it produces a translated code that is more optimized and one that has higher execution performance. In most emulation implementations, the first time a translation is needed, it is done as an interpretation to reduce overhead, after the code is seen (executed) many times, a JIT translation is invoked to create a more optimized translation.

[003] However, the code transformation process still presents a number of problems. The JIT compilation process itself imposes a significant amount of overhead on the processor. This can cause a large delay in the start up of the application. Additionally, managing the storage of transformed code in system memory causes multiple trips back and forth to system memory and includes memory mapping and allocation management overhead, which imposes a significant latency penalty. Furthermore, changes to region of execution in the application involve relocating the transformed code in the system memory and code cache, and starting of the process from scratch. The interpretation process involves less overhead than JIT translation but it's overhead is repeated per instruction and thus is still relatively significant. The code produced is poorly optimized if at all.

SUMMARY OF THE INVENTION

[004] Embodiments of the present invention implement an algorithm and an apparatus that enables a hardware based acceleration of a guest instruction to native instruction translation process.

[005] In one embodiment, the present invention is implemented as a hardware based translation accelerator. The hardware based translation accelerator includes a guest fetch logic component for accessing a plurality of guest instructions; a guest fetch buffer coupled to the guest fetch logic component and a branch prediction component for assembling the plurality of guest instructions into a guest instruction block; and a plurality of conversion tables coupled to the guest fetch buffer for translating the guest instruction block into a corresponding native conversion block.

[006] The hardware based translation accelerator further includes a native cache coupled to the conversion tables for storing the corresponding native conversion block, and a conversion look aside buffer coupled to the native cache for storing a mapping of the guest instruction block to corresponding native conversion block, wherein upon a subsequent request for a guest instruction, the conversion look aside buffer is indexed to determine whether a hit occurred, wherein the mapping indicates the guest instruction has a corresponding converted native instruction in the native cache. In response to the hit the conversion look aside buffer forwards the translated native instruction for execution.

[007] The foregoing is a summary and thus contains, by necessity, simplifications, generalizations and omissions of detail; consequently, those skilled in the art will appreciate that the summary is illustrative only and is not intended to be in any way limiting. Other aspects, inventive features, and advantages of the present invention, as defined solely by the claims, will become apparent in the non-limiting detailed description set forth below.

BRIEF DESCRIPTION OF THE DRAWINGS

[001] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements.

[002] Figure 1 shows an exemplary sequence of instructions operated on by one embodiment of the present invention.

[003] Figure 2 shows a diagram depicting a block-based translation process where guest instruction blocks are converted to native conversion blocks in accordance with one embodiment of the present invention.

[004] Figure 3 shows a diagram illustrating the manner in which each instruction of a guest instruction block is converted to a corresponding native instruction of a native conversion block in accordance with one embodiment of the present invention.

[005] Figure 4 shows a diagram illustrating the manner in which far branches are processed with handling of native conversion blocks in accordance with one embodiment of the present invention.

[006] Figure 5 shows a diagram of an exemplary hardware accelerated conversion system illustrating the manner in which guest instruction blocks and their corresponding native conversion blocks are stored within a cache in accordance with one embodiment of the present invention.

[007] Figure 6 shows a more detailed example of a hardware accelerated conversion system in accordance with one embodiment of the present invention.

[008] Figure 7 shows an example of a hardware accelerated conversion system having a secondary software-based accelerated conversion pipeline in accordance with one embodiment of the present invention.

[009] Figure 8 shows an exemplary flow diagram illustrating the manner in which the CLB functions in conjunction with the code cache and the guest instruction to

native instruction mappings stored within memory in accordance with one embodiment of the present invention.

[010] Figure 9 shows an exemplary flow diagram illustrating a physical storage stack code cache implementation and the guest instruction to native instruction mappings in accordance with one embodiment of the present invention.

[011] Figure 10 shows a diagram depicting additional exemplary details of a hardware accelerated conversion system in accordance with one embodiment of the present invention.

[012] Figure 11A shows a diagram of an exemplary pattern matching process implemented by embodiments of the present invention.

[013] Figure 11B shows a diagram of a SIMD register-based pattern matching process in accordance with one embodiment of the present invention.

[014] Figure 12 shows a diagram of a unified register file in accordance with one embodiment of the present invention.

[015] Figure 13 shows a diagram of a unified shadow register file and pipeline architecture 1300 that supports speculative architectural states and transient architectural states in accordance with one embodiment of the present invention.

[016] Figure 14 shows a diagram of the second usage model, including dual scope usage in accordance with one embodiment of the present invention.

[017] Figure 15 shows a diagram of the third usage model, including transient context switching without the need to save and restore a prior context upon returning from the transient context in accordance with one embodiment of the present invention.

[018] Figure 16 shows an diagram depicting a case where the exception in the instruction sequence is because translation for subsequent code is needed in accordance with one embodiment of the present invention.

[019] Figure 17 shows a diagram of the fourth usage model, including transient context switching without the need to save and restore a prior context upon returning from the transient context in accordance with one embodiment of the present invention.

[020] Figure 18 shows a diagram of an exemplary microprocessor pipeline in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

[021] Although the present invention has been described in connection with one embodiment, the invention is not intended to be limited to the specific forms set forth herein. On the contrary, it is intended to cover such alternatives, modifications, and equivalents as can be reasonably included within the scope of the invention as defined by the appended claims.

[022] In the following detailed description, numerous specific details such as specific method orders, structures, elements, and connections have been set forth. It is to be understood however that these and other specific details need not be utilized to practice embodiments of the present invention. In other circumstances, well-known structures, elements, or connections have been omitted, or have not been described in particular detail in order to avoid unnecessarily obscuring this description.

[023] References within the specification to "one embodiment" or "an embodiment" are intended to indicate that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. The appearance of the phrase "in one embodiment" in various places within the specification are not necessarily all referring to the same embodiment, nor are separate or alternative embodiments mutually

exclusive of other embodiments. Moreover, various features are described which may be exhibited by some embodiments and not by others. Similarly, various requirements are described which may be requirements for some embodiments but not other embodiments.

[024] Some portions of the detailed descriptions, which follow, are presented in terms of procedures, steps, logic blocks, processing, and other symbolic representations of operations on data bits within a computer memory. These descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. A procedure, computer executed step, logic block, process, etc., is here, and generally, conceived to be a self-consistent sequence of steps or instructions leading to a desired result. The steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals of a computer readable storage medium and are capable of being stored, transferred, combined, compared, and otherwise manipulated in a computer system. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[025] 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 following discussions, it is appreciated that throughout the present invention, discussions utilizing terms such as "processing" or "accessing" or "writing" or "storing" or "replicating" or the like, 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 and other computer readable media into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[026] Embodiments of the present invention function by greatly accelerating the process of translating guest instructions from a guest instruction architecture into native instructions of a native instruction architecture for execution on a native processor. Embodiments of the present invention utilize hardware-based units to implement hardware acceleration for the conversion process. The guest instructions can be from a number of different instruction architectures. Example architectures include Java or JavaScript, x86, MIPS, SPARC, and the like. These guest instructions are rapidly converted into native instructions and pipelined to the native processor hardware for rapid execution. This provides a much higher level of performance in comparison to traditional software controlled conversion processes.

[027] In one embodiment, the present invention implements a flexible conversion process that can use as inputs a number of different instruction architectures. In such an embodiment, the front end of the processor is implemented such that it can be software controlled, while taking advantage of hardware accelerated conversion processing to deliver the much higher level of performance. Such an implementation delivers benefits on multiple fronts. Different guest architectures can be processed and converted while each receives the benefits of the hardware acceleration to enjoy a much higher level of performance. The software controlled front end can provide a great degree of flexibility for applications executing on the processor. The hardware acceleration can achieve near native hardware speed for execution of the guest instructions of a guest application. In the descriptions which follow, Figure 1 through Figure 4 shows the manner in which embodiments of the present invention handle guest instruction sequences and handle near branches and far branches within those guest instruction sequences. Figure 5 shows an overview of an exemplary hardware accelerated conversion processing system in accordance with one embodiment of the present invention.

[028] Figure 1 shows an exemplary sequence of instructions operated on by one embodiment of the present invention. As depicted in Figure 1, the instruction sequence 100 comprises 16 instructions, proceeding from the top of Figure 1 to the

bottom. As can be seen in Figure 1, the sequence 100 includes four branch instructions 101-104.

[029] One objective of embodiments of the present invention is to process entire groups of instructions as a single atomic unit. This atomic unit is referred to as a block. A block of instructions can extend well past the 16 instructions shown in Figure 1. In one embodiment, a block will include enough instructions to fill a fixed size (e.g., 64 bytes, 128 bytes, 256 bytes, or the like), or until an exit condition is encountered. In one embodiment, the exit condition for concluding a block of instructions is the encounter of a far branch instruction. As used herein in the descriptions of embodiments, a far branch refers to a branch instruction whose target address resides outside the current block of instructions. In other words, within a given guest instruction block, a far branch has a target that resides in some other block or in some other sequence of instructions outside the given instruction block. Similarly, a near branch refers to a branch instruction whose target address resides inside the current block of instructions. Additionally, it should be noted that a native instruction block can contain multiple guest far branches. These terms are further described in the discussions which follow below.

[030] Figure 2 shows a diagram depicting a block-based conversion process, where guest instruction blocks are converted to native conversion blocks in accordance with one embodiment of the present invention. As illustrated in Figure 2, a plurality of guest instruction blocks 201 are shown being converted to a corresponding plurality of native conversion blocks 202.

[031] Embodiments of the present invention function by converting instructions of a guest instruction block into corresponding instructions of a native conversion block. Each of the blocks 201 are made up of guest instructions. As described above, these guest instructions can be from a number of different guest instruction architectures (e.g., Java or JavaScript, x86, MIPS, SPARC, etc.). Multiple guest instruction blocks can be converted into one or more corresponding native conversion blocks. This conversion occurs on a per instruction basis.

[032] Figure 2 also illustrates the manner in which guest instruction blocks are assembled into sequences based upon a branch prediction. This attribute enables embodiments of the present invention to assemble sequences of guest instructions based upon the predicted outcomes of far branches. Based upon far branch prediction, a sequence of guest instructions is assembled from multiple guest instruction blocks and converted to a corresponding native conversion block. This aspect is further described in Figure 3 and Figure 4 below.

[033] Figure 3 shows a diagram illustrating the manner in which each instruction of a guest instruction block is converted to a corresponding native instruction of a native conversion block in accordance with one embodiment of the present invention. As illustrated in Figure 3, the guest instruction blocks reside within a guest instruction buffer 301. Similarly, the native conversion block(s) reside within a native instruction buffer 302.

[034] Figure 3 shows an attribute of embodiments of the present invention, where the target addresses of the guest branch instructions are converted to target addresses of the native branch instructions. For example, the guest instruction branches each include an offset that identifies the target address of the particular branch. This is shown in Figure 3 as the guests offset, or G_offset. As guest instructions are converted, this offset is often different because of the different lengths or sequences required by the native instructions to produce the functionality of the corresponding guest instructions. For example, the guest instructions may be of different lengths in comparison to their corresponding native instructions. Hence, the conversion process compensates for this difference by computing the corresponding native offset. This is shown in Figure 3 as the native offset, or N_offset.

[035] It should be noted that the branches that have targets within a guest instruction block, referred to as near branches, are not predicted, and therefore do not alter the flow of the instruction sequence.

[036] Figure 4 shows a diagram illustrating the manner in which far branches are processed with handling of native conversion blocks in accordance with one embodiment of the present invention. As illustrated in Figure 4, the guest instructions are depicted as a guest instruction sequence in memory 401. Similarly, the native instructions are depicted as a native instruction sequence in memory 402.

[037] In one embodiment, every instruction block, both guest instruction blocks and native instruction blocks, concludes with a far branch (e.g., even though native blocks can contain multiple guest far branches). As described above, a block will include enough instructions to fill a fixed size (e.g., 64 bytes, 128 bytes, 256 bytes, or the like) or until an exit condition, such as, for example, the last guest far branch instruction, is encountered. If a number of guest instructions have been processed to assemble a guest instruction block and a far branch has not been encountered, then a guest far branch is inserted to conclude the block. This far branch is merely a jump to the next subsequent block. This ensures that instruction blocks conclude with a branch that leads to either another native instruction block, or another sequence of guest instructions in memory. Additionally, as shown in Figure 4 a block can include a guest far branch within its sequence of instructions that does not reside at the end of the block. This is shown by the guest instruction far branch 411 and the corresponding native instruction guest far branch 412.

[038] In the Figure 4 embodiment, the far branch 411 is predicted taken. Thus the instruction sequence jumps to the target of the far branch 411, which is the guest instruction F. Similarly, in the corresponding native instructions, a far branch 412 is followed by the native instruction F. The near branches are not predicted. Thus, they do not alter the instruction sequence in the same manner as far branches.

[039] In this manner, embodiments of the present invention generate a trace of conversion blocks, where each block comprises a number (e.g., 3-4) of far branches. This trace is based on guest far branch predictions.

[040] In one embodiment, the far branches within the native conversion block include a guest address that is the opposite address for the opposing branch path. As described above, a sequence of instructions is generated based upon the prediction of far branches. The true outcome of the prediction will not be known until the corresponding native conversion block is executed. Thus, once a false prediction is detected, the false far branch is examined to obtain the opposite guest address for the opposing branch path. The conversion process then continues from the opposite guest address, which is now the true branch path. In this manner, embodiments of the present invention use the included opposite guest address for the opposing branch path to recover from occasions where the predicted outcome of a far branch is false. Hence, if a far branch predicted outcome is false, the process knows where to go to find the correct guest instruction. Similarly, if the far branch predicted outcome is true, the opposite guest address is ignored. It should be noted that if far branches within native instruction block are predicted correctly, no entry point in CLB for their target blocks is needed. However, once a miss prediction occurs, a new entry for the target block needs to be inserted in CLB. This function is performed with the goal of preserving CLB capacity.

[041] Figure 5 shows a diagram of an exemplary hardware accelerated conversion system 500 illustrating the manner in which guest instruction blocks and their corresponding native conversion blocks are stored within a cache in accordance with one embodiment of the present invention. As illustrated in Figure 5, a conversion look aside buffer 506 is used to cache the address mappings between guest and native blocks ; such that the most frequently encountered native conversion blocks are accessed through low latency availability to the processor 508.

[042] The Figure 5 diagram illustrates the manner in which frequently encountered native conversion blocks are maintained within a high-speed low latency cache, the conversion look aside buffer 506. The components depicted in Figure 5 implement hardware accelerated conversion processing to deliver the much higher level of performance.

[043] The guest fetch logic unit 502 functions as a hardware-based guest instruction fetch unit that fetches guest instructions from the system memory 501. Guest instructions of a given application reside within system memory 501. Upon initiation of a program, the hardware-based guest fetch logic unit 502 starts prefetching guess instructions into a guest fetch buffer 503. The guest fetch buffer 507 accumulates the guest instructions and assembles them into guest instruction blocks. These guest instruction blocks are converted to corresponding native conversion blocks by using the conversion tables 504. The converted native instructions are accumulated within the native conversion buffer 505 until the native conversion block is complete. The native conversion block is then transferred to the native cache 507 and the mappings are stored in the conversion look aside buffer 506. The native cache 507 is then used to feed native instructions to the processor 508 for execution. In one embodiment, the functionality implemented by the guest fetch logic unit 502 is produced by a guest fetch logic state machine.

[044] As this process continues, the conversion look aside buffer 506 is filled with address mappings of guest blocks to native blocks. The conversion look aside buffer 506 uses one or more algorithms (e.g., least recently used, etc.) to ensure that block mappings that are encountered more frequently are kept within the buffer, while block mappings that are rarely encountered are evicted from the buffer. In this manner, hot native conversion blocks mappings are stored within the conversion look aside buffer 506. In addition, it should be noted that the well predicted far guest branches within the native block do not need to insert new mappings in the CLB because their target blocks are stitched within a single mapped native block, thus preserving a small capacity efficiency for the CLB structure. Furthermore, in one embodiment, the CLB is structured to store only the ending guest to native address mappings. This aspect also preserves the small capacity efficiency of the CLB.

[045] The guest fetch logic 502 looks to the conversion look aside buffer 506 to determine whether addresses from a guest instruction block have already been converted to a native conversion block. As described above, embodiments of the present invention provide hardware acceleration for conversion processing. Hence, the

guest fetch logic 502 will look to the conversion look aside buffer 506 for pre-existing native conversion block mappings prior to fetching a guest address from system memory 501 for a new conversion.

