Office de la Propriété Intellectuelle du Canada
Un organisme d'Industrie Canada

Canadian Intellectual Property Office
An agency of Industry Canada

CA 2382151 C 2006/02/07

(11)(21) 2 382 151

(12) BREVET CANADIEN
CANADIAN PATENT

(13) C

(86) Date de dépôt PCT/PCT Filing Date: 2001/04/10
(87) Date publication PCT/PCT Publication Date: 2001/10/25
(45) Date de délivrance/Issue Date: 2006/02/07
(85) Entrée phase nationale/National Entry: 2002/02/15
(86) N° demande PCT/PCT Application No.: US 2001/040482
(87) N° publication PCT/PCT Publication No.: 2001/080502
(30) Priorité/Priority: 2000/04/17 (09/550,413) US

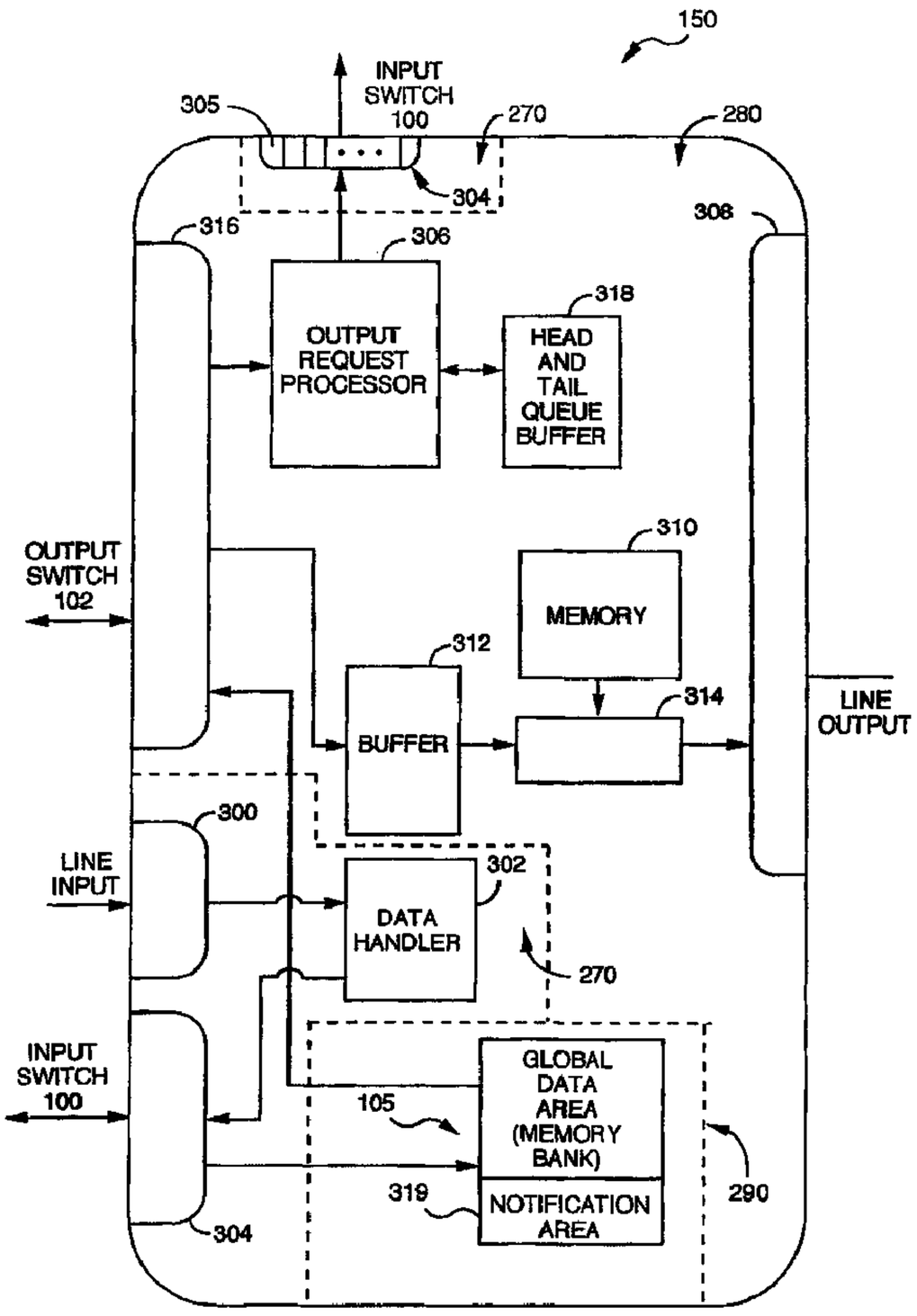
(51) Cl.Int./Int.Cl. *H04L 12/28* (2006.01),
H04L 12/56 (2006.01)

(72) Inventeurs/Inventors:
FERGUSON, DENNIS C., US;
PATEL, RAJIV N., US;
CHEUNG, GERALD, US;
SINDHU, PRADEEP S., US

(73) Propriétaire/Owner:
JUNIPER NETWORKS, INC., US

(74) Agent: SMART & BIGGAR

(54) Titre : FILTRAGE ET CONSULTATION D'ITINERAIRES DANS UN DISPOSITIF DE COMMUTATION
(54) Title: FILTERING AND ROUTE LOOKUP IN A SWITCHING DEVICE



(57) Abrégé/Abstract:
A method and apparatus for performing a lookup (Fig. 3) in a switching device (150) of a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a

CA 2382151 C 2006/02/07

(11)(21) 2 382 151

(13) C

(57) Abrégé(suite)/Abstract(continued):
sequence of operations for the lookup. The method includes determining a first lookup operation to be executed, executing the first lookup operation including returning a result and determining if the result includes a pointer to another lookup operation in the sequence of operations. If the result includes a pointer to another lookup operation, the lookup operation indicated by the result is executed. Else, the lookup is terminated.

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

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
25 October 2001 (25.10.2001)

PCT

(10) International Publication Number
WO 01/80502 A1

(51) International Patent Classification[7]: H04L 12/28, 12/56

(21) International Application Number: PCT/US01/40482

(22) International Filing Date: 10 April 2001 (10.04.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/550,413 17 April 2000 (17.04.2000) US

(71) Applicant *(for all designated States except US)*: **JUNIPER NETWORKS, INC.** [US/US]; 1194 North Mathilda Avenue, Sunnyvale, CA 94089-1206 (US).

(72) Inventors; and

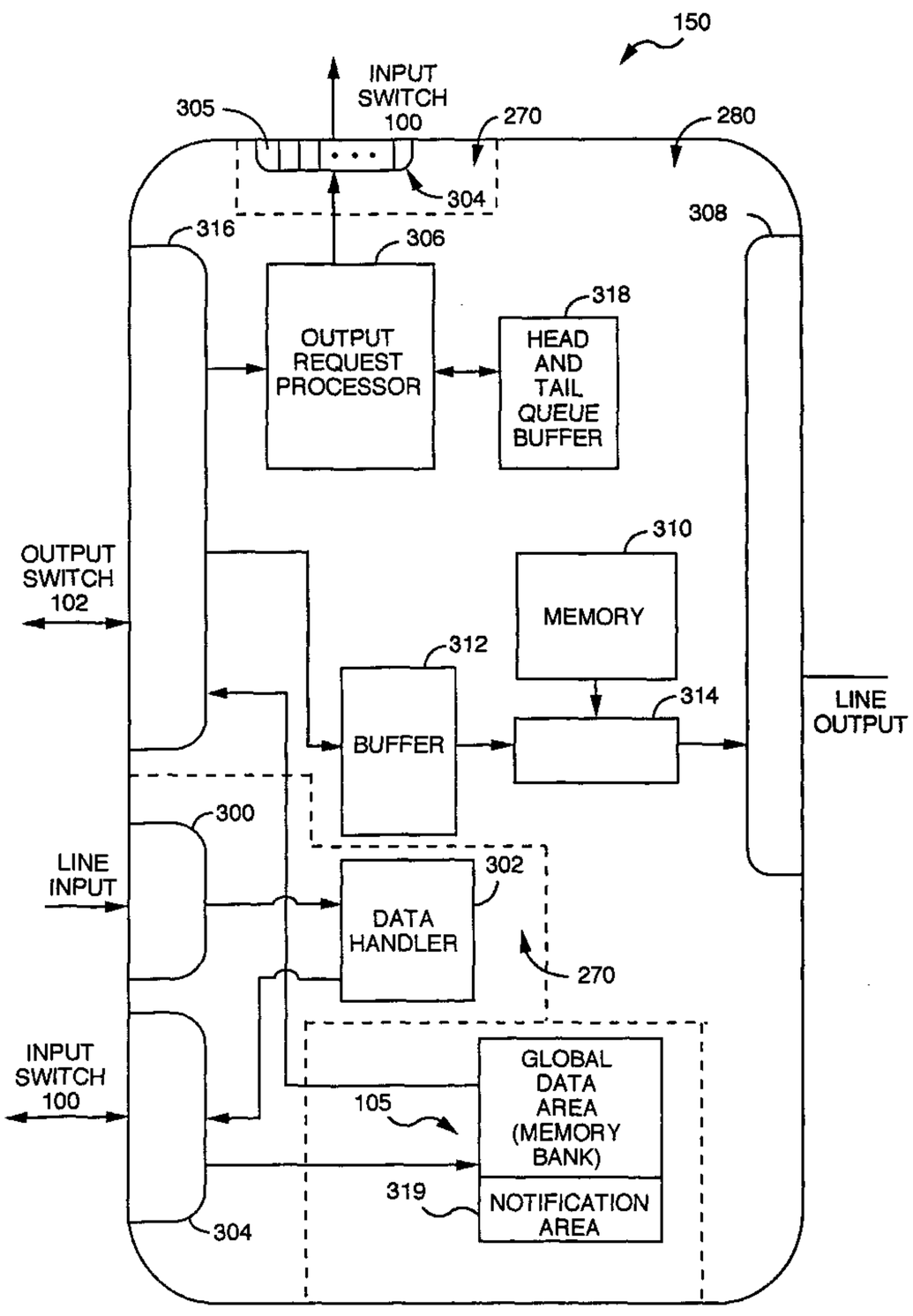
(75) Inventors/Applicants *(for US only)*: **FERGUSON, Dennis, C.** [CA/US]; 203 Orchard Glen Court, Mountain View, CA 94043 (US). **PATEL, Rajiv, N.** [US/US]; 628 Paco Drive, Los Altos, CA 94024 (US). **CHEUNG, Gerald** [US/US]; 725 Ashby Drive, Palo Alto, CA 94301-3148 (US). **SINDHU, Pradeep, S.** [IN/US]; 1557 Montalto Drive, Mountain View, CA 94040 (US).

(74) Agent: **KIRKLAND, Mark, D.**; Fish & Richardson P.C., 2200 Sand Hill Road #100, Menlo Park, CA 94025 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

[Continued on next page]

(54) Title: FILTERING AND ROUTE LOOKUP IN A SWITCHING DEVICE



(57) Abstract: A method and apparatus for performing a lookup (Fig. 3) in a switching device (150) of a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup. The method includes determining a first lookup operation to be executed, executing the first lookup operation including returning a result and determining if the result includes a pointer to another lookup operation in the sequence of operations. If the result includes a pointer to another lookup operation, the lookup operation indicated by the result is executed. Else, the lookup is terminated.

WO 01/80502 A1

WO 01/80502 A1

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *with international search report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

FILTERING AND ROUTE LOOKUP IN A SWITCHING DEVICE

Background

The present invention relates generally to data routing systems, and more particularly to methods and apparatus for efficiently routing packets through a network.

In packet switched communication systems, a router is a switching device which receives packets containing data or control information on one port, and based on destination information contained within the packet, routes the packet out another port to the destination (or an intermediary destination).

Conventional routers perform this switching function by evaluating header information contained within a first data block in the packet in order to determine the proper output port for a particular packet.

Efficient switching of packets through the router is of paramount concern. Referring now to Figure 1A, a conventional router includes a plurality of input ports 2 each including an input buffer (memory) 4, a switching device 6 and a plurality of output ports 8.

Data packets received at an input port 2 are stored, at least temporarily, in input buffer 4 while destination information associated with each packet is decoded to determine the appropriate switching through the switching device 6. The size of input buffer 4 is based in part on the speed with which the destination information may be decoded. If the decoding process takes too long as compared to the rate at which packets are received, large sized memory elements may be required or packets may be dropped.

In addition, the size of input buffer may be influenced by a condition referred to as "blocking". Packets may be forced to remain in the input buffer after the destination information is decoded if the switching device cannot make the connection. Blocking refers to a condition in which a connection cannot be made in the switch due to the unavailability of the desired output port (the port is busy, e.g., routing another packet from a different input port). In summary, the size of input buffer 4 is dependent on a number of factors including the line input rate, the speed of the lookup process, and the blocking characteristics for the switching device. Unfortunately, conventional routers are inefficient in a number of respects. Each input port includes a dedicated input buffer and memory sharing between input ports is not provided for in the design. Each input buffer must be sized to meet the maximum throughput requirements for a given port. However, design trade-offs (cost) often necessitate smaller buffers for each port. With the smaller buffers, the possibility arises for packets to be dropped due to blocking conditions.

