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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
25 September 2014 (25.09.2014)

WIPO | PCT

(10) International Publication Number
WO 2014/151043 Al

(51) International Patent Classification:
G06F 9/455 (2006.01) *G06F 9/46* (2006.01)

(21) International Application Number:
PCT/US20 14/024828

(22) International Filing Date:
12 March 2014 (12.03.2014)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
61/800,487 15 March 2013 (15.03.2013) US

(71) Applicant *(for all designated States except US):* **SOFT MACHINES, INC.** [US/US]; 321 1 Scott Boulevard, Suite 202, Santa Clara, CA 95054 (US).

(72) Inventor; and
(71) Applicant *(for US only):* **ABDALLAH, Mohammad** [US/US]; 3868 Suncrest Ave., San Jose, CA 95 132 (US).

(74) Agent: **BARNES, Glenn, D.;** Murabito, Hao & Barnes LLP, 2 N Market St., 3rd Floor, San Jose, CA 95 113 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available):* AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, KZ, LA, LC, LK, LR, LS, LT, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

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

Published:

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

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

(54) Title: A METHOD FOR EMULATING A GUEST CENTRALIZED FLAG ARCHITECTURE BY USING A NATIVE DIS ¬ TRIBUTED FLAG ARCHITECTURE

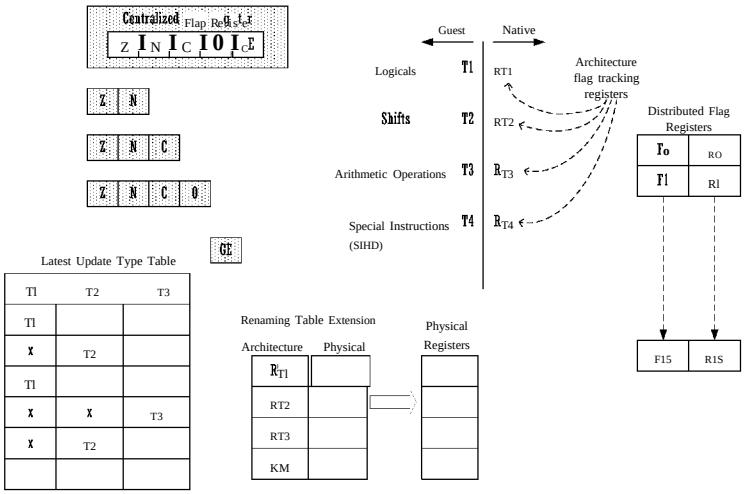


Fig. 37

(57) Abstract: A method for emulating a guest centralized flag architecture by using a native distributed flag architecture. The meth¬ od includes receiving an incoming instruction sequence using a global front end; grouping the instructions to form instruction blocks, wherein each of the instruction blocks comprise two half blocks; scheduling the instructions of the instruction block to ex¬ ecute in accordance with a scheduler; and using a distributed flag architecture to emulate a centralized flag architecture for the emu - lation of guest instruction execution.

WO 2014/151043 A1

A METHOD FOR EMULATING A GUEST CENTRALIZED FLAG ARCHITECTURE BY USING A NATIVE DISTRIBUTED FLAG ARCHITECTURE

This application claims the benefit co-pending commonly assigned US Provisional Patent Application serial number 61/800,487, titled "A METHOD FOR EMULATING A GUEST CENTRALIZED FLAG ARCHITECTURE BY USING A NATIVE DISTRIBUTED FLAG ARCHITECTURE" by Mohammad A. Abdallah, filed on March 15, 2013, and which is incorporated herein in its entirety.

CROSS REFERENCE TO RELATED APPLICATION

This application is related to co-pending commonly assigned US Patent Application serial number 2009/01 13170, titled "APPARATUS AND METHOD FOR PROCESSING AN INSTRUCTION MATRIX SPECIFYING PARALLEL INDEPENDENT OPERATIONS" by Mohammad A. Abdallah, filed on April 12, 2007, and which is incorporated herein in its entirety.

This application is related to co-pending commonly assigned US Patent Application serial number 2010/0161948, titled "APPARATUS AND METHOD FOR PROCESSING COMPLEX INSTRUCTION FORMATS IN A MULTITHREADED ARCHITECTURE SUPPORTING VARIOUS CONTEXT SWITCH MODES AND VIRTUALIZATION SCHEMES" by Mohammad A. Abdallah, filed on November 14, 2007, 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 selecting instructions comprising an instruction sequence.

BACKGROUND OF THE INVENTION

[002] Processors are required to handle multiple tasks that are either dependent or totally independent. The internal state of such processors usually consists of registers that might hold

different values at each particular instant of program execution. At each instant of program execution, the internal state image is called the architecture state of the processor.

[003] When code execution is switched to run another function (e.g., another thread, process or program), then the state of the machine/processor has to be saved so that the new function can utilize the internal registers to build its new state. Once the new function is terminated then its state can be discarded and the state of the previous context will be restored and execution resumes. Such a switch process is called a context switch and usually includes lO's or hundreds of cycles especially with modern architectures that employ large number of registers (e.g., 64, 128, 256) and/or out of order execution.

[004] In thread-aware hardware architectures, it is normal for the hardware to support multiple context states for a limited number of hardware-supported threads. In this case, the hardware duplicates all architecture state elements for each supported thread. This eliminates the need for context switch when executing a new thread. However, this still has multiple draw backs, namely the area, power and complexity of duplicating all architecture state elements (i.e., registers) for each additional thread supported in hardware. In addition, if the number of software threads exceeds the number of explicitly supported hardware threads, then the context switch must still be performed.

[005] This becomes common as parallelism is needed on a fine granularity basis requiring a large number of threads. The hardware thread-aware architectures with duplicate context-state hardware storage do not help non-threaded software code and only reduces the number of context switches for software that is threaded. However, those threads are usually constructed for coarse grain parallelism, and result in heavy software overhead for initiating and synchronizing, leaving fine grain parallelism, such as function calls and loops parallel execution, without efficient threading initiations/auto generation. Such described overheads are accompanied with the difficulty of auto parallelization of such codes using sate of the art compiler or user parallelization techniques for non-explicitly/easily parallelized/threaded software codes.

SUMMARY OF THE INVENTION

[006] In one embodiment, the present invention is implemented as a method for emulating a guest centralized flag architecture by using a native distributed flag architecture. The method includes receiving an incoming instruction sequence using a global front end; grouping the instructions to form instruction blocks, wherein each of the instruction blocks comprise two half blocks; scheduling the instructions of the instruction block to execute in accordance with a scheduler; and using a distributed flag architecture to emulate a centralized flag architecture for the emulation of guest instruction 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

[008] 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.

[009] Figure 1 shows an overview diagram of a process for grouping instructions into a block and tracking dependencies among the instructions by using a register template.

[010] Figure 2 shows an overview diagram of a register view, a source view, and an instruction view in accordance with one embodiment of the present invention.

[01 1] Figure 3 shows a diagram that illustrates an exemplary register template and how the source view is populated by information from the register template in accordance with one embodiment of the present invention.

[012] Figure 4 shows a diagram illustrating a first embodiment for dependency broadcasting within source view. In this embodiment, each column comprises an instruction block.

[013] Figure 5 shows a diagram illustrating a second embodiment for dependency broadcasting within source view.

[014] Figure 6 shows a diagram illustrating the selection of ready blocks for dispatch starting from the commit pointer and broadcasting the corresponding port assignments in accordance with one embodiment of the present invention.

[015] Figure 7 shows an adder tree structure that is used to implement the selector array described in Figure 6 in accordance with one embodiment of the present invention.

[016] Figure 8 shows exemplary logic of a selector array adder tree in greater detail.

[017] Figure 9 shows a parallel implementation of the adder tree for implementing a selector array in accordance with one embodiment of the present invention.

[018] Figure 10 shows an exemplary diagram illustrating how adder X from Figure 9 can be implemented by using carry save adders in accordance with one embodiment of the present invention.

[019] Figure 11 shows a masking embodiment for masking ready bits for scheduling starting from the commit pointer and using the selector array adders in accordance with of the present invention.

[020] Figure 12 shows an overview diagram of how register view entries are populated by register templates in accordance with one embodiment of the present invention.

[021] Figure 13 shows a first embodiment for reduced register view footprint in accordance with one embodiment of the present invention.

[022] Figure 14 shows a second embodiment for reduced register footprint in accordance with one embodiment of the present invention.

[023] Figure 15 shows an exemplary format of the delta between snapshots in accordance with one embodiment of the present invention.

[024] Figure 16 shows a diagram of a process for creating register template snapshots upon allocations of blocks of instructions in accordance with one embodiment of the present invention.

[025] Figure 17 shows another diagram of a process for creating register template snapshots upon allocations of blocks of instructions in accordance with one embodiment of the present invention.

[026] Figure 18 shows an overview diagram of hardware for implementing the serial implementation of creating a subsequent register template from a previous register template in accordance with one embodiment of the present invention.

[027] Figure 19 shows an overview diagram of hardware for implementing a parallel implementation of creating a subsequent register template from a previous register template in accordance with one embodiment of the present invention.

[028] Figure 20 shows an overview diagram of the hardware for instruction block-based execution and how it works with the source view, the instruction view, the register templates, and the register view in accordance with one embodiment of the present invention.

[029] Figure 21 shows an example of a chunking architecture in accordance with one embodiment of the present invention.

[030] Figure 22 shows a depiction of how threads are allocated in accordance with their block numbers and thread ID in accordance with one embodiment of the present invention.

[031] Figure 23 shows an implementation of a scheduler using thread pointer maps that point to physical storage locations in order to manage multithreaded execution in accordance with one embodiment of the present invention.

[032] Figure 24 shows another implementation of a scheduler using thread based pointer maps in accordance with one embodiment of the present invention.

[033] Figure 25 shows a diagram of a dynamic calendar-based allocation of execution resources to threads in accordance with one embodiment of the present invention.

[034] Figure 26 diagrams a dual dispatch process in accordance with one embodiment of the present invention.

[035] Figure 27 diagrams a dual dispatch transient multiply-accumulate in accordance with one embodiment of the present invention.

[036] Figure 28 diagrams a dual dispatch architecturally visible state multiply-add in accordance with one embodiment of the present invention.

[037] Figure 29 shows an overview diagram of a fetch and formation of instruction blocks for execution on grouped execution units process in accordance with one embodiment of the present invention.

[038] Figure 30 shows an exemplary diagram of instruction grouping in accordance with one embodiment of the present invention. In the Figure 30 embodiment two instructions are shown with a third auxiliary operation.

[039] Figure 31 shows how half block pairs within a block stack maps onto the execution block units in accordance with one embodiment of the present invention.

[040] Figure 32 shows a diagram depicting intermediate block results storage as a first level register file in accordance with one embodiment of the present invention.

[041] Figure 33 shows an odd/even ports scheduler in accordance with one embodiment of the present invention.

[042] Figure 34 shows a more detailed version of Figure 33 where four execution units are shown receiving results from the scheduler array and writing outputs to a temporary register file segment.

[043] Figure 35 shows a diagram depicting guest flag architecture emulation in accordance with one embodiment of the present invention.

[044] Figure 36 shows a diagram illustrating the front end of the machine the scheduler and the execution units and a centralized flag register in accordance with one embodiment of the present invention.

[045] Figure 37 shows a diagram of a centralized flag register emulation process as implemented by embodiments of the present invention.

[046] Figure 38 shows a flowchart of the steps of a process 3800 of emulating centralized flag register behavior in a guest setting.

DETAILED DESCRIPTION OF THE INVENTION

[047] 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.

[048] 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.

[049] 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.

[050] 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.

[051] 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.

[052] Figure 1 shows an overview diagram of a process for grouping instructions into a block and tracking dependencies among the instructions by using a register template.

[053] Figure 1 shows an instruction block having a header and a body. The block is created from a group of instructions. The block comprises an entity that encapsulates the group of instructions. In the present embodiment of the microprocessor, the level of abstraction is raised to blocks instead of individual instructions. Blocks are processed for dispatch instead of individual instructions. Each block is labeled with a block number. The machine's out of order management

job is thereby greatly simplified. One key feature is to find a way to manage a larger number of instructions being processed without greatly increasing the management overhead of the machine.

[054] Embodiments of the present invention achieves this objective by implementing instruction blocks, register templates and inheritance vectors. In the block shown in Figure 1, the header of the block lists and encapsulates all the sources and destinations of the instructions of the block and where those sources come from (e.g., from which blocks). The header includes the destinations that update the register template. The sources included in the header will be concatenated with the block numbers stored in the register template.

[055] The number of instructions that are processed out of order determines the management complexity of the out of order machine. More out of order instructions leads to greater complexity. Sources need to compare against destinations of prior instructions in the out of order dispatch window of the processor.

[056] As shown in Figure 1, the register template has fields for each register from R0 to R63. Blocks write their respective block numbers into the register template fields that correspond to the block destinations. Each block reads the register fields that represent its register sources from the register template. When a block retires and writes its destination register contents into the register file, its number is erased from the register template. This means that those registers can be read as sources from the register file itself.

[057] In the present embodiment, the register template is updated each cycle of the machine whenever a block is allocated. As new template updates are generated, prior snapshots of the register templates are stored into an array (e.g., the register view shown in Figure 2), one per block. This information is retained until the corresponding block is retired. This allows the machine to recover from miss-predictions and flushes very quickly (e.g., by obtaining the last known dependency state).

[058] In one embodiment, the register templates stored in the register view can be compressed (thereby saving storage space) by storing only the delta between successive snapshots (incremental changes between snapshots). In this manner the machine obtains a shrunk register

view. Further compression can be obtained by only storing templates for blocks that have a branch instruction.

[059] If a recovery point is needed other than a branch miss-prediction, then a recovery is first obtained at the branch recovery point, then state can be rebuilt out of allocating instructions (but not executing them) until the machine reaches the sought after recovery point.

[060] It should be noted that in one embodiment, the term "register template" as used herein is synonymous with the term "inheritance vectors" as described in the earlier filed commonly assigned patent application "EXECUTING INSTRUCTION SEQUENCE CODE BLOCKS BY USING VIRTUAL CORES INSTANTIATED BY PARTITIONABLE ENGINES" by Mohammad Abdallah, filed on March 23, 2012, serial number 13428440, which is incorporated herein in its entirety.

[061] Figure 2 shows an overview diagram of a register view, a source view, and an instruction view in accordance with one embodiment of the present invention. This figure shows one embodiment of a scheduler architecture (e.g., having a source view, instruction view, register view, etc.). Other implementations of a scheduler architecture that achieves the same functionality by combining or splitting one or more of the above cited structures are possible.

[062] Figure 2 diagrams the functional entities supporting the operation of the register templates and retention of the machine state. The left-hand side of Figure 2 shows register templates T0 through T4, with the arrows indicating the inheritance of information from one register template/inheritance vector to the next. The register view, source view, and instruction view each comprise data structures for storing information which relates to the blocks of instructions. Figure 2 also shows an exemplary instruction block having a header and how the instruction block includes both sources and destinations for the registers of the machine. Information about the registers referred to by the blocks is stored in the register view data structure. Information about the sources referred to by the blocks is stored in the source view data structure. Information about the instructions themselves referred to by the blocks is stored in the instruction view data structure. The register templates/inheritance vectors themselves comprise data structures storing dependency and inheritance information referred to by the blocks.

[063] Figure 3 shows a diagram that illustrates an exemplary register template and how the source view is populated by information from the register template in accordance with one embodiment of the present invention.

[064] In the present embodiment, it should be noted that the goal of the source view is to determine when particular blocks can be dispatched. When a block is dispatched it broadcasts its block number to all remaining blocks. Any matches for sources of the other blocks (e.g., a compare) causes a ready bit (e.g., or some other type of indicator) to be set. When all ready bits are set (e.g., AND gate) the block is ready to be dispatched. Blocks are dispatched based on the readiness of other blocks they depend on.

[065] When multiple blocks are ready for dispatch, the oldest block is chosen for dispatch ahead of younger blocks. For example, in one embodiment a find first circuit can be used to find the oldest block based on proximity to a commit pointer and subsequent blocks based on relative proximity to the commit pointer (e.g., working on each block's ready bit).

[066] Referring still to Figure 3, in this example, the register template snapshot created at the arrival of block 20 is being examined. As described above, the register template has fields for each register from R0 to R63. Blocks write their respective block numbers into the register template fields that correspond to the block destinations. Each block reads the register fields that represent its register sources from the register template. The first number is the block that wrote to the register and the second number is the destination number of that block.

[067] For example, when block 20 arrives, it reads the snapshot of the register template and looks up its own register sources in the register template to determine the latest block that wrote to each of its sources and populate the source view according to the updates that its destinations make to the previous register template snapshot Subsequent blocks , will update the register template with their own destinations. This is shown in the bottom left of Figure 3, where block 20 populates its sources : source 1, source 2, source 3, all the way to source 8.

[068] Figure 4 shows a diagram illustrating a first embodiment for dependency broadcasting within source view. In this embodiment, each column comprises an instruction block.

When a block is allocated it marks (e.g., by writing 0) in all the block's columns where ever its sources have dependency on those blocks. When any other block is dispatched its number is broadcasted across the exact column that relates to that block. It should be noted that writing a 1 is the default value indicating that there is no dependency on that block.

[069] When all ready bits in a block are ready, that block is dispatched and its number is broadcast back to all the remaining blocks. The block number compares against all the numbers stored in the sources of the other blocks. If there is a match, the ready bit for that source is set. For example, if the block number broadcasted on source 1 equals 11 then the ready bit for source 1 of block 20 will be set.

[070] Figure 5 shows a diagram illustrating a second embodiment for dependency broadcasting within source view. This embodiment is organized by sources as opposed to being organized by blocks. This is shown by the sources SI through S8 across the source view data structure. In a manner similar to as described with Figure 4 above, in the Figure 5 embodiment, when all ready bits in a block are ready, that block is dispatched and its number is broadcast back to all the remaining blocks. The block number compares against all the numbers stored in the sources of the other blocks. If there is a match, the ready bit for that source is set. For example, if the block number broadcasted on source 1 equals 11 then the ready bit for source 1 of block 20 will be set.

[071] The Figure 5 embodiment also shows how the compares are only enabled on the blocks between the commit pointer and the allocate pointer. All other blocks are invalid.

[072] Figure 6 shows a diagram illustrating the selection of ready blocks for dispatch starting from the commit pointer and broadcasting the corresponding port assignments in accordance with one embodiment of the present invention. The source view data structure is shown on the left-hand side of Figure 6. The instruction view data structure is shown on the right-hand side of Figure 6. A selector array is shown between the source view and the instruction view. In this embodiment, the selector array dispatches four blocks per cycle via the four dispatch ports PI through P4.

[073] As described above, blocks are selected for dispatch from the commit pointer wrapping around to allocate pointer (e.g., trying to honor dispatching older blocks first). The

selector array is used to find the first 4 ready blocks starting from the commit pointer. It is desired to dispatch the oldest ready blocks. In one embodiment, the selector array can be implemented by using an adder tree structure. This will be described in Figure 7 below.

[074] Figure 6 also shows how the selector array is coupled to each of the four ports that passed through the entries in the instruction view. In this embodiment, the port couplings as port enables, and enable one of the four ports to be activated and for that instruction view entry to pass through down to the dispatch port and on to the execution units. Additionally, as described above, dispatched blocks are broadcast back through the source view. The block numbers of selected blocks for dispatch are broadcast back (up to 4). This is shown on the far right-hand side of Figure 6.

[075] Figure 7 shows an adder tree structure that is used to implement the selector array described in Figure 6 in accordance with one embodiment of the present invention. The depicted adder tree implements the functionality of the selector array. The adder tree picks the first four ready blocks and mounts them to the four available ports for dispatch (e.g., read port 1 through read port 4). No arbitration is used. The actual logic that is used to specifically enable a specific port is explicitly shown in entry number 1. For the sake of clarity, the logic is not specifically show in the other entries. In this manner, Figure 7 shows one specific embodiment of how the direct selection of each particular port for block dispatch is implemented. It should be noted however, that alternatively, an embodiment that uses priority encoders can be implemented.

[076] Figure 8 shows exemplary logic of a selector array adder tree in greater detail. In the Figure 8 embodiment, logic is shown for a range exceed bit. The range exceed bit ensures that no more than four blocks will be selected for dispatch if a fifth block is ready the range exceed bit will not allow it to be dispatched if the first four also ready. It should be noted that the sum bits are S 0 to S 3 are both used to enable the dispatch port as well as propagation to the next adder stage in the serial implementation.

[077] Figure 9 shows a parallel implementation of the adder tree for implementing a selector array in accordance with one embodiment of the present invention. The parallel implementation does not forward the sum from each adder to the next. In the parallel

implementation, each adder uses all its necessary inputs directly using a multiple input addition implementation, such as multi-input carry save adder trees. For example, the adder "X" sums all of the previous inputs. This parallel implementation is desirable in order to execute faster compute times (e.g., single cycle).

[078] Figure 10 shows an exemplary diagram illustrating how adder X from Figure 9 can be implemented by using carry save adders in accordance with one embodiment of the present invention. Figure 10 shows a structure that can add 32 inputs in a single cycle. The structure is put together using 4-by-2 carry save adders.

[079] Figure 11 shows a masking embodiment for masking ready bits for scheduling starting from the commit pointer and using the selector array adders in accordance with of the present invention. In this implementation, the selector array adders are trying to select first 4 ready blocks to dispatch starting from the commit pointer potentially wrapping around to the allocate pointer. In this implementation, multi-input parallel adders are used. Additionally, in this implementation a source of these circular buffer is utilized.

[080] Figure 11 shows how the ready bits are ANDed together with each of the two masks (individually or separately) and applied to the two adder trees in parallel. The first four are selected by using the two adder trees and comparing against the threshold of four. The "X" marks denote "exclude from the selection array for that adder tree" thus the "X" value is zero. On the other hand the "Y" marks denote "do include in the selection array for that adder tree" thus the "Y" value is one.

[081] Figure 12 shows an overview diagram of how register view entries are populated by register templates in accordance with one embodiment of the present invention.

[082] As described above, register view entries are populated by register templates. The register view stores snapshots of register templates for each block in sequence. When a speculation is not valid (e.g., a branch miss-prediction), the register view has a latest valid snapshot before the invalid speculation point. The machine can roll back its state to the last valid snapshot by reading that register view entry and loading it into the base of the register template. Each entry of register

view shows all of the register inheritance states. For example in the Figure 12 embodiment, if the register view for block F is invalid, the machine state can be rolled back to an earlier last valid register template snapshot.

[083] Figure 13 shows a first embodiment for reduced register view footprint in accordance with one embodiment of the present invention. The amount of memory needed to store the register view entries can be reduced by only storing those register view template snapshots that contain branch instructions. When an exception occurs (e.g., a speculation is not valid, a branch miss-prediction, etc.), the last valid snapshot can be rebuilt from the branch instruction that occurred prior to the exception. Instructions are fetched from the branch prior to the exception down to the exception in order to build the last valid snapshot. The instructions are fetched but they are not executed. As shown in Figure 13, only those snapshots that include branch instructions are saved in the reduced register view. This greatly reduces the amount of memory needed to store the register template snapshots.

[084] Figure 14 shows a second embodiment for reduced register footprint in accordance with one embodiment of the present invention. The amount of memory needed to store the register view entries can be reduced by only storing a sequential subset of the snapshots (e.g., one out of every four snapshots). The change between successive snapshots can be stored as a "delta" from an original snapshot using a comparatively smaller amount of memory than full successive snapshots. When an exception occurs (e.g., a speculation is not valid, a branch miss-prediction, etc.), the last valid snapshot can be rebuilt from the original snapshot that occurred prior to the exception. The "delta" from the original snapshot that occurred prior to the exception and the successive snapshots are used to rebuild the last valid snapshot. The initial original state can accumulate deltas to arrive to the state of the required snapshot.

[085] Figure 15 shows an exemplary format of the delta between snapshots in accordance with one embodiment of the present invention. Figure 15 shows an original snapshot and two deltas. In one delta, R5 and R6 are the only registers being updated by B3. The rest of the entries are not changed. In another Delta, R1 and R7 are the only registers being updated by B2. The rest of the entries are not changed.

[086] Figure 16 shows a diagram of a process for creating register template snapshots upon allocations of blocks of instructions in accordance with one embodiment of the present invention. In this embodiment, the left-hand side of Figure 16 shows two de-multiplexers and at the top of Figure 16 is a snapshot register template. Figure 16 shows a diagram for creating a subsequent register template from a previous register template (e.g., a serial implementation).