[046] In one embodiment, the conversion look aside buffer is indexed by guest address ranges, or by individual guest address. The guest address ranges are the ranges of addresses of guest instruction blocks that have been converted to native conversion blocks. The native conversion block mappings stored by a conversion look aside buffer are indexed via their corresponding guest address range of the corresponding guest instruction block. Hence, the guest fetch logic can compare a guest address with the guest address ranges or the individual guest address of converted blocks, the mappings of which are kept in the conversion look aside buffer 506 to determine whether a pre-existing native conversion block resides within what is stored in the native cache 507 or in the code cache of Figure 6. If the pre-existing native conversion block is in either of the native cache or in the code cache, the corresponding native conversion instructions are forwarded from those caches directly to the processor.

[047] In this manner, hot guest instruction blocks (e.g., guest instruction blocks that are frequently executed) have their corresponding hot native conversion blocks mappings maintained within the high-speed low latency conversion look aside buffer 506. As blocks are touched, an appropriate replacement policy ensures that the hot blocks mappings remain within the conversion look aside buffer. Hence, the guest fetch logic 502 can quickly identify whether requested guest addresses have been previously converted, and can forward the previously converted native instructions directly to the native cache 507 for execution by the processor 508. These aspects save a large number of cycles, since trips to system memory can take 40 to 50 cycles or more. These attributes (e.g., CLB, guest branch sequence prediction, guest & native branch buffers, native caching of the prior) allow the hardware acceleration functionality of embodiments of the present invention to achieve application performance of a guest application to within 80% to 100% the application performance of a comparable native application.

[048] In one embodiment, the guest fetch logic 502 continually prefetches guest instructions for conversion independent of guest instruction requests from the processor 508. Native conversion blocks can be accumulated within a conversion buffer "code cache" in the system memory 501 for those less frequently used blocks. The conversion look aside buffer 506 also keeps the most frequently used mappings. Thus, if a requested guest address does not map to a guest address in the conversion look aside buffer, the guest fetch logic can check system memory 501 to determine if the guest address corresponds to a native conversion block stored therein.

[049] In one embodiment, the conversion look aside buffer 506 is implemented as a cache and utilizes cache coherency protocols to maintain coherency with a much larger conversion buffer stored in higher levels of cache and system memory 501. The native instructions mappings that are stored within the conversion look aside buffer 506 are also written back to higher levels of cache and system memory 501. Write backs to system memory maintain coherency. Hence, cache management protocols can be used to ensure the hot native conversion blocks mappings are stored within the conversion look aside buffer 506 and the cold native conversion mappings blocks are stored in the system memory 501. Hence, a much larger form of the conversion buffer 506 resides in system memory 501.

[050] It should be noted that in one embodiment, the exemplary hardware accelerated conversion system 500 can be used to implement a number of different virtual storage schemes. For example, the manner in which guest instruction blocks and their corresponding native conversion blocks are stored within a cache can be used to support a virtual storage scheme. Similarly, a conversion look aside buffer 506 that is used to cache the address mappings between guest and native blocks can be used to support the virtual storage scheme (e.g., management of virtual to physical memory mappings).

[051] In one embodiment, the Figure 5 architecture implements virtual instruction set processor/computer that uses a flexible conversion process that can receive as inputs a number of different instruction architectures. In such a virtual

instruction set processor, the front end of the processor is implemented such that it can be software controlled, while taking advantage of hardware accelerated conversion processing to deliver the much higher level of performance. Using such an implementation, different guest architectures can be processed and converted while each receives the benefits of the hardware acceleration to enjoy a much higher level of performance. Example guest architectures include Java or JavaScript, x86, MIPS, SPARC, and the like. In one embodiment, the "guest architecture" can be native instructions (e.g., from a native application/macro-operation) and the conversion process produces optimize native instructions (e.g., optimized native instructions/micro-operations). The software controlled front end can provide a large degree of flexibility for applications executing on the processor. As described above, the hardware acceleration can achieve near native hardware speed for execution of the guest instructions of a guest application.

[052] Figure 6 shows a more detailed example of a hardware accelerated conversion system 600 in accordance with one embodiment of the present invention. System 600 performers in substantially the same manner as system 500 described above. However, system 600 shows additional details describing functionality of an exemplary hardware acceleration process.

[053] The system memory 601 includes the data structures comprising the guest code 602, the conversion look aside buffer 603, optimizer code 604, converter code 605, and native code cache 606. System 600 also shows a shared hardware cache 607 where guest instructions and native instructions can both be interleaved and shared. The guest hardware cache 610 catches those guest instructions that are most frequently touched from the shared hardware cache 607.

[054] The guest fetch logic 620 prefetches guest instructions from the guest code 602. The guest fetch logic 620 interfaces with a TLB 609 which functions as a conversion look aside buffer that translates virtual guest addresses into corresponding physical guest addresses. The TLB 609 can forward hits directly to the guest hardware

cache 610. Guest instructions that are fetched by the guest fetch logic 620 are stored in the guest fetch buffer 611.

[055] The conversion tables 612 and 613 include substitute fields and control fields and function as multilevel conversion tables for translating guest instructions received from the guest fetch buffer 611 into native instructions.

[056] The multiplexers 614 and 615 transfer the converted native instructions to a native conversion buffer 616. The native conversion buffer 616 accumulates the converted native instructions to assemble native conversion blocks. These native conversion blocks are then transferred to the native hardware cache 600 and the mappings are kept in the conversion look aside buffer 630.

[057] The conversion look aside buffer 630 includes the data structures for the converted blocks entry point address 631, the native address 632, the converted address range 633, the code cache and conversion look aside buffer management bits 634, and the dynamic branch bias bits 635. The guest branch address 631 and the native address 632 comprise a guest address range that indicates which corresponding native conversion blocks reside within the converted lock range 633. Cache management protocols and replacement policies ensure the hot native conversion blocks mappings reside within the conversion look aside buffer 630 while the cold native conversion blocks mappings reside within the conversion look aside buffer data structure 603 in system memory 601.

[058] As with system 500, system 600 seeks to ensure the hot blocks mappings reside within the high-speed low latency conversion look aside buffer 630. Thus, when the fetch logic 640 or the guest fetch logic 620 looks to fetch a guest address, in one embodiment, the fetch logic 640 can first check the guest address to determine whether the corresponding native conversion block resides within the code cache 606. This allows a determination as to whether the requested guest address has a corresponding native conversion block in the code cache 606. If the requested guest address does not reside within either the buffer 603 or 608, or the buffer 630, the guest address and a

number of subsequent guest instructions are fetched from the guest code 602 and the conversion process is implemented via the conversion tables 612 and 613.

[059] Figure 7 shows an example of a hardware accelerated conversion system 700 having a secondary software-based accelerated conversion pipeline in accordance with one embodiment of the present invention.

[060] The components 711-716 comprise a software implemented load store path that is instantiated within a specialized high speed memory 760. As depicted in Figure 7, the guest fetch buffer 711, conversion tables 712-713 and native conversion buffer 716 comprise allocated portions of the specialized high speed memory 760. In many respects, the specialized high-speed memory 760 functions as a very low-level fast cache (e.g., L0 cache).

[061] The arrow 761 illustrates the attribute whereby the conversions are accelerated via a load store path as opposed to an instruction fetch path (e.g., from the fetched decode logic).

[062] In the Figure 7 embodiment, the high-speed memory 760 includes special logic for doing comparisons. Because of this, the conversion acceleration can be implemented in software. For example, in another embodiment, the standard memory 760 that stores the components 711-716 is manipulated by software which uses a processor execution pipeline, where it loads values from said components 711-716 into one or more SIMD register(s) and implements a compare instruction that performs a compare between the fields in the SIMD register and, as needed, perform a mask operation and a result scan operation. A load store path can be implemented using general purpose microprocessor hardware, such as, for example, using compare instructions that compare one to many.

[063] It should be noted that the memory 760 is accessed by instructions that have special attributes or address ranges. For example, in one embodiment, the guest fetch buffer has an ID for each guest instruction entry. The ID is created per guest

instruction. This ID allows easy mapping from the guest buffer to the native conversion buffer. The ID allows an easy calculation of the guest offset to the native offset, irrespective of the different lengths of the guest instructions in comparison to the corresponding native instructions. This aspect is diagramed in Figure 3 above.

[064] In one embodiment the ID is calculated by hardware using a length decoder that calculates the length of the fetched guest instruction. However, it should be noted that this functionality can be performed in hardware or software.

[065] Once IDs have been assigned, the native instructions buffer can be accessed via the ID. The ID allows the conversion of the offset from guest offset to the native offset.

[066] Figure 8 shows an exemplary flow diagram illustrating the manner in which the CLB functions in conjunction with the code cache and the guest instruction to native instruction mappings stored within memory in accordance with one embodiment of the present invention.

[067] As described above, the CLB is used to store mappings of guest addresses that have corresponding converted native addresses stored within the code cache memory (e.g., the guest to native address mappings). In one embodiment, the CLB is indexed with a portion of the guest address. The guest address is partitioned into an index, a tag, and an offset (e.g., chunk size). This guest address comprises a tag that is used to identify a match in the CLB entry that corresponds to the index. If there is a hit on the tag, the corresponding entry will store a pointer that indicates where in the code cache memory 806 the corresponding converted native instruction chunk (e.g., the corresponding block of converted native instructions) can be found.

[068] It should be noted that the term "chunk" as used herein refers to a corresponding memory size of the converted native instruction block. For example, chunks can be different in size depending on the different sizes of the converted native instruction blocks.

[069] With respect to the code cache memory 806, in one embodiment, the code cache is allocated in a set of fixed size chunks (e.g., with different size for each chunk type). The code cache can be partitioned logically into sets and ways in system memory and all lower level HW caches (e.g., native hardware cache 608, shared hardware cache 607). The CLB can use the guest address to index and tag compare the way tags for the code cache chunks.

[070] Figure 8 depicts the CLB hardware cache 804 storing guest address tags in 2 ways, depicted as way x and way y. It should be noted that, in one embodiment, the mapping of guest addresses to native addresses using the CLB structures can be done through storing the pointers to the native code chunks (e.g., from the guest to native address mappings) in the structured ways. Each way is associated with a tag. The CLB is indexed with the guest address 802 (comprising a tag). On a hit in the CLB, the pointer corresponding to the tag is returned. This pointer is used to index the code cache memory. This is shown in Figure 8 by the line "native address of code chunk=Seg#+F(pt)" which represents the fact that the native address of the code chunk is a function of the pointer and the segment number. In the present embodiment, the segment refers to a base for a point in memory where the pointer scope is virtually mapped (e.g., allowing the pointer array to be mapped into any region in the physical memory).

[071] Alternatively, in one embodiment, the code cache memory can be indexed via a second method, as shown in Figure 8 by the line "Native Address of code chunk =seg#+ Index * (size of chunk) + way# * (Chunk size)". In such an embodiment, the code cache is organized such that its way-structures match the CLB way structuring so that a 1:1 mapping exist between the ways of CLB and the ways of the code cache chunks. When there is a hit in a particular CLB way then the corresponding code chunk in the corresponding way of the code cache has the native code.

[072] Referring still to Figure 8, if the index of the CLB misses, the higher hierarchies of memory can be checked for a hit (e.g., L1 cache, L2 cache, and the like). If there is no hit in these higher cache levels, the addresses in the system memory 801

are checked. In one embodiment, the guest index points to a entry comprising, for example, 64 chunks. The tags of each one of the 64 chunks are read out and compared against the guest tag to determine whether there is a hit. This process is shown in Figure 8 by the dotted box 805. If there is no hit after the comparison with the tags in system memory, there is no conversion present at any hierarchical level of memory, and the guest instruction must be converted.

[073] It should be noted that embodiments of the present invention manage each of the hierarchical levels of memory that store the guest to native instruction mappings in a cache like manner. This comes inherently from cache-based memory (e.g., the CLB hardware cache, the native cache, L1 and L2 caches, and the like). However, the CLB also includes "code cache + CLB management bits" that are used to implement a least recently used (LRU) replacement management policy for the guest to native instruction mappings within system memory 801. In one embodiment, the CLB management bits (e.g., the LRU bits) are software managed. In this manner, all hierarchical levels of memory are used to store the most recently used, most frequently encountered guest to native instruction mappings. Correspondingly, this leads to all hierarchical levels of memory similarly storing the most frequently encountered converted native instructions.

[074] Figure 8 also shows dynamic branch bias bits and/or branch history bits stored in the CLB. These dynamic branch bits are used to track the behavior of branch predictions used in assembling guest instruction sequences. These bits are used to track which branch predictions are most often correctly predicted and which branch predictions are most often predicted incorrectly. The CLB also stores data for converted block ranges. This data enables the process to invalidate the converted block range in the code cache memory where the corresponding guest instructions have been modified (e.g., as in self modifying code).

[075] Figure 9 shows an exemplary flow diagram illustrating a physical storage stack cache implementation and the guest address to native address mappings in

accordance with one embodiment of the present invention. As depicted in Figure 9, the cache can be implemented as a physical storage stack 901.

[076] Figure 9 embodiment illustrates the manner in which a code cache can be implemented as a variable structure cache. Depending upon the requirements of different embodiments, the variable structure cache can be completely hardware implemented and controlled, completely software implemented and controlled, or some mixture of software intimidation and control and underlying hardware enablement.

[077] The Figure 9 embodiment is directed towards striking an optimal balance for the task of managing the allocation and replacement of the guest to native address mappings and their corresponding translations in the actual physical storage. In the present embodiment, this is accomplished through the use of a structure that combines the pointers with variable size chunks.

[078] A multi-way tag array is used to store pointers for different size groups of physical storage. Each time a particular storage size needs to be allocated (e.g., where the storage size corresponds to an address), then accordingly, a group of storage blocks each corresponding to that size is allocated. This allows an embodiment of the present invention to precisely allocate storage to store variable size traces of instructions. Figure 9 shows how groups can be of different sizes. Two exemplary group sizes are shown, “replacement candidate for group size 4" and "replacement candidate for group size 2". A pointer is stored in the TAG array (in addition to the tag that correspond to the address) that maps the address into the physical storage address. The tags can comprise two or more sub-tags. For example, the top 3 tags in the tag structure 902 comprise sub tags A1 B1, A2 B2 C2 D2, and A3 B3 respectively as shown. Hence, tag A2 B2 C2 D2 comprises a group size 4, while tag A1 B1 comprises a group size 2. The group size mask also indicates the size of the group.

[079] The physical storage can then be managed like a stack, such that every time there is a new group allocated, it can be placed on top of the physical storage stack. Entries are invalidated by overwriting their tag, thereby recovering the allocated space.

[080] Figure 9 also shows an extended way tag structure 903. In some circumstances, an entry in the tag structure 902 will have a corresponding entry in the extended way tag structure 903. This depends on upon whether the entry and the tag structure has an extended way bit set (e.g., set to one). For example, the extended way bit set to one indicates that there are corresponding entries in the extended way tag structure. The extended way tag structure allows the processor to extend locality of reference in a different way from the standard tag structure. Thus, although the tag structure 902 is indexed in one manner (e.g., index (j)), the extended way tag structure is indexed in a different manner (e.g., index (k)).

[081] In a typical implementation, the index(J) can be much larger number of entries within the index(k). This is because, in most limitations, the primary tag structure 902 is much larger than the extended way tag structure 903, where, for example, (j) can cover 1024 entries (e.g., 10 bits) while (k) can cover 256 (e.g., 8 bits).

[082] This enables embodiments of the present invention to incorporate additional ways for matching traces that have become very hot (e.g., very frequently encountered). For example, if a match within a hot set is not found in the tag structure 902, then by setting an extended way bit, the extended way tag structure can be used to store additional ways for the hot trace. It should be noted that this variable cache structure uses storage only as needed for the cached code/data that we store on the stack, for example, if any of the cache sets (the entries indicated by the index bits) is never accessed during a particular phase of a program, then there will be no storage allocation for that set on the stack. This provides an efficient effective storage capacity increase compared to typical caches where sets have fixed physical data storage for each and every set.

[083] There can be also bits to indicate that a set or group of sets are cold (e.g., meaning they have not been accesses in a long time). In this case the stack storage for those sets looks like bubbles within the allocated stack storage. At that time, their allocation pointers can be claimed for other hot sets. This process is a storage reclamation process, where after a chunk has been allocated within the stack, the whole

set to which that chunk belongs become later cold. The needed mechanisms and structures (not shown in Figure 9 in order not to clutter or obscure the aspects shown) that can facilitate this reclamation are: a cold set indicator for every set (entry index) and a reclamation process where the pointers for the ways of those cold sets are reused for other hot set's ways. This allows those stack storage bubbles (chunks) to be reclaimed. When not in reclamation mode, a new chunk is allocated on top of the stack, when the stack has cold sets (e.g., the set ways/chunks are not accessed in a long time) a reclamation action allow a new chunk that needs to be allocated in another set to reuse the reclaimed pointer and its associated chunk storage (that belongs to a cold set) within the stack.