50483-4

While excess memory capacity typically exists in the router (due to the varied usage of the input ports), no means for taking advantage of the excess is afforded.

To minimize the occurrence of dropping packets, designers developed non head-of-line blocking routers. Referring now to Figure 1B, a conventional non head-of-line blocking router includes a plurality of input ports 2 each including an input buffer (memory) 4, a switching device 6 and a plurality of output ports 8 each having an output buffer (memory) 9. In order to provide non head-of-line blocking, each output port 8 is configured to include an output buffer so that each output port can simultaneously be outputting packets as well as receiving new packets for output at a later time. As the size of the output buffer is increased, fewer packets are dropped due to head-of-line blocking at input ports.

However, these designs are even more inefficient in terms of memory capacity and cost. Again, each output port includes a dedicated output buffer and memory sharing between output ports is not provided for in the design. Each output buffer must be sized to meet the maximum throughput requirements for a given port (in order to maintain its non head-of-line blocking characteristics). Even more excess memory capacity typically exists in the router (due to the varied usage of the input ports and output ports), yet no means for taking advantage of the excess is afforded. Twice the amount and bandwidth of memory has to be used than required to support the amount of data being moved through these types of devices.

What is desirable is to produce a router where the data packets can flow to a common memory, while routing decisions are made off-line. By separating the data path,

the path along which the packet data traverses through the router, and the control path, a path used in evaluating the packet headers, memory can be conserved. In addition, by separating the data and control path, advanced filtering, policing and other operations can be performed without incurring expensive increases in the memory requirements for the router due to the additional time required to perform the extra operations.

Summary of the Invention

In an aspect the invention provides a method for performing a lookup in a switching device of a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the method comprising: determining a first lookup operation to be executed, the first lookup operation being selected from a group including at least a tree search, an index search, and a filter operation; executing the first lookup operation including returning a result; determining if the result includes a pointer to another lookup operation in the sequence of operations; if so, executing the lookup operation indicated by the result; and else, terminating the lookup, wherein a lookup operation includes a function list that specifies one or more functions to execute during the execution of the lookup operation and wherein the function is selected from the group of a management function, accounting function and policing function, wherein the method further comprises identifying when a lookup operation specifies a function and executing the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed, and wherein the execution of the function includes determining when a packet should be

sampled for further processing and including in the result a designation that indicates the packet is to be sampled.

In another aspect the invention provides a route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the route lookup engine comprising: a group of different lookup operation engines for executing lookup operations including returning a result, the group of different lookup operation engines including a tree search lookup engine, an index search engine, and a filter engine; a lookup engine operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network; and a memory configurable to store one or more tree data structures where the pointer returned for invoking the tree search engine includes an indicator point to a particular tree data structure stored in the memory to be searched in the lookup operation.

In another aspect the invention provides a route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the route lookup engine comprising: a group of

different lookup operation engines for executing lookup operations including returning a result, the group of different lookup operation engines including a tree search lookup engine, an index search engine, and a filter engine; a lookup engine operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network; and a memory configurable to store one or more filter data structures and where the pointer returned for invoking the filter engine includes an indicator pointing to a particular filter data structure stored in the memory to be searched in the lookup operation.

In another aspect the invention provides a route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the route lookup engine comprising: a group of different lookup operation engines for executing lookup operations including returning a result; and a lookup engine operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines

50483-4

based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network, wherein the lookup engine is operable to identify when a lookup operation specifies a function and execute the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed, and where the execution of the function includes determining when a packet should be sampled for further processing and including in the result a designation that indicates the packet is to be sampled.

In one aspect the invention provides a method for performing a lookup in a switching device of a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup. The method includes determining a first lookup operation to be executed, executing the first lookup operation including returning a result and determining if the result includes a pointer to another lookup operation in the sequence of operations. If the result includes a pointer to another lookup operation, the lookup operation indicated by the result is executed. Else, the lookup is terminated.

Aspects of the invention can include one or more of the following features. A lookup operation can be selected from the group of a tree search, an index search and a filter. A lookup operation can include a function list that specifies one or more functions to execute during the execution of the lookup operation. The function can be selected from the group of a management function, accounting function and policing function. The method can include identifying when a lookup operation specifies a function and executing the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed. The execution of the function includes determining when a packet should be sampled for further processing and including in the result a designation that indicates the packet is to be sampled.

In another aspect the invention provides a method for performing a lookup to determine routing for a packet through a switching device in a packet switched network. The method includes chaining a plurality of lookup operations in a sequence including linking each operation to a successive operation in the sequence such that an arbitrary sequence of operations can be specified to determine the routing of a packet through the switching device and executing the chain of lookup operations.

In another aspect the invention provides a method for performing a lookup in a switching device. The method includes identifying a first lookup operation in a sequence of lookup operations to be performed on a packet, executing the first lookup operation including returning a result that is a pointer to a subsequent lookup operation in the sequence, executing the subsequent lookup including returning a result that is a pointer to a next lookup operation in the sequence, continuing to execute lookup operations in the sequence until a lookup operation in the sequence returns a result that indicates that no more operations are to be processed and when a result indicates that no more operations are to be processed, returning a notification to the switching device that includes routing information for the routing of the packet through the switching device.

In another aspect the invention provides a method for policing a stream in a switching device in a packet switched network. The method includes, in a single read operation, determining a data rate for the stream in a time interval and a policing decision for a current packet in the stream and, in a single write operation, writing the policy decision and count information for the stream without requiring global overhead to clear the count at each time interval.

In another aspect the invention provides a method for updating a lookup data structure in a lookup process. The lookup data structure includes an arbitrary sequence of lookup operations for determining the routing of a packet through a switching device in a packet switched network. Each lookup operation invokes a distinct lookup algorithm that calls a data structure that when executed returns a result that links to a next lookup operation in the arbitrary sequence. The

method includes determining a location in the sequence of lookup operations where an update is desired. If the update adds a lookup operation to the sequence at the location, the added lookup operation is written to memory and linked to a next lookup operation after the location. Thereafter, a pointer in a lookup operation preceding the location is updated to point to the added lookup operation. If the update deletes a lookup operation from the sequence at the location, a pointer in a lookup operation preceding the location is updated to point to a next operation after the location and thereafter the lookup operation can be deleted from the memory.

In another aspect the invention provides a data structure for a lookup operation. The lookup operation is in a sequence of lookup operations that, when executed by a switching device in a packet switched network, determines routing for a packet through the switching device. The packet includes a key to be used in a lookup operation. The data structure includes a next hop identifier for linking operations in an arbitrary sequence to determine the routing of the packet through the switching device. The next hop identifier includes a pointer, an update and an offset. The pointer points to a particular lookup operation selected from a group of lookup operations. The update includes data for updating a pointer that points to a starting byte in the key to be used in the lookup operation. The offset indicates an offset bit down from the starting byte bit location to use for the lookup operation.

In another aspect the invention provides a method for performing a lookup to determine routing for a packet through a switching device in a packet switched network. The method includes providing plural algorithms in a lookup engine for performing distinct lookup operations, specifying an arbitrary sequence of lookup operations to be performed when the packet is received and executing lookup operations defined in the sequence in the order specified.

In another aspect the invention provides a route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup. The apparatus includes one or more lookup operation engines for executing lookup operations including returning a result and a lookup engine. The lookup engine is operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network.

Aspects of the invention can include one or more of the following features. The lookup operation engines can be selected from the group of a tree search look up engine, a index search index engine and a filter engine. The route lookup engine can include a memory configurable to

store one or more tree data structures and where the pointer returned for invoking the tree search engine includes an indicator pointing to a particular tree data structure stored in the memory to be searched in the lookup operation. The memory can include one or more index data structures and where the pointer returned for invoking the index search engine includes an indicator pointing to a particular index data structure stored in the memory to be searched in the lookup operation. The memory can store one or more filter data structures and where the pointer returned for invoking the filter engine includes an indicator pointing to a particular filter data structure stored in the memory to be searched in the lookup operation.

A lookup operation can include a function list that specifies one or more functions to execute during the execution of the lookup operation and where the lookup engine can be operable to read the function list and execute the one or more functions in the lookup. The function can be selected from the group of a management function, accounting function and policing function. The lookup engine can be operable to identify when a lookup operation specifies a function and execute the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed. The execution of the function can include determining when a packet should be sampled for further processing and including in the result a designation that indicates the packet is to be sampled.

In another aspect the invention provides an apparatus for policing a stream in a switching device in a packet switched network and includes a buffer for storing a count and a threshold for the stream and a policing engine. The policing engine is operable to, in a single read operation, determine a data rate for the stream in a time interval and a make policing decision for a current packet in the stream and, in a single write operation, write count information for the stream after each packet in a stream is processed without requiring global overhead to clear the count at each time interval.

Aspects of the invention can include one or more of the following features. The buffer can include four values including a last time adjustment value that is written in the single write operation to indicate a last time that the data rate was calculated, a current count value that indicates an amount of data that had been written as of the last adjustment time, a threshold value that indicates the threshold amount of data that can be passed in the stream before policing is required, and a credit value indicating the amount of counts to be applied to the current count per unit time. The policing engine can be operable to read, in the read operation, the four values and make the policing decision, and operable to write, in the write operation, a new value for the last time adjustment value and the current count value that reflects the processing of a current packet.

Aspects of the invention can include one or more of the following advantages. A technique is provided to implement traffic policing based on a fixed window monitoring mechanism with a minimal use of memory bandwidth. A method and apparatus are provided for

implementing a general purpose packet filter within a lookup engine for longest match lookups. An apparatus is provided that supports chained lookup operations. The apparatus includes a route lookup engine that includes plural engines each for performing a different type of lookup operation. An apparatus is provided to allow for the chaining of plural lookup techniques in a switching device.

Other advantages and features will be apparent from the following description and claims.

Brief Description of the Drawings

FIG. 1A and 1B are block diagrams of conventional router devices.

FIG. 2A is a schematic block diagram of a data routing system.

FIG. 2B is a schematic block diagram of a router.

FIG. 3 is a schematic block diagram of a multi-function port.

FIG. 4 is a schematic diagram showing the data transfers between components of the router of Figure 2B.

FIG. 5 is a schematic block diagram of an input switch.

FIG. 6 is a schematic diagram of memory structure for the router.

FIG. 7 is a schematic diagram of the global memory for the router.

FIG. 8 is a schematic block diagram of a controller.

FIG. 9 shows a schematic block diagram for a key lookup engine.

FIG. 10 shows a packet processing operation.

Figure 11 is a schematic block diagram of an output switch.

Figure 12 is a schematic block diagram for an output section of a multi-function port.

Figure 13 is a schematic block diagram for a queue system for storing notifications.

Figure 14 is a flow diagram for a process of routing a packet through a router.

Detailed Description

Referring to FIG. 2A, in a packet switching system, a source 10 is connected to one or more routers 20 for transmitting packets to one or more destinations 30. Each router includes a plurality of multi-function multiports that are connected to various sources and destinations. A packet from source 10 may pass through more than one router 20 prior to arriving at its destination.

Referring to FIG. 2B, each router 20 includes an input switch 100, an output switch 102, a global data buffer 104 including one or more memory banks 105, a controller 106 and a plurality of multi-function multiports 150 (150-0 through 150-3), respectively. Associated with

the controller 106 is a controller memory 109 for storing routing information. Input switch 100 and output switch 102 are connected to each multi-function multiport 150 in router 20. In one implementation, router 20 includes plug-and-play multi-function multiports which allow for easy expansion capability. The present invention will be described with reference to a system including eight multi-function multiports 150 (even though FIG. 2B only shows four), with each multi-function multiport including up to sixteen input ports and sixteen output ports. Other configurations may be used depending on user load conditions. Each multi-function multiport includes one or more input ports, one or more output ports and a memory. The configuration and operation of the multi-function multiports will be described in greater detail below.

In operation, packets are received at a multi-function multiport 150, transferred to input switch 100 and stored temporarily in global data buffer 104. When the packet is received by input switch 100, a key and other information is read from the packet and transferred (in the form of a notification) to controller 106. The key contains destination information which is derived from the header field associated with the first block of data in a packet and other information (such as source ID, priority data and flow ID).

A route lookup engine 110 in controller 106 performs a lookup based on the notification information and returns a result which includes the output multiport associated with the destination. The result is coupled with other information (such as source ID, flow ID and packet length) for routing the packet through router 20 and provided as a notification from controller 106 to output switch 102. Output switch 102 transfers the notification to the identified multi-function multiport 150. Upon receiving the notification information, the multi-function multiport 150 initiates the transfer of the packet from global data buffer 104 through output switch 102 to the appropriate multi-function multiport 150.

Multi-function Multiports

Referring to FIG. 3, each multi-function multiport 150 includes an input section 270, an output section 280 and a memory section 290.

Input section 270 includes a line input interface 300, a data handler 302 and an input switch interface 304. Output section 280 includes an output request processor 306, a line output interface 308, a storage device (memory) 310, stream output buffers 312 (one for each output stream), output formatter 314, an output switch interface 316 and head and tail queue buffer 318. In addition, the output section includes a portion of input switch interface 304. Specifically, input switch interface 304 includes read request queues 305, one for each memory bank. The use and operation of the read request queues, stream output buffers, and head and tail queue will be discussed in greater detail below.