[087] This serial implementation shows how register template snapshots are created upon allocation of blocks of instructions. Those snapshots serves to capture the latest register architectural states update that are used for dependency tracking (e.g., as described in Figures 1 through 4) as well as updating the register view for handling miss-predictions/exceptions (e.g., as described in Figures 12 through 15).

[088] The de-mux functions by selecting which incoming source is passed on. For example, register R2 will de-mux to a 1 at the second output, while R8 will de-mux to a 1 at the seventh output, and so on.

[089] Figure 17 shows another diagram of a process for creating register template snapshots upon allocations of blocks of instructions in accordance with one embodiment of the present invention. The Figure 17 embodiment also shows the creating of a subsequent register template from a previous register template. The Figure 17 embodiment also shows an example of register template block inheritance. This Figure shows an example of how the register template is updated from allocated block numbers. For example, block Bf updates R2, R8, and R10. Bg updates R1 and R9. The dotted arrows indicate that the values are inherited from the prior snapshot. This process proceeds all the way down to block Bi. Thus, for example, since no snapshot updated register R7, its original value Bb will have propagated all the way down.

[090] Figure 18 shows an overview diagram of hardware for implementing the serial implementation of creating a subsequent register template from a previous register template in accordance with one embodiment of the present invention. The de-multiplexer is used to control a series of two input multiplexers which of two block numbers will be propagated down to the next stage. It can either be the block number from the previous stage or the current block number.

[091] Figure 19 shows an overview diagram of hardware for implementing a parallel implementation of creating a subsequent register template from a previous register template in accordance with one embodiment of the present invention. This Parallel implementation uses special encoded multiplexer controls to create a subsequent register template from a previous register template.

[092] Figure 20 shows an overview diagram of the hardware for instruction block-based execution and how it works with the source view, the instruction view, the register templates, and the register view in accordance with one embodiment of the present invention.

[093] In this implementation, the allocator scheduler in dispatcher receives instructions fetched by the machine's front end. These instructions go through block formation in the manner we described earlier. As described earlier the blocks yield register templates and these register templates are used to populate the register view. From the source view the sources are transferred to the register file hierarchy and there are broadcasts back to the source view in the manner described above. The instruction view transfers instructions to the execution units. The instructions are executed by the execution units as the sources needed by the instructions coming from the register file hierarchy. These executed instructions are then transferred out of the execution unit and back into the register file hierarchy.

[094] Figure 21 shows an example of a chunking architecture in accordance with one embodiment of the present invention. The importance of chunking is that it reduces the number of write ports into each scheduler entry from 4 to 1 by using the four multiplexers shown, while still densely packing all the entries without forming bubbles.

[095] The importance of chunking can be seen by the following example (e.g., noting that allocation of blocks in each cycle starts at the top position, in this case B0). Assuming in cycle 1, three blocks of instructions are to be allocated to the scheduler entries (e.g., the three blocks will occupy the first 3 entries in the scheduler). In the next cycle (e.g., cycle 2) another two blocks of instructions are to be allocated. In order to avoid creating bubbles in the scheduler array entries, the scheduler array entries have to be built with support for four write ports. This is expensive in terms of power consumption, timing, area, and the like. The chunking structure above simplifies all

scheduler arrays to only have one write port by using the multiplexing structure before allocating to the arrays. In the above example, the B0 in cycle two will be selected by the last mux while B1 in cycle two will be selected by the first mux (e.g., going from left to right).

[096] In this manner, each for entry chunk only needs one write port per entry and four read ports per entry. There is a trade-off in cost because the multiplexers must be implemented, however that cost is made up many times over in the savings from not having to implement four write ports per entry, as there can be very many entries.

[097] Figure 21 also shows an intermediate allocation buffer. If the scheduler arrays cannot accept all the chunks sent to them, then they can be stored temporarily in the intermediate allocation buffer. When the scheduler arrays have free space, the chunks will be transferred from the intermediate allocation buffer to the scheduler arrays.

[098] Figure 22 shows a depiction of how threads are allocated in accordance with their block numbers and thread ID in accordance with one embodiment of the present invention. Blocks are allocated to the scheduler array via a chunking implementation as described above. Each of the thread blocks maintain a sequential order among themselves using the block number. The blocks from different threads can be interleaved (e.g., Blocks for thread Thl and blocks for thread Th2 are interleaved in the scheduler array. In this manner, blocks from different threads are present within the scheduler array.

[099] Figure 23 shows an implementation of a scheduler using thread pointer maps that point to physical storage locations in order to manage multithreaded execution in accordance with one embodiment of the present invention. In this embodiment, management of the threads is implemented through the control of the thread maps. For example here Figure 23 shows thread 1 map and thread 2 map. The maps track the location of the blocks of the individual thread. The entries in the map .2 physical storage locations the entries in the map are allocated to blocks belonging to that thread. In this implementation, each thread has an allocation counter that counts for both threads. The overall count cannot exceed N divided by 2 (e.g., exceeding space available). The allocation counters have adjustable thresholds in order to implement fairness in the allocation of

the total entries from the pool. The allocation counters can prevent one thread from using all of the available space.

[0100] Figure 24 shows another implementation of a scheduler using thread based pointer maps in accordance with one embodiment of the present invention. Figure 24 shows a relationship between the commit pointer and the allocation pointer. As shown, each thread has a commit pointer and an allocate pointer the arrow shows how reality pointer for thread 2 can wrap around the physical storage allocating blocks B1 and B2, but it cannot allocate block B9 until the commit pointer for thread 2 moves down. This is shown by the position of the commit pointer of thread 2 and the strikethrough. The right-hand side of Figure 24 shows a relationship between the allocation of blocks and the commit pointer as it moves around counterclockwise.

[0101] Figure 25 shows a diagram of a dynamic calendar-based allocation of execution resources to threads in accordance with one embodiment of the present invention. Fairness can be dynamically controlled using the allocate counters based on the forward progress of each thread. If both threads are making substantial forward progress, then both allocation counters are set to the same threshold (e.g., 9). However if one thread makes slow forward progress, such as suffering from an L2 cache miss or such events, then the ratio of the threshold counters can be adjusted in the favor of the thread that is still making substantial forward progress. If one thread is stalled or suspended (e.g., is in wait or spin state waiting on an OS or IO response) the ratio can be completely adjusted to the other thread with the exception of a single return entry that is reserved for the suspended thread to signal the release of the wait state.

[0102] In one embodiment, the process starts off with a ratio of 50%: 50%. Upon the L2 cache miss detection on block 22, the front end of the pipeline stalls any further fetch into the pipeline or allocation into the scheduler of thread 2 blocks. Upon retirement of thread 2 blocks from the scheduler, those entries will be made available for thread 1 allocation until the point where the new dynamic ratio of thread allocation is achieved. For example, 3 out the recently retired thread 2 blocks will be returned to the pool for allocation to thread 1 instead of thread 2, making the thread 1 to thread 2 ratio 75% : 25%.

[0103] It should be noted that a stall of thread 2 blocks in the front of the pipeline might require flushing those blocks from the front of the pipeline if there is no hardware mechanism to bypass them (e.g., by thread 1 blocks by passing the stalled thread 2 blocks).

[0104] Figure 26 diagrams a dual dispatch process in accordance with one embodiment of the present invention. Multi- dispatch generally encompasses dispatching a block (having multiple instruction within) multiple times such that different instructions with the block can execute on each pass through the execution units. One example would be a dispatch of an address calculation instruction followed by a subsequent dispatch that consumes the resulting data. Another example would be a floating point operation, where the first part is executed as fixed point operation and the second part is executed to complete the operation by performing rounding, flag generation/calculation, exponent adjustment or the like. Blocks are allocated, committed and retired atomically as a single entity.

[0105] A main benefit of multi-dispatch is that it avoids allocating multiple separate blocks into the machine window, thereby making the machine window effectively larger. A larger machine window means more opportunities for optimization and reordering.

[0106] Looking at the bottom left the Figure 26, there is an instruction block depicted. This block cannot be dispatched in a single cycle because there is latency between the load address calculation and the load returning data from the caches/memory. So this block is first dispatched with its intermediate result being held as a transient state (its result is being delivered on the fly to the second dispatch without being visible to the architectural state). The first dispatch sends the two components 1 and 2 that are used in the address calculation and the dispatch of the LA. The second dispatch sends components 3 and 4 which are the execution parts of the load data upon the load returning data from the caches/memory.

[0107] Looking at the bottom right of Figure 26 there is a floating point multiply accumulate operation depicted. In the case where the hardware does not have sufficient bandwidth of incoming sources to dispatch the operation in a single phase, then dual dispatch is used, as the multiply accumulate figure shows. The first dispatch is a fixed point multiply as shown. The second

dispatch is a floating point addition rounding as shown. When both of these dispatched instructions execute, they effectively perform the floating point multiply/accumulate.

[0108] Figure 27 diagrams a dual dispatch transient multiply-accumulate in accordance with one embodiment of the present invention. As shown in Figure 27, the first dispatch is the integer 32 bit multiply, and the second dispatch is the integer accumulate add. State communicated between the first dispatch and the second dispatch (the result of the multiply) is transient and not architecturally visible. The transient storage in one implementation can hold results of more than one multiplier and can tag them to identify the corresponding multiply accumulate pair, thereby allowing intermix of multiple multiply accumulate pairs being dispatch in an arbitrary fashion (e.g., interleaved, etc.).

[0109] Note that other instructions can use this same hardware for their implementation (e.g., floating point, etc.).

[0110] Figure 28 diagrams a dual dispatch architecturally visible state multiply-add in accordance with one embodiment of the present invention. The first dispatch is the single precision multiply, and the second dispatch is the single precision add. In this implementation, state information communicated between the first dispatch and the second dispatch (e.g., the result of the multiply) is architecturally visible since this storage is an architecture state register.

[0111] Figure 29 shows an overview diagram of a fetch and formation of instruction blocks for execution on grouped execution units process in accordance with one embodiment of the present invention. Embodiments of the present invention utilize a process whereby instructions are fetched and formed as blocks by the hardware or dynamic converter/JIT. The instructions in the blocks are organized such that a result of an early instruction in the block feeds a source of a subsequent instruction in the block. This is shown by the dotted arrows in the block of instructions. This property enables the block to execute efficiently on the stacked execution units of the execution block. Instructions can also be grouped even if they can execute in parallel, such as if they share the same source (not shown explicitly in this figure).

[01 12] One alternative to forming the blocks in hardware is to form them in software (statically or at runtime) where instruction pairs, triplets, quads, etc., are formed.

[01 13] Other implementations of instruction grouping functionality can be found in commonly assigned US patent 8,327,1 15.

[01 14] Figure 30 shows an exemplary diagram of instruction grouping in accordance with one embodiment of the present invention. In the Figure 30 embodiment two instructions are shown with a third auxiliary operation. The left-hand side of Figure 3 1 instruction block comprising an upper half block/1 slot and a lower half block/1 slot. The vertical arrows coming down from the top indicates sources coming into the block while the vertical arrows going down from the bottom indicate destinations going back to memory. Proceeding from the left-hand side of Figure 3 towards the right-hand side, different instruction combinations that are possible are illustrated. In this implementation, each half block can receive three sources and can pass on two destinations. OP1 and OP2 are normal operations. AuxiliaryOPs are auxiliary operations such as a logical, a shift, a move, a sign extend, a branch, etc. The benefit of dividing the block into two halves is to allow the benefit of having each half dispatch on its own independently or otherwise together as one block dynamically (either for port utilization or because of resource constrains) based on dependency resolution, thus having better utilization of execution times, at the same time having the 2 halves correspond to one block allows the machine to abstract the complexity of 2 half blocks to be managed like one block(i.e. at allocate and retirement).

[01 15] Figure 3 1 shows how half block pairs within a block stack maps onto the execution block units in accordance with one embodiment of the present invention. As shown in the execution block, each execution block has two slots, slot 1 and slot 2. The objective is to s map the block onto the execution units such that the first half block executes on slot 1 and the second half block executes on slot 2. The objective is to allow the 2 half blocks to dispatch independently if the instruction group of each half block does not depend on the other half. The paired arrows coming into the execution block from the top are two 32-bit words of a source. The paired arrows leaving the execution block going down are two 32-bit words of a destination. Going from left to right of Figure 31, different exemplary combinations of instructions are shown that are capable of being stacked onto the execution block units.

[01 16] The top of Figure 3 1 summarizes how the pairs of half blocks execute in a full block context or any half block context. Each of the s Execution blocks have two slots/half blocks and each one of the half bocks/execution slots executes either a single, paired or triplet grouped operations. There are four types of block execution types. The first is parallel halves (which allows each half block to independently execute once its own sources are ready but the 2 half blocks can still execute as one block on one execution unit if both halves are ready at the same time. The second is atomic parallel halves (which refers to half blocks that can execute in parallel because there is no dependency between the 2 halves but they are forced to execute together as one block because the resource sharing between the 2 halves make it preferred or necessary for the two halves to execute together atomically within the constraint of the resources available in each execution block). The third type is atomic serial halves s (which requires the first half to forward data to the second half, through transient forwarding with or without internal storage). The fourth type is sequential halves (as in dual dispatch) where the 2nd half depend on the first half and is dispatched on a later cycle than the first one and forwards the data through external storage that are tracked for dependency resolution, similar to the dual dispatch case..

[01 17] Figure 32 shows a diagram depicting intermediate block results storage as a first level register file in accordance with one embodiment of the present invention. Each group of registers represent a block of instructions (representing two half blocks) in which both 32 bit results as well as 64 bits results can be supported by using two 32 bit registers to support one 64 bit register. The storage per block assumes a virtual block storage, which means two half blocks from different blocks can write into the same virtual block storage. Combined results' storage of two half blocks that make up one virtual block storage.

[01 18] Figure 33 shows an odd/even ports scheduler in accordance with one embodiment of the present invention. In this implementation, the result storage is asymmetrical. Some of the result storage is three 64 bit result registers per half block while others are one 64 bit result register per half block, however alternative implementation can use symmetrical storage per half block and additionally could also employ 64-bit and 32-bit partition as described in Figure 32. In these embodiments, storage is assigned per half block, as opposed to per block. This implementation reduces the number of ports needed for dispatch by using them as odd or even.

[01 19] Figure 34 shows a more detailed version of Figure 33 where four execution units are shown receiving results from the scheduler array and writing outputs to a temporary register file segment. The ports are attached at even and odd intervals. The left side of the scheduling array shows block numbers and the right side shows half block numbers.

[0120] Each core has even and odd ports into the scheduling array, where each port is connected to an odd or even half block position. In one implementation, the even ports and their corresponding half blocks can reside in a different core than the odd ports and their corresponding half blocks. In another implementation, the odd and even ports will be distributed across multiple different cores as shown in this figure. As described in the prior earlier filed commonly assigned patent application "EXECUTING INSTRUCTION SEQUENCE CODE BLOCKS BY USING VIRTUAL CORES INSTANTIATED BY PARTITIONABLE ENGINES" by Mohammad Abdallah, filed on March 23, 2012, serial number 13428440, which is incorporated herein in its entirety, the cores can be physical cores or virtual cores.

[0121] In certain types of blocks, one half of a block can be dispatched independently from the other half of the block. In other types of blocks, both halves of a block need to be dispatched simultaneously to the same execution block units. In still other types of blocks, the two halves of a block need to be dispatched sequentially (the second half after the first half).

[0122] Figure 35 shows a diagram depicting guest flag architecture emulation in accordance with one embodiment of the present invention. The left-hand side of Figure 35 shows a centralized flag register having five flags. The right-hand side of Figure 35 shows a distributed flag architecture having distributed flag registers wherein the flags are distributed amongst registers themselves.

[0123] During architecture emulation, it is necessary for the distributed flag architecture to emulate the behavior of the centralized guest flag architecture. Distributed flag architecture can also be implemented by using multiple independent flag registers as opposed to a flag field associated with a data register. For example, data registers can be implemented as R0 to R15 while independent flag registers can be implemented as F0 to F3. Those flag registers in this case are not associated directly with the data registers.

[0124] Figure 36 shows a diagram illustrating the front end of the machine the scheduler and the execution units and a centralized flag register in accordance with one embodiment of the present invention. In this implementation, the front end categorizes incoming instructions based on the manner in which they update guest instruction flags. In one embodiment, the guest instructions are categorized into 4 native instruction types, Tl, T2, T3, and T4. T1-T4 are instruction types that indicate which flag fields that each guest instruction type updates. Guest instruction types update different guest instruction flags, based on their type. For example, logical guest instructions update T1 native instructions.

[0125] Figure 37 shows a diagram of a centralized flag register emulation process as implemented by embodiments of the present invention. The actors in Figure 37 comprise a latest update type table, a renaming table extension, physical registers, and distributed flag registers. Figure 37 is now described by the flowchart of Figure 38.

[0126] Figure 38 shows a flowchart of the steps of a process 3800 of emulating centralized flag register behavior in a guest setting.

[0127] In step 3801, the front end/dynamic converter (hardware or software) categorizes incoming instructions based on the manner in which they update guest instruction flags. In one embodiment, the guest instructions are categorized into four flag architectural types, Tl, T2, T3, and T4. T1-T4 are instruction types that indicate which flag fields that each guest instruction type updates. Guest instruction types update different guest flags, based on their type. For example, logical guest instructions update T1 type flags, shift guest instructions update T2 type flags, arithmetic guest instructions update T3 type flags, and special guest instructions update type T4 flags. It should be noted that guest instructions can be architectural instruction representation while native can be what the machine internally executes (e.g., microcode). Alternatively, guest instructions can be instructions from an emulated architecture (e.g., x86, java, ARM code, etc.).

[0128] In step 3802, the order in which those instruction types update their respective guest flags is recorded in a latest update type table data structure. In one embodiment, this action is performed by the front end of the machine.

[0129] In step 3803, when those instruction types reach the Scheduler (the in-order part of the allocation/renaming stage), the scheduler assigns an implicit physical destination that corresponds to the architectural type and records that assignment in a renaming/mapping table data structure.

[0130] And in step 3804, when a subsequent guest instruction reaches the allocation/renaming stage in the scheduler, and that instruction wants to read guest flag fields, (a) the machine determines which flag architectural types need to be accessed to perform the read. (b) if all needed flags are found in the same latest update flag type (e.g., as determined by the latest update type table), then the corresponding physical register (e.g., that maps to that latest flag type) is read to obtain the needed flags. (c) if all needed flags cannot be found in a same latest update flag type, then each flag needs to be read from the corresponding physical register that maps to the individual latest update flag type.

[0131] And in step 3805, each flag is being read individually from the physical register that holds its latest value that was lastly updated, as tracked by the latest update flag type table.

[0132] It should be noted that if a latest update type is inclusive of another type then all then all subset types have to map to the same physical registers of the super set type.

[0133] At retirement, that destination flag fields are merged with a cloned centralized/guest flag architecture register. It should be noted that the cloning is performed due to the fact that the native architecture utilizes a distributed flag architecture as opposed to a single register centralized flag architecture.

[0134] Examples of instructions that update certain flag types:

[0135] CF,OF,SF,ZR- arithmetic instruction and load/write flags instructions

[0136] SF, ZF and conditional CF - logicals and shifts

[0137] SF, ZF - moves/loads, EXTR, some multiplies

[0138] ZF - POPCNT and STREX[P]

[0139] GE - SIMD instructions ???

[0140] Examples of conditions/predications that read certain flags:

[0141] 0000 EQ Equal Z == 1

[0142] 0001 NE Not equal, or Unordered Z == 0

[0143] 0010 CS b Carry set, Greater than or equal, or Unordered C == 1

[0144] 0011 CC c Carry clear, Less than C == 0

[0145] 0100 MI Minus, negative, Less than N == 1

[0146] 0101 PL Plus, Positive or zero, Greater than or equal to, Unordered N == 0011 0 VS Overflow, Unordered V == 1

[0147] 0111 VC No overflow, Not unordered V == 0

[0148] 1000 HI Unsigned higher, Greater than, Unordered C == 1 and Z == 0

[0149] 1001 LS Unsigned lower or same, Less than or equal C == 0 or Z == 1

[0150] 1010 GE Signed greater than or equal, Greater than or equal N == V

[0151] 1011 LT Signed less than, Less than, Unordered N != V

[0152] 1100 GT Signed greater than, Greater than Z == 0 and N == V

[0153] 1101 LE Signed less than or equal, Less than or equal, Unordered Z == 1 or N != V

[0154] 1110 None (AL), Always (unconditional), Any flag set to any value.

[0155] 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 method for emulating a guest centralized flag architecture by using a native distributed flag architecture, comprising:

receiving an incoming instruction sequence using a global front end;

grouping the instructions to form instruction blocks, wherein each of the instruction blocks comprise two half blocks;

scheduling the instructions of the instruction block to execute in accordance with a scheduler; and

using a distributed flag architecture to emulate a centralized flag architecture for the emulation of guest instruction execution.

2. The method of claim 1, wherein the distributed flag architecture emulates the behavior of a centralized guest flag architecture.

3. The method of claim 1, wherein a distributed flag architecture can be implemented by using multiple independent flag registers.

4. The method of claim 1, wherein , guest instructions are categorized into 4 native instruction types.

5. The method of claim 1, wherein guest instructions are categorized into 4 native instruction types and guest instruction types update different guest instruction flags, based on their type.

6. The method of claim 1, wherein a front end/dynamic converter categorizes incoming instructions based on the manner in which they update guest instruction flags.

7. A non-transitory computer readable media having computer readable code which when executed by a computer system causes the computer system to perform a method for emulating a guest centralized flag architecture by using a native distributed flag architecture, comprising:

receiving an incoming instruction sequence using a global front end;

grouping the instructions to form instruction blocks, wherein each of the instruction blocks comprise two half blocks;

scheduling the instructions of the instruction block to execute in accordance with a scheduler; and

using a distributed flag architecture to emulate a centralized flag architecture for the emulation of guest instruction execution.

8. The computer readable media of claim 7, wherein the distributed flag architecture emulates the behavior of a centralized guest flag architecture.

9. The computer readable media of claim 7, wherein a distributed flag architecture can be implemented by using multiple independent flag registers.

10. The computer readable media of claim 7, wherein , guest instructions are categorized into 4 native instruction types.

11. The computer readable media of claim 7, wherein guest instructions are categorized into 4 native instruction types and guest instruction types update different guest instruction flags, based on their type.

12. The computer readable media of claim 7, wherein a front end/dynamic converter categorizes incoming instructions based on the manner in which they update guest instruction flags.

13. A computer system having a processor coupled to a memory, the memory having computer readable code which when executed by the computer system causes the computer system to implement a method for emulating a guest centralized flag architecture by using a native distributed flag architecture, comprising:

receiving an incoming instruction sequence using a global front end;

grouping the instructions to form instruction blocks, wherein each of the instruction blocks comprise two half blocks;

scheduling the instructions of the instruction block to execute in accordance with a scheduler; and

using a distributed flag architecture to emulate a centralized flag architecture for the emulation of guest instruction execution.

14. The computer system of claim 13, wherein the distributed flag architecture emulates the behavior of a centralized guest flag architecture.

15. The computer system of claim 13, wherein a distributed flag architecture can be implemented by using multiple independent flag registers.

16. The computer system of claim 13, wherein , guest instructions are categorized into 4 native instruction types.

17. The computer system of claim 13, wherein guest instructions are categorized into 4 native instruction types and guest instruction types update different guest instruction flags, based on their type.

18. The computer system of claim 13, wherein a front end/dynamic converter categorizes incoming instructions based on the manner in which they update guest instruction flags.

19. A method for executing dual dispatch of blocks and half blocks, comprising:

receiving an incoming instruction sequence using a global front end;

grouping the instructions to form instruction blocks, wherein each of the instruction blocks comprise two half blocks;

scheduling the instructions of the instruction block to execute in accordance with a scheduler; and

performing a dual dispatch of the two half blocks for execution on an execution unit.