[084] It should be noted that the Figure 9 embodiment is well-suited to use standard memory in its implementation as opposed to specialized cache memory. This attribute is due to the fact that the physical storage stack is managed by reading the pointers, reading indexes, and allocating address ranges. Specialized cache-based circuit structures are not needed in such an implementation.

[085] It should be noted that in one embodiment, the Figure 9 architecture can be used to implement data caches and caching schemes that do not involve conversion or code transformation. Consequently, the Figure 9 architecture can be used to implement more standardized caches (e.g., L2 data cache, etc.). Doing so would provide a larger effective capacity in comparison to a conventional fixed structure cache, or the like.

[086] Figure 10 shows a diagram depicting additional exemplary details of a hardware accelerated conversion system 1000 in accordance with one embodiment of the present invention. The line 1001 illustrates the manner in which incoming guests instructions are compared against a plurality of group masks and tags. The objective is to quickly identify the type of guest instruction and assign it to a corresponding group. The group masks and tags function by matching subfields of the guest instruction in order to identify particular groups to which the guest instruction belongs. The mask obscures irrelevant bits of the guest instruction pattern to look particularly at the

relevant bits. The tables, such as for example, table 1002, stores the mask-tag pairs in a prioritized manner.

[087] A pattern is matched by reading into the table in the priority direction, which is depicted in this case being from the top down. In this manner, a pattern is matched by reading in the priority direction of the mask-tag storage. The different masks examined in order of their priority and the pattern matching functionality is correspondingly applied in order of their priority. When a hit is found, then the corresponding mapping of the pattern is read from a corresponding table storing the mappings (e.g., table 1003). The 2nd level tables 1004 illustrates the hierarchical manner in which multiple conversion tables can be accessed in a cascading sequential manner until a full conversion of the guest instruction is achieved. As described above, the conversion tables include substitute fields and control fields and function as multilevel conversion tables for translating guest instructions received from the guest fetch buffer into native instructions.

[088] In this manner, each byte stream in the buffer sent to conversion tables where each level of conversion table serially detects bit fields. As the relevant bit fields are detected, the table substitutes the native equivalence of the field.

[089] The table also produces a control field that helps the substitution process for this level as well as the next level table (e.g., the 2nd level table 1004). The next table uses the previous table control filed to identify next relevant bit field, which is in substituted with the native equivalence. The second level table can then produce control field to help a first level table, and so on. Once all guest bit fields are substituted with native bit fields, the instruction is fully translated and is transmitted to the native conversion buffer. The native conversion buffer is then written into the code cache and its guest to native address mappings are logged in the CLB, as described above.

[090] Figure 11A shows a diagram of an exemplary pattern matching process implemented by embodiments of the present invention. As depicted in Figure 11A,

destination is determined by the tag, the pattern, and the mask. The functionality of the pattern decoding comprises performing a bit compare (e.g., bitwise XOR), performing a bit AND (e.g., bitwise AND), and subsequently checking all zero bits (e.g., NOR of all bits).

[091] Figure 11B shows a diagram 1100 of a SIMD register based pattern matching process in accordance with one embodiment of the present invention. As depicted in diagram 1100, four SIMD registers 1102-1105 are shown. These registers implement the functionality of the pattern decoding process as shown. An incoming pattern 1101 is used to perform a parallel bit compare (e.g., bitwise XOR) on each of the tags, and the result performs a bit AND with the mask (e.g., bitwise AND). The match indicator results are each stored in their respective SIMD locations as shown. A scan is then performed as shown, and the first true among the SIMD elements encountered by the scan is the element where the equation (Pi XOR Ti) AND Mi = 0 for all i bits is true, where Pi is the respective pattern, Ti is the respective tag and Mi is the respective mask.

[092] Figure 12 shows a diagram of a unified register file 1201 in accordance with one embodiment of the present invention. As depicted in Figure 12, the unified register file 1201 includes 2 portions 1202-1203 and an entry selector 1205. The unified register file 1201 implements support for architecture speculation for hardware state updates.

[093] The unified register file 1201 enables the implementation of an optimized shadow register and committed register state management process. This process supports architecture speculation for hardware state updating. Under this process, embodiments of the present invention can support shadow register functionality and committed register functionality without requiring any cross copying between register memory. For example, in one embodiment, the functionality of the unified register file 1201 is largely provided by the entry selector 1205. In the Figure 12 embodiment, each register file entry is composed from 2 pairs of registers, R & R', which are from portion 1 and the portion 2, respectively. At any given time, the register

that is read from each entry is either R or R', from portion 1 or portion 2. There are 4 different combinations for each entry of the register file based on the values of x & y bits stored for each entry by the entry selector 105.

[094] The values for the x & y bits are as follows.

00 : R not Valid;	R' committed	(upon read request R' is read)
01: R speculative;	R' committed	(upon read request R is read)
10: R committed ;	R' speculative	(upon read request R' is read)
11: R committed ;	R' not Valid	(upon read request R is read)

[095] The following are the impact of each instruction/event. Upon Instruction Write back, 00 becomes 01 and 11 becomes 10. Upon instruction commit, 01 becomes 11 and 10 becomes 00. Upon the occurrence of a rollback event, 01 becomes 00 and 10 becomes 11.

[096] These changes are mainly changes to the state stored in the register file entry selector 1205 and happen based on the events as they occur. It should be noted that commit instructions and roll back events need to reach a commit stage in order to cause the bit transition in the entry selector 1205.

[097] In this manner, execution is able to proceed within the shadow register state without destroying the committed register state. When the shadow register state is ready for committing, the register file entry selector is updated such that the valid results are read from which portion in the manner described above. In this manner, by simply updating the register file entry selector as needed, speculative execution results can be rolled back to most recent commit point in the event of an exception. Similarly, the commit point can be advanced forward, thereby committing the speculative execution results, by simply updating the register file entry selectors. This functionality is provided without requiring any cross copying between register memory.

[098] In this manner, the unified register file can implement a plurality of speculative scratch shadow registers (SSSR) and a plurality of committed registers (CR) via the register file entry selector 1205. For example, on a commit, the SSSR registers become CR registers. On roll back SSSR state is rolled back to the CR registers.

[099] Figure 13 shows a diagram of a unified shadow register file and pipeline architecture 1300 that supports speculative architectural states and transient architectural states in accordance with one embodiment of the present invention.

[0100] The Figure 13 embodiment depicts the components comprising the architecture 1300 that supports instructions and results comprising architecture speculation states and supports instructions and results comprising transient states. As used herein, a committed architecture state comprises visible registers and visible memory that can be accessed (e.g., read and write) by programs executing on the processor. In contrast, a speculative architecture state comprises registers and/or memory that is not committed and therefore is not globally visible.

[0101] In one embodiment, there are four usage models that are enabled by the architecture 1300. A first usage model includes architecture speculation for hardware state updates, as described above in the discussion of Figure 12.

[0102] A second usage model includes dual scope usage. This usage model applies to the fetching of 2 threads into the processor, where one thread executes in a speculative state and the other thread executes in the non-speculative state. In this usage model, both scopes are fetched into the machine and are present in the machine at the same time.

[0103] A third usage model includes the JIT (just-in-time) translation or compilation of instructions from one form to another. In this usage model, the reordering of architectural states is accomplished via software, for example, the JIT. The third usage model can apply to, for example, guest to native instruction translation,

virtual machine to native instruction translation, or remapping/translating native micro instructions into more optimized native micro instructions.

[0104] A fourth usage model includes transient context switching without the need to save and restore a prior context upon returning from the transient context. This usage model applies to context switches that may occur for a number of reasons. One such reason could be, for example, the precise handling of exceptions via an exception handling context. The second, third, and fourth usage models are further described in the discussions of Figures 14-17 below.

[0105] Referring again to Figure 13, the architecture 1300 includes a number of components for implementing the 4 usage models described above. The unified shadow register file 1301 includes a first portion, committed register file 1302, a second portion, the shadow register file 1303, and a third portion, the latest indicator array 1304. A speculative retirement memory buffer 1342 and a latest indicator array 1340 are included. The architecture 1300 comprises an out of order architecture, hence, the architecture 1300 further includes a reorder buffer and retirement window 1332. The reorder and retirement window 1332 further includes a machine retirement pointer 1331, a ready bit array 1334 and a per instruction latest indicator, such as indicator 1333.

[0106] The first usage model, architecture speculation for hardware state updates, is further described in detail in accordance with one embodiment of the present invention. As described above, the architecture 1300 comprises a out of order architecture. The hardware of the architecture 1300 able to commit out of order instruction results (e.g., out of order loads and out of order stores and out of order register updates). The architecture 1300 utilizes the unified shadow register file in the manner described in discussion of Figure 12 above to support speculative execution between committed registers and shadow registers. Additionally, the architecture 1300 utilizes the speculative load store buffer 1320 and the speculative retirement memory buffer 1342 to support speculative execution.

[0107] The architecture 1300 will use these components in conjunction with reorder buffer and retirement window 1332 to allow its state to retire correctly to the committed register file 1302 and to the visible memory 1350 even though the machine retired those in out of order manner internally to the unified shadow register file and the retirement memory buffer. For example, the architecture will use the unified shadow register file 1301 and the speculative memory 1342 to implement rollback and commit events based upon whether exceptions occur or do not occur. This functionality enables the register state to retire out of order to the unified shadow register file 1301 and enables the speculative retirement memory buffer 1342 to retire out of order to the visible memory 1350. As speculative execution proceeds and out of order instruction execution proceeds, if no branch has been missed predicted and there are no exceptions that occur, the machine retirement pointer 1331 advances until a commit event is triggered. The commit event causes the unified shadow register file to commit its contents by advancing its commit point and causes the speculative retirement memory buffer to commit its contents to the memory 1350 in accordance with the machine retirement pointer 1331.

[0108] For example, considering the instructions 1-7 that are shown within the reorder buffer and retirement window 1332, the ready bit array 1334 shows an "X" beside instructions are ready to execute and a "/" beside instructions that are not ready to execute. Accordingly, instructions 1, 2, 4, and 6 are allowed to proceed out of order. Subsequently, if an exception occurs, such as the instruction 6 branch being miss-predicted, the instructions that occur subsequent to instruction 6 can be rolled back. Alternatively, if no exception occurs, all of the instructions 1-7 can be committed by moving the machine retirement pointer 1331 accordingly.

[0109] The latest indicator array 1341, the latest indicator array 1304 and the latest indicator 1333 are used to allow out of order execution. For example, even though instruction 2 loads register R4 before instruction5, the load from instruction 2 will be ignored once the instruction 5 is ready to occur. The latest load will override the earlier load in accordance with the latest indicator.

[0110] In the event of a branch prediction or exception occurring within the reorder buffer and retirement window 1332, a rollback event is triggered. As described above, in the event of a rollback, the unified shadow register file 1301 will rollback to its last committed point and the speculative retirement memory buffer 1342 will be flushed.

[0111] Figure 14 shows a diagram 1400 of the second usage model, including dual scope usage in accordance with one embodiment of the present invention. As described above, this usage model applies to the fetching of 2 threads into the processor, where one thread executes in a speculative state and the other thread executes in the non-speculative state. In this usage model, both scopes are fetched into the machine and are present in the machine at the same time.

[0112] As shown in diagram 1400, 2 scope/traces 1401 and 1402 have been fetched into the machine. In this example, the scope/trace 1401 is a current non-speculative scope/trace. The scope/trace 1402 is a new speculative scope/trace. Architecture 1300 enables a speculative and scratch state that allows 2 threads to use those states for execution. One thread (e.g., 1401) executes in a non-speculative scope and the other thread (e.g., 1402) uses the speculative scope. Both scopes can be fetched into the machine and be present at the same time, with each scope set its respective mode differently. The first is non-speculative and the other is speculative. So the first executes in CR/CM mode and the other executes in SR/SM mode. In the CR/CM mode, committed registers are read and written to, and memory writes go to memory. In the SR/SM mode, register writes go to SSSR, and register reads come from the latest write, while memory writes the retirement memory buffer (SMB).

[0113] One example will be a current scope that is ordered (e.g., 1401) and a next scope that is speculative (e.g., 1402). Both can be executed in the machine as dependencies will be honored because the next scope is fetched after the current scope. For example, in scope 1401, at the "commit SSSR to CR", registers and memory up to this point are in CR mode while the code executes in CR/CM mode. In scope 1402, the code executes in SR and SM mode and can be rolled back if an exception happens. In

this manner, both scopes execute at the same time in the machine but each is executing in a different mode and reading and writing registers accordingly.

[0114] Figure 15 shows a diagram 1500 of the third usage model, including transient context switching without the need to save and restore a prior context upon returning from the transient context in accordance with one embodiment of the present invention. As described above, this usage model applies to context switches that may occur for a number of reasons. One such reason could be, for example, the precise handling of exceptions via an exception handling context.

[0115] In the 3rd usage model occurs when the machine is executing translated code and it encounters a context switch (e.g., exception inside of the translated code or if translation for subsequent code is needed). In the current scope (e.g., prior to the exception), SSSR and the SMB have not yet committed their speculative state to the guest architecture state. The current state is running in SR/SM mode. When the exception occurs the machine switches to an exception handler (e.g., a convertor) to take care of exception precisely. A rollback is inserted, which causes the register state to roll back to CR and the SMB is flushed. The convertor code will run in SR/CM mode. During execution of convertor code the SMB is retiring its content to memory without waiting for a commit event. The registers are written to SSSR without updating CR. Subsequently, when the convertor is finished and before switching back to executing converted code, it rolls back the SSSR (e.g., SSSR is rolled back to CR). During this process the last committed Register state is in CR.

[0116] This is shown in diagram 1500 where the previous scope/trace 1501 has committed from SSSR into CR. The current scope/trace 1502 is speculative. Registers and memory and this scope are speculative and execution occurs under SR/SM mode. In this example, an exception occurs in the scope 1502 and the code needs to be re-executed in the original order before translation. At this point, SSSR is rolled back and the SMB is flushed. Then the JIT code 1503 executes. The JIT code rolls back SSSR to the end of scope 1501 and flushes the SMB. Execution of the JIT is under SC/CM mode. When the JIT is finished, the SSSR is rolled back to CR and the current

scope/trace 1504 then re-executes in the original translation order in CR/CM mode. In this manner, the exception is handled precisely at the exact current order.

[0117] Figure 16 shows a diagram 1600 depicting a case where the exception in the instruction sequence is because translation for subsequent code is needed in accordance with one embodiment of the present invention. As shown in diagram 1600, the previous scope/trace 1601 concludes with a far jump to a destination that is not translated. Before jumping to a far jump destination, SSSR is committed to CR. The JIT code 1602 then executes to translate the guess instructions at the far jump destination (e.g., to build a new trace of native instructions). Execution of the JIT is under SR/CM mode. At the conclusion of JIT execution, the register state is rolled back from SSSR to CR, and the new scope/trace 1603 that was translated by the JIT begins execution. The new scope/trace continues execution from the last committed point of the previous scope/trace 1601 in the SR/SM mode.

[0118] Figure 17 shows a diagram 1700 of the fourth usage model, including transient context switching without the need to save and restore a prior context upon returning from the transient context in accordance with one embodiment of the present invention. As described above, this usage model applies to context switches that may occur for a number of reasons. One such reason could be, for example, the processing inputs or outputs via an exception handling context.

[0119] Diagram 1700 shows a case where a previous scope/trace 1701 executing under CR/CM mode ends with a call of function F1. Register state up to that point is committed from SSSR to CR. The function F1 scope/trace 1702 then begins executing speculatively under SR/CM mode. The function F1 then ends with a return to the main scope/trace 1703. At this point, the register state is rollback from SSSR to CR. The main scope/trace 1703 resumes executing in the CR/CM mode.

[0120] Figure 18 shows a diagram of an exemplary microprocessor pipeline 1800 in accordance with one embodiment of the present invention. The microprocessor pipeline 1800 includes a hardware conversion accelerator 1810 that implements the

functionality of the hardware acceleration conversion process, as described above. In the Figure 18 embodiment, the hardware conversion accelerator 1810 is coupled to a fetch module 1801 which is followed by a decode module 1802, an allocation module 1803, a dispatch module 1804, an execution module 1805 and a retirement modules 1806. It should be noted that the microprocessor pipeline 1800 is just one example of the pipeline that implements the functionality of embodiments of the present invention described above. One skilled in the art would recognize that other microprocessor pipelines can be implemented that include the functionality of the decode module described above.

[0121] The foregoing description, for the purpose of explanation, has been described with reference to specific embodiments. However, the illustrated discussions above are not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations are possible in view of the above teachings. Embodiments were chosen and described in order to best explain the principles of the invention and its practical applications, to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as may be suited to the particular use contemplated.