Memory section 290 includes a memory bank 105 (which represents a portion of the global data buffer 104) and a notification area 319. The use an operation of the memory section will be discussed in greater detail below.

The multi-function multiport is used in conjunction with the input switch, output switch and controller as is shown in FIG. 4. The various piece components of the input section, output section and memory section are described in greater detail below. The combination of the devices into a single unit simplifies the interfaces between the components.

Referring again to FIG. 3, packets are received at line input interface 300. As the packets are received, data handler 302 divides the packets received into fixed lengths cells. In one implementation, the length of each cell is 80 bytes, with 16 bytes of internal header (control information) and 64 bytes of cell data. As the data handler divides the incoming packets into fixed length cells, it synchronously outputs the cells to input switch 100 through input switch interface 304.

Each cell transferred from a multi-function multiport 150 to the input switch contains a cell header and cell data. The cell header can include a type field, stream field, and packet header fields. In addition, the cell header can include an independent read request in the form of a multi-function multiport identifier and address.

The type field indicates the type of cell to be transferred from the multi-function multiport. At each cell slot (20 clock cycles in one implementation), a multi-function multiport may transfer either a data cell, an indirect cell placeholder, or a delayed indirect cell placeholder. Data cells contain data associated with an incoming packet. An indirect cell placeholder is an empty cell, and is used in conjunction with indirect addressing for the storage of the cells in the global data buffer 104. Delayed indirect cell placeholders arise when a data stream that requires indirect addressing terminates at a time prior to the designated time for writing the last indirect addressing cell associated with the data stream to global data buffer 104. The generation and operation of indirect placeholders and delayed indirect placeholders will be discussed in greater detail below.

The stream field indicates the stream to which the cell data belongs. In one implementation, each multi-function multiport is capable of handling up to sixteen separate streams of data at a time, one on each of its respective 16 input ports.

The packet header field contains header information associated with a given packet and includes start offset information, packet length and interface index information.

The multi-function multiport identifier identifies the multi-function multiport which is sourcing the read request. The address indicates the address in global data buffer 104 to be read.

A single cell can be transferred from a multi-function multiport 150 to input switch 100 at each cell (time) slot "T". For a given cell slot "T", input switch 100 receives a total of "N" cells,

where "N" is equal to the number of multi-function multiports. Similarly, a single cell can be transferred from the input switch 100 to memory 104, from the memory 104 to the output switch 102, and finally from the output switch 102 to a multi-function multiport 150 at each cell (time) slot "T" as is shown in FIG. 4.

In one implementation, cells from a given stream may be written to memory in an order that is different from the arrival order. These out of order writes are performed to make efficient use of scarce bandwidth between the multi-function multiports and the input switch. When a packet comes in to the multi-function multiport, it is broken up into cells as the bytes arrive and the cells are placed in per-bank output queues on the way to the input switch. These queues are designed to share scarce interconnect bandwidth between the streams of a multi-functional multiport in the most efficient way possible, but they have the detrimental effect of reordering cells at the interface between the multi-function multiport and the input switch. Thus the cells from a given stream may arrive at the input switch out of order. The multi-function multiport marks the data cells of a stream with one of four codes: first cell (FC); intermediate data cell (DC); last cell (LC); or first cell which happens to be also a last cell (FLC).

Input Switch

Referring to FIGs. 2B and 5, input switch 100 includes a round robin data handler 500, one or more input interfaces (501-0 through 501-7, one for each multi-function multiport 150), one or more memory interfaces 502 (502-0 through 502-7, one associated with each memory bank), a like plurality of pointers 504 (504-0 through 504-7), an output processor 505, one or more output interfaces 506 (506-0 through 506-7, one for each multi-function multiport 150), a reservation table 508, an indirect cell processor 510, controller interface 512 and read controller 517.

a) Transfers through the Input Switch

Round robin data handler 500 receives cells from each multi-function multiport and transfers them to output processor 505 for output to an appropriate memory bank 105 in global data buffer 104. Round robin data handler 500 services the inputs (cells) received on input interfaces 501 in a round robin, time division multiplexed manner. That is, for a given cell slot, one cell from each multi-function multiport is received at the round robin data handler 500 and subsequently transferred to output processor 505 for transfer at the next cell slot to a memory bank 105 in global data buffer 104. At the next time cell slot, data handler 500 transfers the next cell received from the same multi-function multiport to output processor 505 for transfer to a different memory bank. In one implementation, the next cell received is transferred to the next memory bank (next in numerical order, modulo *N*) in the memory array. Alternatively, another

time dependent permutation may be used to control the transfer of successive cells from the same multi-function multiport.

Round robin data handler 500 and output processor 505 within the input switch 100 transfer cells out to global data buffer 104 on transmission lines. Output processor 505 outputs one cell to each memory bank in a single cell slot. One cell from each multifunction multiport is written to global data buffer 104 per cell slot. Round robin data handler 500 time division multiplexes the transfers to output processor 505 such that consecutive cells from the same multi-function multiport are written to consecutive memory banks 105 (modulo *N*) in global data buffer 104.

Pointer 504 indicates the location in an associated memory bank to which the next cell will be written. Output processor 505 writes a cell to a memory location in a particular memory bank based on the next available address in the bank as is indicated by the associated pointer 504.

b) Key Reading and the Linking Process

Round robin data handler 500 includes a key reading engine 514 for determining the key information associated with a first cell in a packet and a linking engine 515 for linking cells in the same packet.

The process of reading key information is known in the art. After the key is determined for a given packet, it is stored temporarily in key buffer 516 in input switch 100 until the entire packet has been stored in global data buffer 104. Each entry in the key buffer is referred to as a notification or "info cell" and includes a key, full address, offsets and an indirect cell indicator and can include other information.

Linking engine 515 determines the starting address (full address) in memory for where the first cell in a given packet is to be stored in memory. The starting address includes the bank number in global data buffer 104 (the bank number which is assigned to store the cell by round robin data handler 500) and the first available address location in the designated bank (as is indicated by the associated pointer 504). The starting address is stored in key buffer 516 along with the associated key for the packet. When the next cell associated with the same packet arrives at switch 100, an offset associated with the offset at which the cell is to be written (relative to the full address) is computed and stored in key buffer 516. In one implementation, up to four offsets are stored. Each offset address is computed based on the relative offset in memory between the location of the last cell in memory and the value of the pointer 504 associated with the current memory bank which is to be written.

If more than five data cells are included in a packet, then the indirect cell indicator for that packet is set, and the last offset indicates the address in memory where the first indirect cell associated with the packet is stored. Indirect cells are described in greater detail below.

After the packet has been stored in memory, the associated notification in key buffer 516 (a route lookup request) is forwarded through the controller interface 512 to the controller 106 for processing. Alternatively, the notification may be transferred after the first five cells have been stored in memory.

As described above, the data cells are stored in the global buffer upon receipt. The data path for the data packets flows directly from the input port on which a packet is received (the multi-function multiport 150) to the global data buffer 104. The data packets remain in the global data buffer 104 while a routing decision is made in a separate control path using controller 106. The separation of the data path and control path allows for the sharing of the memory resources among all of the input ports.

The linking or threading of cells for a packet is performed by using the offsets described above and indirect cells. Offsets are used to link cells in a packet. Offsets may be stored along with key information and routed through controller 106 (Figure 2B) or may be stored in indirect cells. In one implementation, if a cell contains 5 cells or less, no indirect cells are required to be used. Indirect cell processor 510 performs the linking of cells in memory for a given packet. Indirect cell processor 510 generates indirect cells for storage in global data buffer 104. Indirect cells contain offset information associated with the relative offset in memory space between contiguous cells in the packet. Indirect cell processor includes indirect cell memory 520 for storing indirect cell data during the formation of indirect cells.

As was described above, when a packet is received, the linking engine processes the first five cells and stores linking information in the form of a start address and four offsets in key buffer 516. In the event more than five cells are contained within a packet, the indirect cell processor takes over for the linking engine and computes the offsets associated with the locations in memory where the remaining cells in the packet are stored. Round robin processor 500 passes cells to the output processor 505 for transfer to an associated memory bank in global data buffer 104. Round robin processor 500 enables the indirect cell processor when the packet being processed contains more than 5 cells (based on header information included within the first cell). At the time for writing the fifth cell to memory, indirect cell processor 510 stores in indirect cell memory 520 the address (the "indirect cell address") associated with the location in memory at which the fifth cell would have been written if it had been the last cell in the packet. The indirect cell address indicates the location in memory where the indirect cell is to be written when full (or when the last cell of the packet is processed).

When an indirect cell is full (having stored offsets in all available locations except the last field), then the indirect cell processor stores the offset associated with the location in memory where the next indirect cell is located. Thereafter, the full indirect cell is written to its appropriate place in memory. The writing of the indirect cell to memory coincides with the receipt of an indirect cell placeholder by the input switch 100 from the associated multi-function multiport 150. This process continues until the last cell in a packet is stored in memory. At that time, the last indirect cell is written to memory, and the associated entry from the key buffer 516 is transferred to the controller 106 for processing. For a given packet, all indirect cells are written to the same memory bank in the global memory buffer.

As often will be the case, the last cell of a packet will not coincide with the timing required to write the completed indirect cell immediately into memory. This is because packet length is completely arbitrary. The end of a packet will likely not coincide with the last available entry of an indirect cell. When a packet has completed (all cells have been received by the input switch) and a last entry in the indirect cell is written, the indirect cell is free to be written to memory. However, the writing will be delayed until the proper time, hence the term delayed indirect cell. A delayed indirect cell is a indirect cell that is the last indirect cell associated with a packet. It is delayed, because it is written to memory after the rest of the packet has been written to memory. The timing of the write to memory is dictated by the address which is reserved for the indirect cell. As was described above, at the time for the creation of an indirect cell, its position in memory is reserved. The delayed indirect cell will be written to memory at the next time slot available for the particular multi-function multiport to write to the particular memory bank after the packet has been completed. The timing of the write to memory of delayed indirect cells coincides with the receipt of a delayed indirect placeholder from the appropriate multi-function multiport 150.

c) Transfers to Memory

At each cell slot, output processor 505 generates a cell that includes a read request source field, read address, write address and data field (cell data received from multiport 150). The read request source field indicates the output port (in the particular multi-function multiport 150) requesting the read (destination output port). Output processor 505 receives read requests from read controller 517 and bundles the read request with any write request received from round robin data handler 500 destined for the same memory bank. At each cell slot, output processor 505 provides a cell which may include a write and read request to each memory bank 105 in global data buffer 104.

Read controller 517 controls the transfer of read request signals flowing from input switch 100 out memory interface 502 to the individual memory banks in global data buffer 104.

Read controller 517 receives read requests from each multi-function multiport through output interfaces 506. The format of each request includes source identification (output port) and a full address in memory which is to be read. At each cell slot, each multifunction multiport port may generate a read request for processing by switch 100 to read a memory location in global data buffer 104, resulting in the reading of a cell (a read reply) from a memory bank 105 (on a subsequent cell slot) to output switch 102.

Read controller 517 loads a reservation table 508 as requests to transfer packets are received from the various multi-function multiports 150. The reservation table is loaded such that at every cell slot a single read request is generated for each bank of memory 105. The structure of the reservation table is described in greater detail in "Separation of Data and Control in a Switching Device". At each cell slot, each multi-function multiport is capable of requesting a read from a single memory bank 105 in global data buffer 104. Associated with reservation table 508 is a read pointer. The pointer points to a next row in the reservation table to be read. Rows ahead of the read pointer correspond to requests that will be queued at a later cell slot time. In one implementation, the pointer moves at least one row in each cell slot time.

Memory Architecture

Referring now to FIG. 6, main memory 104 is used as temporary buffer storage for packets flowing into the system on input streams 1052 and out of the system on output streams 1054. Main memory is divided into two distinct parts: a global data buffer 104 that is used to store incoming packets while one or more lookup engines in the controller 106 determine the outgoing stream for each packet; and packet notification queues 319 that are used to store packet pointers (notifications) after the outgoing stream has been determined. Notification queues 319 are associated with outgoing streams, whereas the global data buffer 104 forms a common pool shared amongst all the streams.

Referring now to FIG. 7, main memory includes a plurality of memory banks. Associated with each memory bank is an input switch interface (an input port) 304 and output switch interface (an output port) 316. At each cell slot, each memory bank receives at most one write and one read request via input switch interface 304. The write requests are associated with cells received from a multi-function multiport 150. Read requests reflect a request for cell data to be transferred from a memory bank to output switch 102 for ultimate transfer to a requesting multi-function multiport 150.

The memory in the multi-function multiport configuration is physically distributed across a number of banks b, one bank for each active multi-function multiport in the system. Each bank is divided into two contiguous, non-overlapping regions referred to as global data area 105 and the notification area (notification queues 319). The global data area for a bank constitutes $1/b$ of

the memory of the global data buffer 104. The notification area provides space for queuing notifications that will be sent out on the line output interface 308 for a given multi-function multiport. Typically, the global data area is four times larger than the notification area; this factor derives from the ratio between data size and notification size for the shortest packet.