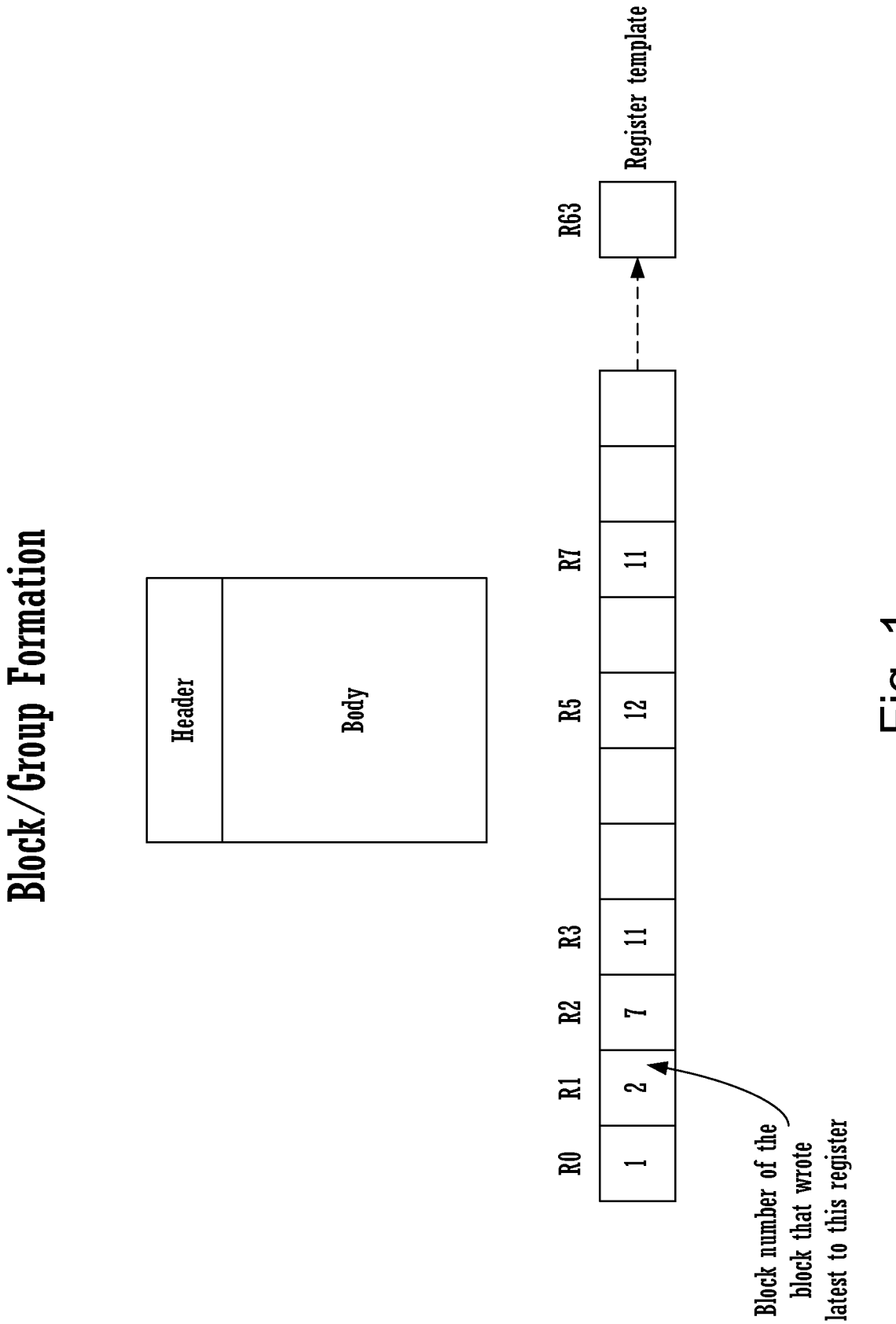

Block/Group Formation
Header
Body
R0
R1
R2
R3
R5
R7
R63
Register template
1
2
7
11
12
11
Block number of the block that wrote latest to this register

Fig. 1

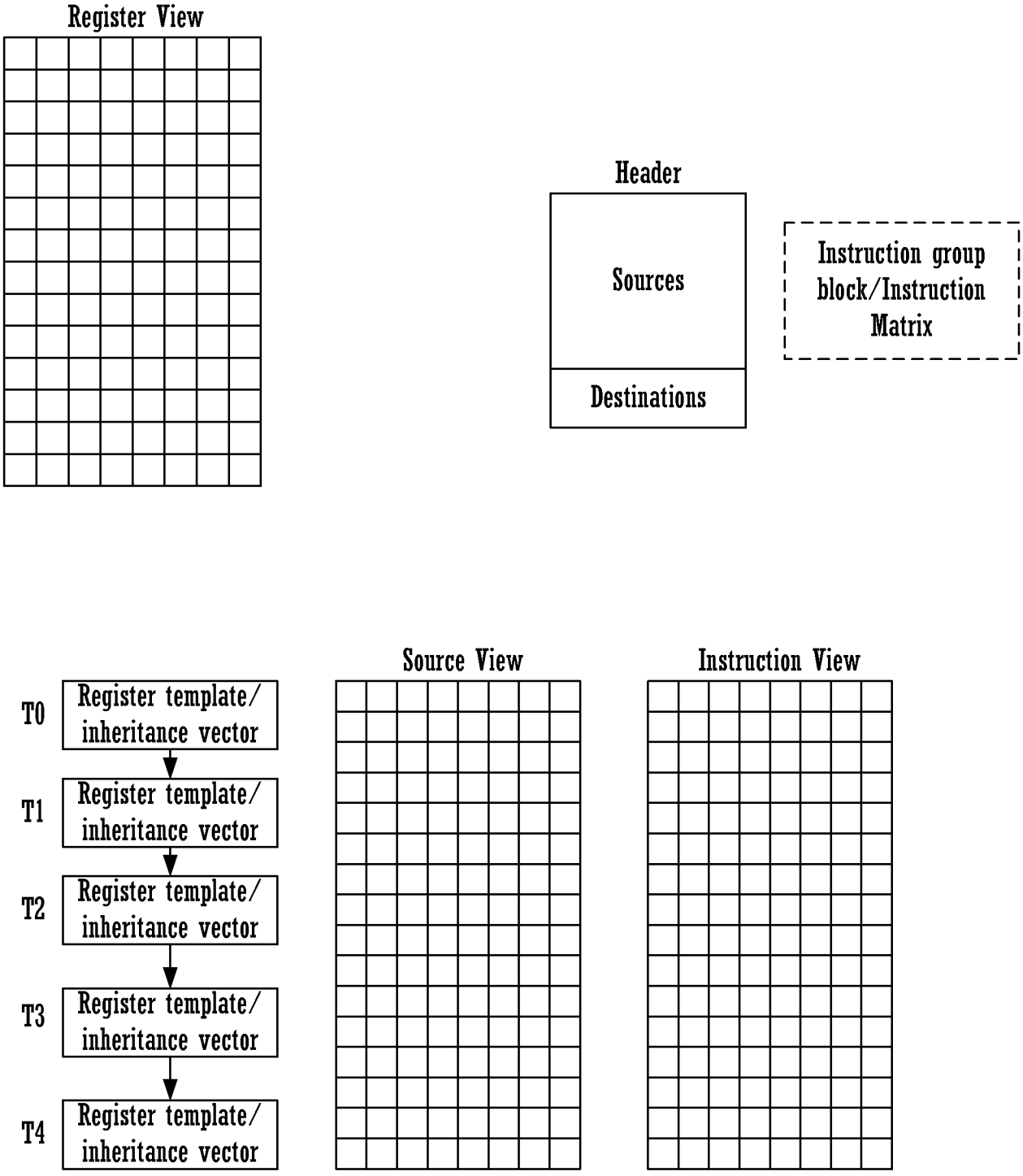

Register View
Header
Sources
Destinations
Instruction group block/Instruction Matrix
T0
Register template/ inheritance vector
T1
Register template/ inheritance vector
T2
Register template/ inheritance vector
T3
Register template/ inheritance vector
T4
Register template/ inheritance vector
Source View
Instruction View

Fig. 2

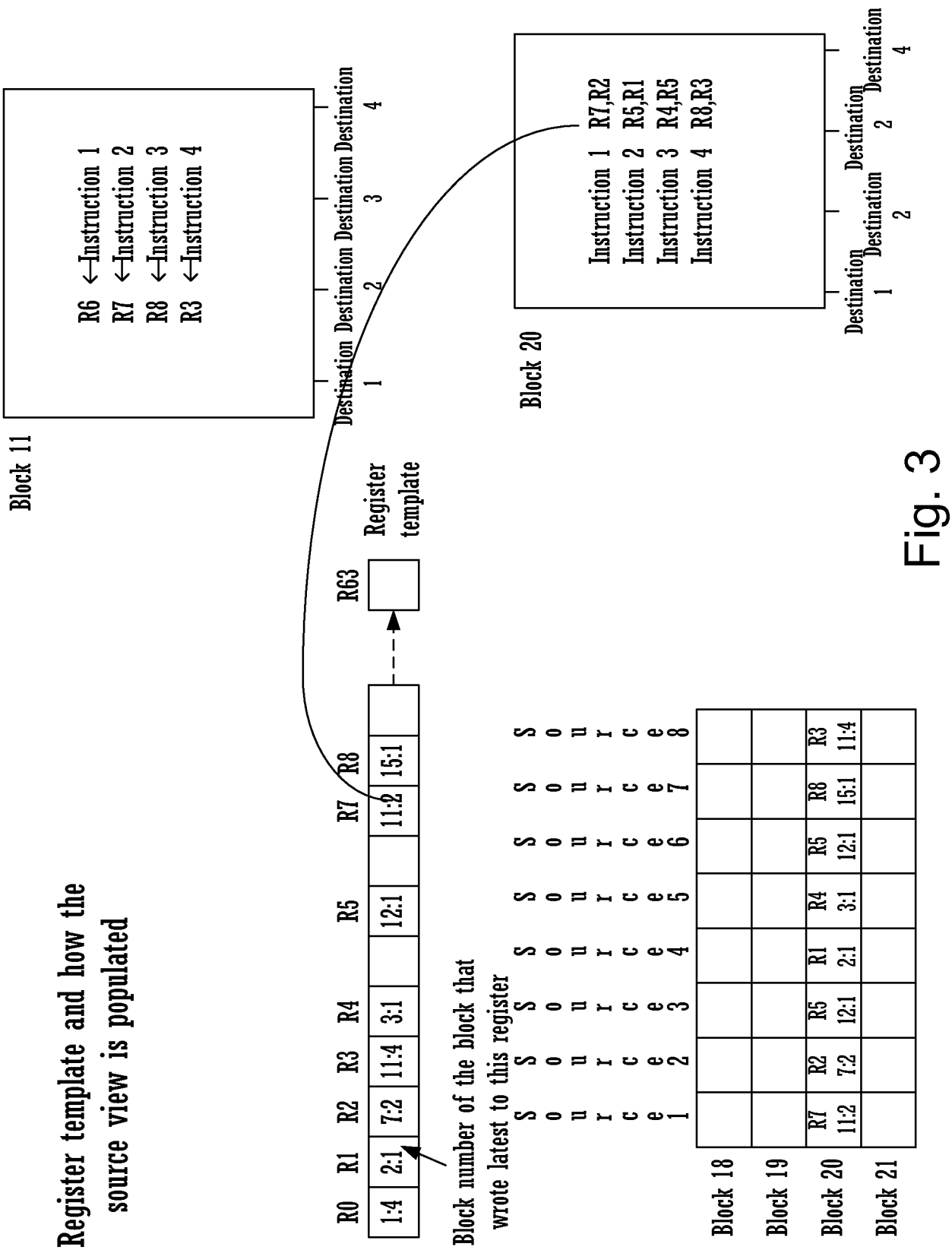

Register template and how the source view is populated
Block 11
R6 ←Instruction 1
R7 ←Instruction 2
R8 ←Instruction 3
R3 ←Instruction 4
Destination 1
Destination 2
Destination 3
Destination 4
Block 20
Instruction 1 R7,R2
Instruction 2 R5,R1
Instruction 3 R4,R5
Instruction 4 R8,R3
Destination 1
Destination 2
Destination 3
Destination 4
R63
Register template
R0 R1 R2 R3 R4 R5 R7 R8
1:4 2:1 7:2 11:4 3:1 12:1 11:2 15:1
Block number of the block that wrote latest to this register
S S S S S S S S
0 0 0 0 0 0 0 0
u u u u u u u u
r r r r r r r r
c c c c c c c c
e e e e e e e e
1 2 3 4 5 6 7 8
Block 18
Block 19
Block 20
Block 21
R7 11:2
R2 7:2
R5 12:1
R1 2:1
R4 3:1
R5 12:1
R8 15:1
R3 11:4

Fig. 3

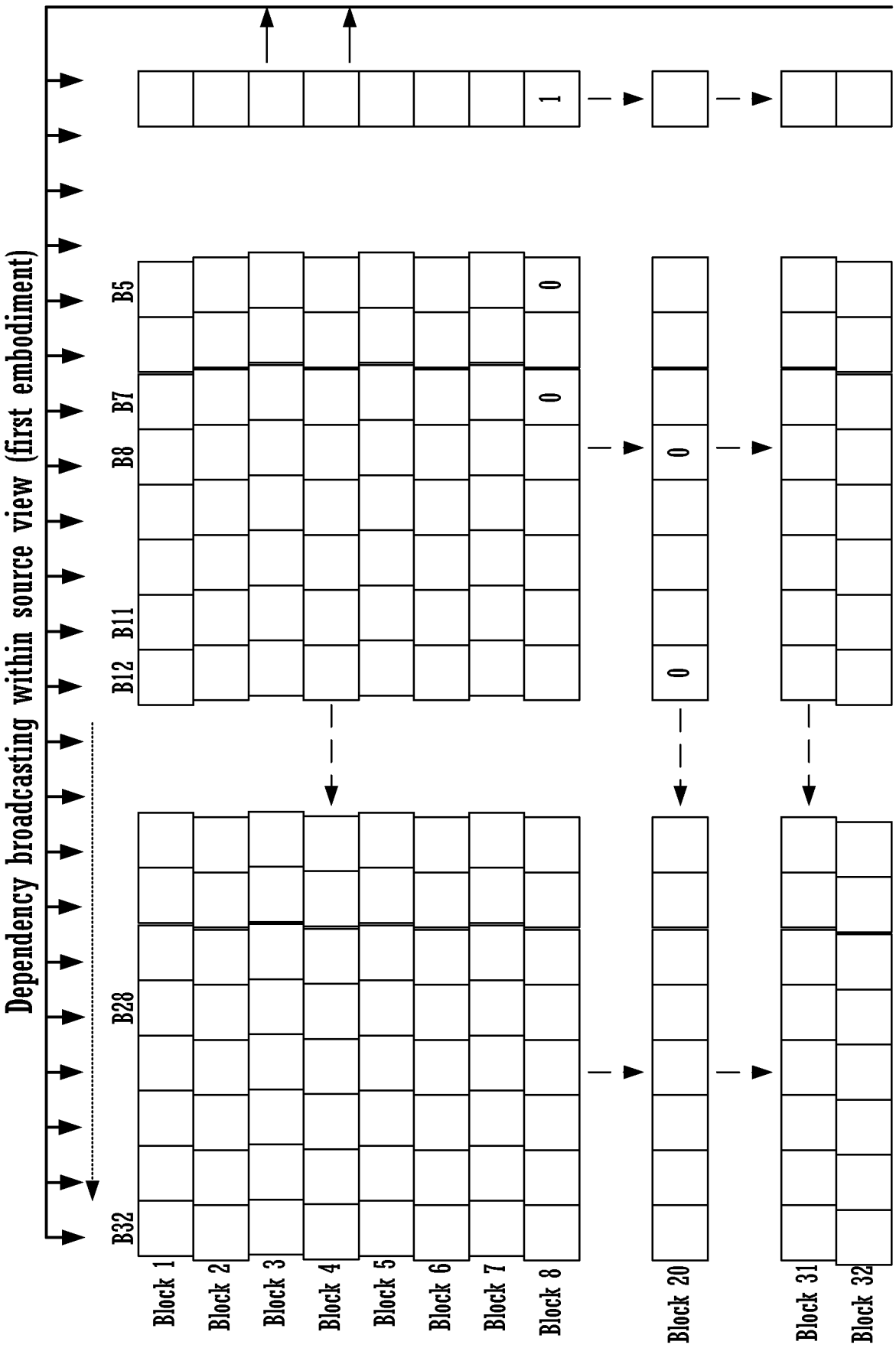

Dependency broadcasting within source view (first embodiment)
B5
B7
B8
B11
B12
B28
B32
1
0
0
0
0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
Block 8
Block 20
Block 31
Block 32

Fig. 4

Dispatch Broadcasting within source view (Second Embodiment)

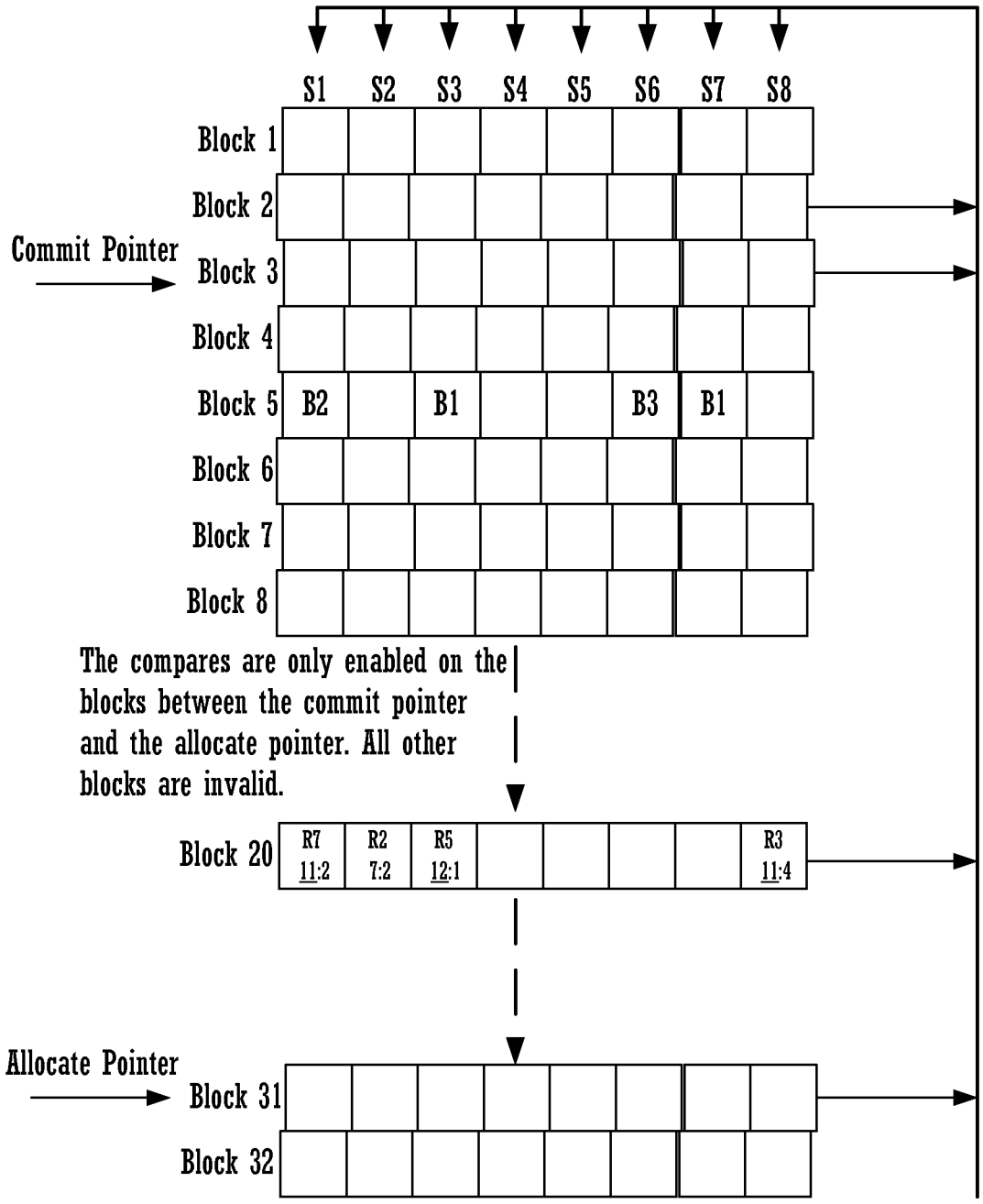


Fig. 5

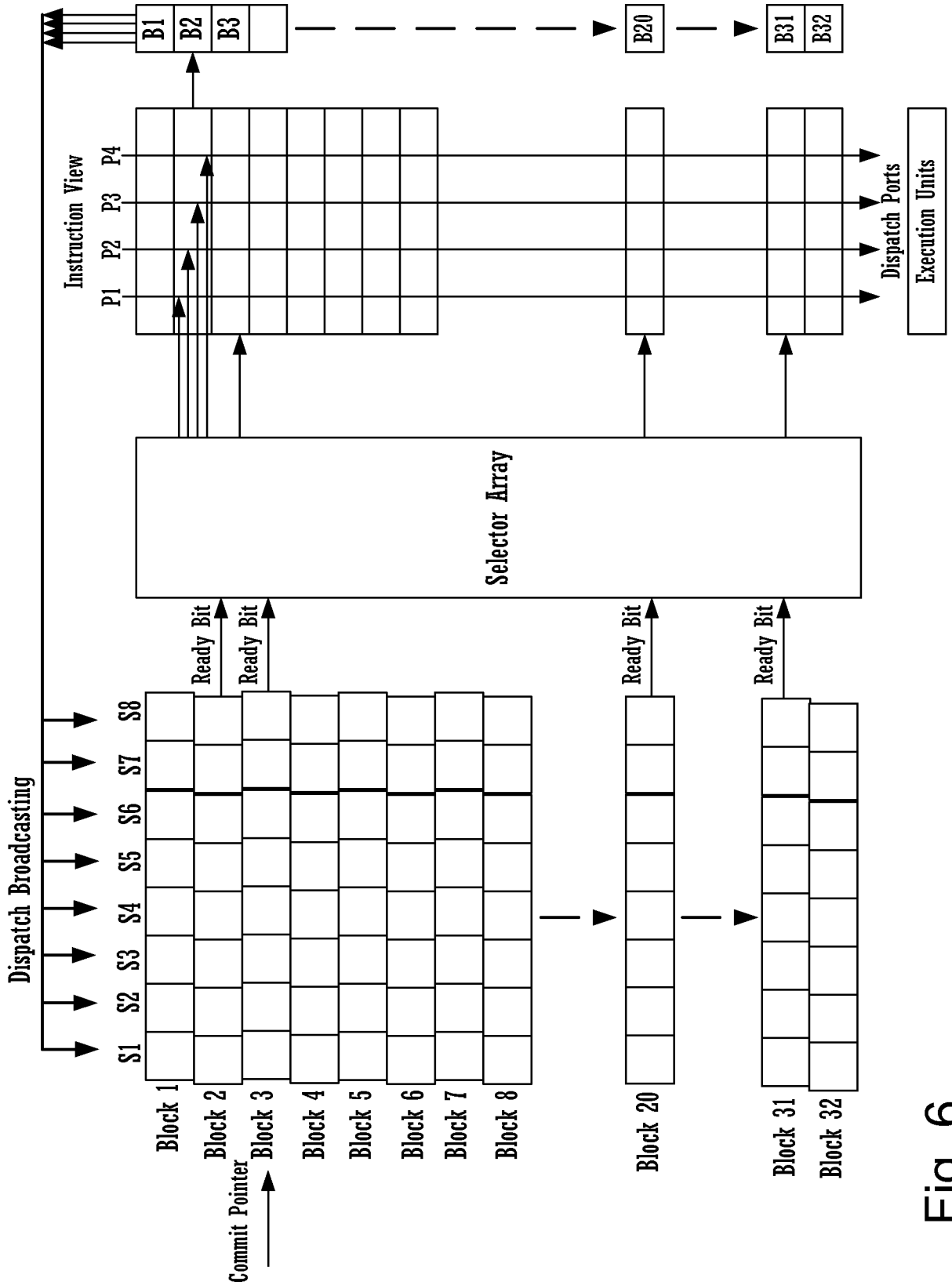

Dispatch Broadcasting
S1
S2
S3
S4
S5
S6
S7
S8
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
Block 8
Block 20
Block 31
Block 32
Commit Pointer
Ready Bit
Ready Bit
Ready Bit
Ready Bit
Selector Array
Instruction View
P1
P2
P3
P4
B1
B2
B3
B20
B31
B32
Dispatch Ports
Execution Units