CLAIMS

What is claimed is:

1. A hardware based translation accelerator, comprising:

a guest fetch logic component for accessing a plurality of guest instructions;

a guest fetch buffer coupled to the guest fetch logic component and a branch prediction component for assembling the plurality of guest instructions into a guest instruction block;

a plurality of conversion tables coupled to the guest fetch buffer for translating the guest instruction block into a corresponding native conversion block;

a native cache coupled to the conversion tables for storing the corresponding native conversion block;

a conversion look aside buffer coupled to the native cache for storing a mapping of the guest instruction block to corresponding native conversion block;

wherein upon a subsequent request for a guest instruction, the conversion look aside buffer is indexed to determine whether a hit occurred, wherein the mapping indicates the guest instruction has a corresponding converted native instruction in the native cache; and

in response to the hit the conversion look aside buffer forwards the translated native instruction for execution.

2. The hardware based translation accelerator of claim 1, wherein a hardware fetch logic component such as the plurality of guest instructions independent of the processor.

3. The hardware based translation accelerator of claim 1, wherein the conversion look aside buffer comprises a cache that uses a replacement policy to maintain most frequently encountered native conversion blocks stored therein.

4. The hardware based translation accelerator of claim 1, wherein a conversion buffer is maintained within a system memory and cache coherency is maintained between the conversion look aside buffer and the conversion buffer.

5. The hardware based translation accelerator of claim 4, wherein the conversion buffer is larger than the conversion look aside buffer, and a write back policy is used to maintain coherency between the conversion buffer and the conversion look aside buffer.

6. The hardware based translation accelerator of claim 1, wherein the conversion look aside buffer is implemented as a high-speed low latency cache memory coupled to a pipeline of the processor.

7. A system for accelerating the translation of guest instructions to native instructions for a processor, comprising:

a guest fetch logic component for accessing a plurality of guest instructions;

a guest fetch buffer coupled to the guest fetch logic component and a branch prediction component for assembling the plurality of guest instructions into a guest instruction block;

a plurality of conversion tables coupled to the guest fetch buffer for translating the guest instruction block into a corresponding native conversion block;

a native cache coupled to the conversion tables for storing the corresponding native conversion block;

a conversion look aside buffer coupled to the native cache for storing a mapping of the guest instruction block to corresponding native conversion block;

wherein upon a subsequent request for a guest instruction, the conversion look aside buffer is indexed to determine whether a hit occurred, wherein the mapping indicates the guest instruction has a corresponding converted native instruction in the native cache; and

in response to the hit the conversion look aside buffer forwards the translated native instruction for execution.

8. The system of claim 7, wherein a hardware fetch logic component such as the plurality of guest instructions independent of the processor.

9. The system of claim 7, wherein the conversion look aside buffer comprises a cache that uses a replacement policy to maintain most frequently encountered native conversion blocks stored therein.

10. The system of claim 7, wherein a conversion buffer is maintained within a system memory and cache coherency is maintained between the conversion look aside buffer and the conversion buffer.

11. The system of claim 10, wherein the conversion buffer is larger than the conversion look aside buffer, and a write back policy is used to maintain coherency between the conversion buffer and the conversion look aside buffer.

12. The system of claim 7, wherein the conversion look aside buffer is implemented as a high-speed low latency cache memory coupled to a pipeline of the processor.

13. A microprocessor that implements a method of translating instructions, said microprocessor comprises:

a microprocessor pipeline;

a hardware accelerator module coupled to the microprocessor pipeline, wherein the hardware accelerator module further comprises:

a guest fetch logic component for accessing a plurality of guest instructions;

a guest fetch buffer coupled to the guest fetch logic component and a branch prediction component for assembling the plurality of guest instructions into a guest instruction block;

a plurality of conversion tables coupled to the guest fetch buffer for translating the guest instruction block into a corresponding native conversion block;

a native cache coupled to the conversion tables for storing the corresponding native conversion block;

a conversion look aside buffer coupled to the native cache for storing a mapping of the guest instruction block to corresponding native conversion block;

wherein upon a subsequent request for a guest instruction, the conversion look aside buffer is indexed to determine whether a hit occurred, wherein the mapping indicates the guest instruction has a corresponding converted native instruction in the native cache; and

in response to the hit the conversion look aside buffer forwards the translated native instruction for execution.

14. The microprocessor of claim 13, wherein a hardware fetch logic component such as the plurality of guest instructions independent of the processor.

15. The microprocessor of claim 13, wherein the conversion look aside buffer comprises a cache that uses a replacement policy to maintain most frequently encountered native conversion blocks stored therein.

16. The microprocessor of claim 13, wherein a conversion buffer is maintained within a system memory and cache coherency is maintained between the conversion look aside buffer and the conversion buffer.

17. The microprocessor of claim 16, wherein the conversion buffer is larger than the conversion look aside buffer, and a write back policy is used to maintain coherency between the conversion buffer and the conversion look aside buffer.

18. The microprocessor of claim 13, wherein the conversion look aside buffer is implemented as a high-speed low latency cache memory coupled to a pipeline of the processor.

19. The microprocessor of claim 13, wherein the hardware accelerator module functions comprises a parallel guest instruction fetch pipeline that functions in parallel to a native microprocessor fetch pipeline.

20. A microprocessor that implements a method of translating instructions, said microprocessor comprises:

a microprocessor pipeline;

an accelerator module comprising high speed memory coupled to the microprocessor pipeline, wherein the accelerator module further comprises:

a guest fetch logic for accessing a plurality of guest instructions;

a guest fetch memory coupled to the guest fetch logic for assembling the plurality of guest instructions into a guest instruction block;

a plurality of conversion tables for translating the guest instruction block into a corresponding native conversion block;

a native conversion buffer for storing the corresponding native conversion block;

wherein upon a subsequent request for a guest instruction, a conversion look aside buffer storing a mapping of the guest instruction block to corresponding native conversion block is indexed to determine whether a hit occurred, wherein the mapping indicates the guest instruction has a corresponding converted native instruction in the native cache; and

in response to the hit the conversion look aside buffer forwards the translated native instruction for execution.

21. The microprocessor of claim 20, wherein the high speed memory comprises an L0 cache of the microprocessor.

22. The microprocessor of claim 20, wherein the accelerator module further comprises a load store instruction fetch path of the microprocessor.

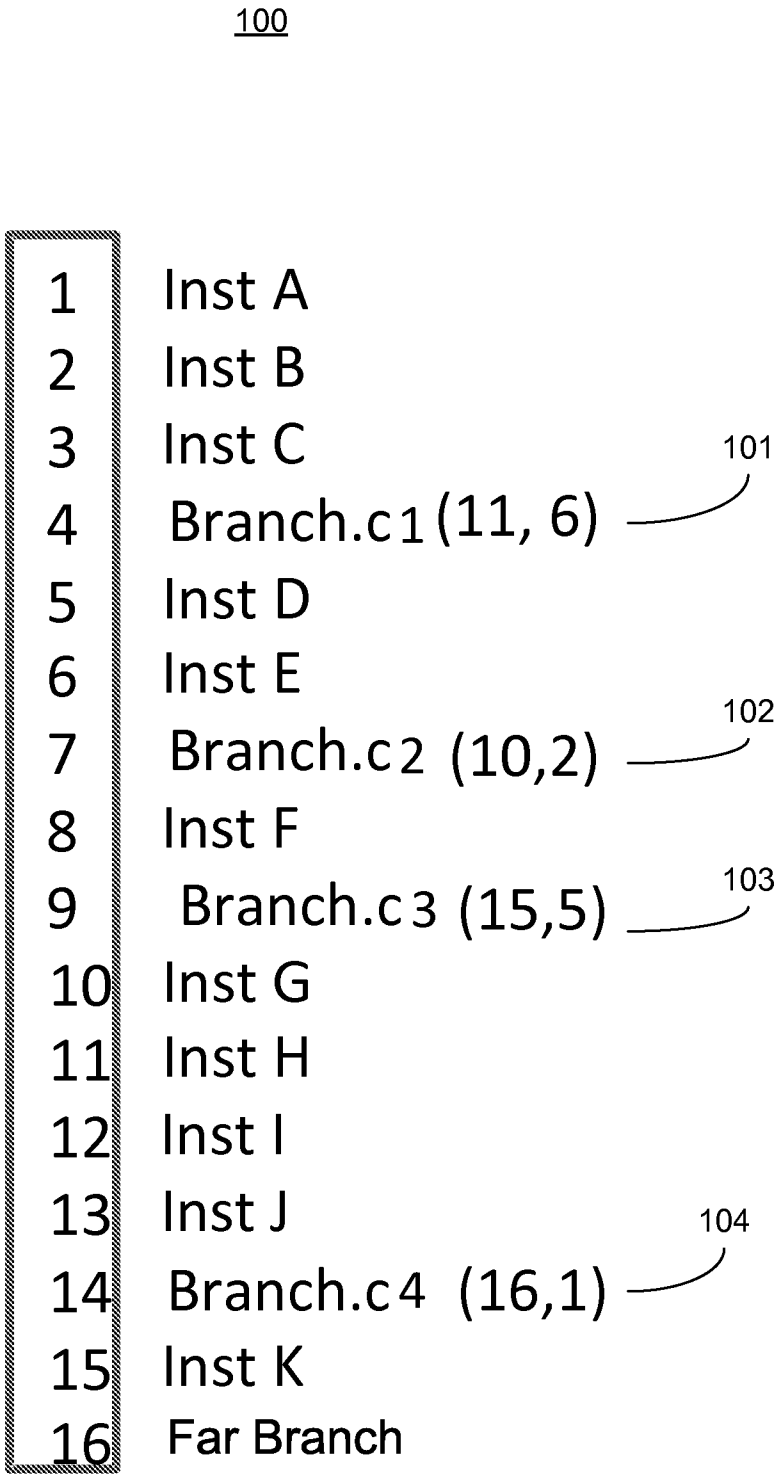
100
1 Inst A
2 Inst B
3 Inst C
4 Branch.c1 (11, 6)
5 Inst D
6 Inst E
7 Branch.c2 (10,2)
8 Inst F
9 Branch.c3 (15,5)
10 Inst G
11 Inst H
12 Inst I
13 Inst J
14 Branch.c4 (16,1)
15 Inst K
16 Far Branch
101
102
103
104

FIG. 1

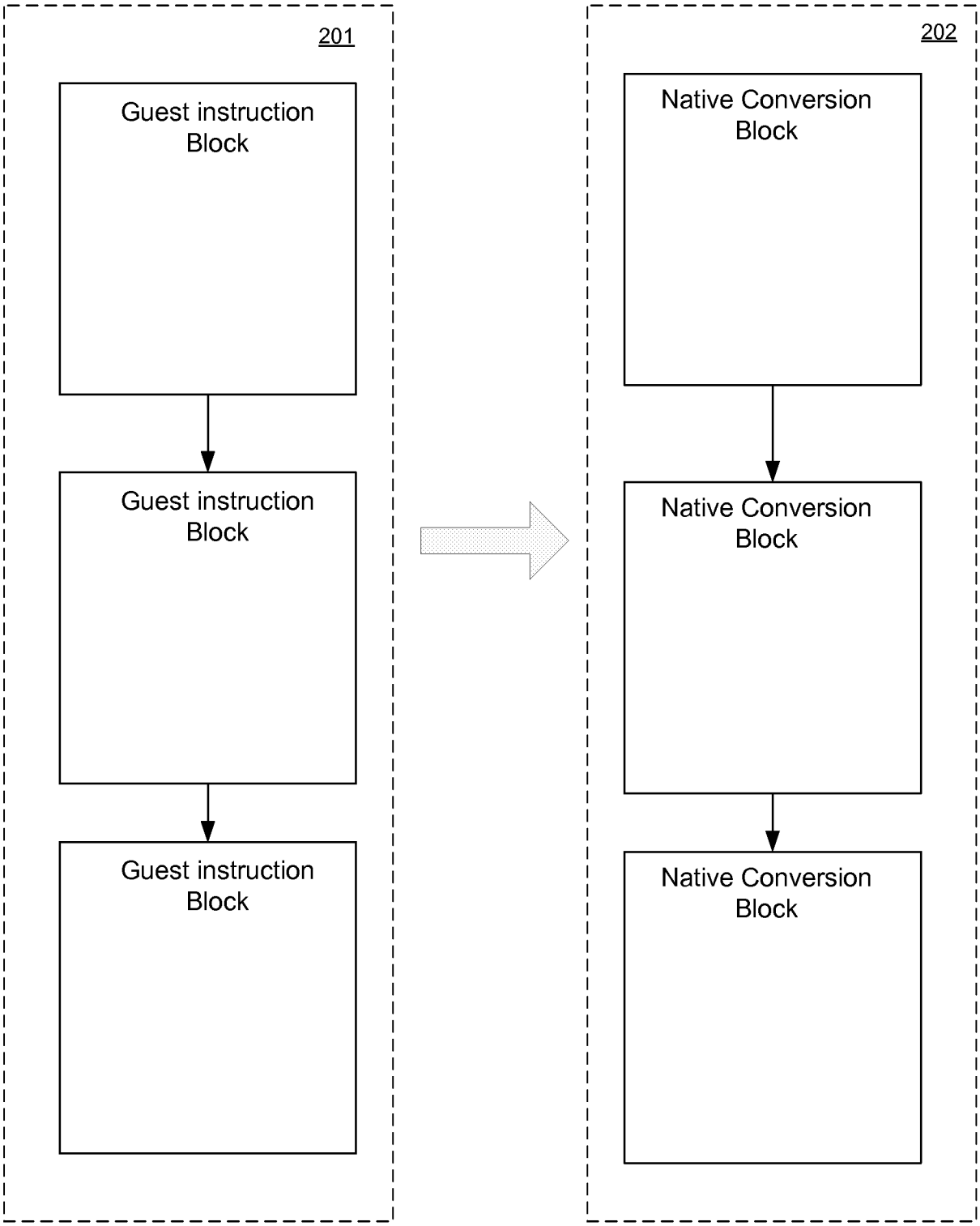
201
Guest instruction Block
Guest instruction Block
Guest instruction Block
202
Native Conversion Block
Native Conversion Block
Native Conversion Block

FIG. 2

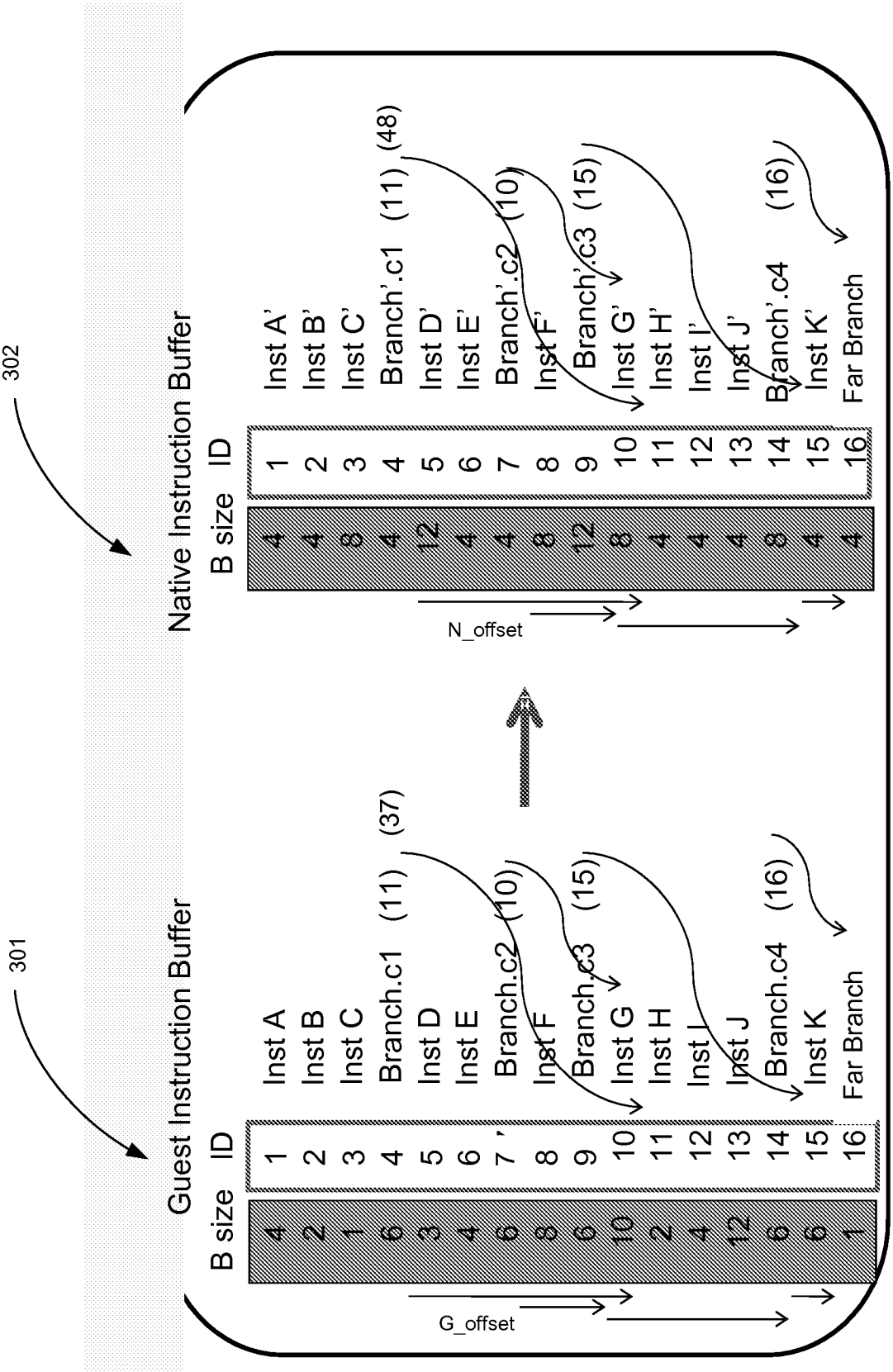
301
302
Guest Instruction Buffer
B size
ID
Inst A
Inst B
Inst C
Branch.c1 (11) (37)
Inst D
Inst E
Branch.c2 (10)
Inst F
Branch.c3 (15)
Inst G
Inst H
Inst I
Inst J
Branch.c4 (16)
Inst K
Far Branch
G_offset
Native Instruction Buffer
B size
ID
Inst A'
Inst B'
Inst C'
Branch'.c1 (11) (48)
Inst D'
Inst E'
Branch'.c2 (10)
Inst F'
Branch'.c3 (15)
Inst G'
Inst H'
Inst I'
Inst J'
Branch'.c4 (16)
Inst K'
Far Branch
N_offset