In one implementation, each bank's memory bandwidth is sufficient for reading and writing packets from a full-duplex OC-48 interface as well as for queuing and dequeuing notifications for the worst-case example of single-cell packets. Thus, both the aggregate memory size and the aggregate memory bandwidth scale linearly with the number of active multi-function multiports *b* in the system.

In one implementation, each memory bank is implemented as two sub-banks using two 72-bit wide SDRAM (static dynamic random access memory) DIMM's (dynamic in-line memory modules) cycling at 125 MHZ. The sub-banks are transparent to the input and output switch resulting in what appears to be one continuous bank from the perspective of the switches. However, the sub-bank architecture allows for better throughput. Each DIMM has a 72-bit wide ECC (error correction code) protected data path going to 9 SDRAM chips each of which is 8 bits wide. The two DIMM's have separate address busses and are addressed independently of one another. The DIMM's are interleaved on bit 0 of the 23-bit address. In one implementation, the smallest memory bank configuration is 32 MBytes, using 16 Mbit chips and the largest is 512 MBytes, using 256 Mbit chips.

As was described above, a bank can receive at most one read request and one write request every cell slot. Since a cell slot is 20 clock cycles at 125 MHZ, this works out to a peak bandwidth demand of 400 MBytes/sec for reads and 400 MBytes/sec for writes. The worst case notification load occurs for single cell packets. For unicast traffic, this load is exactly 1/4 the data bandwidth which works out to 100 MBytes/sec for reads and 100 MBytes/sec for writes. In this implementation, the total peak memory bandwidth needed is therefore 1 GByte/sec.

In this implementation, the peak transfer rate of each DIMM is 1 GByte/sec, but the sustained rate depends on the actual mix of reads and writes and how the addresses are distributed over the internal DIMM banks. In practice, each DIMM is expected to deliver a sustained data rate of around 650 MBytes/sec. The total of 1.3 GBytes/sec supplied by the two groups is 30% larger than the maximum sustained requirement of 1 GByte/sec. The 30% headroom provides a way to sustain instantaneous loads where one DIMM has more references directed to it than the other. The memory controller for the two DIMM's resides in the multi-function multiport.

In one implementation, all banks are made the same size and approximately 1/5th of the memory in each bank is allocated to the notification area and 4/5th to the global data area. The purpose of this allocation is to make it exceedingly unlikely for a stream to run out of memory

because of space in its notification queue. With a worst case packet size of 64 bytes, notifications (sized at 16 bytes) need 1/4th the amount of storage that packet data needs, which is exactly the proportion allocated. Any cell in the global data buffer may be accessed via its physical cell pointer, which identifies the physical bank number and the address of the cell within the bank. The physical cell pointer defines a system-wide physical address space. To simplify address computations, as well as to provide a mechanism to detect old packets, accesses to the global packet buffer are performed through a system-wide virtual address space that maps to the physical address space.

Incoming packets are broken up into as many cells as needed and the cells are written to the global packet data buffer as they arrive as described above. The global data buffer is treated as a single large circular buffer. The input switch maintains an array of write pointers, one per active bank, to keep track of where to write the next cell. The pointers start out at one end of the buffer and advance until they eventually wrap around and overwrite packets that were written a long time ago. An ageing mechanism is used to guard against reading cells that may have been overwritten by subsequent packets. The cells of packets arriving on a given stream are interleaved strictly across the active banks to spread the bandwidth load.

Controller

Referring now to Figure 8, controller 106 includes controller memory 109, route lookup engine 110, input switch interface 800 and output switch interface 802. Controller 106 receives a route lookup request from input switch 100 at the input switch interface 800. In one implementation, a plurality of route lookup engines 110 are included in controller 106, each receiving lookup requests in round-robin fashion so as to speed the routing process. In one implementation, controller memory 109 is a four-bank static random access memory (SRAM) that requires thirty six route lookup engines 110 to service at full bandwidth.

The present invention is scalable with respect to performance. That is, the number of route lookup engines 110 included within the controller may be increased to provide higher performance without requiring an increase in memory size. In one implementation, the number of route lookup engines is nines times as great as the number of memory banks in controller memory 109. Alternatively, lesser cost and performance units may use lesser numbers of route lookup engines 110 or more engines as required.

a) Controller operation

Referring to Figs. 2B, 3 and 8, in operation, packets are received at an input port 150, transferred to input switch 100 and stored temporarily in memory 104. When the packet is received by switch 100, a key extraction engine reads the key from the packet and transfers the

key and other information (the notification) to controller 106. The input switch also includes a transfer engine for transferring packets received from an input port 150 to memory 104.

The key includes at least destination information and may also include source information, a flow identifier and physical source information (input port ID). The key can be located in the header field associated with the first block of data in a packet. The header may contain other information (ISO layer 2 and layer 3 headers), such information is passed to memory for storage. The process of reading key information from a packet is known in the art. The present invention accommodates keys of various types. For example, keys for various protocols may be designated (IPV4, IPV6, etc.). The length of the key is user definable. In general, the key is derived from the header, but portions may also be derived from the payload (data field associated with the packet).

When the controller receives the notification information, it must determine a key type. In one implementation, a plurality of key types are defined. The user may define up to 4 types of keys, each having variable length. The key type can be defined by a two bit field in the header. A lookup of the two bit field is used to determine an appropriate starting hop (as described below).

Thereafter, an assigned route lookup engine 110 performs a lookup for the notification. The lookup can include a plurality of chained lookup operations, one of which can be a jtree search. A jtree (jtrie) is a data structure that is used to locate the best (longest) matching route for a given key. At the completion of the lookup, the route lookup engine returns a result which includes the output port associated with the destination. The result and other information (source ID, flow ID, packet length, quality of service and statistical information) for routing the packet through the router combine to form a result notification. The result notification is transferred from the controller 106 to the output switch 102. Upon receiving the result notification, the output switch 102 initiates the transfer of the packet from memory 104 to the respective output port 150 associated with the result.

In one implementation, the data structure for the result notification includes a destination mask, a next hop index pointer, full address, offsets and packet length. The destination mask is used to indicate which multi-function multiport connected to output switch 102 is to transfer the packet. In one implementation, the result notification may be sent to more than one multi-function multiport resulting in the broadcast of the associated packet. Associated with each multi-function multiport 150 is a storage 310. The next hop index pointer points to a location in storage (memory) 310. Storage 310 is used to store media header information associated with a particular type of packet transfer. Next hop addresses, media headers and storage 310 will be described in greater detail below in association with the output section of multi-function multiport 150. The full address indicates the starting address in the global data buffer where the

first cell in the packet is stored. As was described above, offsets provide linking information for retrieving cells or an indirect cell associated with the packet. The packet length indicates the length of the associated packet and may be used to determine if indirect cells will have to be retrieved.

b) Route Lookup Engine

Each route lookup engine performs packet (key) processing. Packet processing is the process of examining the contents of a packet header and performing functions such as route lookup, filtering, or flow policing based on the values of fields in the header. The result of packet processing determines how a packet should be forwarded in the router.

Referring now to FIG. 9, each route lookup engine 110 includes a key buffer 902, a result buffer 904, a key engine 905, one or more specialized engines for processing packets 906, a current key pointer 908 and starting hop table 910. In one implementation, each route lookup engine 110 includes a general purpose key engine 905 and plural specialized engines 906. The general purpose key engine 905 receives the key from the input switch, loads the key and result buffers, performs initializations, unloads the buffers and performs other operations in support of the lookup process. Specialized engines operate on instructions or data structures stored in memory 920 to perform a particular function. Functions can be selected from lookup operations, filtering, policing, management or other functions. In one implementation, the specialized engines can be selected from the group of a firewall engine 906a, a policing engine 906b, index engine 906c and trie search engine 906d. Each of these engines can be invoked to perform an operation and assist in determining a forwarding decision for a packet. As will be described below, more than one engine can be invoked to operate on each packet.

Key engine 905 stores the fields from a packet that have been selected to be part of the key for packet processing in key buffer 902. Any part of a packet can be selected to be part of the key, depending on the application. The key extraction process is completed in the input switch 102 as described above. The results of the key extraction process (the extracted key) and other information forms the notification that is passed to the controller 106. Any part of the notification can be extracted by the key engine 905 and written to the key buffer 902. A 'key' can consist of two parts. In one implementation, the first eight bytes of the key are constructed either from the contents of the notification for the packet, or built from intermediate results of

route lookups. The remaining bytes of the key, which are variable in length up to 41 bytes, are those extracted from the payload of the packet. In one implementation, key buffer 902 is a 64 byte buffer, physically located in the first 8 double words in the key engine's memory (not shown). Fixed data is stored in bytes 0-7 of the buffer while the variable key data is stored in bytes 8 and beyond.

The first 2 bytes of the key buffer are used as an overwrite area; various intermediate next hops may write data in this area to be accessed by subsequent lookup stages. These two bytes are initialized to zero. The first 4 bits of the second word are also used as an overwrite area for sampling indications. Sampling refers to a process of forwarding a copy of a packet to an external system for processing. A packet that is designated to be sampled is switched by the router as defined in the notification, but a copy of the packet (or portion of the packet) is created and forwarded to a system for further processing. The lookup process executed by the route lookup engine may include the designation of a packet for sampling. The further processing can include a management function that can be provided either on or, more typically, off the router. Packets can be designated to be forwarded to the management function for analysis. For example, a sampling of all of the packets that are from a particular source can be sent to the management function for further analysis. The sampling bits can be set to designate a packet as requiring further processing. In this way, when the (result) notification for the packet is processed, a copy of the packet (or portion of the packet) can be forwarded to the management function for further analysis. The sampling bits can be set and modified in the lookup process. As such, whether a particular packet is a candidate for a sampling operation can be decided based on a lookup result. In one implementation, the low order bit (sn[0]) is initialized to the value of an incoming sampling (SN) bit in the notification from the input switch, and the other three bits are initialized to zero. In this way, a device upstream from the controller (e.g., the input switch or the multifunction multiport) can designate some of the packets for sampling. In one implementation, the sampling bits are a mask for a predefined number of sampling operations. That is, the setting of a bit in the sample bits indicates a sampling operation to be performed. Alternatively, the sampling bits can be a set to indicate that a particular packet is a candidate for sampling. The decision as to whether or not the candidate is actually sampled can be made outside the lookup engine based on a programmable probability algorithm.

As various algorithms process the key, the key buffer can be used to communicate from one processing step to another. More specifically, the key buffer can include one or more rewrite bytes. The rewrite bytes form a data area that can be used by one processing step in a lookup (one lookup operation) to directly pass data to another processing step. In one implementation, key buffer 902 includes 2 rewrite bytes.

In one implementation, the format of the fixed and variable areas in the key buffer 902 is as shown in Table 1-1.

Table 1-1 Key Buffer Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
rewrite1								rewrite0								packet length														
sample sn[3:0]				TE	TCP	Q1	Q0	OP	0							0	incoming interface index													
variable length key extracted from packet.....																														
.....																														

Associated with key buffer 902 is the current key pointer 908 that points to the location in the key buffer 902 that should be treated as the beginning of the key in the current processing step. The current key pointer 908 can be moved from field to field in the key buffer 902 for different lookups that are chained together. By default, at the start of a lookup, the current key pointer points to the start of the variable length key. The current key pointer 908 is maintained by key engine 905.

The key engine 905 stores the result of packet processing which is information on how the packet should be forwarded in the result buffer 904. When multiple lookups are performed for the same packet, the result may be modified at the end of each lookup. The contents of the result buffer 904 at the end of the last lookup for a packet is the final result. In one implementation, the contents of the result buffer are as shown in Table 1-2.

Table 1-2 ***Result Buffer Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
next_hop_index																dest_mask									n	0	s_x	x	s_PR	PR

where:

- n = next hop set bit. If n is 1, some next hop has caused the next hop index and destination mask (dest_mask) fields to be set. If n is zero the next hop has not been set and the dest_mask is effectively zero.
- s_PR = priority set bit. Defaults to zero. If s_PR is 1, some next hop has caused the PR bits to be overwritten.
- PR = the priority bits. Set by default to the priority bits in the incoming notification. May be modified by next hops with the s_PR bit set.
- s_x = 'x' set bit. Defaults to zero. If s_x is 1, some next hop has caused the 'x' bit to be overwritten.
- x = 'extra' bit. Set by default to the P[2] bit in the incoming notification. May be modified by next hops with the s_x bit set.