Fig. 6

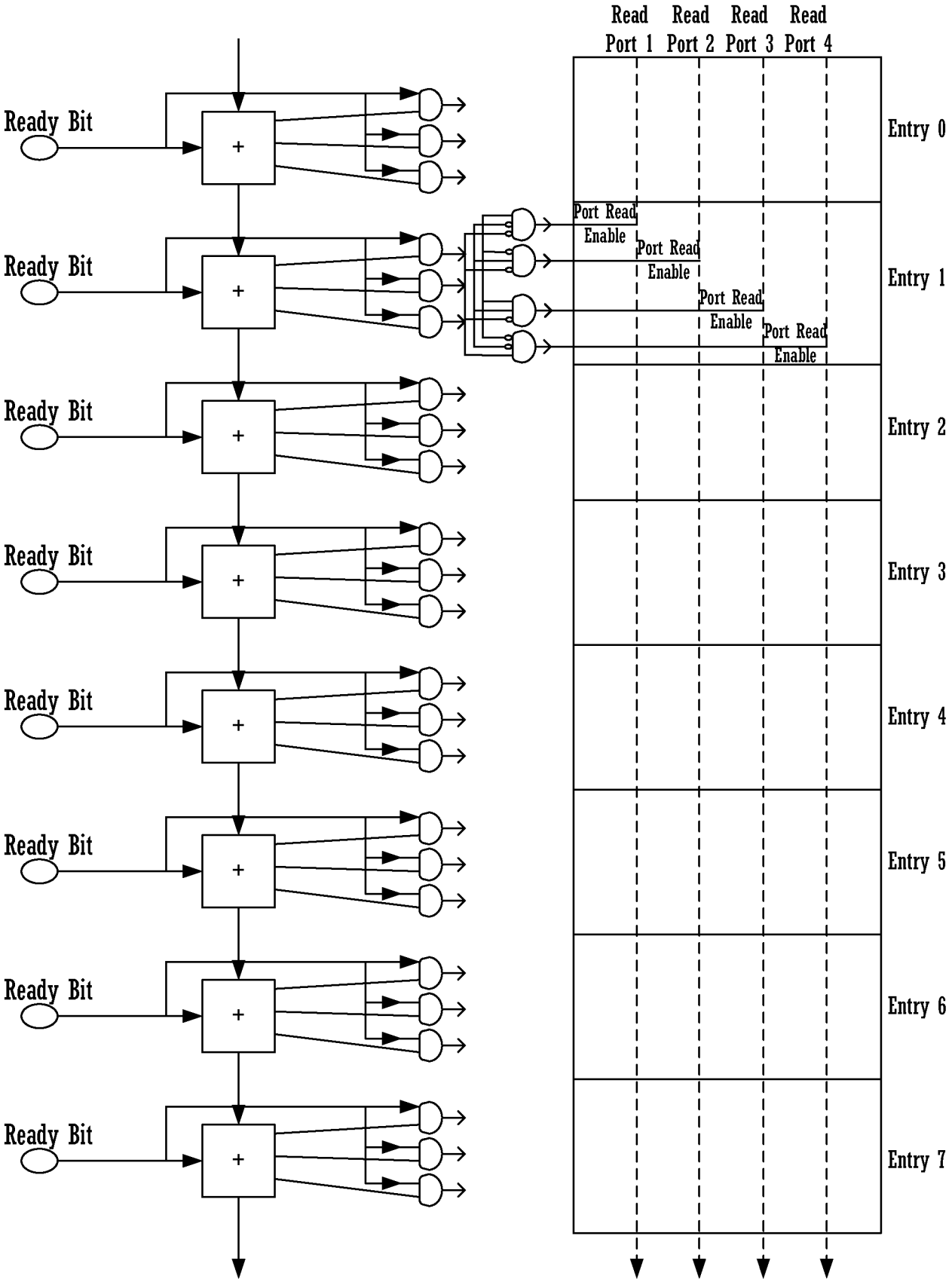

Read Port 1
Read Port 2
Read Port 3
Read Port 4
Ready Bit
Ready Bit
Ready Bit
Ready Bit
Ready Bit
Ready Bit
Ready Bit
Ready Bit
Port Read Enable
Port Read Enable
Port Read Enable
Port Read Enable
Entry 0
Entry 1
Entry 2
Entry 3
Entry 4
Entry 5
Entry 6
Entry 7

Fig. 7

Direct Selection of each particular port with Range Exceed Bit

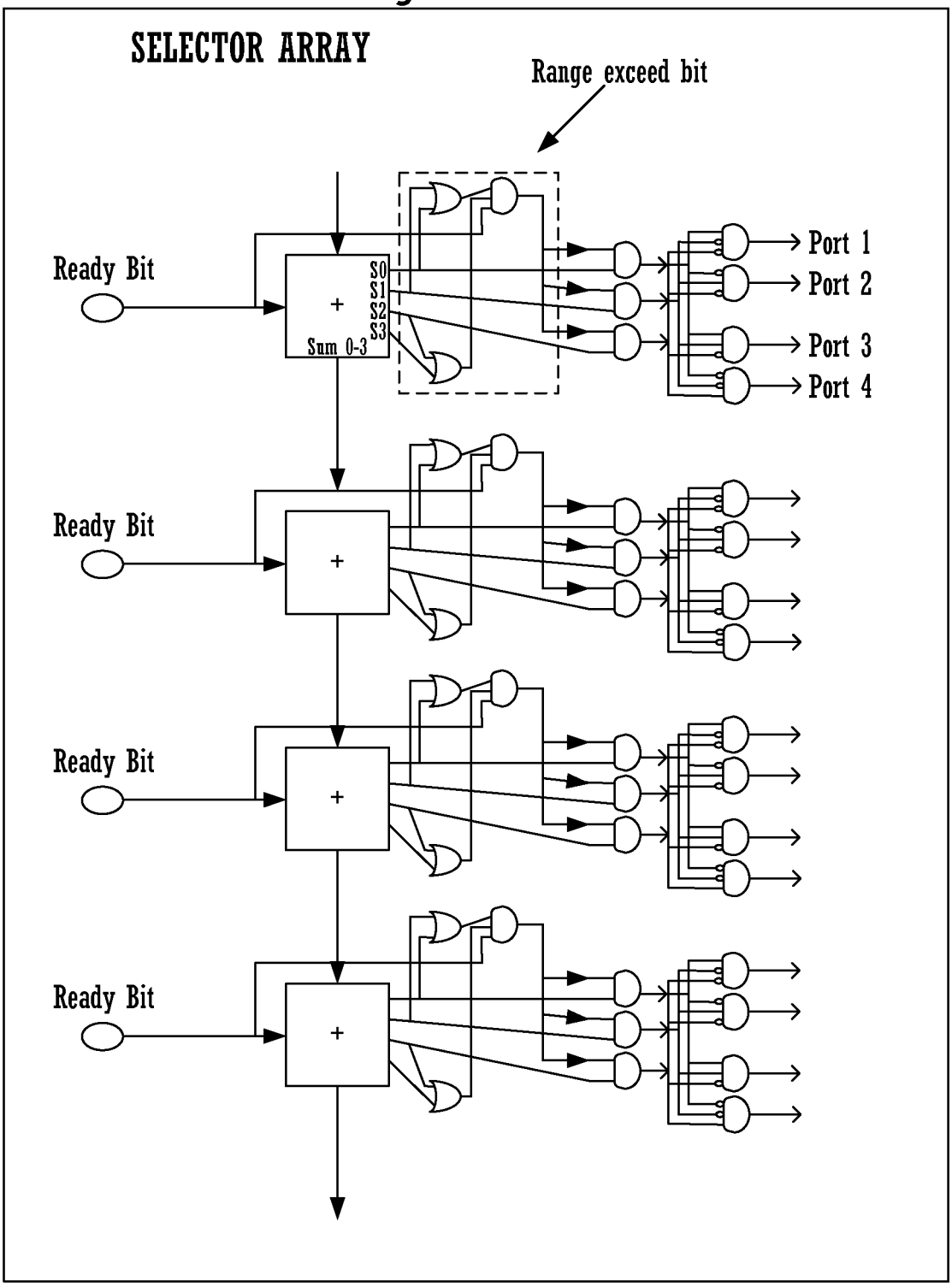


Fig. 8

Parallel implementation of the adder tree

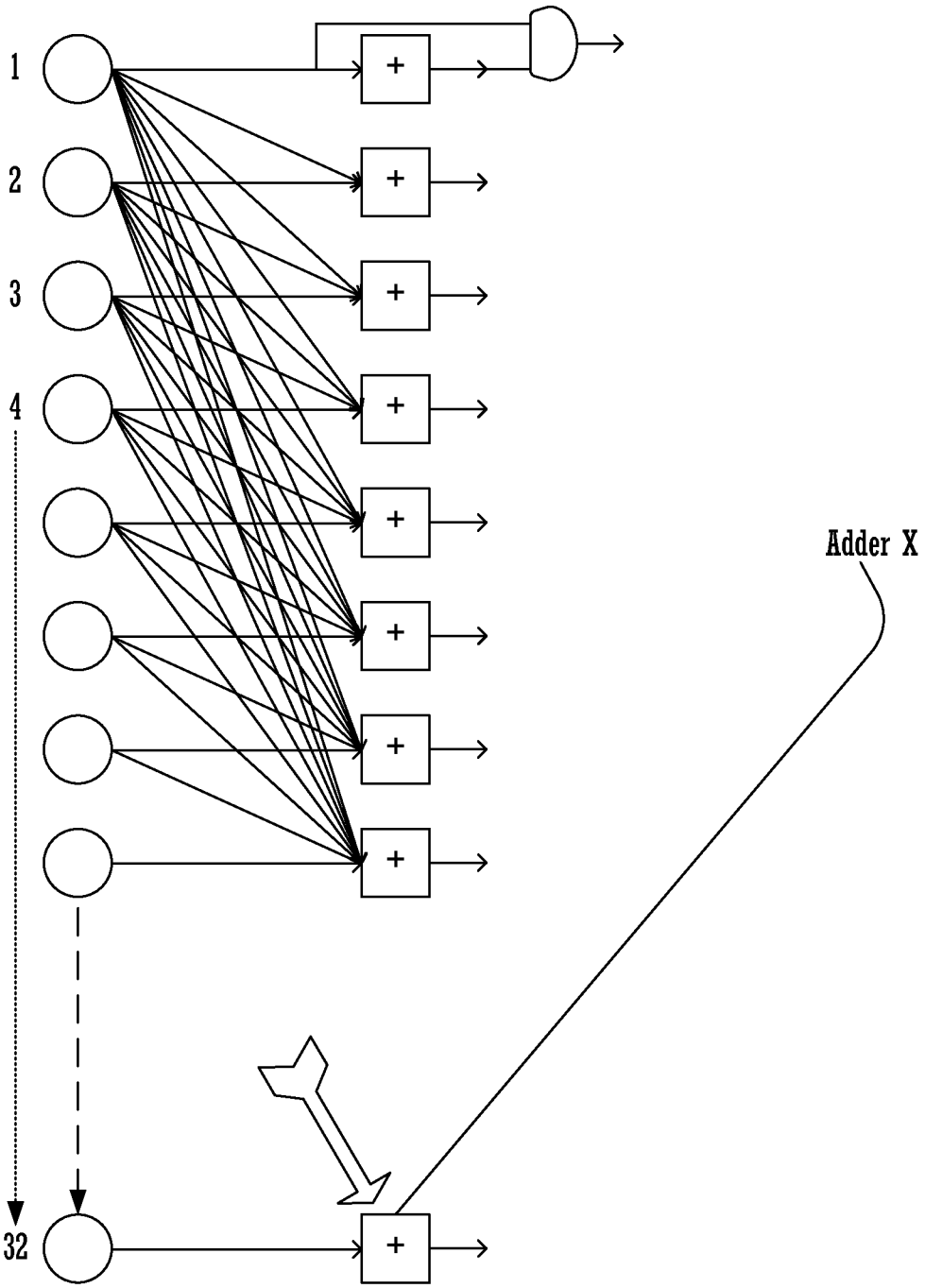


Fig. 9

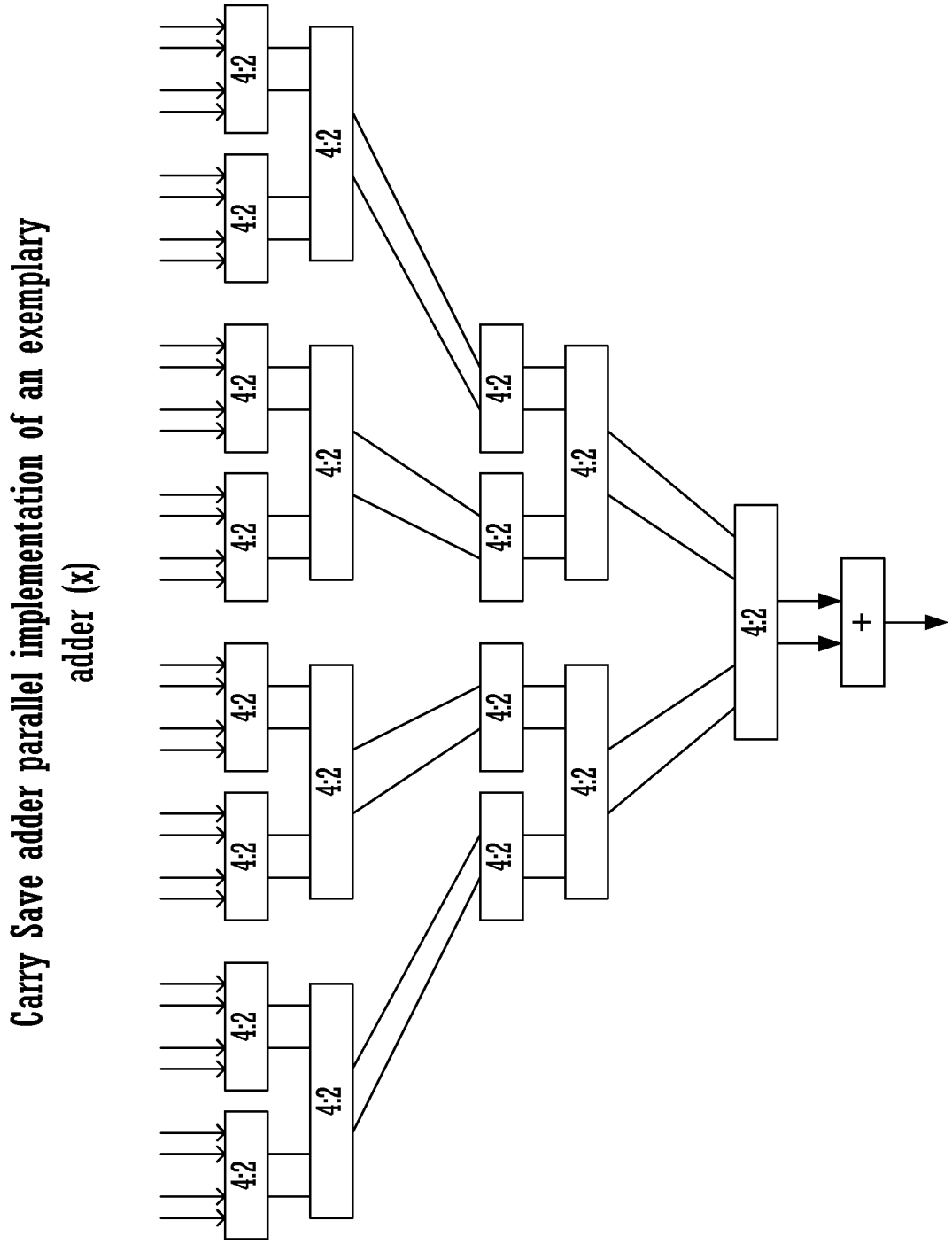

Carry Save adder parallel implementation of an exemplary adder (x)
4:2
+

Fig. 10

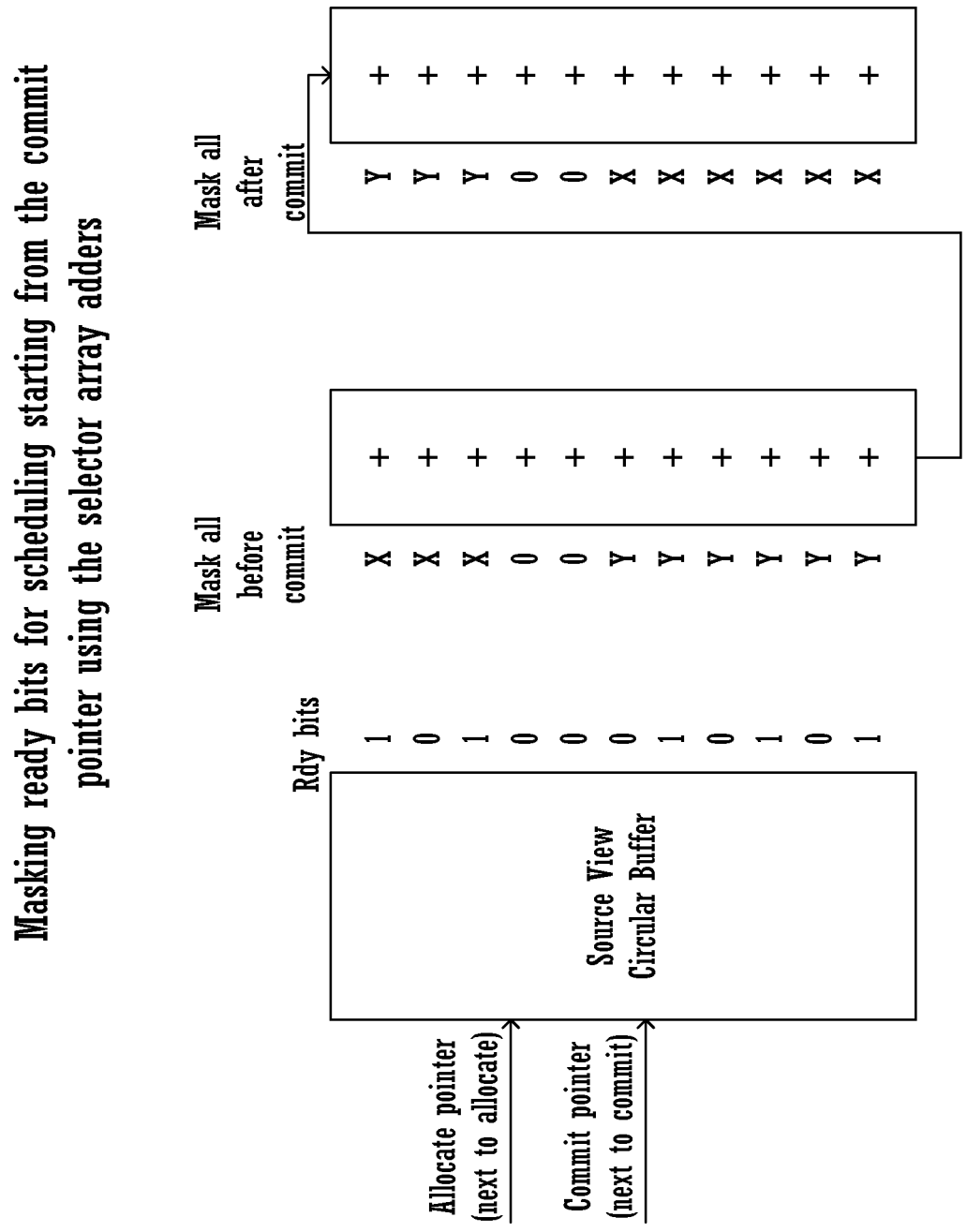

Masking ready bits for scheduling starting from the commit pointer using the selector array adders
Rdy bits
1 0 1 0 0 0 1 0 1 0 1
Source View Circular Buffer
Allocate pointer (next to allocate)
Commit pointer (next to commit)
Mask all before commit
X X X 0 0 Y Y Y Y Y Y
Mask all after commit
Y Y Y 0 0 X X X X X X

Fig. 11

Register view entries populated by register templates

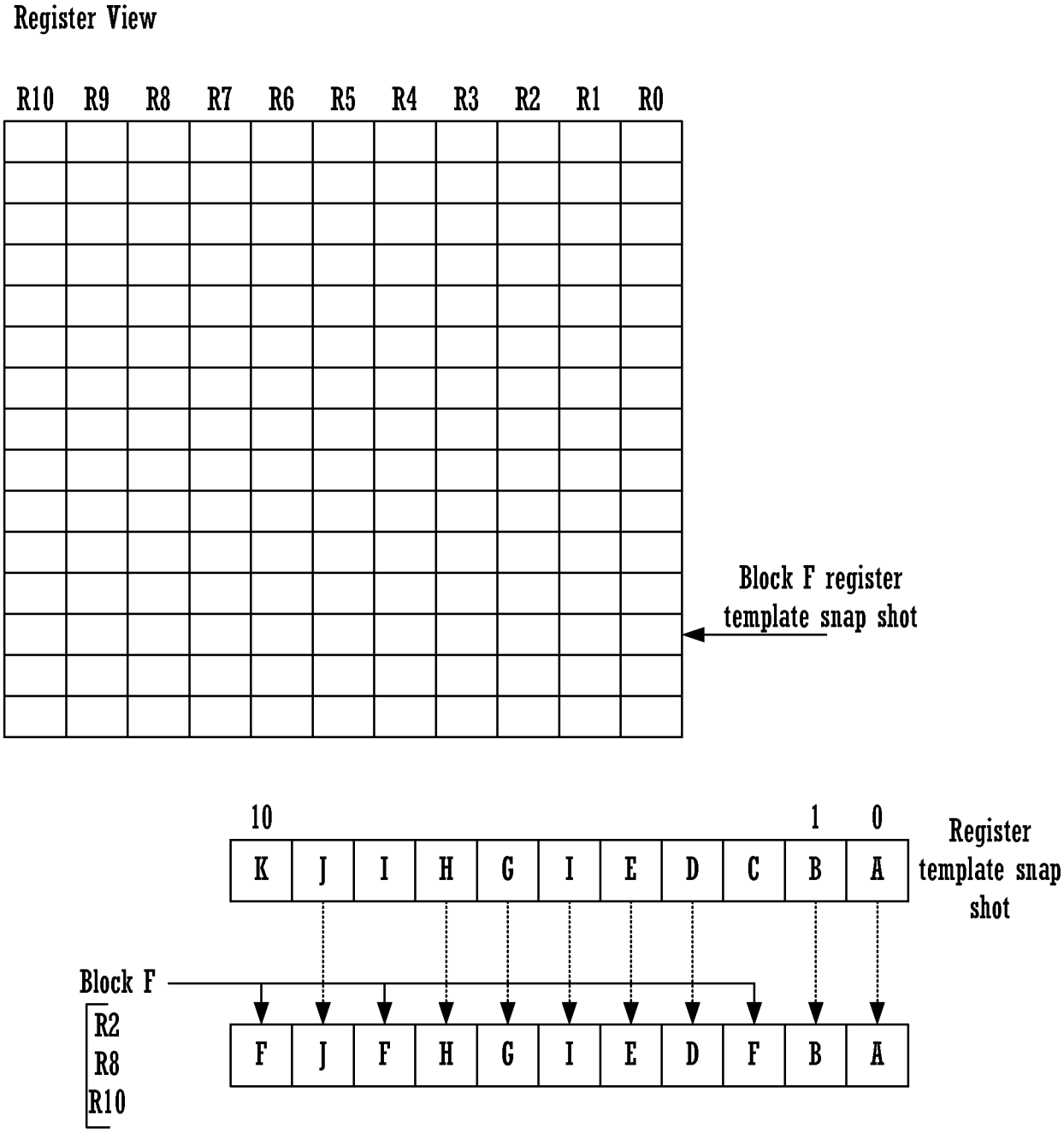


Fig. 12

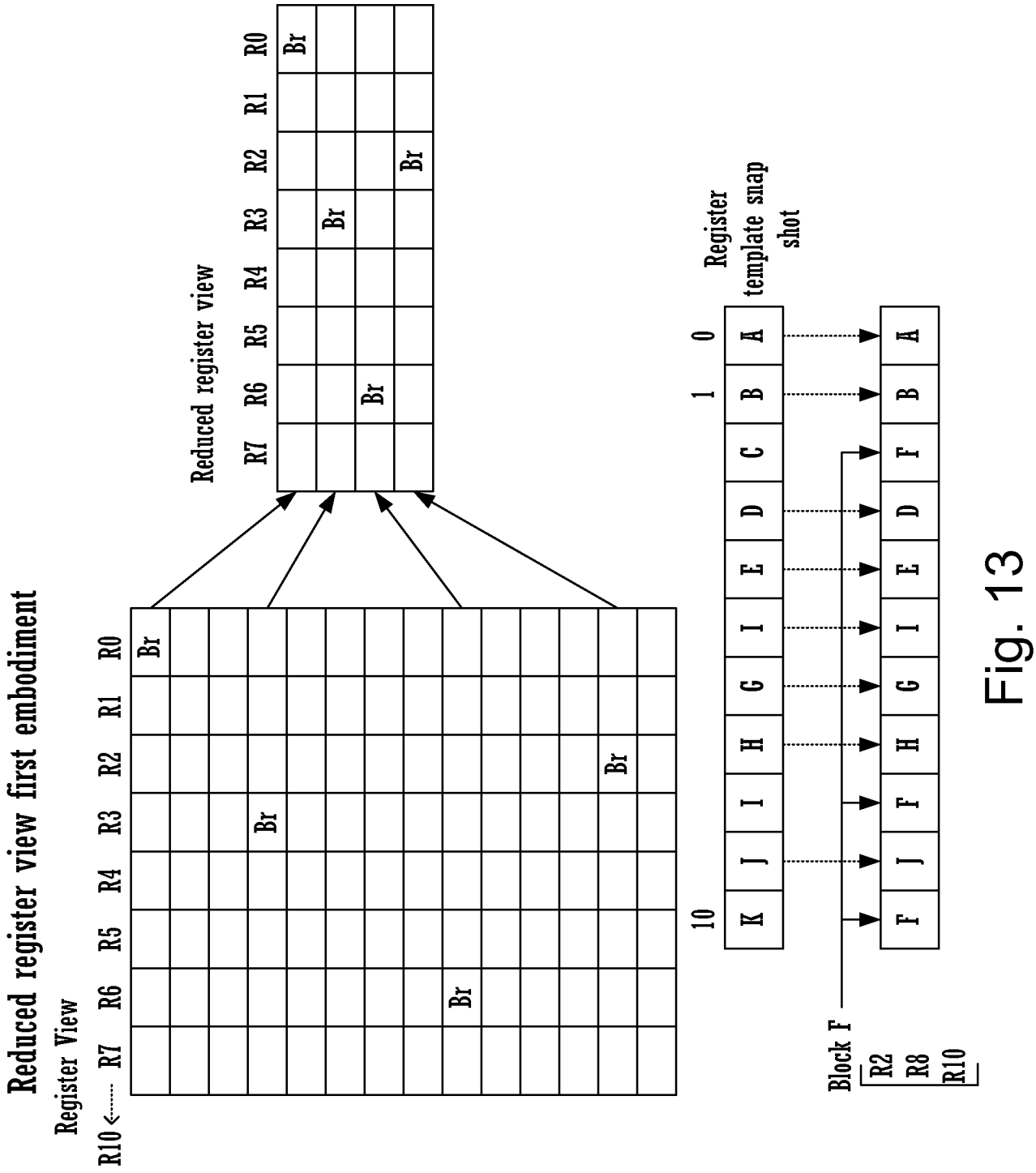

Reduced register view first embodiment
Register View
R10 ← R7 R6 R5 R4 R3 R2 R1 R0
Br
Reduced register view
R7 R6 R5 R4 R3 R2 R1 R0
Br
Register template snap shot
10 1 0
K J I H G I E D C B A
F J F H G I E D F B A
Block F
R2
R8
R10