FIG. 3

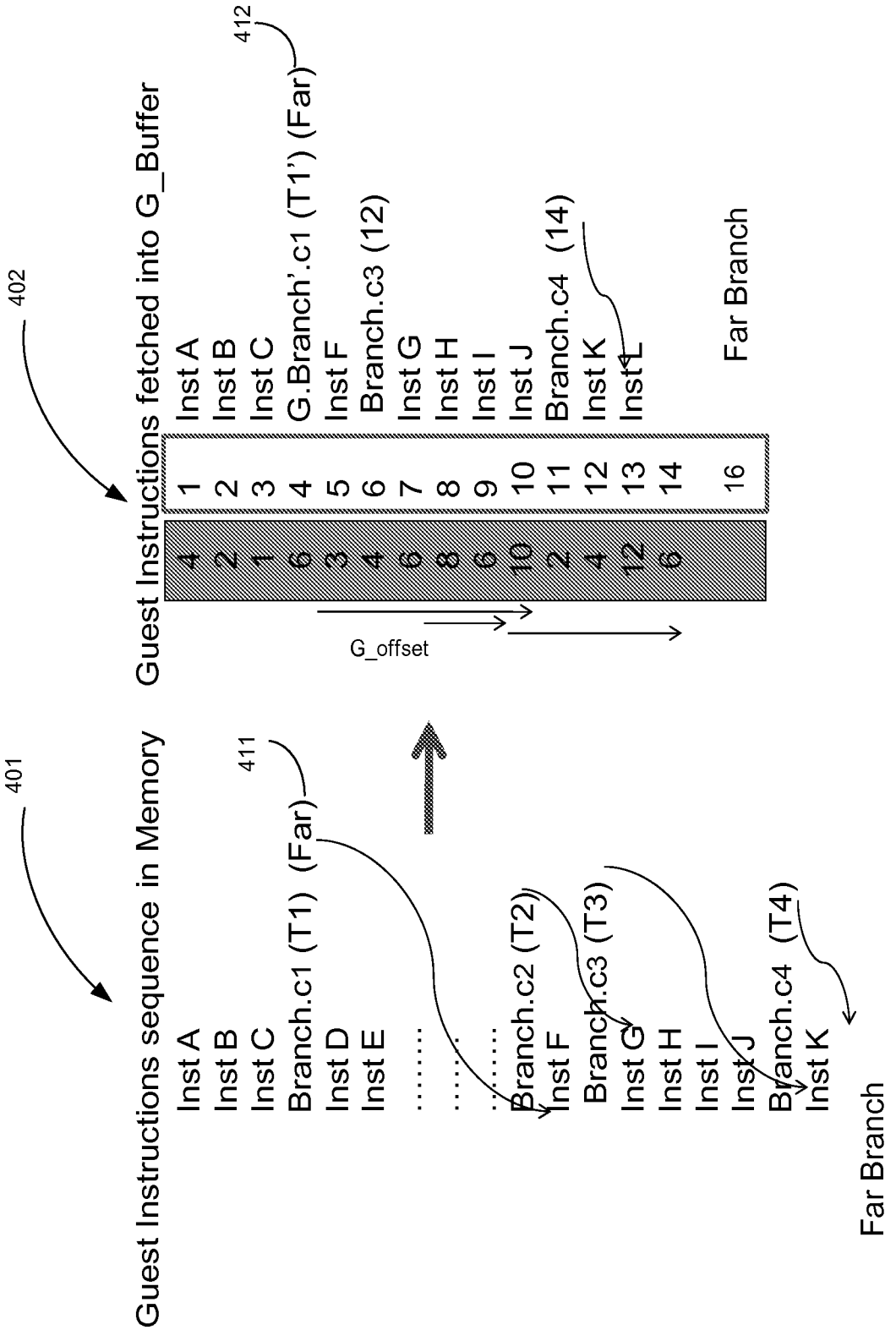
401
Guest Instructions sequence in Memory
Inst A
Inst B
Inst C
Branch.c1 (T1) (Far)
Inst D
Inst E
Branch.c2 (T2)
Inst F
Branch.c3 (T3)
Inst G
Inst H
Inst I
Inst J
Branch.c4 (T4)
Inst K
Far Branch
411
402
Guest Instructions fetched into G_Buffer
Inst A
Inst B
Inst C
G.Branch'.c1 (T1') (Far)
Inst F
Branch.c3 (12)
Inst G
Inst H
Inst I
Inst J
Branch.c4 (14)
Inst K
Inst L
Far Branch
412
G_offset

FIG. 4

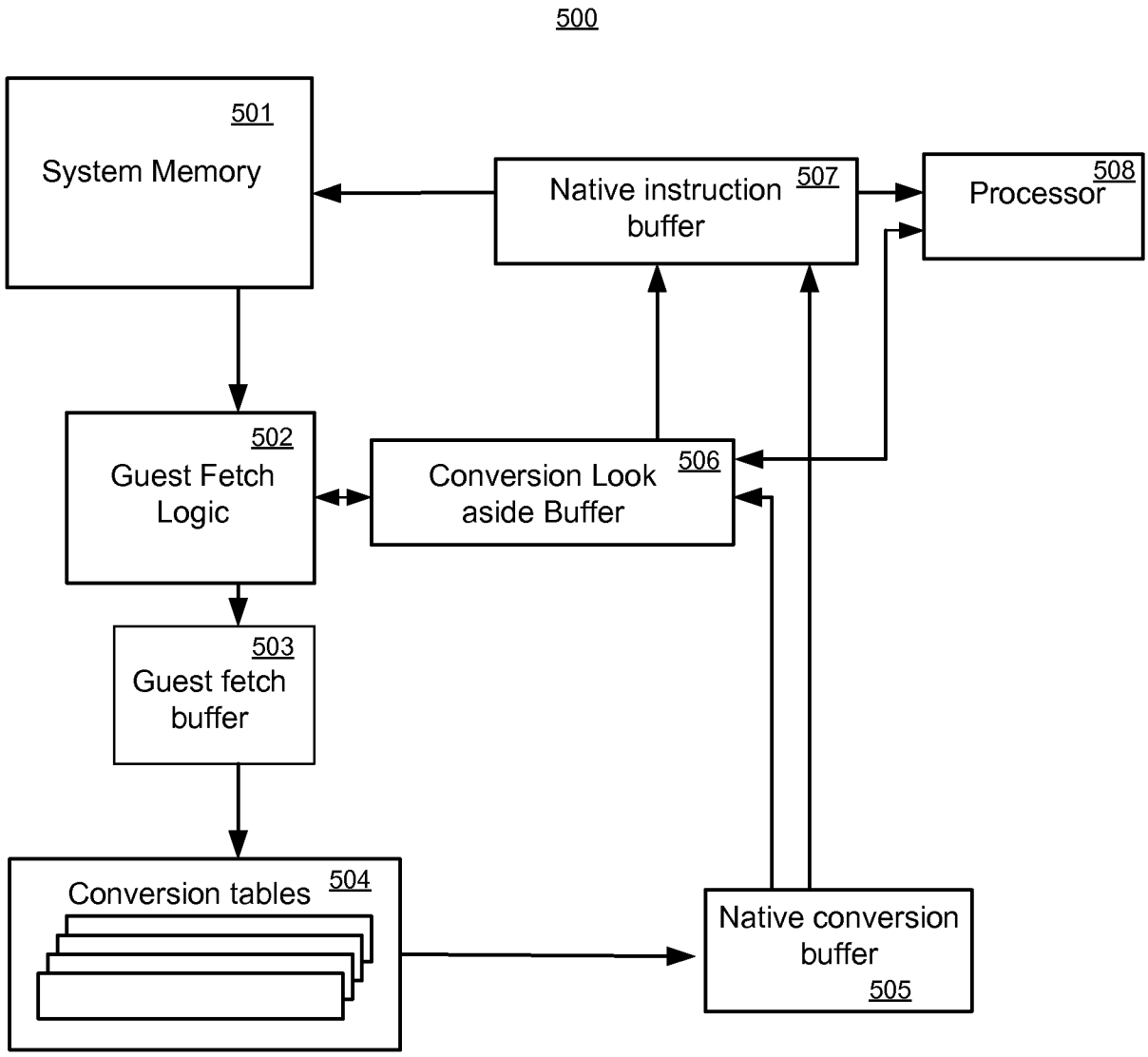
500
501
System Memory
Native instruction buffer
507
Processor
508
502
Guest Fetch Logic
Conversion Look aside Buffer
506
503
Guest fetch buffer
Conversion tables
504
Native conversion buffer
505

FIG. 5

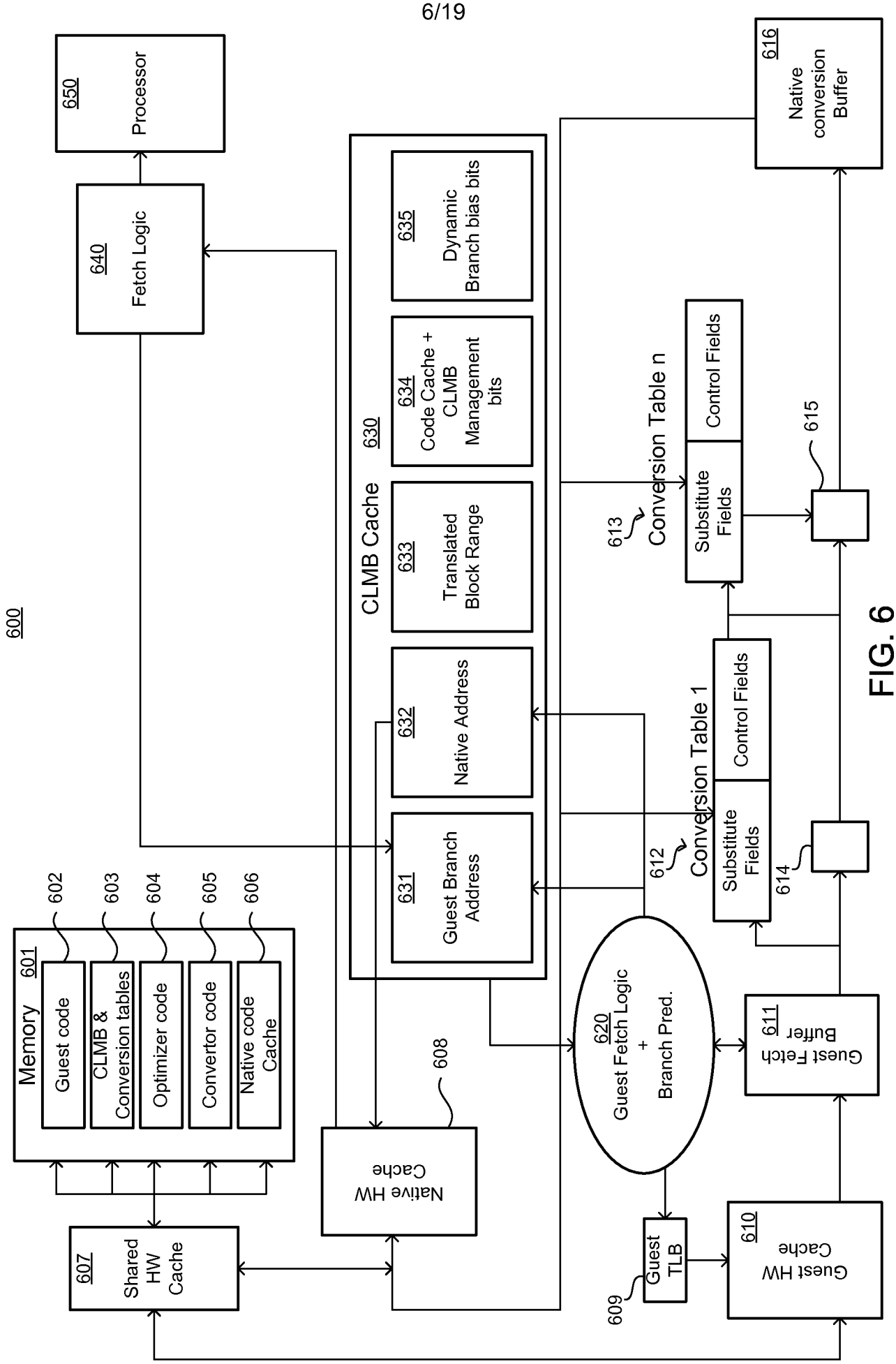
600
650 Processor
640 Fetch Logic
616 Native conversion Buffer
CLMB Cache 630
635 Dynamic Branch bias bits
634 Code Cache + CLMB Management bits
633 Translated Block Range
632 Native Address
631 Guest Branch Address
Conversion Table n
Control Fields
Substitute Fields
613
615
Conversion Table 1
Control Fields
Substitute Fields
612
614
Memory 601
602 Guest code
603 CLMB & Conversion tables
604 Optimizer code
605 Convertor code
606 Native code Cache
608 Native HW Cache
620 Guest Fetch Logic + Branch Pred.
611 Guest Fetch Buffer
607 Shared HW Cache
609 Guest TLB
610 Guest HW Cache