Starting hop table 910 includes starting hops for each packet type. The starting hop table 920 can be shared by all of the route lookup engines 110 in controller 106. Entries in the starting hop table 920 can be of the form of a final next hop or an intermediate next hop. Next hop data structures and the processing of next hops are described in greater detail below.

c) Packet Processing

Packets are processed in accordance with a next hop instruction. A next hop is a data structure stored in memory that either contains the final result of packet processing for a packet or acts as a link to another lookup for the same packet. The key engine 905 receives a next hop instruction, and either processes the instruction directly, or invokes a specialized engine within the route lookup engine to process the next hop. A "final next hop" contains information on the final destination of the packet. An "intermediate next hop" is a link to the next lookup step and contains a command specifying the lookup algorithm type, a memory address pointer to the beginning of the lookup data structure, and an optional pointer offset for moving the current key

pointer to a different key buffer field. An "extended next hop" is a special type of intermediate next hop that contains a memory address pointer to a list of instructions (more than one next hop can be pointing to the same list). These instructions specify whether to modify certain fields in the key buffer and may place lookup results in the result buffer. The list can end with an intermediate next hop, a final next hop or without a next hop (where no next hop is specified, the lookup process ends and the current contents of the result buffer are used as the result). A "starting next hop" is the next hop specifying the first lookup step for a packet and is either of the form of a final next hop or an intermediate next hop. Starting next hops are stored in the starting hop table 910. The data structure for the next hops is described in greater detail below.

Referring now to FIGs. 9 and 10, a method 1000 for packet processing is shown. Packets are processed by the key engine 905 in four steps: initiate, lookup, link, and terminate.

In the initiate step 1002, the key buffer 902 is loaded with pre-selected fields from the notification 1004. The result buffer 904 and current key pointer 908 are initialized to a default values 1006. A pre-selected field from the packet is then used to index the starting hop table 910 1008. Different starting next hops correspond to independent packet processing paths. The protocol type of a packet can be used as the index for selecting the "starting next hop" so that each protocol can be processed in different ways.

Once the starting next hop is obtained, the link portion 1024 of the process begins. The link portion 1024 of the process includes three checks. Each of the checks evaluates a current next hop. By current next hop we refer to either the starting next hop, intermediate next hop returned from a lookup operation or a result returned after processing an extended next hop. First, a check is made to determine if the current next hop (the starting next hop, intermediate next hop or result) is an extended next hop 1020. If the next hop is an extended next hop, then the function specified by the extended next hop (e.g., policing, sampling, counting or other function) is executed 1022. The contents of the key buffer and the result buffer may be modified before the next lookup step is performed. By allowing the modification of the contents of the key buffer 902, a subsequent lookup operation can use the results from an earlier lookup step as part of its key. By allowing the modification of the contents of the result buffer 904, intermediate results can be stored. If not modified by a subsequent lookup step, the intermediate result, or some fields from it, may eventually form the final result. At the completion of the execution of the associated function and the modification of the buffers, a result is returned (1023). The result can be in the form of a next hop. Thereafter, the process continues back at step 1020.

In the second check of the link portion 1024, the current next hop is evaluated to determine if it is of the form of an intermediate next hop 1016. The check can be performed after the first check performed in step 1020 fails (i.e., the current next hop is not an extended next

hop). If the current next hop is an intermediate next hop, then the process continues at step 1018 where the current key pointer is set to the location specified by the intermediate next hop and a lookup is performed on the packet. The intermediate next hop acts as a link between two lookups. The intermediate next hop specifies the type of the next lookup (e.g., lookup engine 906 to invoke), the memory location of the lookup data structure (e.g. index table, jtree, firewall filter program), and the new location of the current key pointer 908.

After the current key pointer is set to the new location in step 1018 (as required), the lookup portion 1014 of the process is invoked. In one implementation, the lookup portion can include the invocation of one or more specialized engines in the route lookup engine 110. In the lookup portion 1014, the key (or portion of the key or other notification data) to be operated on is extracted from the key buffer (1010) and a specified lookup operation is executed (1012). The lookup operation may go on for any number of clock cycles and any number of memory references until a result, in the form of a next hop, is obtained 1012. Thereafter, the process continues at step 1020.

If the current next hop returned is not an intermediate next hop, the third check of the link process 1024 is invoked. More specifically, if the check in step 1016 determines that the current next hop is not an intermediate next hop, then a check is made to determine if the current next hop is a final next hop (1017). This completes the linking portion 1024.

After the check in step 1017 is performed the terminate portion 1030 of the process is invoked. In the terminate portion, packet processing for the current packet is terminated and a result is returned. More specifically, if the current next hop is not a final next hop the process continues at step 1028. If the current next hop is a final next hop, then the process continues at step 1026 where any information in the current next hop relating to the final destination of the packet can be copied into the appropriate fields in the result buffer (1026), as required. The contents of the result buffer is then unloaded (1028) and used to forward the current packet through the router. Thereafter the process ends.

d) Processing Algorithms

The basic routing in a IP (internet protocol) network is done based on a longest match lookup on a field of bits in the packet header. To enhance this routing and make it more intelligent, the system adds processing

features to be able to filter the packets based on some fields in the packet header. Enhancements have been added to allow some accounting ability and by also providing flow control based on a policing engine.

Packet processing includes processing the fields of a packet header, sometimes also known as a key, to perform the required functions like route lookup, filtering or flow policing. Key information for a packet can be processed using several different algorithms to generate a resultant notification (result) which is then used to forward the data packet appropriately.

In one implementation, three base algorithms can be selected from for packet processing and include an index table lookup, variable length best match lookup (i.e., a jtree lookup) and a firewall lookup. Each of the algorithms uses a next hop data structure to initiate processing. At the end of each processing step (after invoking an algorithm to operate on a designated portion of the key or executing a specified function) the result is also a data structure in the form of a next hop. Based on the variety of nexthops one can initiate new processing steps or end the overall packet processing function. The next hops thus form the primary data structure that can be used to initiate a lookup, chain different lookups and terminate a lookup. In addition, the next hop data structure also include provisions for supporting a variety of added features like packet counting, packet sampling and flow based policing of packets.

1) Index Engine

An index table lookup is performed by invoking the index engine 906c to extract the specified bits of index from the key and add the specified bits to a base pointer to compute the address of a next hop to read (see below for next hop data structure). Associated with the index engine 906c are one or more index tables 916 stored in a memory 920. Memory 920 includes the various data structures operated on by the various engines in the route lookup engine 110. In one implementation, memory 920 is separate from the route lookup engine 110 and can be accessed through a memory interface 911. In the implementation shown, memory 920 includes a single index table 916 but plural index tables can be included in memory 920. The key engine 905 reads a current next hop, and if the lookup calls for an index search, invokes the index engine 906c to perform an index lookup on a specified index table 916. An index search next hop includes a designator for the position in the key to use in the index search and the index table 916 to use.

In one implementation, an index table 916 includes a variable number of doublewords (at least one) and has a starting address located at a doubleword boundary. The first word of the index table 916 contains the table size and description of the location (relative to the current key pointer) and number of the key bits to be extracted as an index. The second word of the index table 916 holds the default next hop, which is read if the index exceeds the size of the index table 916. The base address of the index table 916, to which the index is added to compute an address to read if no overflow occurs, is implicitly the address of the third word (i.e. second doubleword) of the table. Each entry in an index table 916 includes a one word 'next hop'. The memory allocation for routes is rounded up to a doubleword boundary, but otherwise is the minimum necessary to hold the route information. An index table 916 can be defined to be a multiple of two words long in contiguous memory. In one implementation, the format of the start of an index table is shown in Table 1-3.

Table 1-3 ***Index Table Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
table size in doublewords																						dont care			i_off			idx_nbits		
default next hop																														
next hop 0																														
.....																														

where:

- the i_off field is a bit offset from the location of the current key pointer to the first bit of the index. The value can be set to 0-7 inclusive, which allows the start of the index to be anywhere in the byte the current key pointer points to.
- the idx_nbits field indicates the number of bits that should be extracted to form the index. Valid values are 0-22 inclusive. A value of 0 in this field causes the default next hop to be unconditionally read. A value larger than 22 is invalid and causes the notification to be discarded.
- The number of pairs of next hops in the table (exclusive of the default next hop) is indicated by the table size field. If the index extracted is index[21:0], the address of the (descriptor word of the) index table is table_address, and the table size in doublewords is table_descriptor[31:11], the address from which the next hop result is extracted is computed as:

```
if (index[21:1] >= table_description[31:11]) then
    next_hop_address = table_address + 1;
else
    next_hop_address = table_address + 2 + index.
```

2) Trie Search Engine (i.e., Variable length best match lookup)

If the current next hop indicates a tree based search is to be performed, the key engine 905 invokes the trie search engine 906d to perform a lookup operation that includes a longest match lookup traversing a radix trie data structure (referred to herein as a "jtree"). The search is based on the specified number of bits at a particular starting point in the key buffer. The process for performing the longest best match lookup is described in greater detail in copending application "Separation of Data and Control in a Switching Device". The result of the longest match lookup of the key bits is a next hop. More specifically, a route in a jtree consists of a one word 'next hop', at a double-word aligned memory location, followed by zero or more words of prefix information. One or more jtrees 914 are stored in memory 920. A next hop specifying a jtree search includes identifying information for the particular jtree to be searched. The storage of a jtree in memory 920 is described in greater detail in "Separation of Data and Control in a Switching Device".

3) Firewall Engine

The firewall engine 906a is an instruction-based, protocol-independent match engine which operates on data in the key buffer. When a next hop specifies a firewall lookup, the key engine 905 invokes the firewall engine 906a which in turn retrieves a firewall filter program from memory 920. The program includes a stream of one or more instructions that can be executed to perform the requested firewall services. Most instructions are 'match' instructions, that is, the instruction takes some data from the key buffer and matches it to the instruction's operands. A 'true' match causes the next instruction in the stream to be executed, a 'false' match causes a forward branch to be taken. A byte index to the data item in the key buffer to be matched, and an offset for the branch forward, is included in the instruction word. The match data is included in the operands.

The other instruction type is an 'action' instruction. Action instructions may carry some fields unique to the instruction in the remainder of the 32-bit word, but typically don't have

operands. Some action instructions terminate the search and indicate the result to return in the form of a next hop structure. The firewall engine 906a and firewall filter programs are described in greater detail below.

e) Next Hop

There are several 'next hop' formats. One of these is a "final" next hop, which terminates the search and contains a final result for the lookup. The others are "intermediate" next hops, which indicate how a further search should be performed.

1) Final Next Hop

In one implementation, bit 10 of the next hop is the 'final' bit. If the final bit is set, the next hop is in 'final' format, otherwise it is one of the 'intermediate' formats. In one implementation, the format for a final hop is as shown in Table 1-4.

Table 1-4 ***Final Next Hop Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
next hop index																x	a	s	PR		f	m	dest_mask							

where:

- f = final bit. set to 1 for final next hops, set to 0 otherwise
- m = multicast bit. If set, and if this next hop is attached to a route in a jtree, a 14-bit 'interface index' is appended to the route prefix data which must be matched against the incoming interface index in the key buffer. If there is a mismatch the packet is discarded. The m-bit is ignored in index table lookups.
- a = accounting bit. If set, and if this next hop is attached to a route in a jtree, a three word packet+byte counter pair is appended to the prefix (and multicast interface index, if present) data. These counters must be incremented to account for packets and bytes matching this route. This bit is ignored in index table lookups.
- s = set bit for the x and PR bits. If 's' is 1, x and PR are copied into the corresponding fields in the result buffer. If 's' is 0, x and PR in the result buffer remain unchanged. The outgoing notification will have the final values of x and PR in the result buffer. Note that {x, PR} in the result buffer default to the incoming priority bits {P[2], P[1:0]}, so the outgoing notification will have the incoming priority bits if no next hop modifies them.

- PR = new priority bits. If 's' is 1, the corresponding field in the result buffer will be overwritten by PR. As a result, the outgoing notification will have these new priority bits. The priority bits are not changed if 's' is 0.
- x = new 'extra' bit. If 's' is 1, the corresponding field in the result buffer will be overwritten by 'x'. As a result, the outgoing notification will have this new 'x' bit. 'X' bit is not changed if 's' is 0. The hardware picks up this 'x' bit and forwards it to the output port.
- the destination mask (dest_mask) field is a 9-bit mask indicating where the packet should be forwarded to. Bits 0 through 7, when set, indicate that the packet should be sent to physical banks 0 through 7, inclusive. When bit 8 is set the packet is locally destined.
- the 'next hop index' is a 16-bit value used by other parts of the forwarding system to determine outgoing processing of the packet.

The execution of the final next hop causes the destination mask and next hop index fields to be copied into the result buffer. The lookup process then ends and key engine 905 uploads the latest results in the result buffer including forming an outgoing notification that includes the results. In one implementation, packet filtering (filtering or policing) can be performed based on the data included in the final next hop. For example, in one implementation, if the destination mask in the result buffer is set to a value of 9'h0 at the end of a lookup, the packet is discarded and a discard counter (CF_DBR_CNTR) is incremented. Final next hops cause the lookup to end, so final next hops with dest_mask=9'h0 will always cause the packet to be dropped.

2) Intermediate Next Hops

All other next hops are 'intermediate format' next hops (hereinafter, intermediate next hop). Intermediate next hops can be chained to allow for plural lookup operations to be performed on a single packet. For example, a packet can be filtered, subject to a jtree lookup to determine a route, then subsequently filtered again prior to determining a final routing decision. The three lookup operations (filter, jtree search, filter) can be specified by chaining next hops. The process for chaining next hops is described in greater detail below. One implementation for the format of an intermediate next hop is as shown in Table 1-5.