Fig. 13

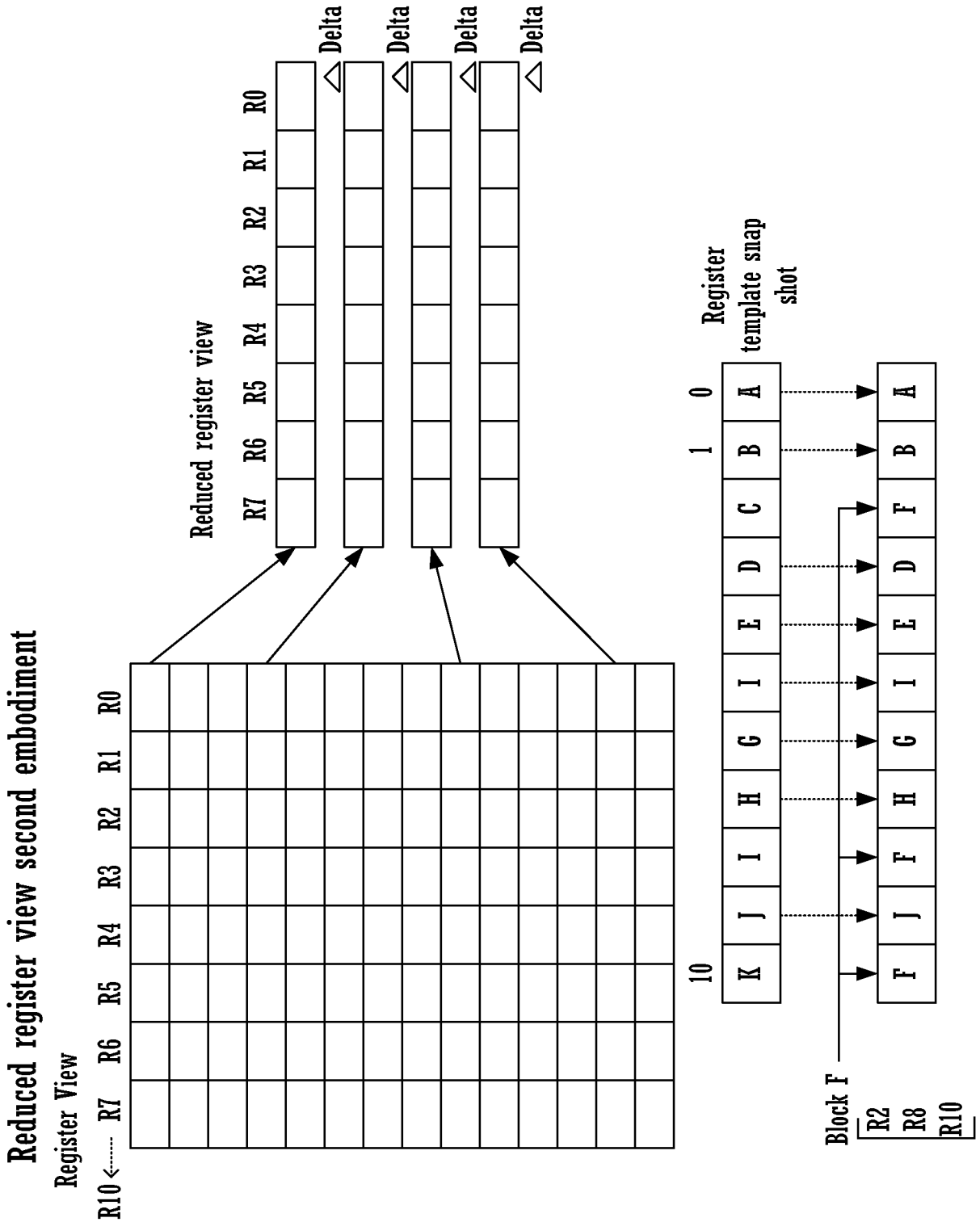

Reduced register view second embodiment
Register View
R10 ←---- R7 R6 R5 R4 R3 R2 R1 R0
Reduced register view
R7 R6 R5 R4 R3 R2 R1 R0
Delta
Delta
Delta
Delta
Register template snap shot
1 0
A B C D E F G H I J
10
K J I H G I F J F
A B F D F I G H F
Block F
R2
R8
R10

Fig. 14

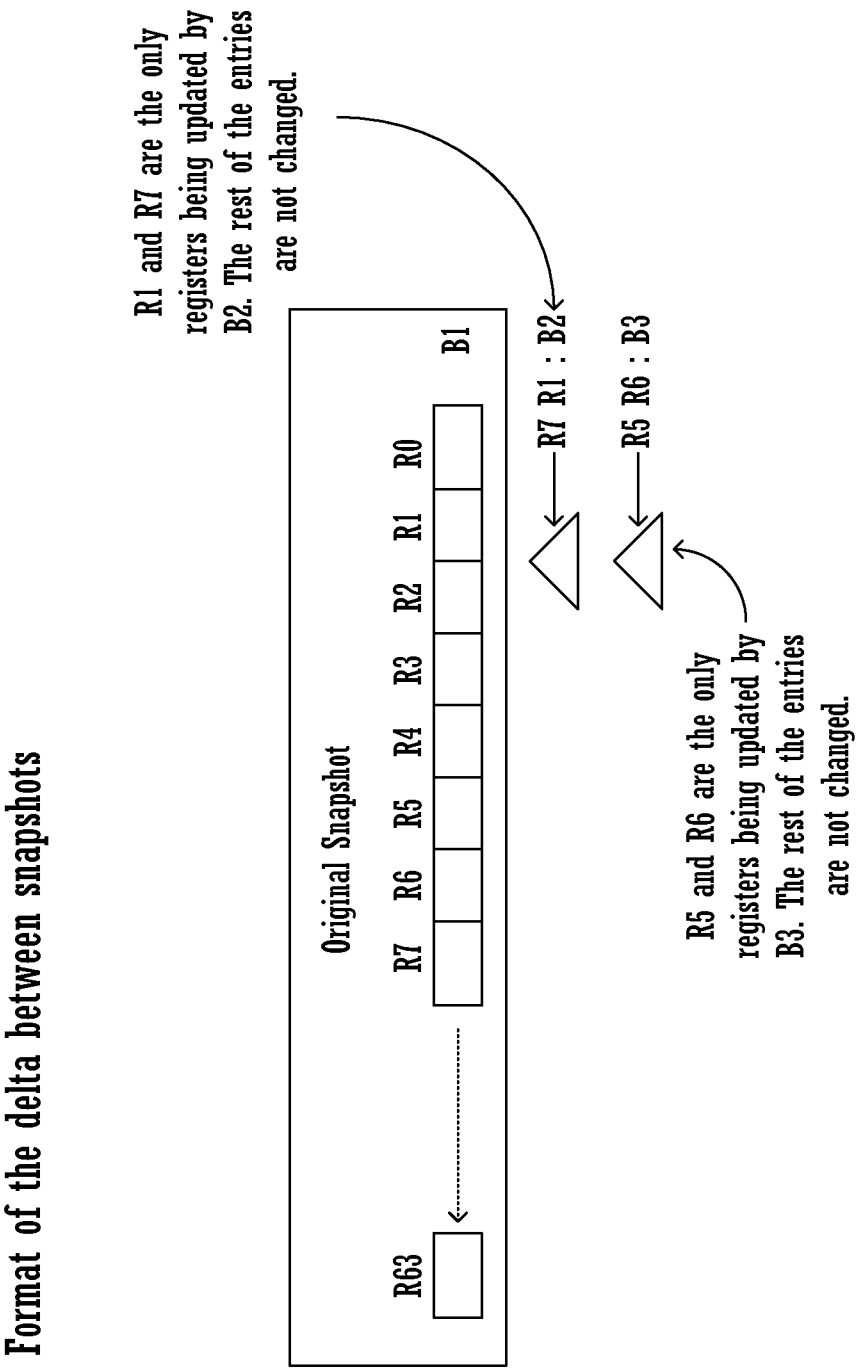

Format of the delta between snapshots
Original Snapshot
R7 R6 R5 R4 R3 R2 R1 R0
B1
R63
R1 and R7 are the only registers being updated by B2. The rest of the entries are not changed.
R7 R1 : B2
R5 R6 : B3
R5 and R6 are the only registers being updated by B3. The rest of the entries are not changed.

Fig. 15

Creating register template snapshots upon allocation of blocks of instructions

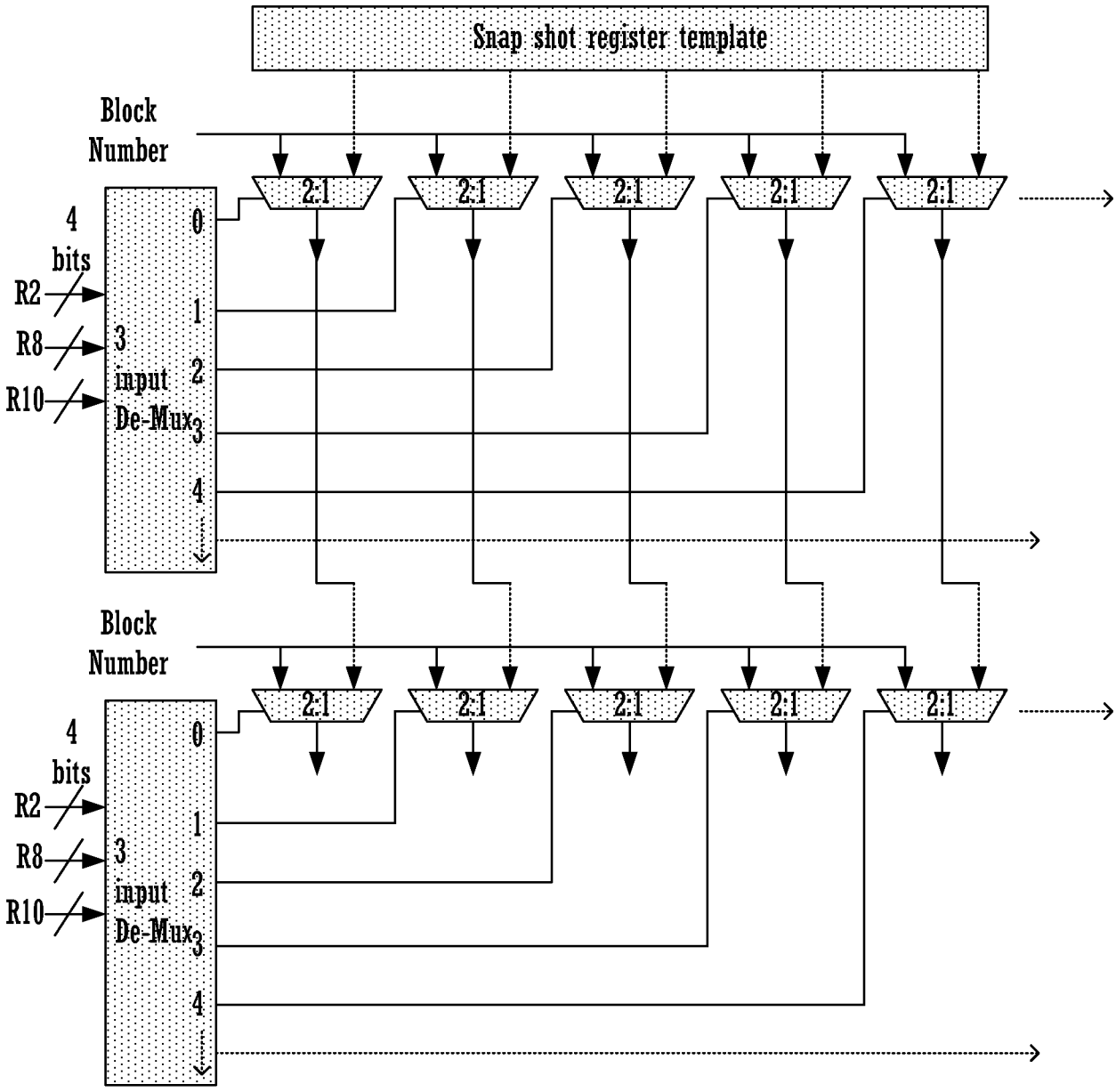


Fig. 16

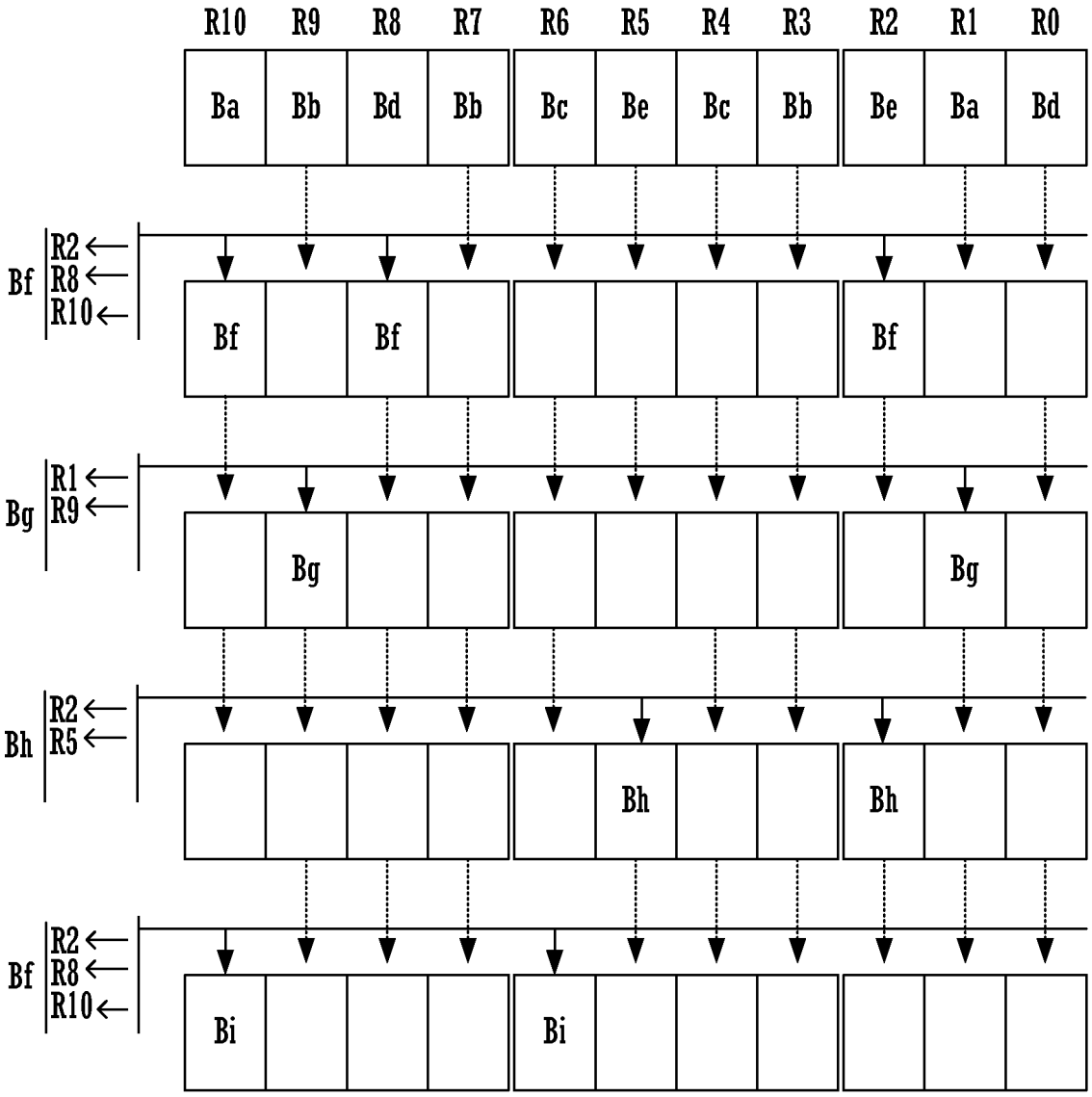

R10 R9 R8 R7 R6 R5 R4 R3 R2 R1 R0
Ba Bb Bd Bb Bc Be Bc Bb Be Ba Bd
Bf
R2
R8
R10
Bf Bf Bf
Bg
R1
R9
Bg Bg
Bh
R2
R5
Bh Bh
Bf
R2
R8
R10
Bi Bi

Fig. 17

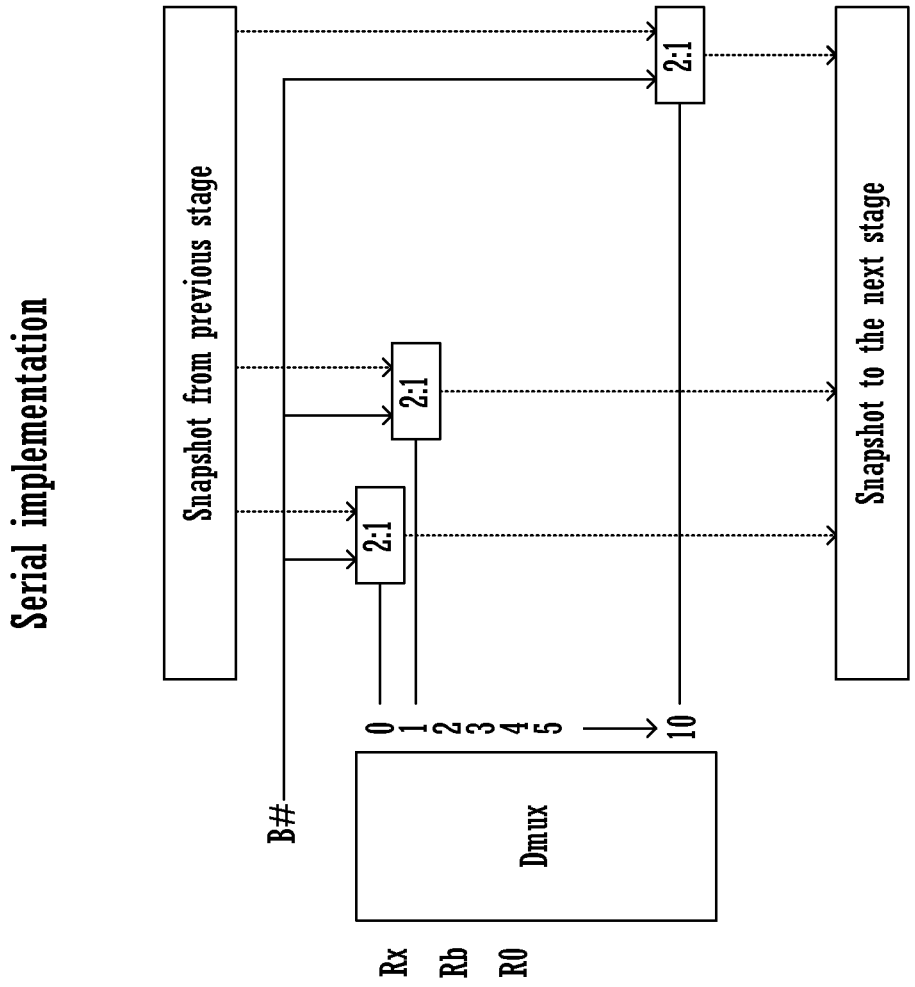

Serial implementation
Snapshot from previous stage
2:1
2:1
2:1
Snapshot to the next stage
0
1
2
3
4
5
10
B#
Dmux
Rx
Rb
R0