FIG. 6

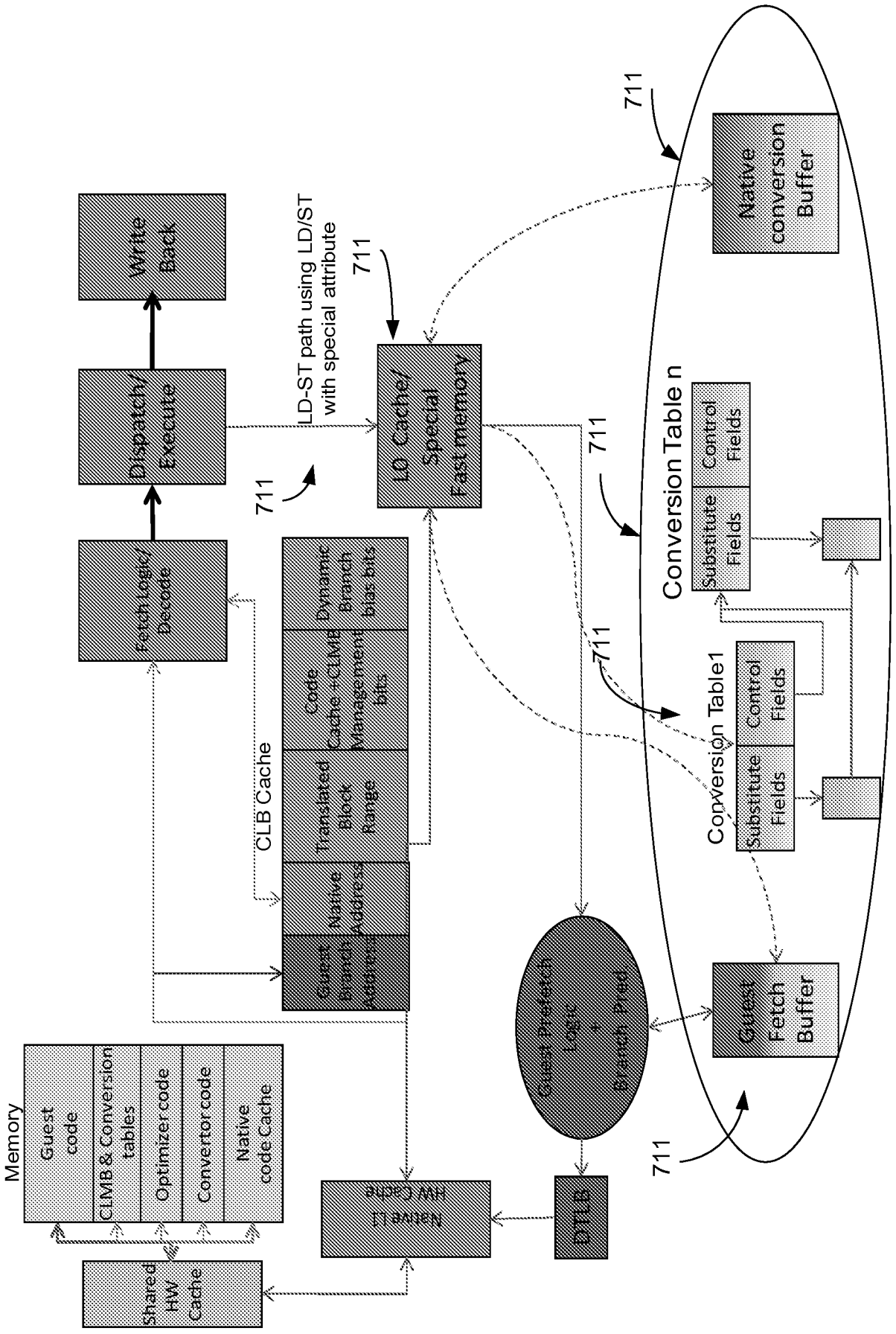
Write Back
Dispatch/ Execute
Fetch Logic/ Decode
LD-ST path using LD/ST with special attribute
711
L0 Cache/ Special Fast memory
CLB Cache
Guest Branch Address
Native Address
Translated Block Range
Code Cache +CLMB Management bits
Dynamic Branch bias bits
Native conversion Buffer
Conversion Table n
Substitute Fields
Control Fields
Conversion Table1
Guest Fetch Buffer
Guest Prefetch logic + Branch Pred.
DTLB
L2 Cache HW Cache
Shared HW Cache
Memory
Guest code
CLMB & Conversion tables
Optimizer code
Convertor code
Native code Cache

FIG. 7

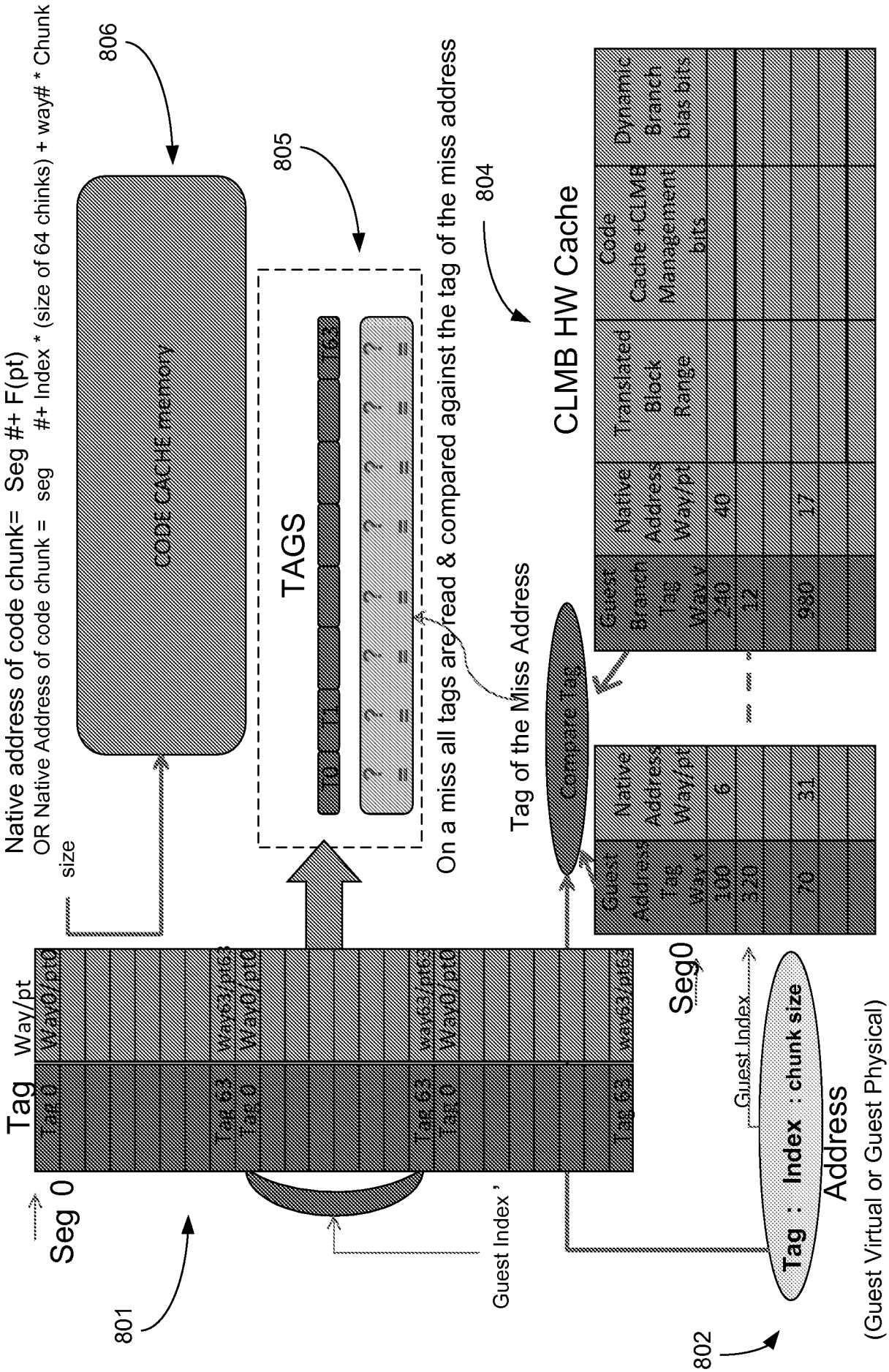
Native address of code chunk= Seg #+ F(pt)
OR Native Address of code chunk = seg #+ Index * (size of 64 chinks) + way# * Chunk size
806
CODE CACHE memory
805
TAGS
804
CLMB HW Cache
On a miss all tags are read & compared against the tag of the miss address
Tag of the Miss Address
Compare Tag
Guest Branch Tag Way x
Native Address Way/pt
Translated Block Range
Code Cache +CLMB Management bits
Dynamic Branch bias bits
240
12
930
17
Guest Address Tag Way x
Native Address Way/pt
100
6
320
31
70
Tag
Way/pt
Way0/pt0
Way63/pt63
Tag 0
Tag 63
Seg 0
Seg0
Guest Index '
Guest Index
Tag : Index : chunk size
Address
(Guest Virtual or Guest Physical)
801
802

FIG. 8

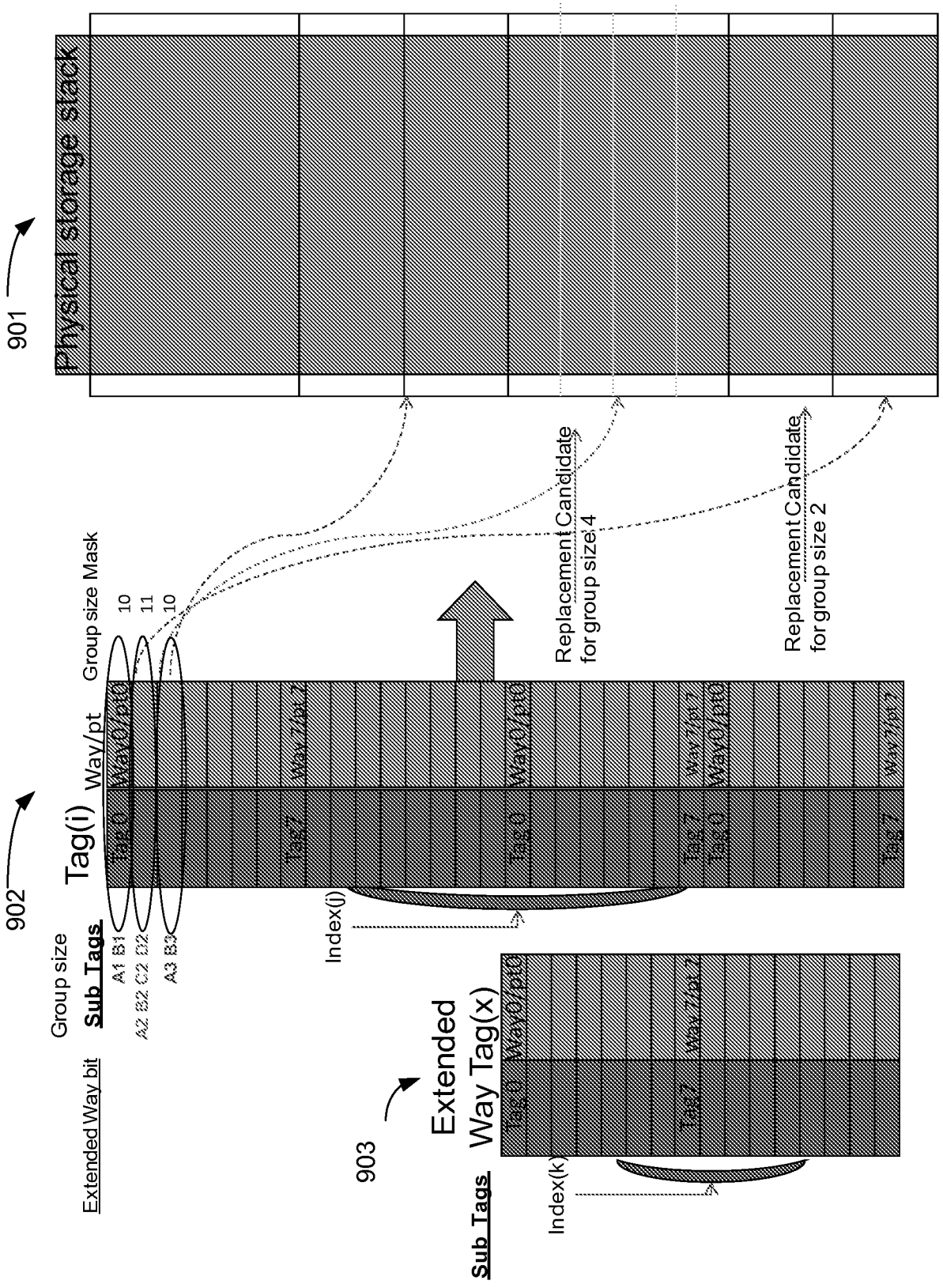
901
Physical storage stack
902
Tag(i)
Way/pt
Group size Mask
Group size
Sub Tags
Extended Way bit
A1 B1
A2 B2 C2 D2
A3 B3
10
11
10
Tag0
Way0/pt0
Tag7
Way7/pt7
Index(j)
Replacement Candidate for group size 4
Replacement Candidate for group size 2
903
Extended Way Tag(x)
Sub Tags
Index(k)

FIG. 9

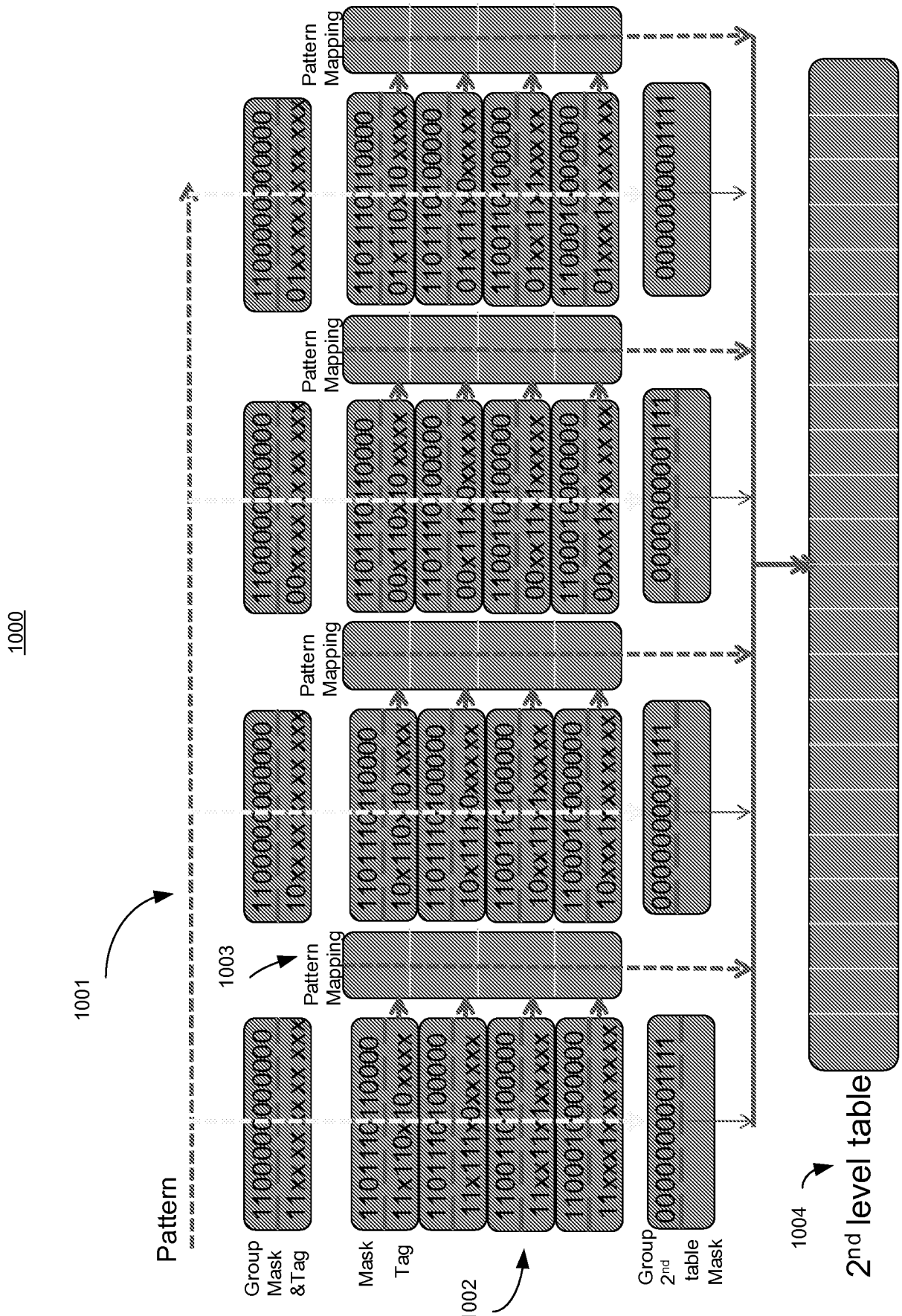
1000
Pattern
1001
1003
Pattern Mapping
Pattern Mapping
Pattern Mapping
Pattern Mapping
Group Mask &Tag
Mask
Tag
1002
Group 2nd table Mask
1004
2nd level table