Table 1-5 ***Intermediate Next Hop Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
																					0		Fid							

where

fid is the format identifier (ID) for the intermediate next hop. In one implementation, there are 5 fid's including:

- 0 - jtree lookup
- 1 - firewall/index table lookup
- 2 - jtree/index table ID lookup
- 3 - multiple next hops
- 4 - extended next hop

In one implementation, if an undefined intermediate next hop (e.g., fid=3'h5, 3'h6, 3'h7) is encountered, the lookup will end immediately, the packet is dropped, and a discard counter (CF_DBSFT_CNTR counter) is incremented.

a) Jtree Lookup Intermediate Next Hop (fid=0)

In one implementation, a jtree lookup has an intermediate next hop format as is shown in Table 1-6.

Table 1-6 ***Jtree Lookup Intermediate next hop format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
8-byte double-word memory address																					0	R	000			byte_offset				

where:

- bits [31:6] of the next hop, with bits [5:0] hardwired to zero, is a tree pointer to the root of a jtree to be searched.
- the byte_offset is the offset to change the current key pointer by. The value of the byte_offset field is added to the current key pointer modulus 64. That is, a byte_offset of 0 will keep the current key pointer unchanged, a byte_offset of 1 would move the current key pointer to the right (i.e. forward) one byte in the key, and a byte_offset of 63 would move the current key pointer one byte left

(i.e. backward) in the key. If the current key pointer is moved to a location beyond the end of the key delivered from the input switch, an error occurs (equivalent to a truncated key error when doing tree lookups). The packet is dropped, and a discard counter (CF_DTK_CNTR discard counter) is incremented.

b) Firewall / Index Table Lookup Intermediate Next Hop (fid=1)

In one implementation, a firewall/index table lookup has a next hop format as is shown in Table 1-7.

Table 1-7 ***Firewall / Index Table Intermediate Next Hop Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
8-byte double-word memory address																						0	i	001		byte_offset				

where:

- i = index table bit. If this bit is 1, the next lookup to be done is an index table lookup. If this bit is 0 the next lookup to be done is a firewall filter program.
- bits [31:10] of the next hop is the word address of the first word of the firewall filter program or index table, depending on the setting of the 'i' bit.
- the byte_offset is the offset to change the current key pointer by.

c) Jtree / Index Table ID Lookup (fid=2)

In one implementation, a jtree/index table ID lookup has a next hop format as is shown in Table 1-8.

Table 1-8 ***Jtree / Index Table ID Intermediate Next Hop Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
rewrite byte								rb	o	S_x	X	s_PR	PR		Byte_offset							0	i	010		table_id				

where:

- i = index table lookup. If the 'i' bit is set to 1, the lookup that is done next is an index table lookup. The table_id refers to the entry in the 64-entry on-chip index table directory from which to get the index table descriptor and start the lookup. If the 'i' bit is set to zero, the lookup that is done next is a jtree lookup. The table_id refers to the on-chip jump (for table_id 0 or 1 and jump table enabled) or tid table (table_id 2 to 63 or jump table not enabled) entry from which to start the lookup.

- table_id is a 6 bit index. If the 'i' bit is set to 1, table_id is the index table descriptor pointing to the index table into which the search will be continued. If the 'i' bit is set to zero, table_id is the jtree root pointer pointing to the tree into which the search will be continued.
- byte_offset is the offset to the current key pointer (i.e. as in the fid=0 intermedaite next hop above).
- s_x = set bit for the 'x' bit. If s_x is 1, 'x' is copied into the corresponding field in the result buffer. If s_x is 0, 'x' in the result buffer remains unchanged.
- x = extra bit. Copied into the corresponding field in the result buffer if s_x is set.
- s_PR = set bit for the PR bits. If s_PR is 1, PR is copied into the corresponding field in the result buffer. If s_PR is 0, PR in the result buffer remains unchanged.
- PR = the priority bits. Copied into the corresponding field in the result buffer if s_PR is set.
- o = set bit for the rewrite field. If 'o' is 1, the rewrite byte in location [31:24] is copied into one of the first 2 bytes of the key, with the rb (i.e. rewrite byte) field specifying which byte is rewritten.
- rb = specifies whether rewrite byte 0 (rb=0) or rewrite byte 1 (rb=1) should be overwritten if the 'o' bit is set.

The rewrite, x, and PR updates take effect before the jtree or index table ID lookup begin. For example, if {rb, o} = {1, 1}, and byte_offset moves the current key pointer to point to rewrite1, the key buffer is updated with the new rewrite byte in the next hop, and then the fid=2 lookup is performed using the new rewrite byte value as the key.

d) Multiple Next Hop Intermediate Next Hop (fid=3)

A multiple next hop can be used to perform load balancing operations. Multiple next hops can be designated to alleviate loading conditions for any particular hop. In one implementation, the multiple next hops are listed, and a selection can be made based on loading or other considerations. In one implementation, the router can be configured for a balanced, unbalanced or incoming interface mode of operation.

In balanced mode, the particular next hop is randomly selected from the designated group of next hop candidates. In one implementation, the selection is made by taking a hash of the key

and based on the hashed value, assigning the packet to a particular one of the candidate next hops. Alternatively, the selection can be made by selecting a random number or by pseudo random means that take into account history or other considerations in making the determination.

In unbalanced mode, weights can be assigned to individual candidate next hops, and a weight affected decision can be made to select a next hop from the candidate next hops. The weights can be assigned to support certain quality of service or priority service considerations.

The incoming interface mode can be used to screen packets depending on the type of next hop being processed. A multiple next hop includes a list that specifies candidate hops. The incoming interface mode can be used to select among the candidates. More specifically, if the incoming interface designator for a packet matches the incoming interface designator associated with a determined route, a first next hop in the multiple next hop list is selected as the next hop to be used in forwarding the packet. If the designators do not match, the second entry in the list can be selected as the next hop to be used in forwarding the packet. For all other next hop types, if the match bit (m-bit described below) is set and no match is detected, then the packet can be dropped and an appropriate discard counter can be set.

One implementation of a multiple next hop format is as shown in Table 1-9.

Table 1-9 ***Multiple Next Hop Intermediate Next Hop Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
8-byte double-word memory address																					0	m	011				a	h	#hops	

where:

- m = multicast bit. If set, and if this next hop is attached to a route in a jtree, a 14-bit 'interface index' is appended to the route prefix data which must be matched against the incoming interface index in the key buffer. If there is a mismatch the packet is discarded unless the 'h' bit is set (see below). The m-bit is ignored in index table lookups.
- a = accounting bit. If set, and if this next hop is attached to a route in a jtree, a three word packet+byte counter pair is appended to the prefix (and multicast interface index, if present) data. These counters must be incremented to account for packets and bytes matching this route. The a-bit is ignored in index table lookups.

- bits [31:10] of the next hop form the memory address of the first entry in a list of next hops for this route, any of which may be used to forward the packet.
- #hops has a value between 1 and 15 inclusive, or 0. If the value is 0, the number of next hops is 16, otherwise the number of next hops is #hops. If #hops is 1, there is only 1 next hop, so the hardware will always choose the same next hop, regardless of the mode or the hash value.
- h = hash bit. Along with the 'm' bit, indicates how the hash value is reduced to the index of a next hop in the list. The modes can be selected from a balanced mode (m=X and h=0), an unbalanced mode (m=0 and h=1) and an incoming interface mode (iif mode) (m=1 and h=1). If the multiple next hop is read from a data structure that does not support multicast, the m-bit is interpreted as 0 even if it is set. For example, if a multiple next hop with {m, h} = {1, 1} is read from an index table, unbalanced mode processing is performed, not iif mode. In the iif mode, there must be at least 2 next hops in the multiple next hops list.

e) Extended Next Hop Intermediate Next Hop (fid=4)

The extended next hop format allows the lookup process to implement one or more functions. Associated with an extended next hop is a extended next hop list that includes one or more functions to be executed. The functions can include policing operations. One implementation for an extended next hop format is as shown in Table 1-10.

Table 1-10 ***Extended Next Hop Intermediate Next Hop Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8-byte double-word memory address																					0	m	100			a	CA	I	CI	O	F

where:

- m = multicast bit. If set, and if this next hop is attached to a route in a jtree, a 14-bit 'interface index' is appended to the route prefix data which must be matched against the incoming interface index in the key buffer. If there is a mismatch the packet is discarded. The m-bit is ignored in index table lookups.
- a = accounting bit. If set, and if this next hop is attached to a route in a jtree, a three word packet+byte counter pair is appended to the prefix (and multicast interface index, if present) data. These counters must be incremented to account for packets and bytes matching this route. The a-bit is ignored in index table lookups.
- bits [31:10] of the next hop form the memory address of the first entry in the extended next hop list. The F, O, CI, I, and CA bits determine what is located in the extended next hop list.

- F = modified final bit. If this bit is set, the word pointed to by the address contains a modified final next hop, which causes results to be set into the result register.
- O = overwrite bit. If this bit is set, the extended next hop list includes an overwrite word, which includes data to be inserted in the rewrite bytes in the key, as well as settings for the precedence bits. The overwrite next hop is included after the modified final next hop, or at the start of the list if the F bit is clear.
- CI = counter indirect bit. If set, the next hop list includes a 32-bit counter pointer. This pointer contains information about the location of a counter and its type. If present this pointer is located after the modified final and overwrite words.
- I = intermediate bit. If set, the last word entry in the list is an intermediate next hop of one of the formats described previously. If no intermediate next hop is included in the list the lookup terminates. If the I-bit is not set, the lookup terminates immediately and the outgoing notification is built based on the current results in the result buffer. Software can terminate a lookup using an extended next hop with the I-bit not set or even none of the F, O, CI, I, CA bits set. In one implementation, the I-bit can be set and a final next hop can be put in the extended next hop list. Note that the modified final next hop is for placing a lookup result in result buffer to be used when the lookup ends. For example, a result can be put in result buffer before an output filter program is run. If the packet is not dropped by firewall filter, the lookup result stored in the result buffer will be used.
- CA = counter attached bit. If set, the next hop list includes an attached, standard, byte-and-packet counter. The counter is attached to the end of the list, possibly with a single pad word to align the counter to an odd-word boundary.

f) Policing

In networking systems, there is a need to be able to count the data bits in a flow (flow is a logical unit of data transmitted for one place to another). Additionally it is also useful to be able to measure the rate of the data bits (in say bits per second) and further be able to shape the data flow as per requirements. The data flows can be shaped to be capped at a maximum bandwidth usage or otherwise restricted (e.g., after a bandwidth usage has been achieved, reduce the priority of service for the flow).

A number of terms are used in this section and are defined as follows. "Data Rate" as used herein refers to the measure of data transferred in a specified duration of time and is

typically measured in bits per second (bps, or Kbps, Mbps, Gbps). "Accounting" as used herein refers to the ability to count the number of bits per flow. "Rate Measure" as used herein refers to the ability to measure the data rate of a flow. "Policing" as used herein refers to the ability to measure the rate and then shape it to a specified threshold.

In conventional systems, policing operations include two steps: measurement and flow policing. For a stream of data, a conventional system needed to have a counter which keeps a count of the number of bits of data in that stream that have been transferred. Additionally, a timer was needed which was used to set a data sampling interval. Given the number of data bits and a time reference, say 100 microseconds, a conventional system could count the number of data bits in that time and multiply by, say 10000, to get a bits per second data rate measure for the stream. If there are thousands of such streams monitored by the system, a large memory may be required to store the count information for the data bits.

In the monitoring step, for each chunk of data per stream, the system would read the counter and increment it with the number of bits in a packet. So for each packet transferred for a stream, the system would need to read the memory once and write to it once. Additionally if during this continuous process of counting the bits
per stream, the system needed to measure the data rate for individually policing each flow, then an additional read of a threshold-count value is required every time the data counter is updated. Furthermore, this counter ideally is cleared every time interval so that the rate can be measured over a next time interval. Thus for rate measurements to make policing decisions for a flow, an additional read (every counter update) and a write (every time interval) to memory may be required. Assuming a time interval of 100 microseconds and have 10000 streams, then there is a fixed overhead of 100 million writes to memory per second required in such a system. The actual number of reads and writes to memory for counting the data bits per stream and the read for the threshold-count value varies with the number of packets/cells transferred in a time interval for each particular stream.

In addition, the time interval and the total number of streams determine the fixed overhead required for the computations. The smaller (finer) the time interval the more accurate the rate measurement would be however the proportion of fixed overhead bandwidth also increases. The overall memory bandwidth requirement is proportional to the fixed overhead and

the rate of packet/cell transfers in the system. The drawbacks of this type of measurement technique is that one needs a significant memory bandwidth overhead. This requirement grows as the time interval reduces and the number of streams increases. The number of accesses required to count the data bits per packet/cell for each stream depends on the number of packets/cells transferred. This is very difficult to predict and thus any technique used for data rate computations will need to adapt to this need.

In the present system, a new approach is proposed. The approach is based on the observation that for policing a flow a system needs to make a memory reference to update the counts every time a packet/cell flows thru the device (i.e. router, switch). In a relatively busy system, where memory accesses are precious, the assumption is that in a set time interval there is a very high probability that each stream would have at least one packet/cell transfer per unit time interval. In such an environment the goal is to be able to compute the data bits, data rate and make a decision on policing each flow in a burst read and write operation to memory without the need for any global overhead to clear all the counters each time interval. This can be accomplished by saving (in memory) the last access time (referred to herein as last adjustment time) together with the data bit count for every flow. The policing decision can be made by computing how many bits of data are allowed in the time interval between the arrival of two packets belonging to the same flow, triggering access to the same counter.