Fig. 18

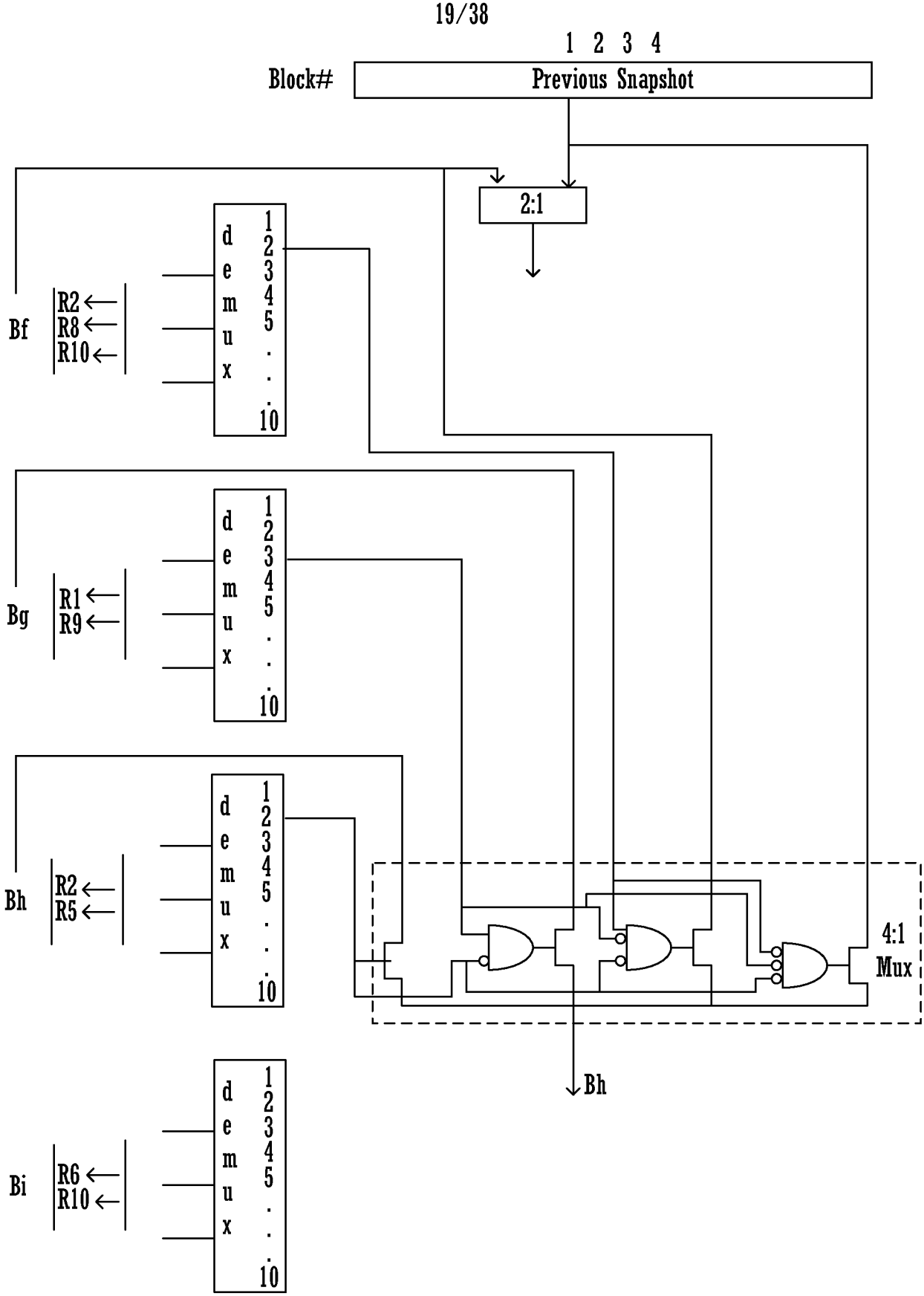

1 2 3 4
Block#
Previous Snapshot
2:1
Bf
R2
R8
R10
Bg
R1
R9
Bh
R2
R5
Bi
R6
R10
demux
1 2 3 4 5 . . . 10
4:1
Mux
Bh

Fig. 19

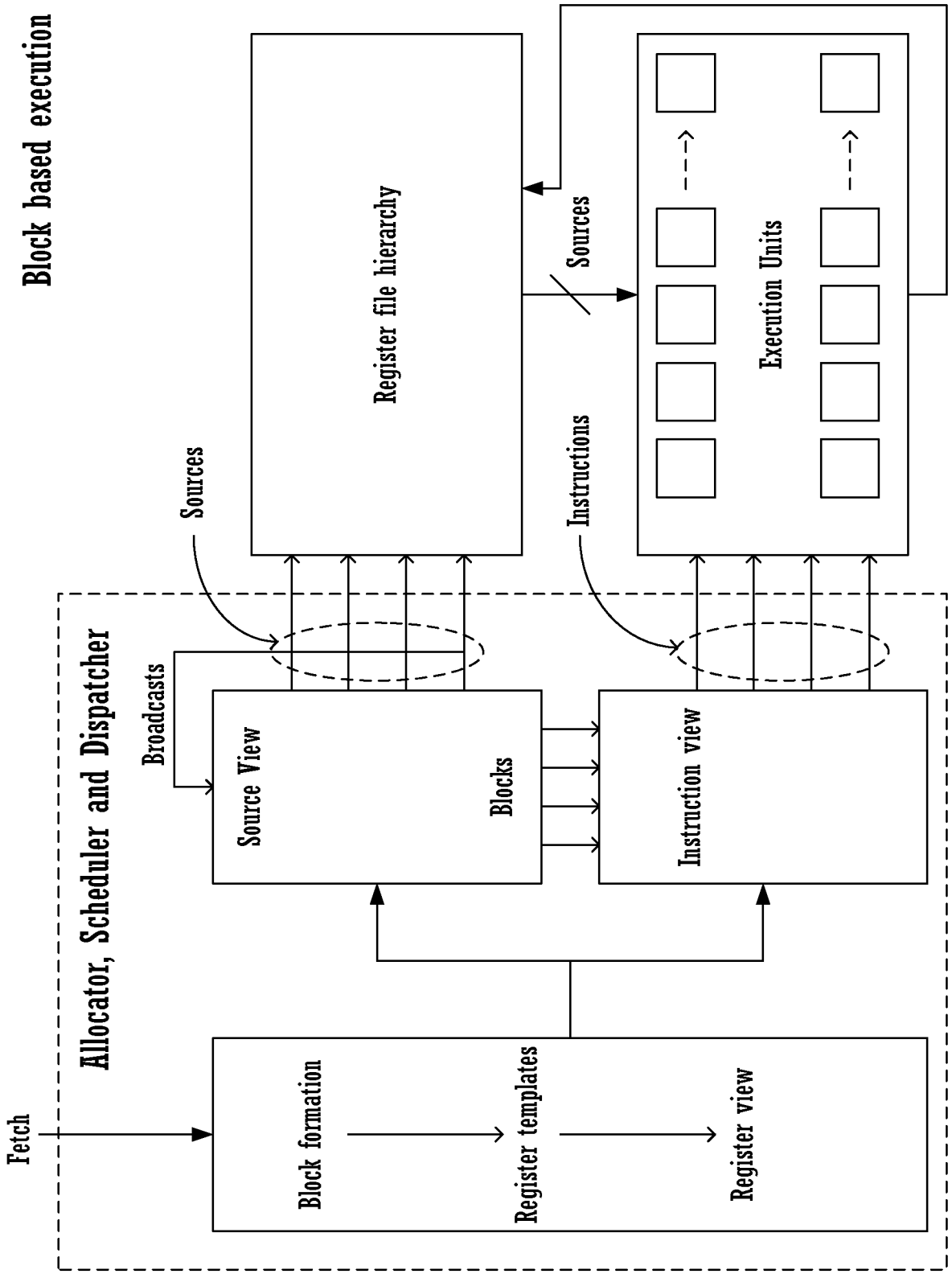

Block based execution
Register file hierarchy
Sources
Execution Units
Instructions
Sources
Broadcasts
Source View
Blocks
Instruction view
Allocator, Scheduler and Dispatcher
Fetch
Block formation
Register templates
Register view

Fig. 20

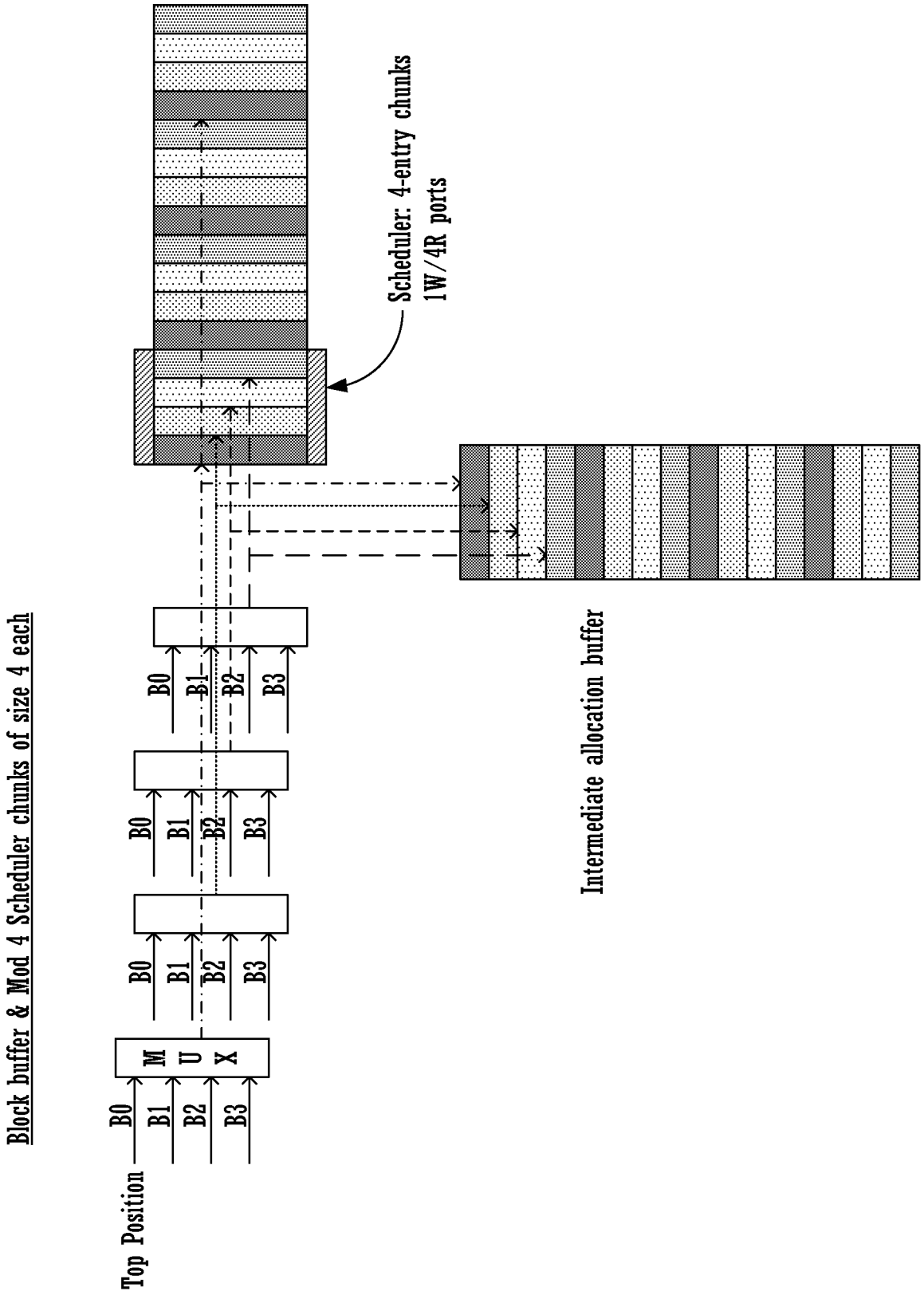

Block buffer & Mod 4 Scheduler chunks of size 4 each
Top Position
B0
B1
B2
B3
MUX
Scheduler: 4-entry chunks
1W/4R ports
Intermediate allocation buffer

Fig. 21

Depiction of how threads are allocated in accordance with their block numbers and thread ID

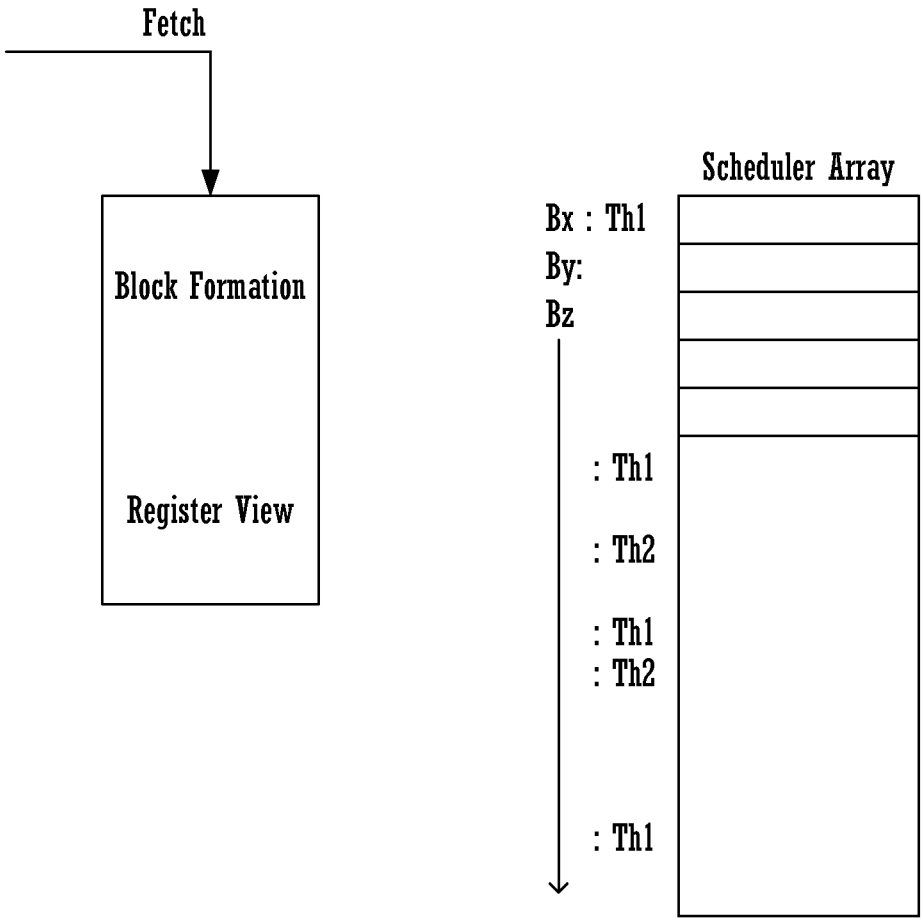


Fig. 22

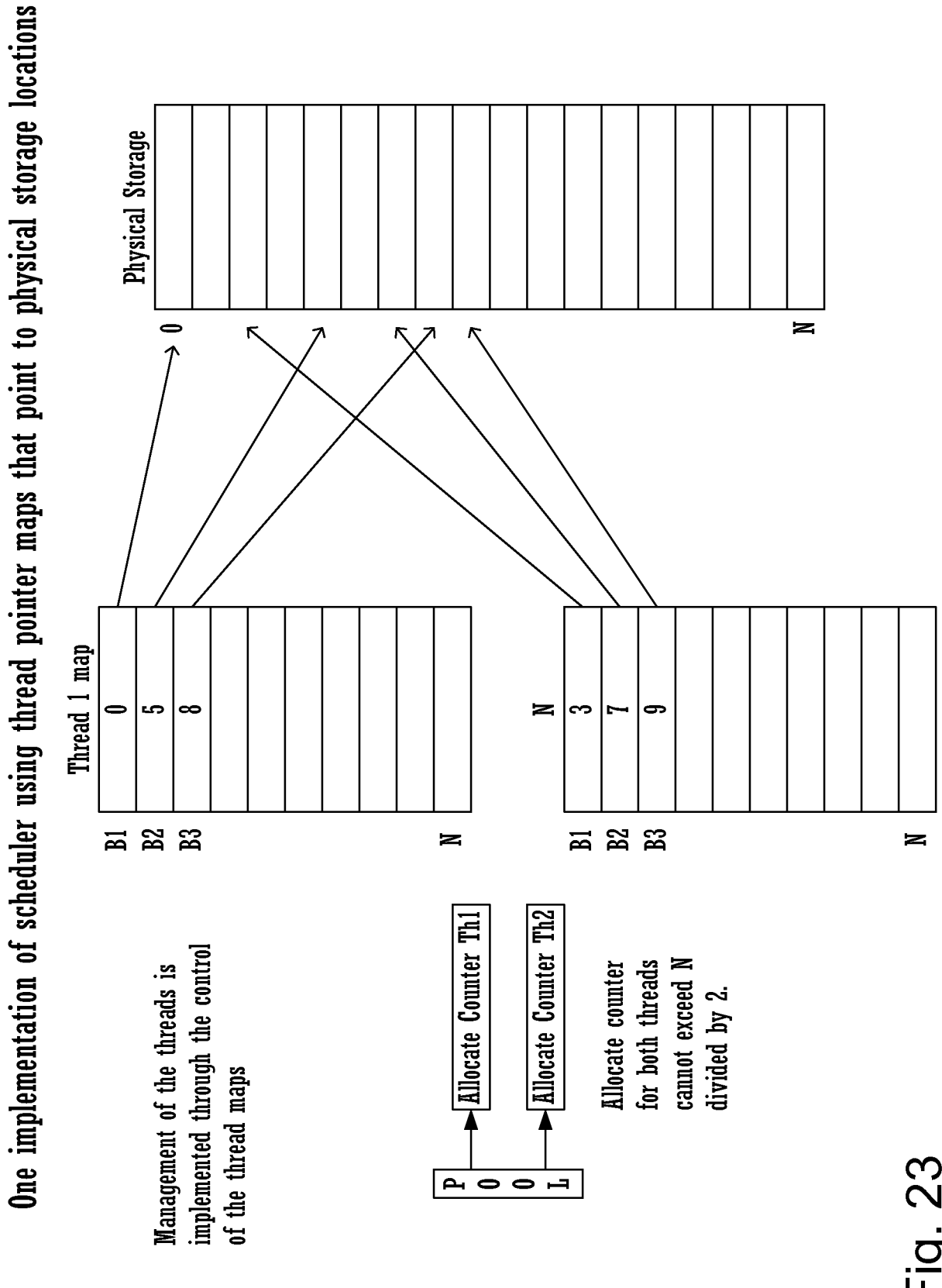

One implementation of scheduler using thread pointer maps that point to physical storage locations
Physical Storage
0
N
Thread 1 map
0
5
8
N
3
7
9
B1
B2
B3
N
B1
B2
B3
N
Management of the threads is implemented through the control of the thread maps
P
O
O
L
Allocate Counter Th1
Allocate Counter Th2
Allocate counter for both threads cannot exceed N divided by 2.

Fig. 23

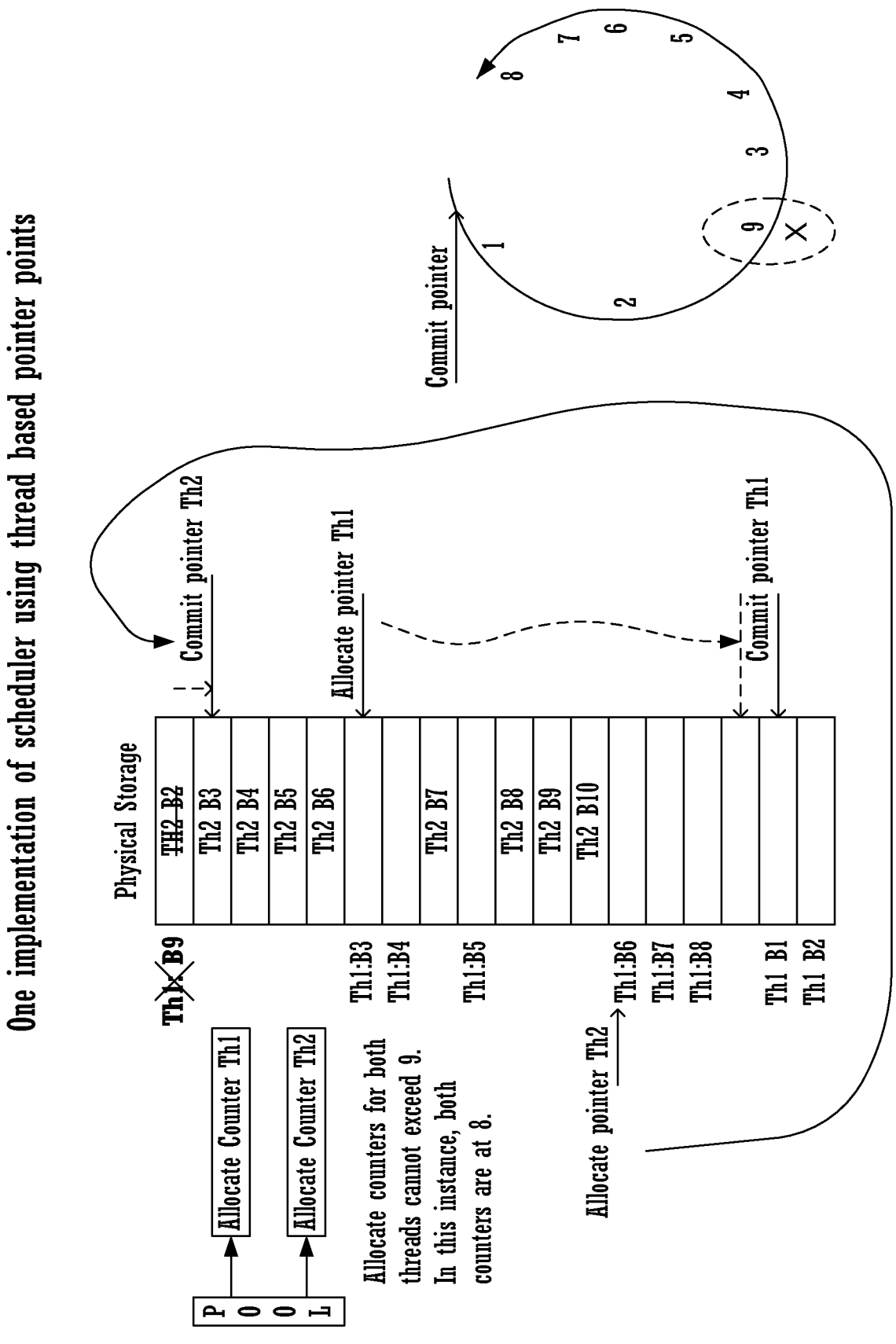

One implementation of scheduler using thread based pointer points
Commit pointer
1
2
3
4
5
6
7
8
9
X
Physical Storage
TH2 B2
Th2 B3
Th2 B4
Th2 B5
Th2 B6
Th2 B7
Th2 B8
Th2 B9
Th2 B10
Th1 B1
Th1 B2
Commit pointer Th2
Allocate pointer Th1
Commit pointer Th1
Th1 B9
Th1:B3
Th1:B4
Th1:B5
Th1:B6
Th1:B7
Th1:B8
Allocate pointer Th2
Allocate counters for both threads cannot exceed 9. In this instance, both counters are at 8.
P
O
O
L
Allocate Counter Th1
Allocate Counter Th2