FIG. 10

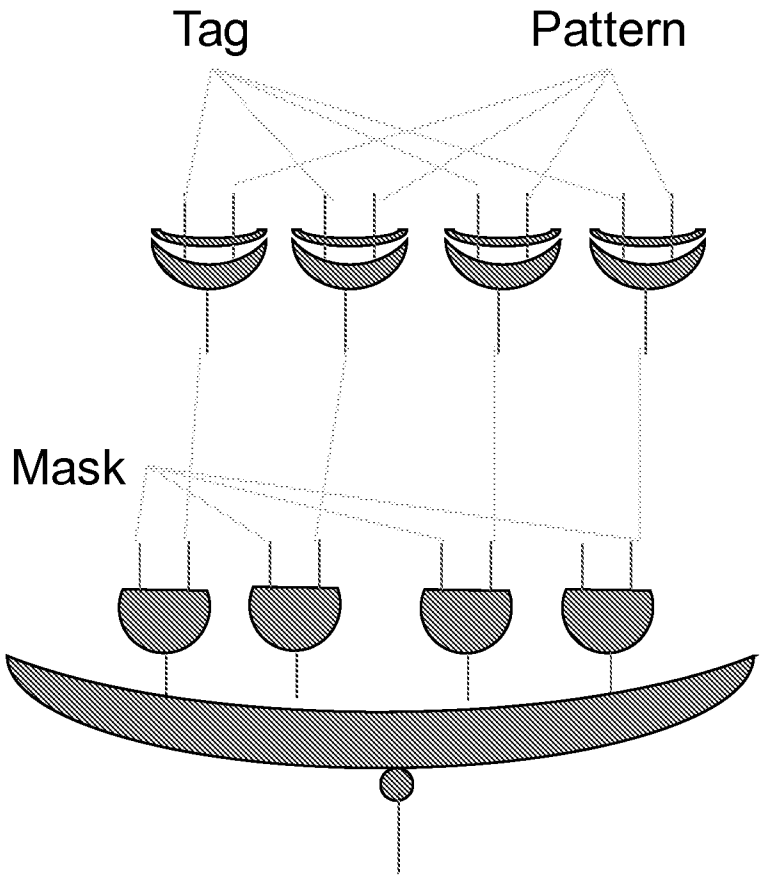

FIG. 11A

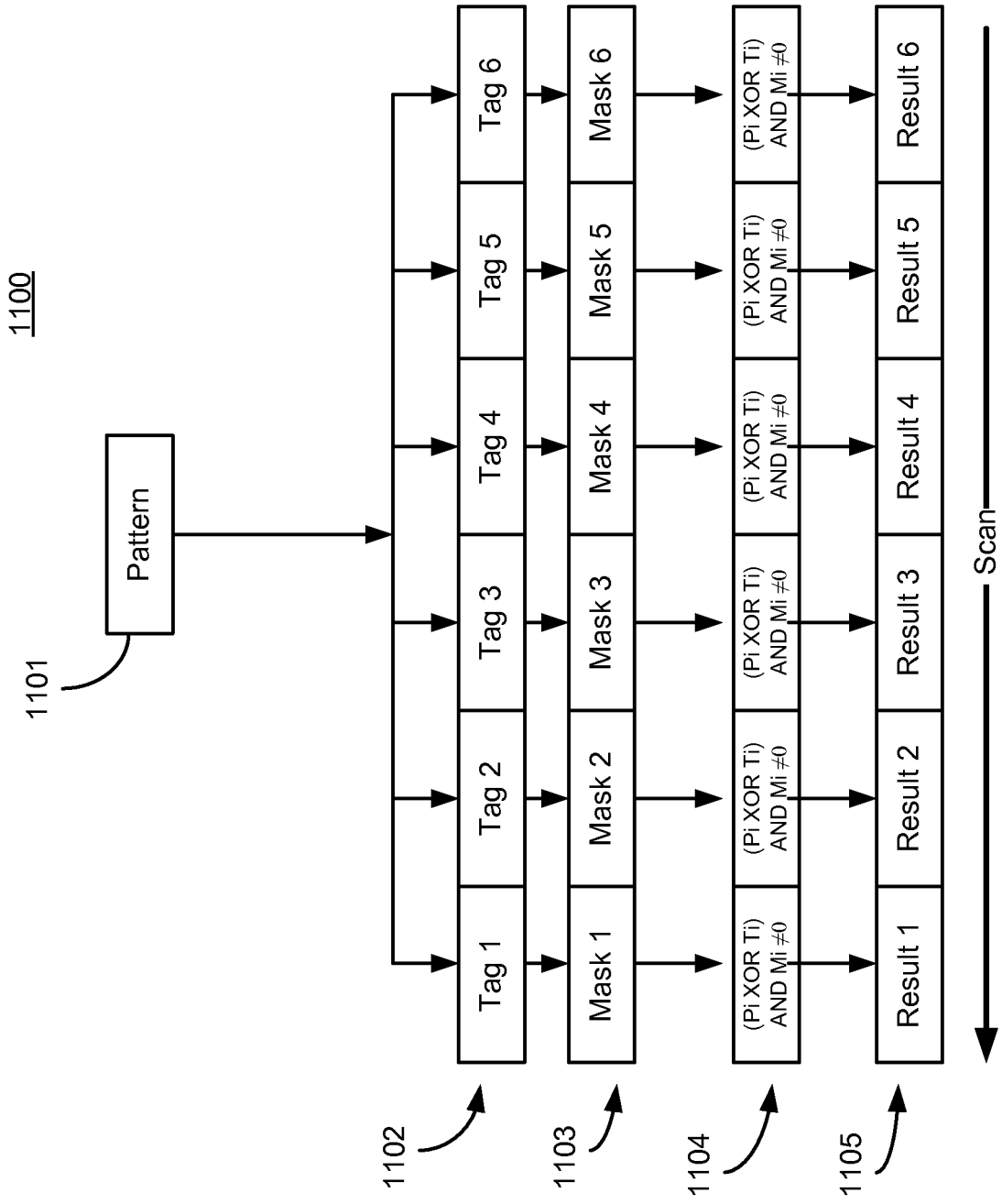
1100
Pattern
1101
Tag 1
Tag 2
Tag 3
Tag 4
Tag 5
Tag 6
Mask 1
Mask 2
Mask 3
Mask 4
Mask 5
Mask 6
(Pi XOR Ti) AND Mi ≠0
Result 1
Result 2
Result 3
Result 4
Result 5
Result 6
Scan
1102
1103
1104
1105

Fig. 11B

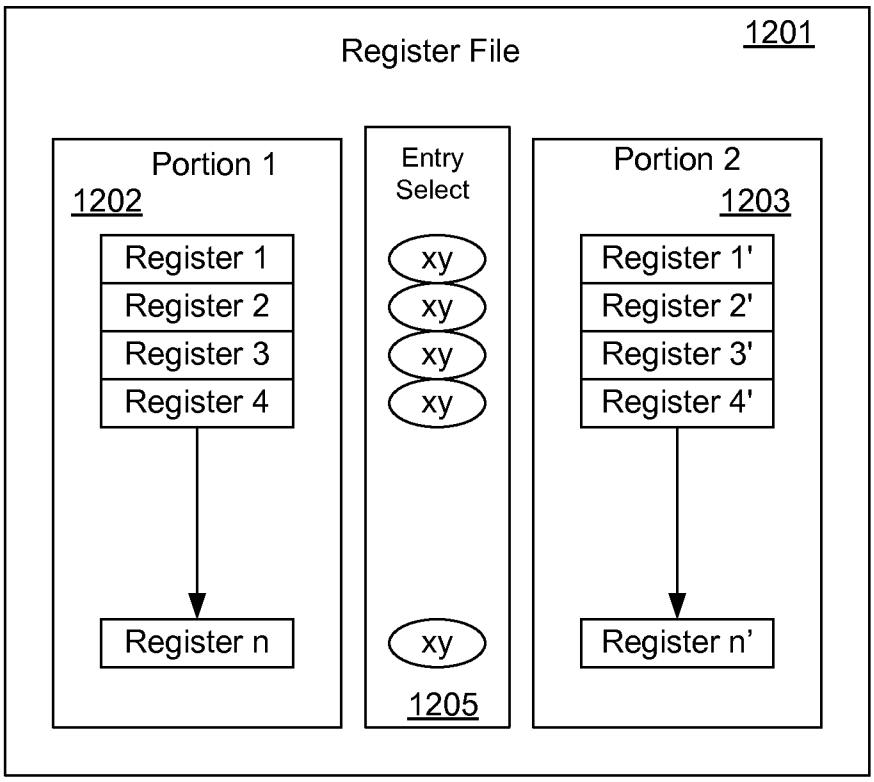
Register File
1201
Portion 1
1202
Register 1
Register 2
Register 3
Register 4
Register n
Entry
Select
xy
xy
xy
xy
xy
1205
Portion 2
1203
Register 1'
Register 2'
Register 3'
Register 4'
Register n'

FIG. 12

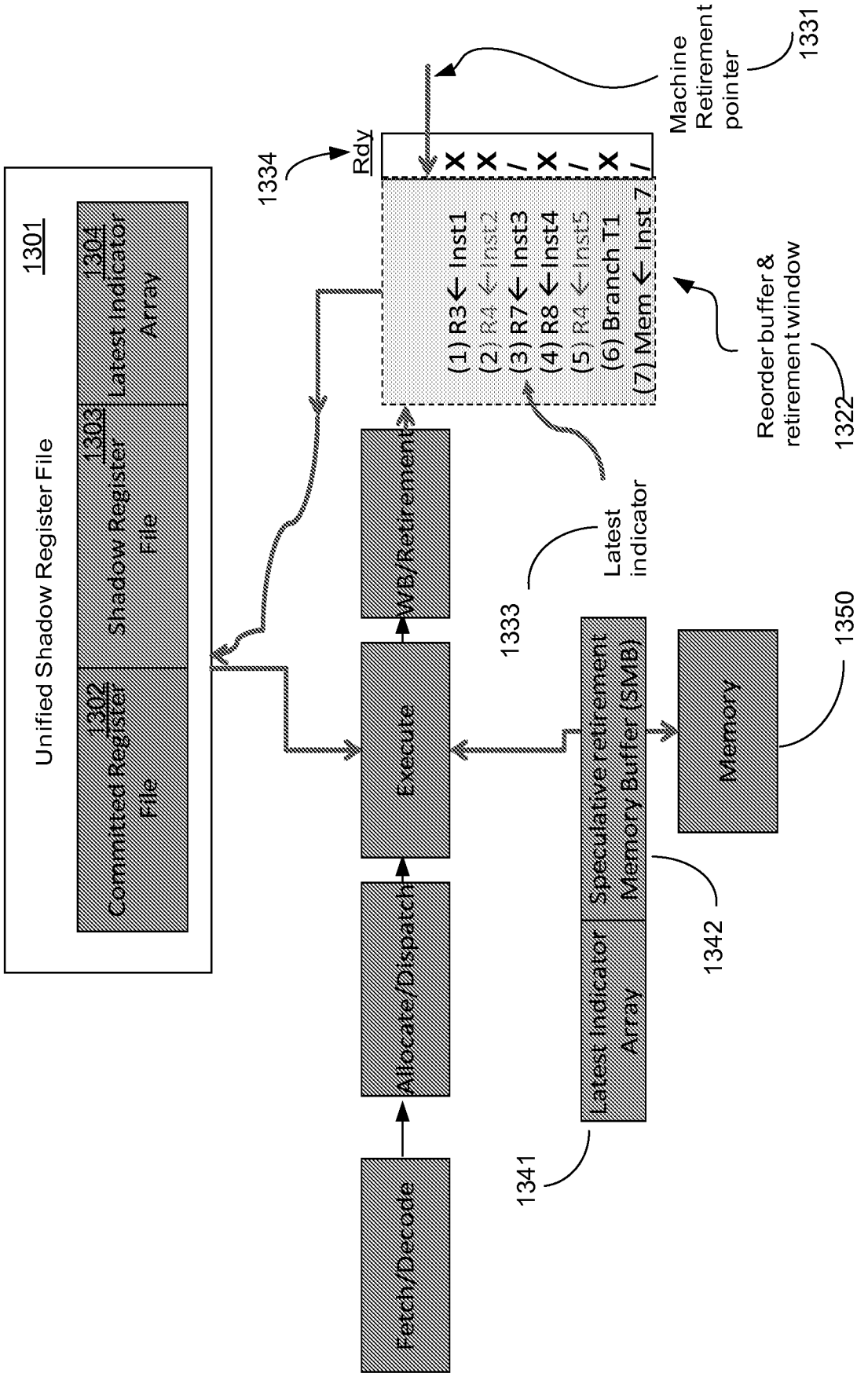
Unified Shadow Register File
1301
Committed Register File
1302
Shadow Register File
1303
Latest Indicator Array
1304
Fetch/Decode
Allocate/Dispatch
Execute
WB/Retirement
Speculative retirement Memory Buffer (SMB)
Latest indicator Array
Memory
1341
1342
1350
1333
Latest indicator
1334
Rdy
(1) R3 ← Inst1
(2) R4 ← Inst2
(3) R7 ← Inst3
(4) R8 ← Inst4
(5) R4 ← Inst5
(6) Branch T1
(7) Mem ← Inst 7
Machine Retirement pointer
1331
Reorder buffer & retirement window
1322

FIG. 13

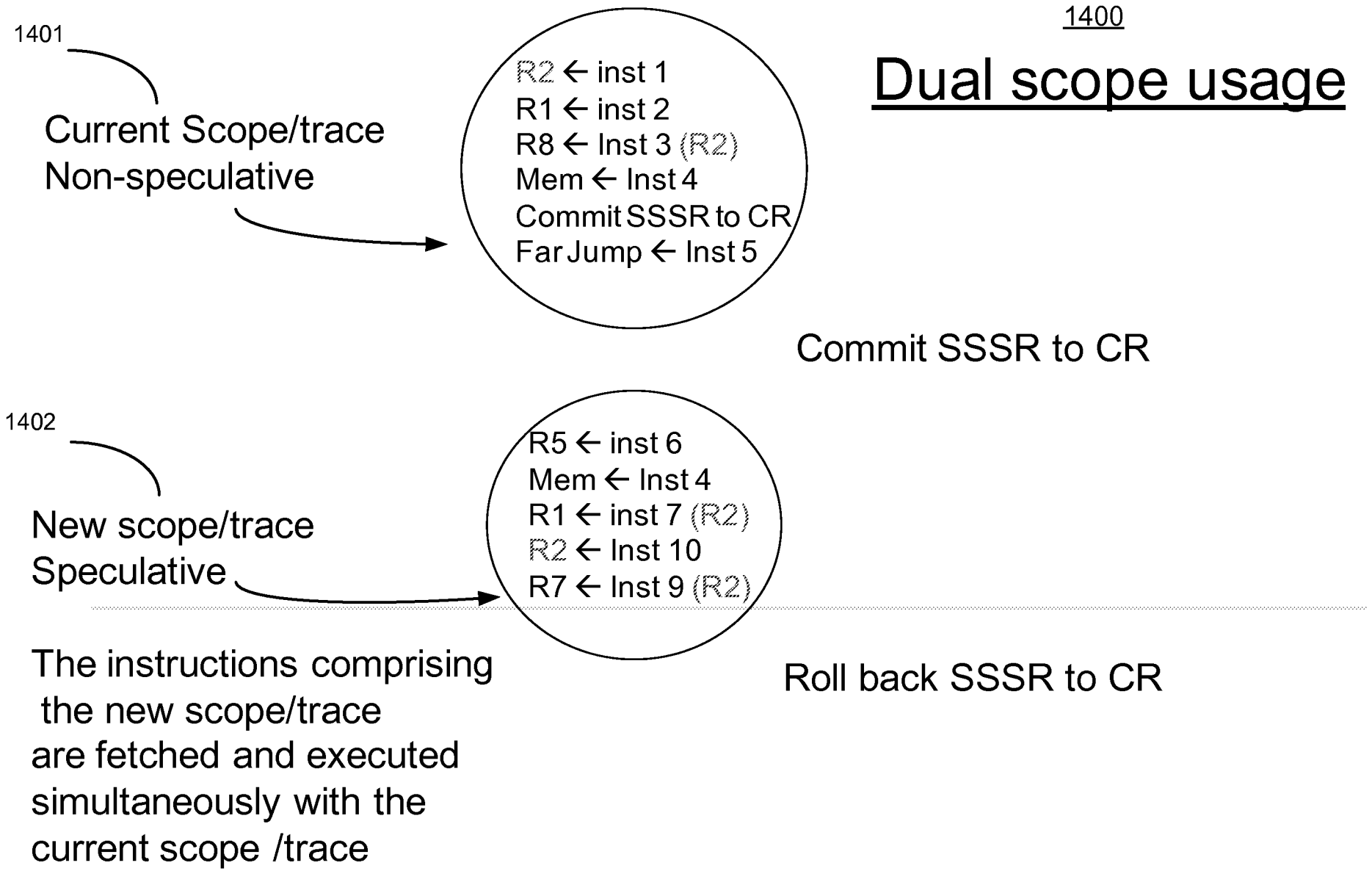
1400
Dual scope usage
1401
Current Scope/trace
Non-speculative
R2 ← inst 1
R1 ← inst 2
R8 ← Inst 3 (R2)
Mem ← Inst 4
Commit SSSR to CR
Far Jump ← Inst 5
Commit SSSR to CR
1402
New scope/trace
Speculative
R5 ← inst 6
Mem ← Inst 4
R1 ← inst 7 (R2)
R2 ← Inst 10
R7 ← Inst 9 (R2)
Roll back SSSR to CR
The instructions comprising
the new scope/trace
are fetched and executed
simultaneously with the
current scope /trace