More specifically, a policing function can be implemented as part of a lookup chain by designating an intermediate next hop that includes an extended next hop format having a list that includes a counter function. The counter can be updated by the policing engine as described below. Depending on the updates, policing decisions can be made on a packet by packet basis. A single burst read operation can be performed to read all of the data required to make the policing decision [the threshold count value (credit_limit), the data counter value (current_credit), the last time updated (last_adjustment_time), the amount to be updated per unit time (time_credit)].

The last time updated reflects the time of arrival of the last packet received for a given flow. A single burst write operation is performed to update the counter with a new count value and the time of arrival for the last packet processed. In the present system, the counters are not

updated at a fixed time interval, and instead are updated only when traffic is received in the given flow. The burst operations result in a large savings on memory bandwidth in the system.

In one implementation, the format for a policing counter for use in an extended next hop is shown in Table 1-11.

Table 1-11 ***Policing Counter Format***

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
(double word aligned address) unused pad word																															
out-of-spec packet counter																															
credit_limit																current_credit															
time_credit																last-adjustment_time															

where:

- last_adjustment_time indicates the last time this policing counter was incremented. The timestamp is referenced to one of two on-chip global time counters, with the particular counter selected by the setting of the 'r' bit in the counter pointer data structure pointed to by an extended next hop. In one implementation, the high rate policing counter increments once every 8.192 us, the low rate every 262.144 us.
- time_credit indicates the amount of credit this counter receives per time increment. The units are as specified in the counter-pointer units field.
- credit_limit indicates the limit to which the current_credit field is allowed to increase. Units are as specified in the counter-pointer units field, times 16.
- current_credit indicates the amount of credit currently accumulated.

An algorithm for updating the counter is as follows. The algorithm can be executed by the policing engine 906b in the route lookup engine 110. The policing engine 906b receives packet length from the notification, and rate and units information from the key engine 905. The packet length is first adjusted by rounding to units specified in the counter pointer data structure pointed to by an extended next hop, i.e.

```
if (units == 2'b00) {
    adj_packet_length = packet_length;
} else if (units == 2'b01) {
    adj_packet_length = (packet_length + 2) >> 2;
} else if (units == 2'b10) {
    adj_packet_length = (packet_length + 8) >> 4;
```

```
} else {
    adj_packet_length = (packet_length + 32) >> 6;
}
```

The adjustment can be performed before the first word of the counter is read. When the first word of the counter is read, then the new credit is computed based on the last adjustment time and the current time, where the latter is selected by the rate bit. The new credit computation is defined by: new_credit = time_credit * (current_time - last_adjustment_time). In this implementation, the new credit value gets the lower 18 bits of the result of the multiplication. If the upper 14 bits of the result are non-zero then the notification is considered to be within specification and the current credit value is updated with the credit limit rather than the equation below. If the upper 14 bits are zero then the computations below should be performed to check if the notification will be out of specification and how the data structure needs to be updated.

When the second word arrives the boolean policing result can be computed as: out_of_spec = (adj_packet_length > (current_credit + new_credit)); where the add must maintain 19 bit precision. The values written back to the locations in the counter are:

```
time_credit = time_credit;
last_adjustment_time = current_time;
credit_limit = credit_limit;
if (out_of_spec) {
    temp = current_credit + new_credit;
} else {
    temp = current_credit + new_credit - adj_packet_length;
}
current_credit = min({credit_limit, 4'h0}, temp); where any out_of_spec
result is returned to the key engine for packet disposal.
```

g) Firewall Filter

The firewall engine provides a filter by running an instruction engine over a portion of data extracted from the key buffer. For IP packets this data can include interface class, IP option flags, incoming interface, fragment offset, destination address, source address, protocol, source port, destination port and tcp flags.

A filter program includes a stream of instructions. The filter program is produced by a user and subsequently compiled to form executable instructions that are stored in a memory (i.e.,

memory 920 of FIG. 9). The filter is executed by a firewall engine (i.e., firewall engine 906a) in accordance with the processing of a next hop. In one implementation, the user can create a filter program using two basic types of instructions: match instructions and action instructions. Each basic instruction has a value to compare to a data quantity, and occasionally a mask (defaults to 0) to bit-fiddle the data before comparison. The match instructions are described in greater detail below. A branch can be taken when the comparison is true or false.

Each (logical) interface family (i.e., incoming interface in the routing device) may (or may not) have a filter program designated for incoming packets, a separate filter program for outgoing packets, and a configurable `interface class' byte for use in choosing output filters based on the incoming interface for a packet.

The structure of a filter program is as follows. Each filter program includes a set of rules. Each rule has a set of zero or more match conditions, and an action which is taken if all match conditions are true. Rules are logically executed in the order they appear in the filter, with the action of the first matching rule being executed.

An example of an IP packet filter produced by a user is shown in Table 1-12 below.

```
filter 3 ip {
    rule 5 {
        protocol udp,50-82 ;
        action count accept ;
    }
    rule 10 {
        protocol tcp,udp,50-82 ;
        source-port 7-64,512-777 ;
        destination-port 121-153 ;
        options lsrr|ssrr ;
        from 192.168/17 ;
        to 128.100/16 ;
        tcp-flags (syn & !ack)        # same as `setup'
                | (rst | ack) ;# same as `established'
    fragment-flags (df | mf) ;    # also `dont-fragment'
                                  #   and `more-fragments'
    fragment-offset 1-8191 ;    # same as `fragment'
    action count log reject host ;
}
rule 20 {
    protocol icmp ;
```

```
    icmptype 1-52 ;
    action discard ;
}
rule 30 {
    action accept count ;
}
}
```

Table 1-12 *IP Filter*

Rule numbering is provided to allow the user to identify either a rule itself (for modifications), or its relative ordering with respect to other rules (for additions).

In one implementation, matches can be of plural kinds. Matches on numeric fields (e.g. protocol, port numbers, fragment offset, icmp type) can specify a separate list of numeric ranges. Matches on bit fields (e.g. tcpflags, options, fragment-flags) can specify a set of bits combined with boolean operators. Matches on addresses are used to match prefixes. Some match conditions can have abbreviations (e.g. `established' for `tcp-flags rst|ack').

The result of a successful match is to take the `action'. The action may be one of `accept', `discard' or `reject', with modifiers`count' and/or `log'. `Count' counts the number of matches for a given rule, while `log' can be used to sample a packet for further processing. The log action can include the setting of sample bits in the result buffer. The setting of sample bits can result in the copying of a packet to an external processor for evaluation in addition to the routing of the packet to its intended destination. Sampling is described in greater detail above.

Each filter program is provided to a compiler. The compiler can be separate from the route lookup engine. The compiler generates the final instructions which can be executed by the firewall engine. The firewall engine can be implemented in a software or hardware engine.

The compiler compiles each rule in the order it appears in the filter program, but can be configured to change the ordering of match condition evaluations to suit its own purposes. The compiler evaluates the match conditions one by one in the order it has decided upon, branching forward into another rule when a match evaluates false. When a match is found false, however, the filter engine knows that (1) all prior match conditions in the rule were true, while (2) the failing match condition was false. The compiler can reorganize the match order to make use of

this type of information. For example, the compiler can skip over any immediately subsequent rules that cannot match (i.e. if `protocol tcp;' fails, all subsequent rules with`protocol tcp;' will also fail and may be skipped) and as far into the list of match conditions to a first feasible subsequent rule. The compiler can then remove dead match conditions before compiling to instruction code. This skip over feature allows the compiler to generate better instruction variants. For example, a set of rules with identical match conditions except for the source prefix, for example, can be compiled into a single evaluation of the common match conditions plus a tree lookup on the source address.

1) Instructions

As described above, the firewall engine is an instruction-based, protocol-independent match engine inside each route lookup engine that operates on the data in the 64-byte key buffer. In one implementation, each 'instruction' is a 4-byte quantity followed by zero or more 4-byte operands. Most instructions are 'match' instructions, that is they take some data from the key buffer and match it to their operands. A 'true' match causes the next instruction in the stream to be executed, a 'false' match causes a forward branch to be taken. A byte index to the data item in the key buffer to be matched, and an offset for the branch forward, is included in the instruction word. The match data is included in the operands. The other instruction type is an 'action' instruction. Action instructions may carry some fields unique to the instruction in the remainder of the 32-bit word, but usually don't include operands. Some action instructions terminate the search and indicate the result to return.

The firewall engine includes a current data pointer (not shown) that is a byte pointer pointing to the data in the 64-byte key buffer to be operated on by the firewall engine. The current data pointer is independent of the current key pointer used outside of firewall programs. In fact, the current key pointer remains at a fixed location during firewall program processing. When a firewall intermediate next hop (fid=1, i=0) is encountered and control is passed to the firewall engine for executing the indicated filter program, the current data pointer is initialized to the value of the current key pointer (current key pointer value is the value after adding byte offset specified in the intermediate next hop). The current data pointer is updated every time an instruction containing the data offset field is executed. The data offset is an absolute byte

address, not an offset from the current value. This means that the current data pointer simply takes on the value of the data offset.

The data offset points to the data in the key buffer to be used for the current instruction (in the filter program). It is the byte offset from the beginning of the key buffer, not an offset from the current data pointer value. For instructions that do not have the data offset field, the current data offset, i.e. the current data pointer value is used as the data offset. In one implementation, the branch offsets in firewall instructions are self-referential word offsets. The firewall engine includes a program counter which keeps track of the 22-bit word address of the current instruction word being executed. The next instruction to be executed after a branch is set to be: the program counter (22-bit word addr) + 1 + branch offset (word offset, variable number of bits). By expressing all memory references in terms of an offset relative to the previous location read, the firewall program can be independent of the position into which it is placed in memory.

In one implementation, there are eight 'match' operations supported. The operation (op) field contains the unique operation code (opcode) indicating the type of match operation to be performed. In one implementations, each match operation is performed on 1-, 2- or 4-byte-wide data. The operation expects to operate on 'data' (that is data extracted at some offset into the key buffer), a 'value' (a value extracted from the instruction) and a 'mask' (a second value sometimes included with the instruction where the default is all-zeroes). In one implementation, the eight operations include: an equivalence operation, a greater than operation, exclusive "or", exclusive "and", non-equivalence operation, less than, exclusive "nor " and exclusive "nand" operations.

a) One-Byte Match Instructions

One-Byte Match with Mask

Byte-wide instructions are identified by a '01' in the low order bits. The first of these carries both data and a mask, but uses the current data offset.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
byte value								byte mask								branch offset (10-bit)										op			1	01

where 'value' and 'mask' are included in the instruction and 'data' is the byte of data at the current data offset.

One-Byte Match without Mask

This second form of operation explicitly sets the data offset, but uses a default mask.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
Byte value								branch offset (12-bit)												data offset						op			0	01

where 'value' is extracted from the instruction and 'mask' is all zeroes (by default). The current data offset is set from the instruction, and 'data' is extracted from the byte at that offset.

b) Two-byte Match Instructions

Two-Byte Match with Mask

In a two byte match operation, the first byte match sets the data offset and both a 'value' and 'mask'. Note that the branch offset is actually an offset from the word after the second instruction word.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
branch offset (20-bit)																				data offset						op			1	10
short value																short mask														

Two-Byte Match without Mask

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
short value																branch offset (10-bit)										op			0	10

c) Four-byte Match Instructions

Finally, there are 4 byte instructions. For these the 'value' and 'mask' words, if any, are always appended in subsequent instructions.

Four-Byte Match without Mask

In this format, the branch offset is actually an offset from the word after the second instruction word.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
branch offset (20-bit)																				data offset						op			0	11
long value																														

Four-Byte Match with Mask

Here the branch offset is actually an offset from the word after the third instruction word.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
branch offset (20-bit)																				data offset						op			1	11
long value																														
long mask																														

h) Action Instructions

The remaining instructions do not include a numeric match (as required by each of the matching instructions set forth above). As such, they are special instructions in the firewall instruction set.

Long Branch Instruction Format

The canonical format for a long branch instruction is:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
branch offset (22-bit)																						data offset						0	0	00

This instruction sets the data offset to that specified, and fetches the next instruction from 'branch offset' words past the next word. Note that setting the 'branch offset' to zero provides a method to reset the 'data offset' without branching.