Fig. 24

Dynamic counter based fair allocation of execution resources to threads

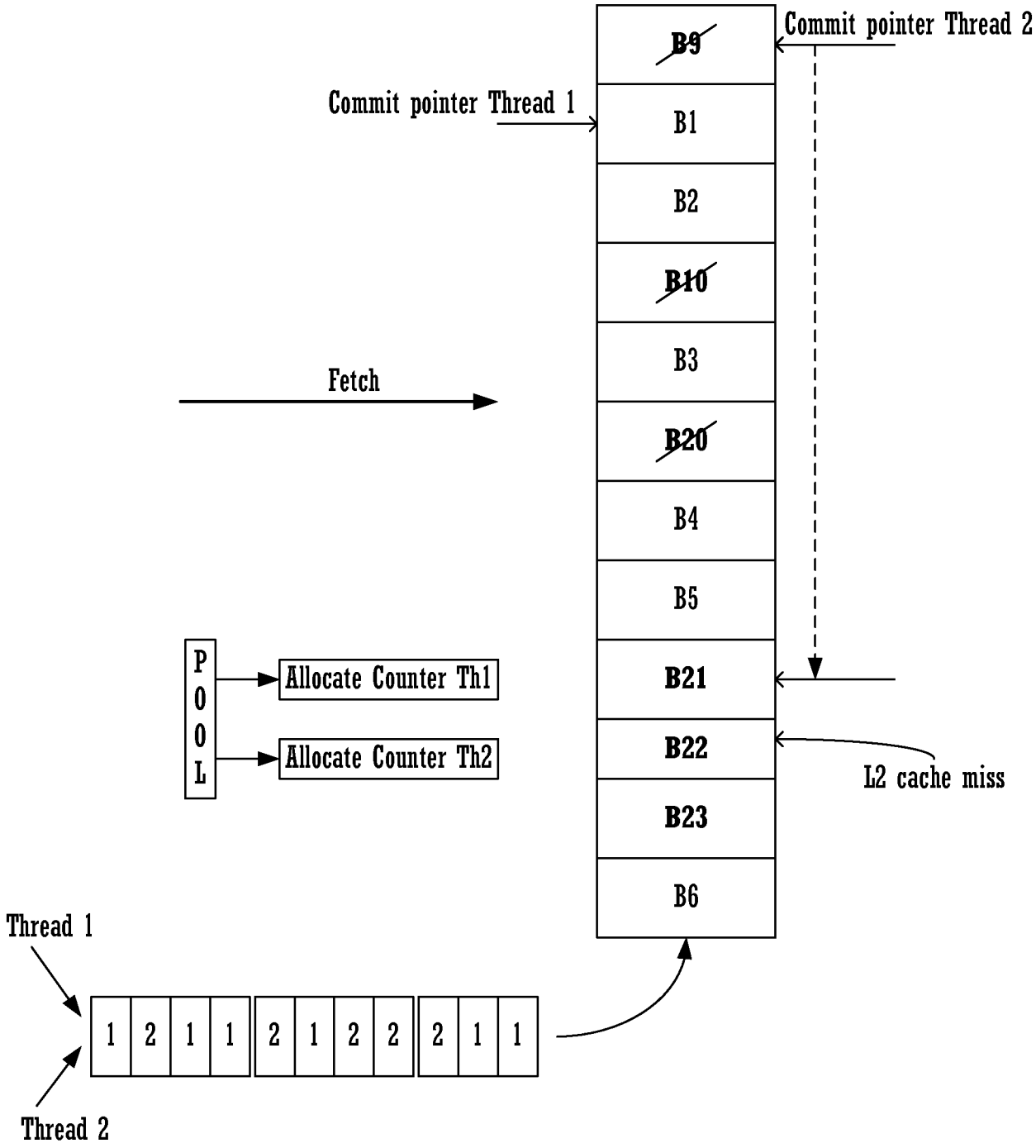


Fig. 25

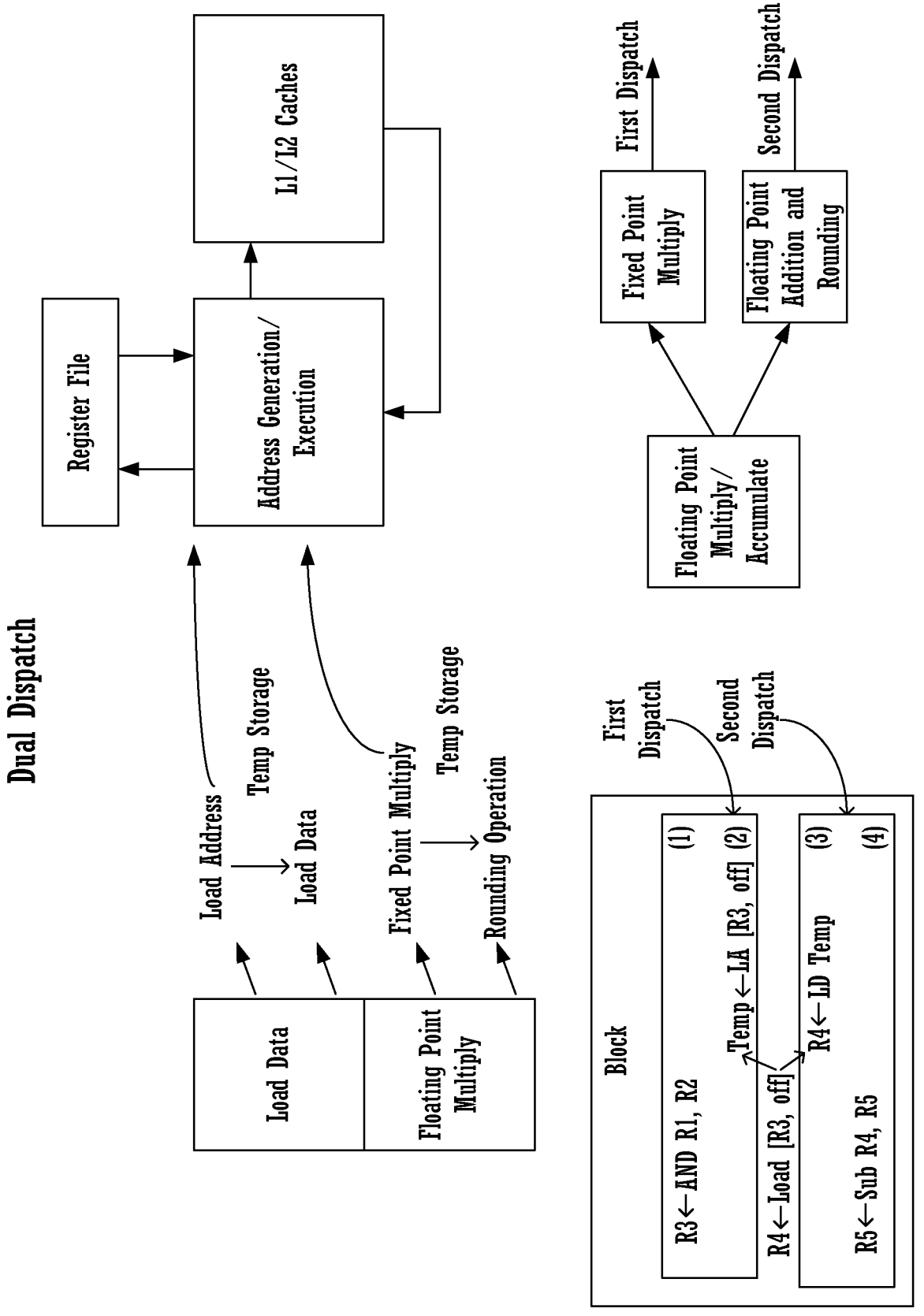

Dual Dispatch
Register File
Address Generation/ Execution
L1/L2 Caches
Load Address
Temp Storage
Load Data
Fixed Point Multiply
Temp Storage
Rounding Operation
Load Data
Floating Point Multiply
Floating Point Multiply/ Accumulate
Fixed Point Multiply
Floating Point Addition and Rounding
First Dispatch
Second Dispatch
First Dispatch
Second Dispatch
Block
R3←AND R1, R2
(1)
Temp←LA [R3, off] (2)
R4←Load [R3, off]
R4←LD Temp
(3)
R5←Sub R4, R5
(4)

Fig. 26

Dual dispatch transient Multiply-Accumulate

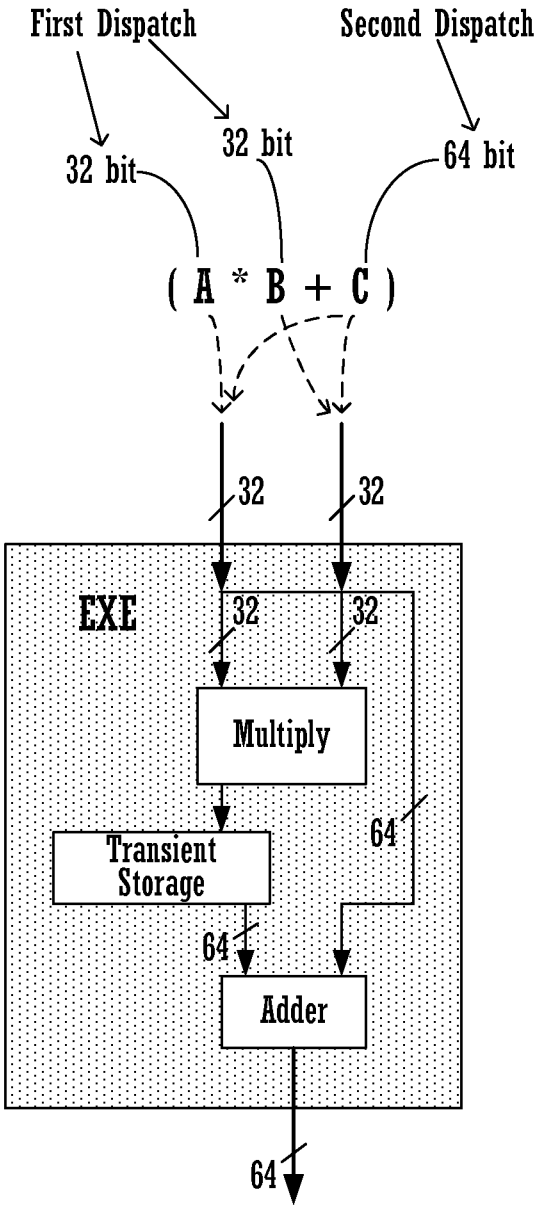


Fig. 27

Dual dispatch architecturally visible state
Multiply-Add

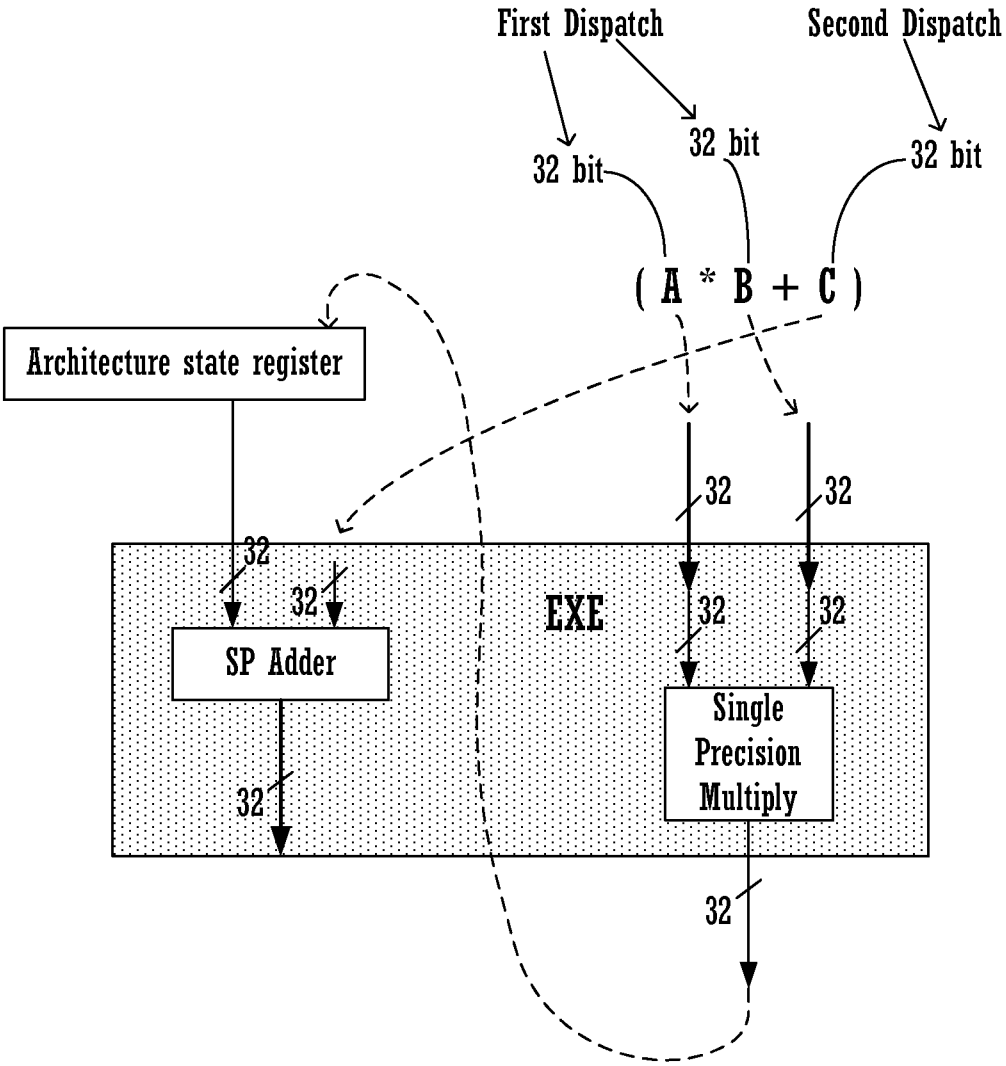


Fig. 28

The fetch and formation of instruction blocks for execution on grouped execution units

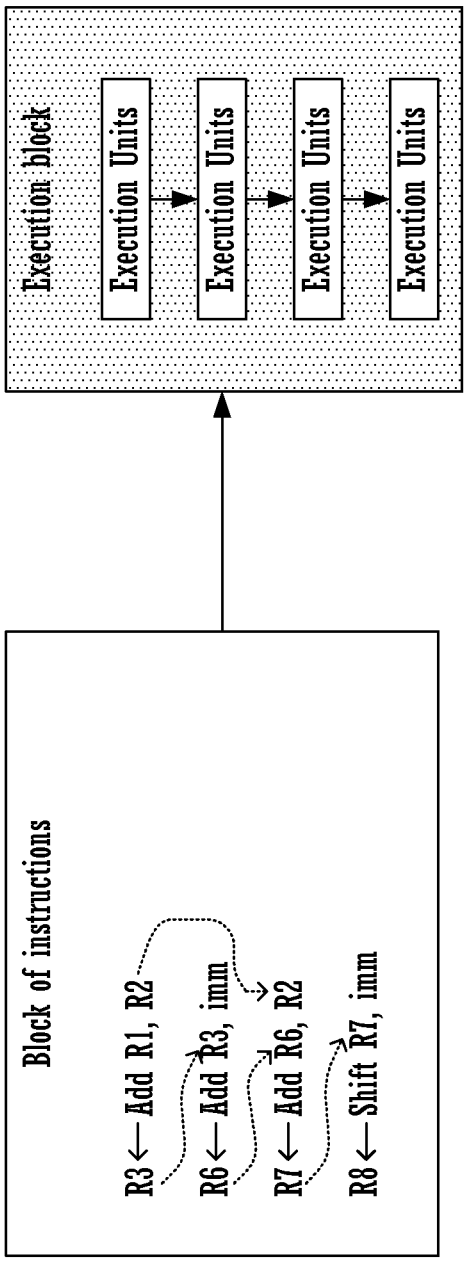


Fig. 29

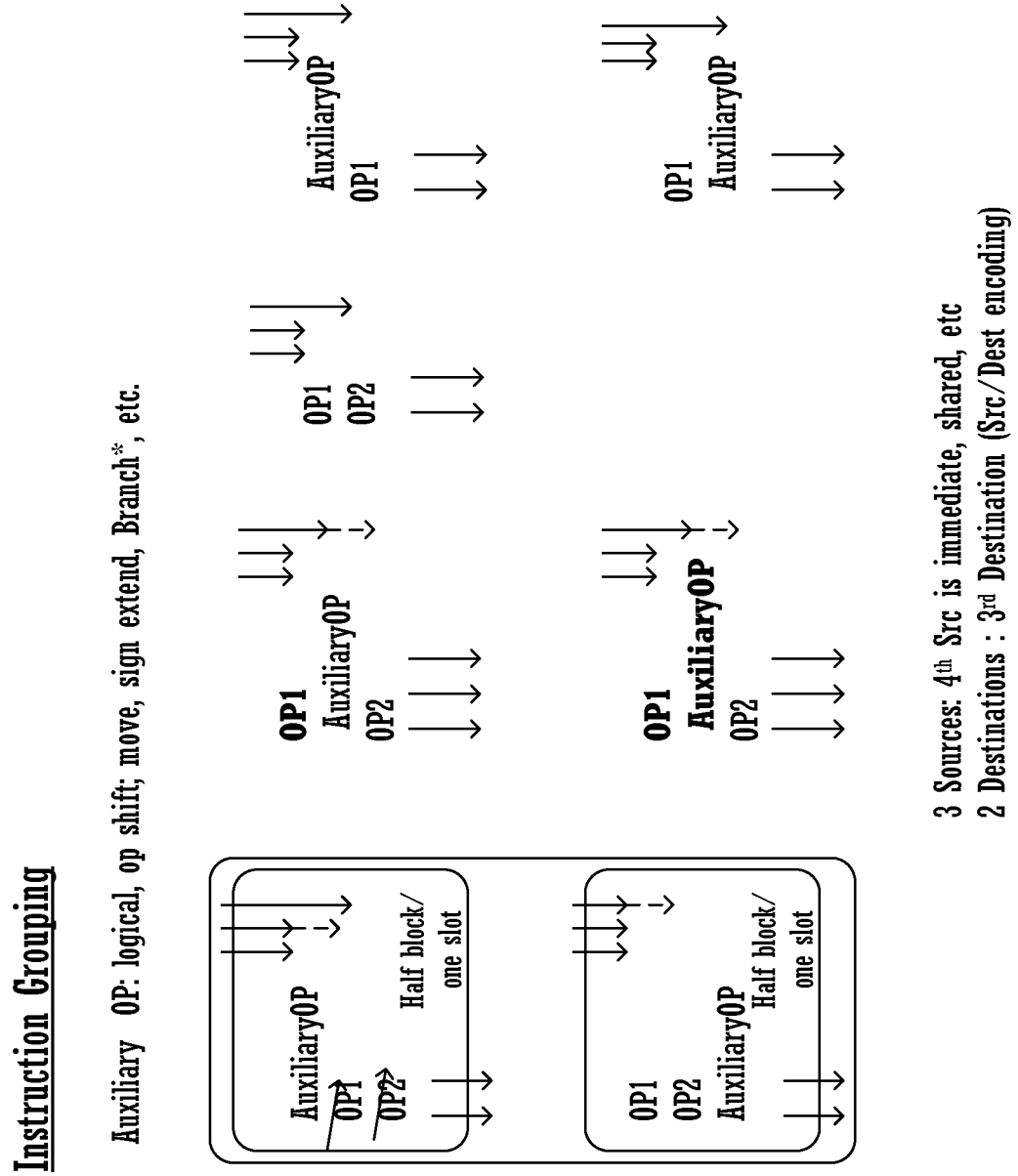

Instruction Grouping
Auxiliary OP: logical, op shift; move, sign extend, Branch*, etc.
AuxiliaryOP
OP1
OP2
Half block/ one slot
OP1
OP2
AuxiliaryOP
OP1
AuxiliaryOP
OP1
AuxiliaryOP
OP2
OP1
AuxiliaryOP
3 Sources: 4th Src is immediate, shared, etc
2 Destinations : 3rd Destination (Src/Dest encoding)

Fig. 30

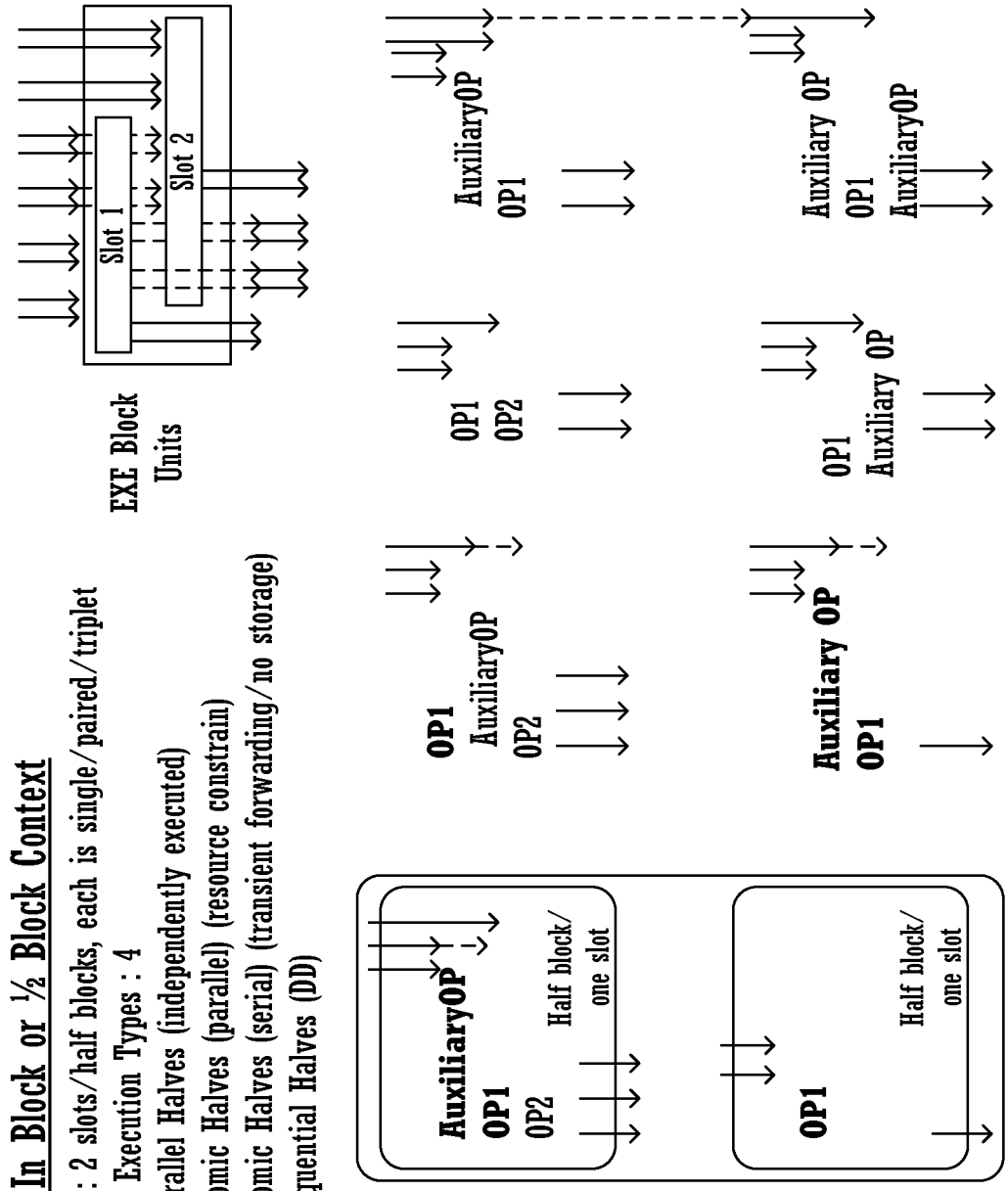

Pairs In Block or ½ Block Context
Slots : 2 slots/half blocks, each is single/paired/triplet
Block Execution Types : 4
1: Parallel Halves (independently executed)
2: Atomic Halves (parallel) (resource constrain)
3: Atomic Halves (serial) (transient forwarding/no storage)
4: Sequential Halves (DD)
EXE Block Units
Slot 1
Slot 2
OP1
Auxiliary OP
OP2
Half block/ one slot

Fig. 31

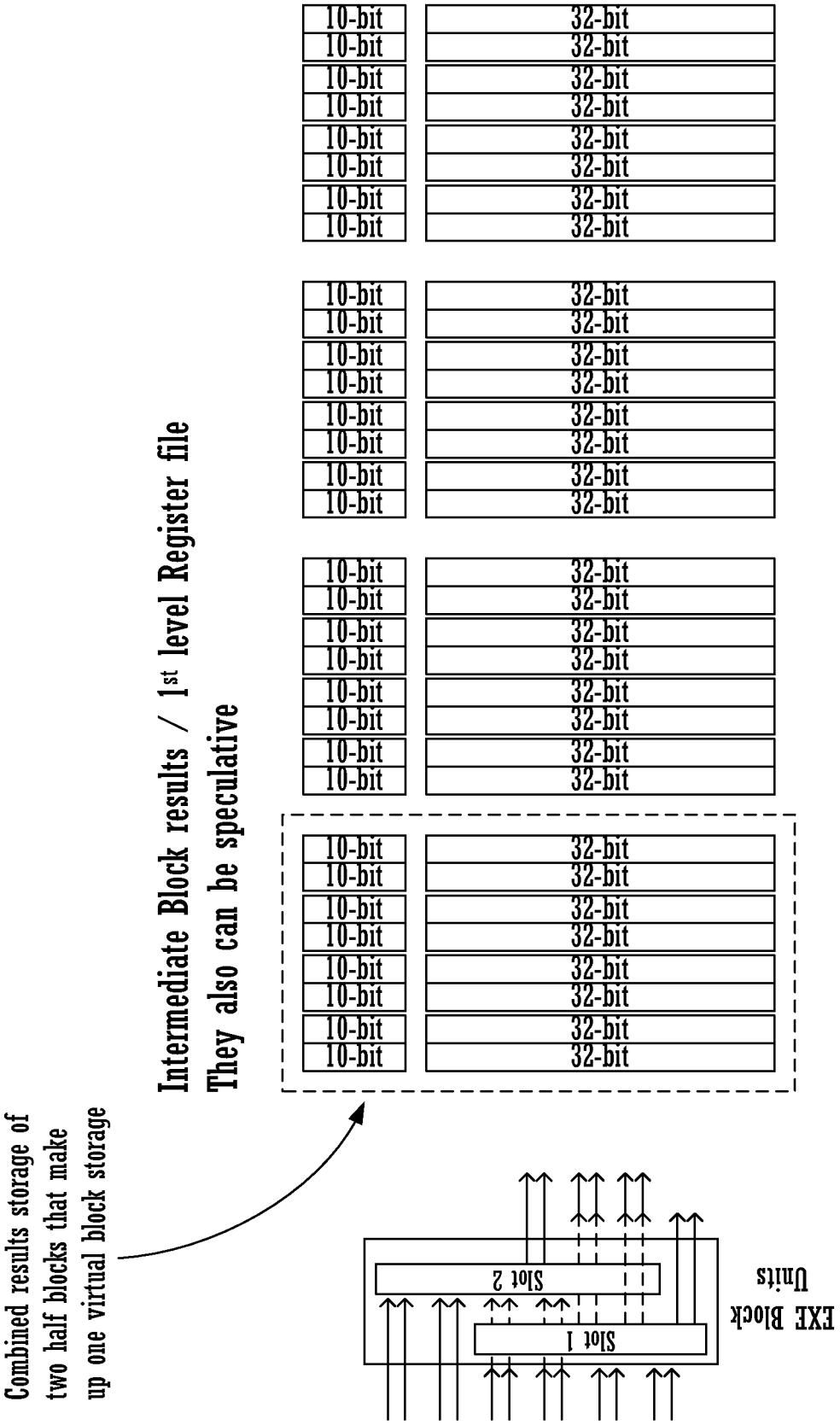

Intermediate Block results / 1st level Register file
They also can be speculative
10-bit
32-bit
Combined results storage of
two half blocks that make
up one virtual block storage
EXE Block
Units
Slot 1
Slot 2