FIG. 14

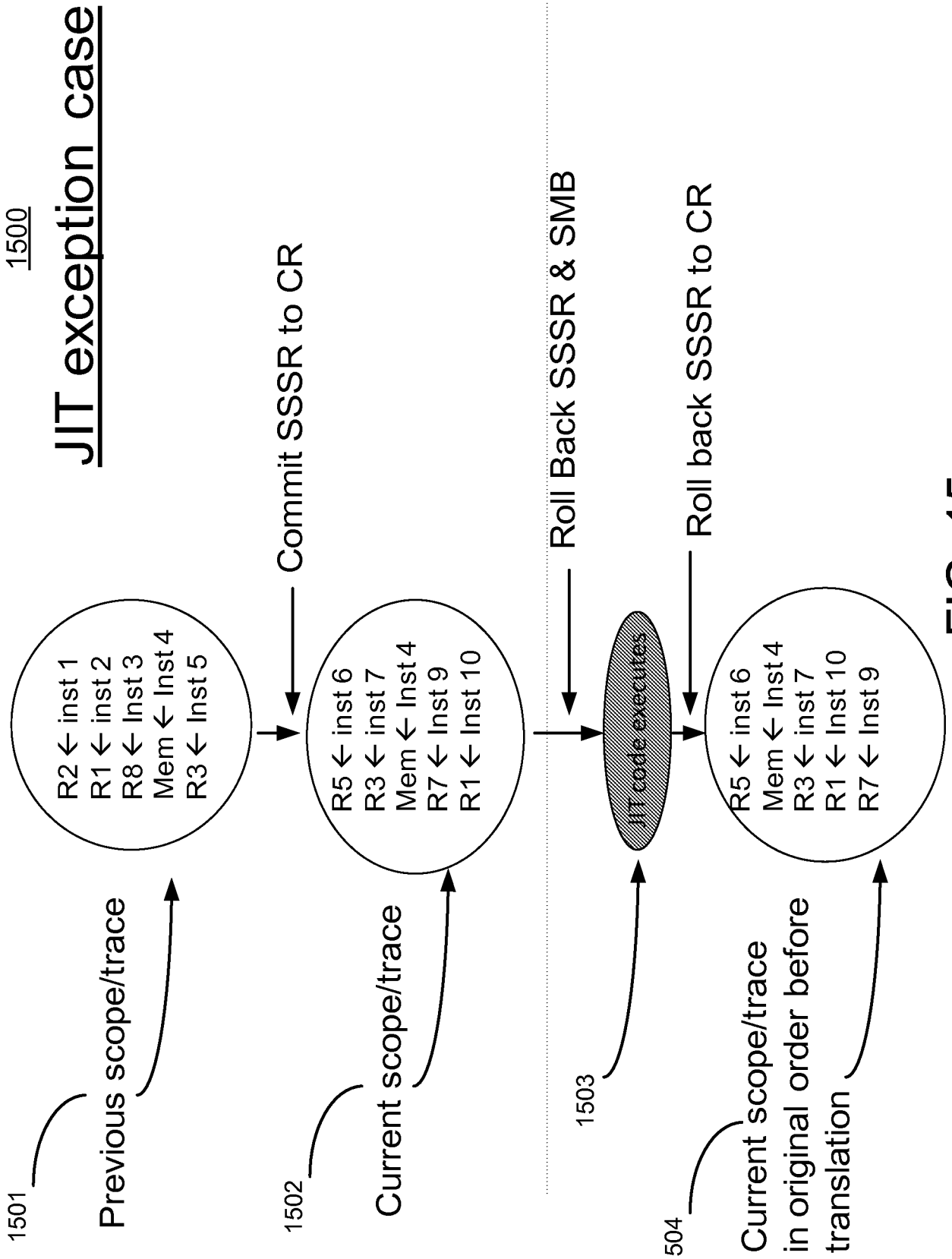
1500
JIT exception case
1501
Previous scope/trace
R2 ← inst 1
R1 ← inst 2
R8 ← Inst 3
Mem ← Inst 4
R3 ← Inst 5
Commit SSSR to CR
1502
Current scope/trace
R5 ← inst 6
R3 ← inst 7
Mem ← Inst 4
R7 ← Inst 9
R1 ← Inst 10
Roll Back SSSR & SMB
1503
JIT code executes
Roll back SSSR to CR
1504
Current scope/trace
in original order before
translation
R5 ← inst 6
Mem ← Inst 4
R3 ← inst 7
R1 ← Inst 10
R7 ← Inst 9

FIG. 15

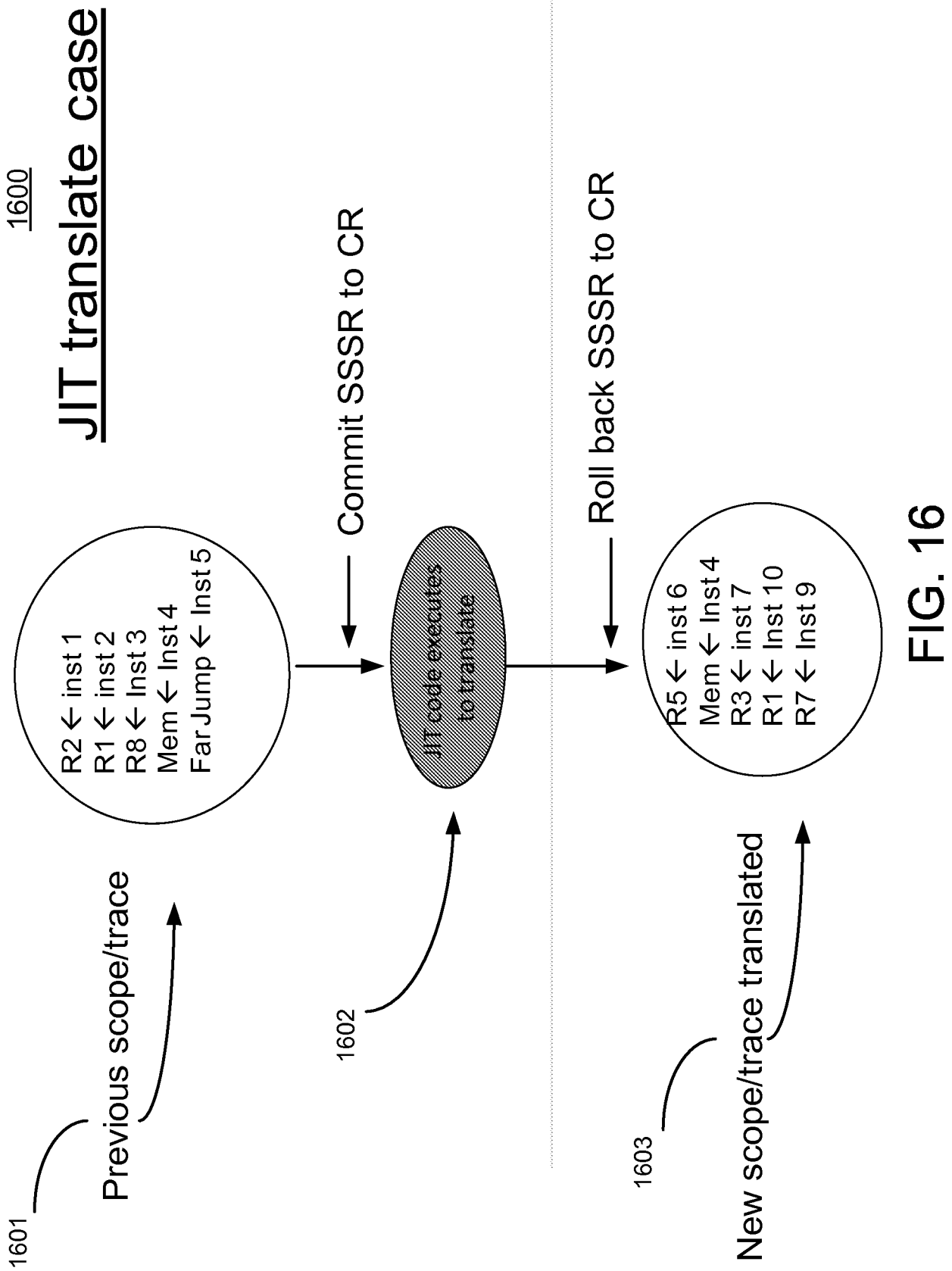
1600
JIT translate case
R2 ← inst 1
R1 ← inst 2
R8 ← Inst 3
Mem ← Inst 4
Far Jump ← Inst 5
Commit SSSR to CR
JIT code executes to translate
Roll back SSSR to CR
R5 ← inst 6
Mem ← Inst 4
R3 ← inst 7
R1 ← Inst 10
R7 ← Inst 9
1601
Previous scope/trace
1602
1603
New scope/trace translated

FIG. 16

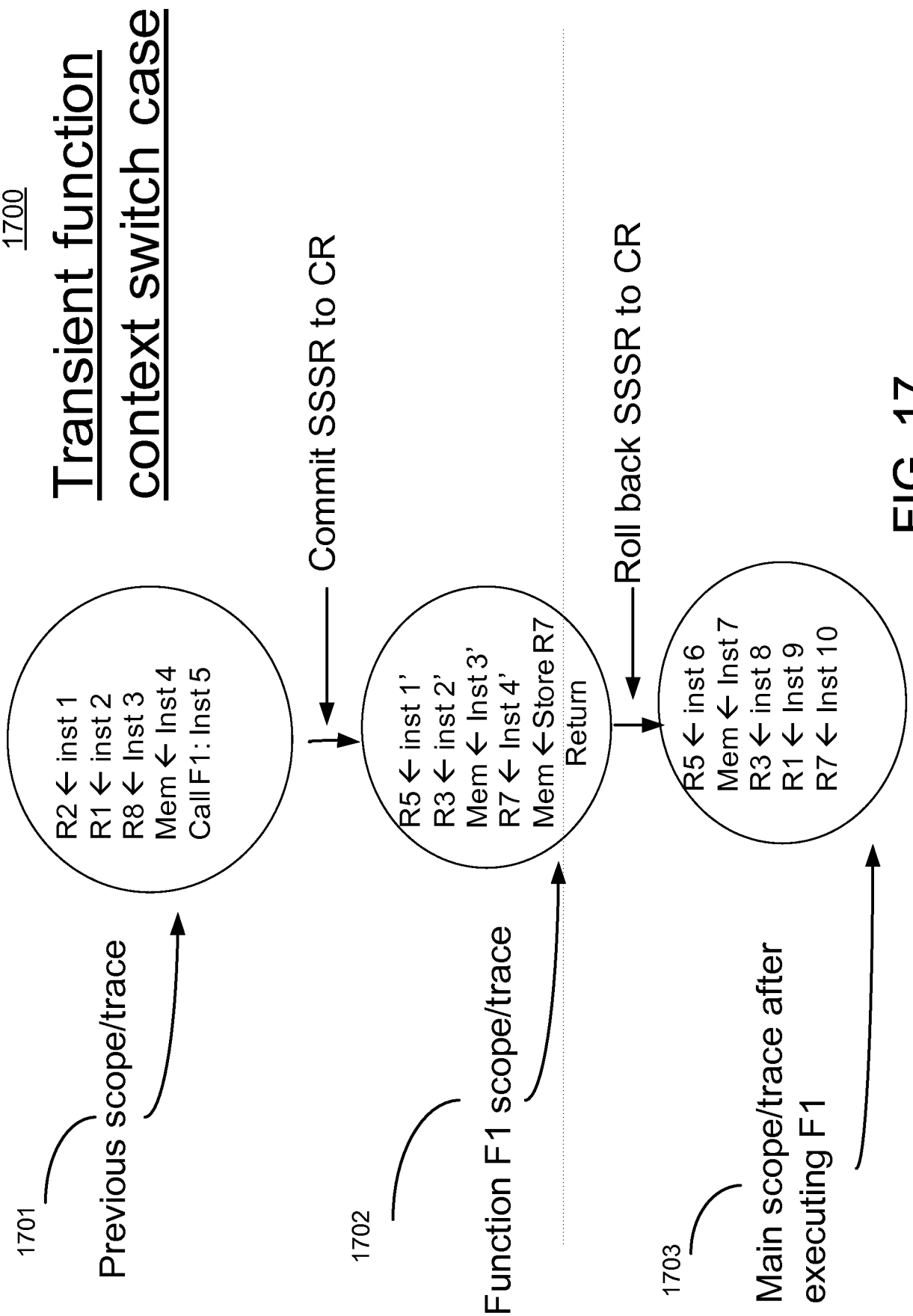
1700
Transient function context switch case
1701
Previous scope/trace
R2 ← inst 1
R1 ← inst 2
R8 ← Inst 3
Mem ← Inst 4
Call F1: Inst 5
Commit SSSR to CR
1702
Function F1 scope/trace
R5 ← inst 1'
R3 ← inst 2'
Mem ← Inst 3'
R7 ← Inst 4'
Mem ← Store R7
Return
Roll back SSSR to CR
1703
Main scope/trace after executing F1
R5 ← inst 6
Mem ← Inst 7
R3 ← inst 8
R1 ← Inst 9
R7 ← Inst 10

FIG. 17

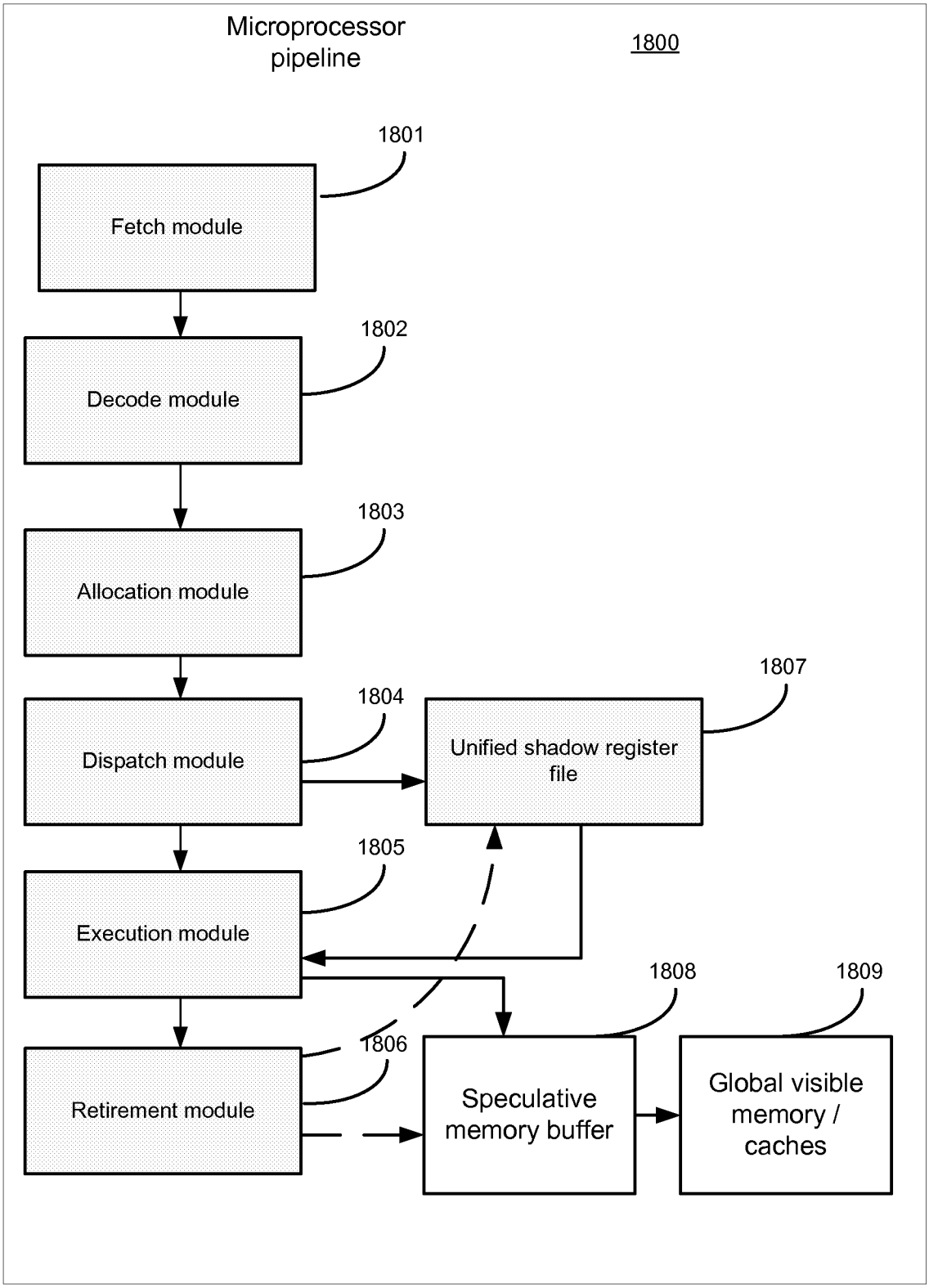
Microprocessor pipeline
1800
1801
Fetch module
1802
Decode module
1803
Allocation module
1804
Dispatch module
1807
Unified shadow register file
1805
Execution module
1808
1809
1806
Retirement module
Speculative memory buffer
Global visible memory / caches

FIG. 18