Fig. 32

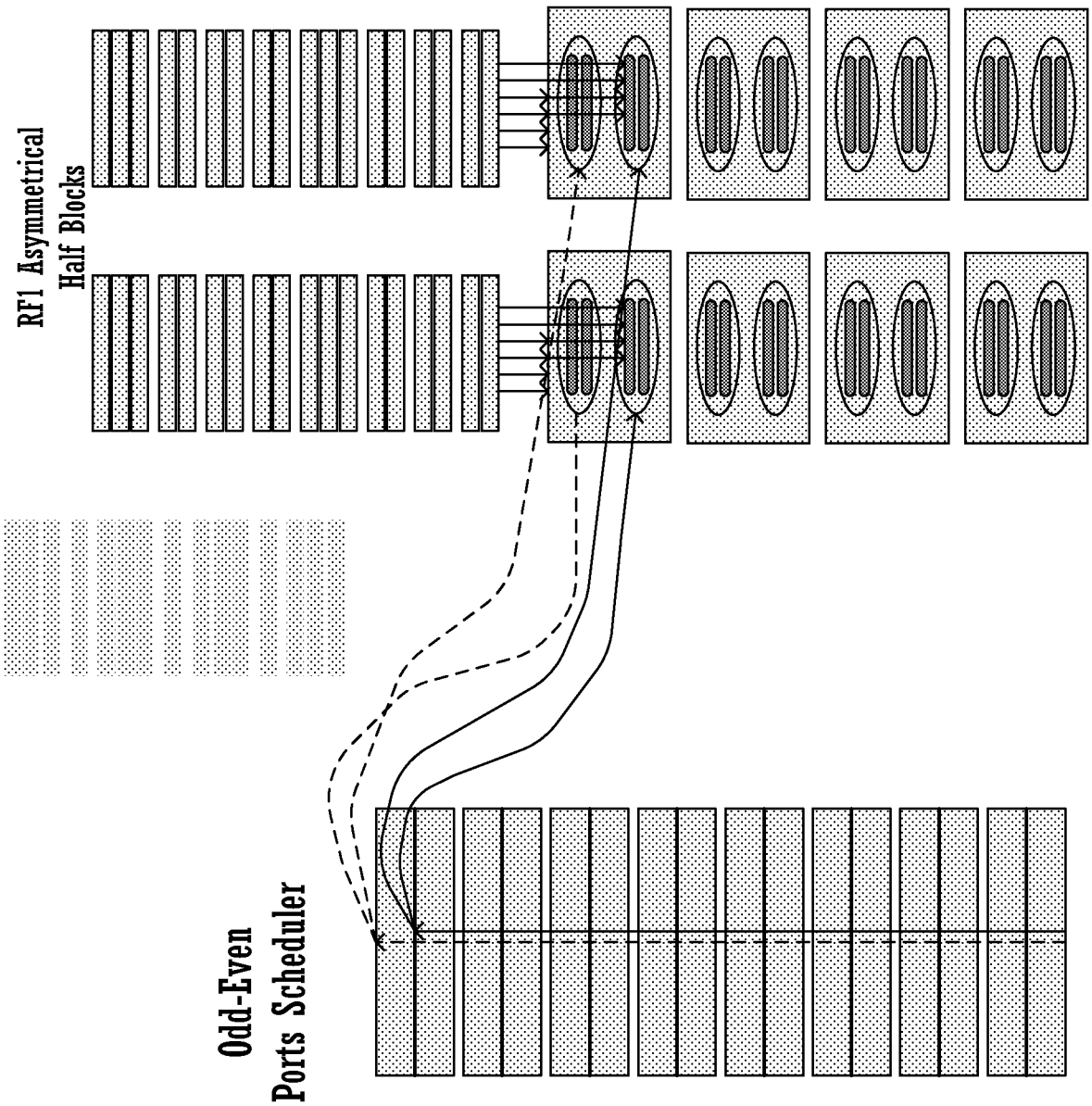


Fig. 33

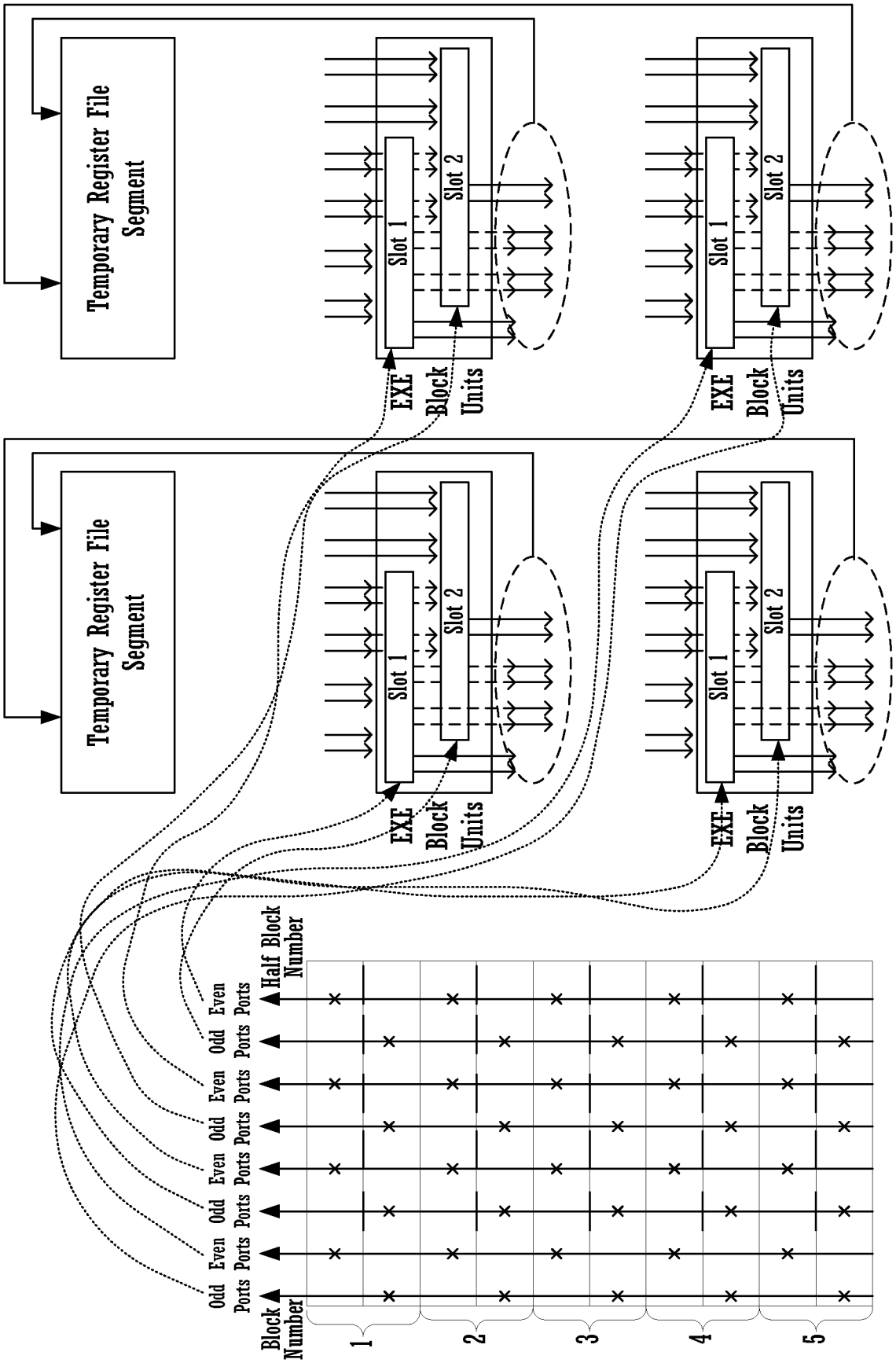
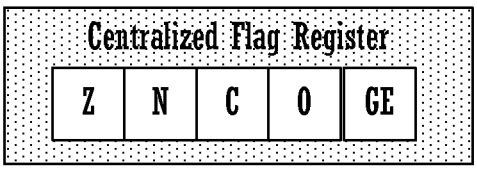

Temporary Register File Segment
Slot 1
Slot 2
EXE Block Units
Half Block Number
Even Ports
Odd Ports
Block Number
1
2
3
4
5

Fig. 34

Guest flag architecture emulation

Centralized flag architecture



Distributed flag architecture

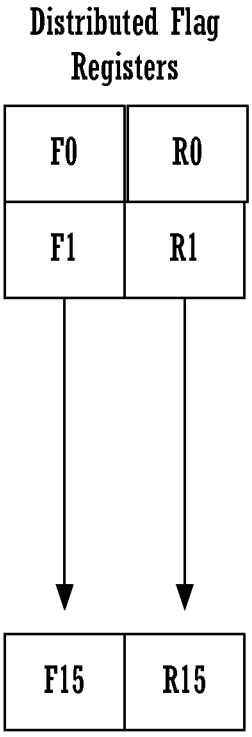


Fig. 35

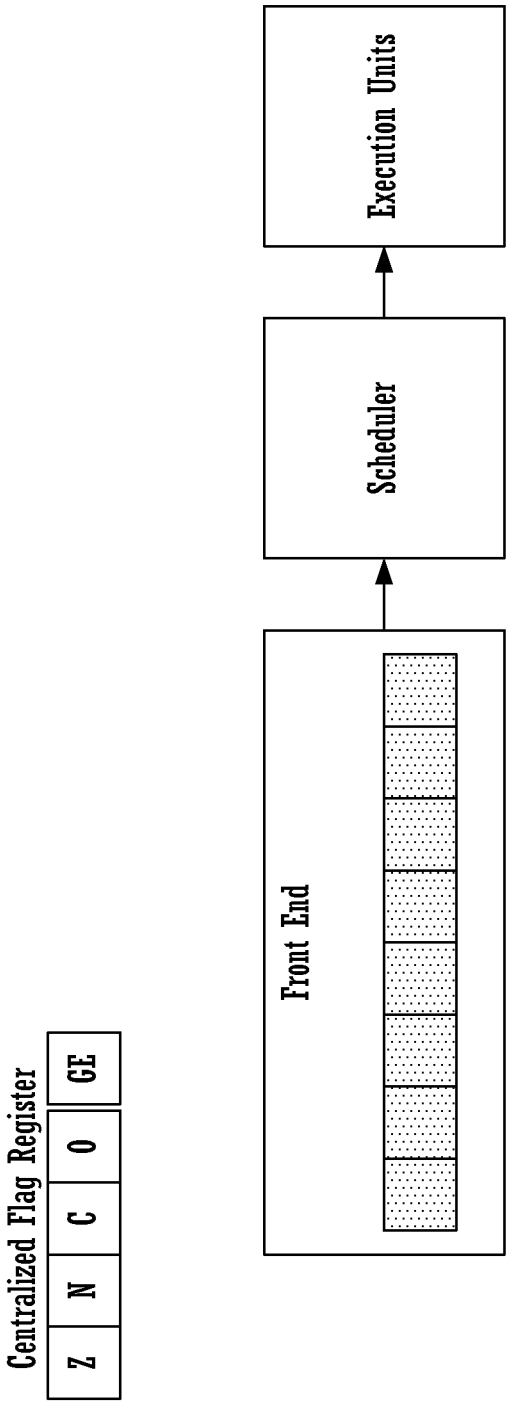

Execution Units
Scheduler
Front End
Centralized Flag Register
Z
N
C
O
GE

Fig. 36

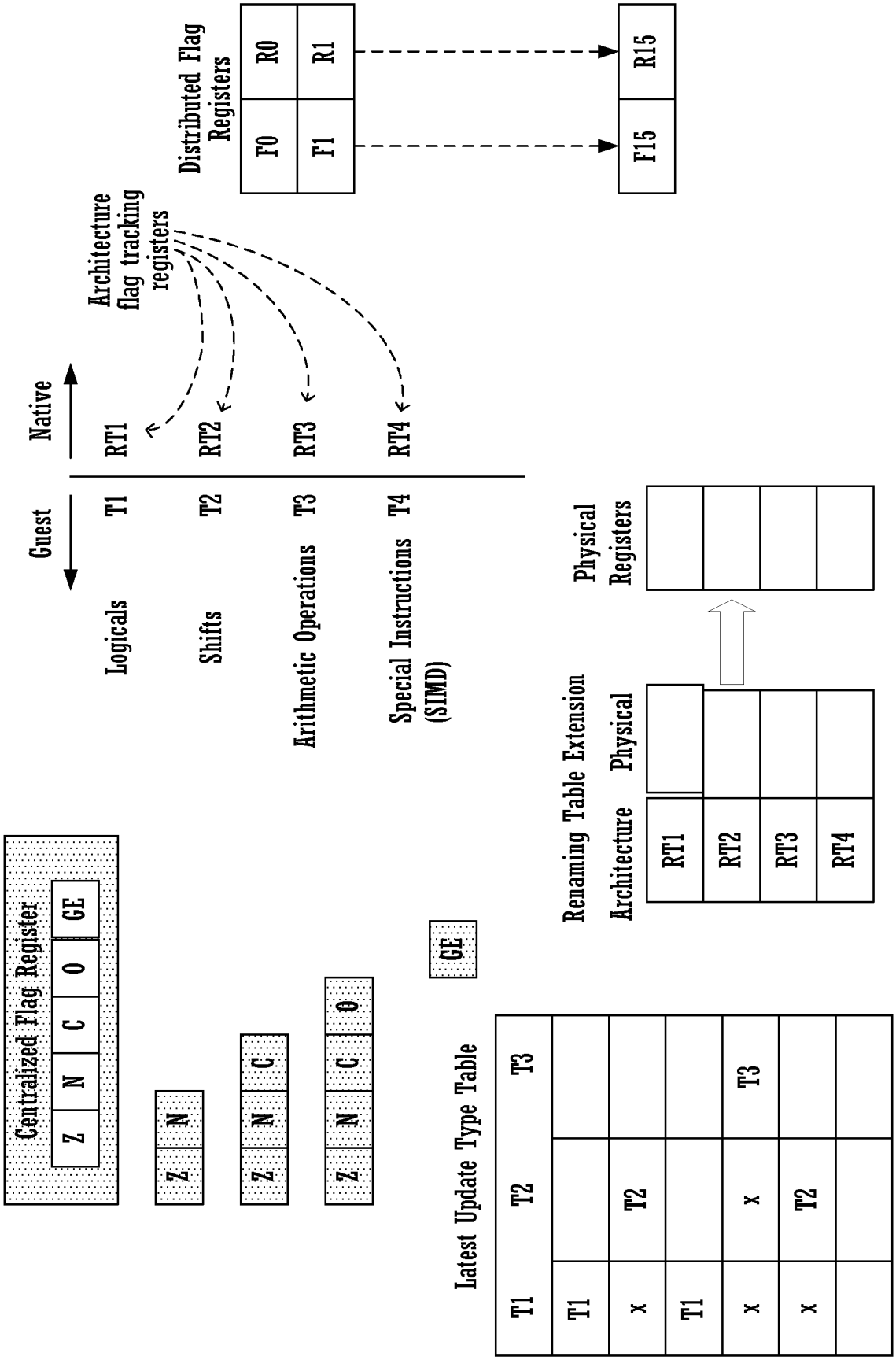

Distributed Flag Registers
F0
F1
F15
R0
R1
R15
Architecture flag tracking registers
Native
Guest
RT1
RT2
RT3
RT4
T1
T2
T3
T4
Logicals
Shifts
Arithmetic Operations
Special Instructions (SIMD)
Physical Registers
Renaming Table Extension
Architecture
Physical
Centralized Flag Register
Z
N
C
O
GE
Latest Update Type Table
T1
T2
T3
x

Fig. 37

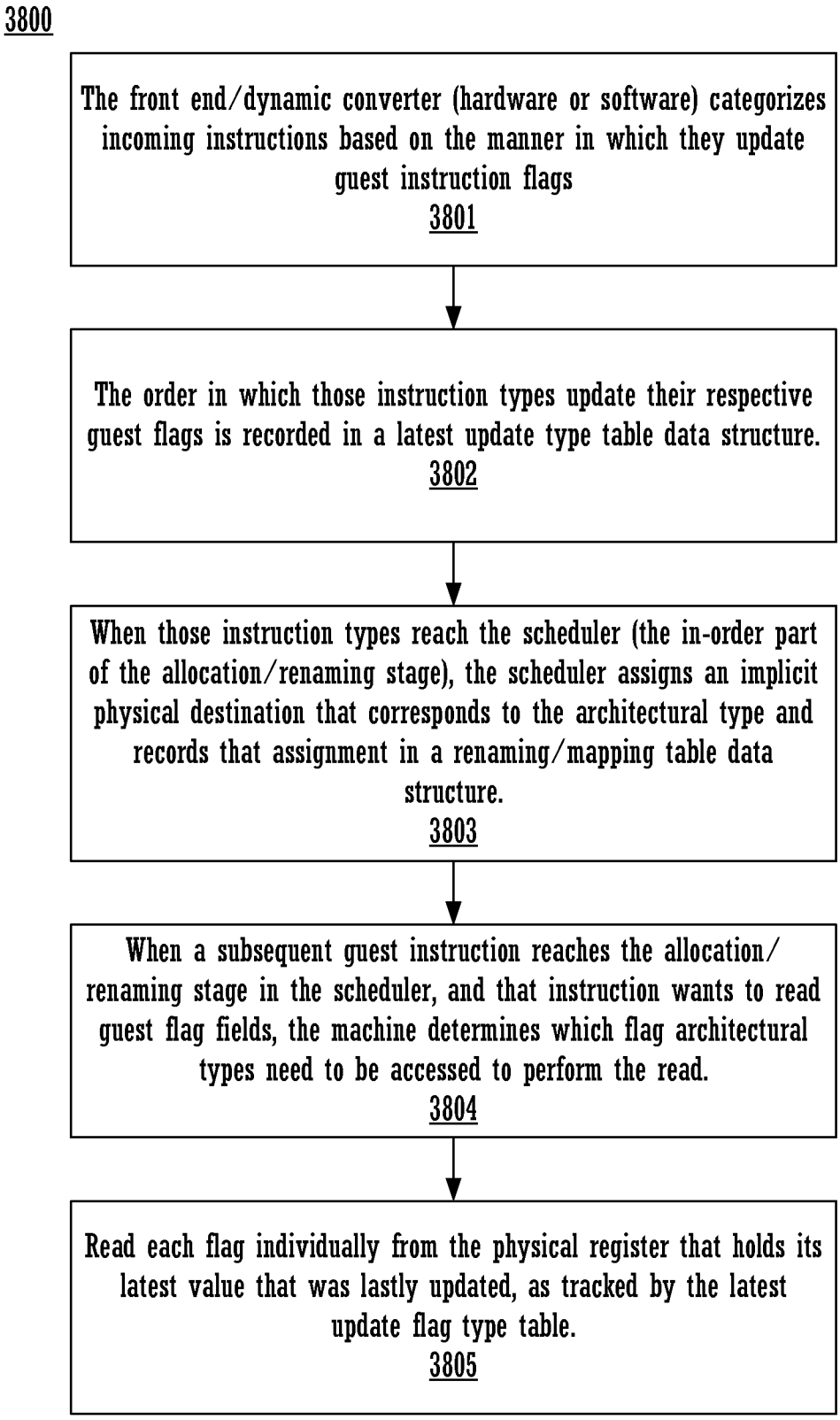

3800
The front end/dynamic converter (hardware or software) categorizes incoming instructions based on the manner in which they update guest instruction flags
3801
The order in which those instruction types update their respective guest flags is recorded in a latest update type table data structure.
3802
When those instruction types reach the scheduler (the in-order part of the allocation/renaming stage), the scheduler assigns an implicit physical destination that corresponds to the architectural type and records that assignment in a renaming/mapping table data structure.
3803
When a subsequent guest instruction reaches the allocation/ renaming stage in the scheduler, and that instruction wants to read guest flag fields, the machine determines which flag architectural types need to be accessed to perform the read.
3804
Read each flag individually from the physical register that holds its latest value that was lastly updated, as tracked by the latest update flag type table.
3805

Fig. 38

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US2014/024828

A. CLASSIFICATION OF SUBJECT MATTER

G06F 9/455(2006.01)i, G06F 9/46(2006.01)i

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

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F 9/455; G06F 15/00; G06F 13/38; G06F 12/00; G06F 9/45; G06F 9/46

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched
Korean utility models and applications for utility models
Japanese utility models and applications for utility models

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
eKOMPASS(KIPO internal) & Keywords: emulation, flag, architecture, instruction, schedule

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 2002-0029308 Al (BABAIAN, B. et al.) 7 March 2002 See abstract, paragraphs [0016]- [0026], and fig. 1.	1-19
A	US 05751982 A (MORLEY, J. E.) 12 May 1998 See abstract, col. 4, line 26 - col. 5, line 31, and fig. 3.	1-19
A	US 2004-0158822 Al (SANDHAM, J. H. et al.) 12 August 2004 See abstract, paragraphs [0043]- [0053], and fig. 2.	1-19
A	US 2003-0093776 Al (HILTON, R.) 15 May 2003 See abstract, paragraphs [0023]- [0026], and fig. 1.	1-19
A	US 2005-0289530 Al (ROBISON, A. D.) 29 December 2005 See abstract, paragraphs [0038]- [0044], and fig. 3.	1-19

☐ Further documents are listed in the continuation of Box C. ☒ See patent family annex.

* Special categories of cited documents:
"A" document defining the general state of the art which is not considered to be of particular relevance
"E" earlier application or patent but published on or after the international filing date
"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)
"O" document referring to an oral disclosure, use, exhibition or other means
"P" document published prior to the international filing date but later than the priority date claimed
"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
25 July 2014 (25.07.2014)	**28 July 2014 (28.07.2014)**

Name and mailing address of the ISA/KR
International Application Division
Korean Intellectual Property Office
189 Cheongsa-ro, Seo-gu, Daejeon Metropolitan City, 302-701, Republic of Korea
Facsimile No. +82-42-472-7140

Authorized officer
YU, Jintae
Telephone No. +82-42-481-8530

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No.
PCT/US2014/024828

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2002-0029308 Al	07/03/2002	US 2002-0046305 Al	18/04/2002
		US 2002-0059268 Al	16/05/2002
		US 2002-0092002 Al	11/07/2002
		US 6732220 B2	04/05/2004
		US 6820255 B2	16/11/2004
		US 7065750 B2	20/06/2006
US 05751982 A	12/05/1998	EP 0817996 Al	15/04/1998
		EP 0817996 Bl	16/08/2001
		WO 96-30829 Al	03/10/1996
US 2004-0158822 Al	12/08/2004	GB 2388218 A	05/11/2003
		JP 2004-038923 A	05/02/2004
		JP 4911868 B2	04/04/2012
		US 2003-0149963 Al	07/08/2003
		US 2008-0177985 Al	24/07/2008
		US 7331040 B2	12/02/2008
		US 8024555 B2	20/09/2011
US 2003-0093776 Al	15/05/2003	US 7092869 B2	15/08/2006
US 2005-0289530 Al	29/12/2005	None	