Termination Action Instructions

There are 4 different regular termination instructions. They are distinguished by a field (the "res" field) in the instruction. The format for a regular termination instruction is:

- **res=2'b00** - Discard notification. The lookup is terminated immediately and the packet is discarded. The CF_DBR_CNTR (a 32-bit discard counter) is incremented. (If the corresponding discard diagnostic bit is set, a copy of the notification is sent to the host.) Note that the packet is discarded regardless of whether the OP or TE (TE enabled) bit is set.
- **res=2'b01** - Send to Host. The lookup is terminated immediately and a notification is sent to the host by setting the destination mask (dest_mask=9'h100). The next_hop_index in the outgoing notification is set to the value of the next_hop_index field in the termination instruction. PR[1:0] is set to 2'b00 (i.e. low priority) regardless of its current value in the result buffer. The x field is not changed, i.e. goes out with its current result buffer value. The {rewrite, rb, o} fields in the instruction are used to modify the corresponding rewrite1 field in the key buffer which gets copied into the outgoing notification. The res=2'b01 termination can be used as a firewall "reject" instruction and the rewrite1 byte can be the "reject code".
- **res=2'b10** - Next lookup is a jtree lookup (fid=2, i=0) in the jtree specified by table_id. Byte_offset is used to set the current key pointer for the next lookup. The {s_x, x} and {s_PR, PR} fields are used to modify the corresponding priority bits in the result register.
- **res=2'b11** - Next lookup is an index table lookup (fid=2, i=1) in the table specified by table_id. Byte_offset is used to set the current key pointer for the next lookup. The {s_x, x} and {s_PR, PR} fields are used to modify the corresponding priority bits in the result register.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
reserved																										res		01		00

Regular Termination res=2'b00 Instruction Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
next_hop_index																rewrite								rb	o	res		01		00

Regular Termination res=2'b01 Instruction Format

where,

- o = set bit for the rewrite field. If 'o' is 1, the rewrite byte in location [15:8] is copied into one of the first 2 bytes in the key buffer, with the rb (i.e. rewrite byte) field specifying which byte is rewritten.
- rb = specifies whether rewrite byte 0 (rb=0) or rewrite byte 1 (rb=1) should be overwritten if the 'o' bit is set.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
rewrite1								s_x	x	s_PR	PR		o1	byte_offset						table_id						res		01		00

Regular Termination res=2'b10, 2'b11 Instruction Format

- o1 = set bit for the rewrite1 field. If 'o1' is 1, the rewrite1 byte in location [31:24] is copied into the first byte in the key buffer.

Extended Next Hop Termination Instruction Format

An extended next hop termination instruction can invoke an extended next hop instruction set as described above. The branch offset to an extended next hop is used to compute the address of an extended next hop list. The extended next hop list is located at: program counter (22-bit word addr) + 1 + branch offset to extended next hop list (22-bit word offset). Note that an extended next hop list linked to a firewall instruction may not be double word aligned because the branch offset is a word offset. The CA, I, CI, O, and F bits are as in the jtree/index table extended next hop format. The extended next hop termination with the I-bit not set can be used as a firewall "accept" instruction. The outgoing notification will be built from the

current contents of the result buffer. The result buffer should already have picked up the final next_hop_index and dest_mask from a route encountered before entering the output firewall filter. Note that an "accept"instruction may simply be an extended next hop termination with none of the 5 control bits set. If the I-bit is set, and the corresponding next hop in the extended next hop list is a final next hop, the result_buffer is updated and the lookup terminates normally.

An extended next hop termination instruction looks like:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
branch offset to extended next hop list (22-bit)																						0	CA	I	CI	O	F	10		00

Tree Search Action Instructions

A tree search action instruction invokes a longest match operation on the field in the key buffer pointed to by the data offset using a jtree stored in memory 920. A tree search instruction can have the following form:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	10
branch offset to jtree (11-bit)											r	R	bit to test									data offset							11	00

Data offset sets the current data pointer prior to starting the tree search. The instruction word immediately following the word indicated by the current data pointer (can be any firewall instruction) is executed if no match is found during the tree lookup. No match here means the lookup falls off the top of the stack, i.e. stack underflow. The remaining fields are used to form a jtree pointer which points at the next tree node at: program counter (22-bit word addr) + 1 + branch offset to jtree (11-bit word offset). The jtree lookup performed is identical to a 'normal' jtree lookup, with the following exceptions. The 21-bit '8-byte double-word memory address' in any jtree pointer is a branch offset rather than an absolute address. This keeps firewall filter programs independent of memory location. The address of the next tree node is calculated like this: program counter (22-bit word addr) + 1 + branch offset (21-bit word offset). Note that since the tree search instruction and jtree pointers in firewall programs have branch offsets that are word offsets, tree nodes in firewall jtrees may not be double word aligned. In one

implementation, multicast iif comparisons and route accounting is not done. The 'next hop' in a firewall jtree must be one of a long branch instruction, a regular termination instruction (any of the 4 **res** types) or an extended next hop termination instruction. The 'next hop' is restricted to these instructions because the word in memory immediately following it is the prefix. If the 'next hop' is not one of the allowed instructions, the packet is discarded as a bad software discard, and CF_DBSFT_CNTR (a 16-bit discard counter) is incremented. If a corresponding discard diagnostic bit is set, a copy of the notification is sent to the host.

Tree Search Fail Instruction

The tree search fail instruction allows firewall jtrees to have prefixes whose next hop means 'this search failed, execute the instruction after the tree search instruction'. In one implementation, the fail instruction is actually a tree search instruction. In this implementation, the tree search instruction found in a firewall tree search is interpreted as a fail instruction.

h) Example of a Chained Lookup

As described above, lookups can be chained to allow for a combination of filtering and lookup operations to be invoked for each packet. An example of a chained lookup is: Index Table Lookup -> Firewall Filter -> Jtree Lookup -> Firewall Filter. This sequence corresponds to index table lookup that is used to index a longest prefix match route lookup (jtree) with firewall filtering applied both before and after the jtree lookup. As described above, the data structures stored in memory in the lookup engine 110 include a table of "starting next hops". The starting next hop can be used to point to the first lookup operation (e.g., the index table lookup). "Intermediate Next Hops" are used to point to the subsequent lookup operations (to the first firewall filter program, one or more Jtrees for longest match prefix route lookup and one or more different firewall filter programs). The intermediate next hops are the result returned from each intermediate lookup operation.

The example above begins with a starting next hop of index table lookup type. That is, the index engine 906c is invoked to perform a lookup in an indicated index table 916. The index step points to an intermediate next hop that chooses an input firewall filter for the packet. In this example, the firewall filter could be selected based on the "incoming interface index". That is, the pointer offset field in the starting next hop moves the current key pointer to the incoming interface index field of the key. This index is used to look up the table in memory pointing to the different firewall filter programs.

The firewall filter programs are stored and associated with firewall engine 906a. The selected firewall filter program is executed by the firewall engine 906a just like a processor executing cpu instructions. Some firewall filter instructions refer to data in the key buffer to perform match operations. Depending on the contents of the packet, the program may decide to filter (drop) the packet and terminate the lookup, or it may exit and return an intermediate next hop (in this example of jtree lookup type) that updates the current key pointer (in this example to the "IP Destination Address" field of the key).

In this example, the third lookup operation is a jtree lookup. The tree search engine 906d performs a jtree lookup on the IP destination address in the key buffer to find the route with the longest matching prefix. The result is an extended next hop which contains instructions to place the final destination for the packet based on the jtree lookup result in the result buffer. This is the destination the packet should be forwarded to if the packet does not get dropped by the firewall filter in the final lookup step. The next hop returned as a result from this jtree lookup operation also contains an intermediate extended next hop of firewall filter type which points to a firewall program. The program may decide to filter (drop) the packet in which case the lookup will terminate and the packet will be dropped. If the program decides to keep the packet, the filter program will simply terminate without taking any special action. In this case, the current contents of the result buffer, which is the result of the jtree lookup in the previous step, will be used to forward the packet to its destination appropriately.

Output Switch

Referring now to FIG. 11, output switch 102 includes a controller interface 1505, one or more memory inputs 1502 (1502-0 through 1502-7, one for each memory bank), one or more outputs 1504 (1504-0 through 1504-7, one for each multi-function multiport), a result processor 1506 and an output processor 1508. Output switch 102 performs four functions: receive output results, process output results, receive cells from memory and output cells to output ports.

a) **Transfers from Memory to the Output Switch**

Cells from memory are received at memory inputs 1502 and transferred to output processor 1508. Cells are transferred based on read requests received at the input switch from multi-function multiports. Each cell includes an output port identifier and cell data.

Output processor 1508 decodes the destination multi-function multiport from the cell information received from memory and transfers the cell data to the appropriate outputs 1502. At each cell slot, output switch 102 may receive a cell for processing from each bank in global data buffer 104.

b) Transfers from the Output Switch to the Multi-function Multiports

Output switch 102 receives notification from controller 106 on controller interface 1505. Result processor 1506 decodes the result (route) and determines which multi-function multiport(s) 150 is (are) to receive the route data. Based on the mask in the notification, result processor 1506 transfers the notification to output processor 1508 for transfer to each multi-function multiport 150 indicated. At each cell slot, output processor 1508 provides (via outputs 1504) a notification cell to each multi-function multiport 150.

A notification cell includes a header and data field. The header includes memory bank source information and route information. The memory bank source information includes a source identifier for indicating which memory bank provided the cell in data field. The route information contains data from the notification including a next hop index, packet length, full address and offsets.

Output Section of a Multi-function multiport

Referring now to Figures 3 and 12 each multi-function multiport 150 includes an output switch interface 316, an input switch interface 304 including read request queues 305, head and tail queue buffer 318, an output request processor 306, an line output interface 308, storage device (memory) 310, stream output buffers 312 and output formatter 314.

a) Notification Queues

A multi-function multiport 150 receives notification that a packet is to be processed in the form of a notification cell received at the output switch interface 316.

Output request processor 306 processes notifications, storing each in an appropriate location in head and tail queue buffer 318 and servicing notification requests as they make their way through the various priority queues in head and tail queue buffer 318. The servicing of requests results in the generation of a read request to input switch 100 associated with the first address in memory where the packet (associated with the particular notification) is stored.

Referring now to Figure 13, head and tail queue buffer 319 includes a plurality of notification queues Q 1700, where $Q=4*s$, and where s is the number of active streams in the multi-function multiport. Unlike the global data buffer, the queues Q are implemented on a per-port basis. The queues in a respective multi-function multiport store only those notifications associated with streams to be outputted from the respective port of the multi-function multiport. Each queue is itself divided into a head region 1702, a tail region 1704 and a body region 1706. The head and tail region for a queue are stored in the head and tail queue buffer 318.

The size of the portion of the head and tail queue buffer dedicated to each stream is fixed at initialization time and is proportional to the peak bandwidth of its stream. The partitions

between portions are "hard" in the sense that a stream cannot use more than the memory allocated to it. The partitions between queues associated with the same stream are "soft". The size of an individual queue is proportional to the nominal bandwidth allocated to its queue. The body region of the notification queue is stored in the notification area 319 (Figure 3) of the memory section 290 of the multi-function multiport 150. Each stream is assigned 4 queues (the body portions of the priority queues) in the notification area 319 (Figure 3). The body region can be sized to be 1/5 of the overall memory section.

Each queue associated with a given stream is serviced according to a priority scheme. Notifications that are received by the output request processor 306 are loaded into an appropriate queue associated with a stream based on the priority of the notification. Priority for notifications can be set by an external source and may be included in the packet received by the router. Alternatively, controller 106 (Figure 3A) may set the priority depending on the amount of time required to perform the route lookup or other criteria.

Once a queue 1700 has been identified based on the priority information and stream ID, the output request processor 306 loads the notification into the appropriate tail queue 1704. Notifications are transferred between the respective head, tail and body portions of a queue based on available space by a queue manager (not shown). In one implementation, each notification is 16 bytes, and the notification area 319 is sized to hold 64 bytes. Accordingly, for reasons of bandwidth efficiency, all reads and writes to the notification area are done using 64-byte cells containing four 16-byte notifications each. The head and tail of each queue is sized to store only a small number of notifications, the bulk of queue storage being provided by the notification area in the multi-function multiport memory bank. As long as space is available on-chip (on the multiport) to hold the notifications for a queue, the notification area is completely bypassed. When on-chip space runs out, the notification area acts as the large "middle" of the queue, with a few notifications at the head and tail being held on-chip.

While the size of the notification area will tend to limit the numbers of dropped packets, occasionally a queue will become full. Output request processor includes a drop engine (not shown) for determining which entries in a particular queue are to be dropped based on a predefined algorithm. In one implementation, the drop engine institutes a programmable random early drop routine. The routine is programmable in that the user can define one or more parameters, random in that a random number generator is used to determine whether a entry will be dropped. Early refers dropping from the head of the queue.

The programmable random early drop routine may be implemented in software and when executed performs the following sequence of operations. The process begins by calculating the amount of data stored in a particular queue. This information is stored in the form of a fraction (or percentage) of fullness. Thereafter, a drop criterion is determined based on the

fraction of fullness. In one implementation, a table of drop criterion values ranging from zero to one is mapped against fullness fractional values. The drop engine then derives a random number from zero to one. The random number may be generated by a random number generator or other means as is known in the art. A comparison is made between the random number generated and the drop criterion value. Thereafter, the entry at the head of the particular queue is dropped if the random number generated is larger than the drop criterion. Alternatively, the drop engine could avoid the drop if the random number generated is less than the drop criterion. The drop engine operates on each queue at a regular interval to assure that the queues do not overflow and a orderly method of dropping packets is achieved if required. This process is extremely helpful when transmitting packets across the Internet.

b) Per Bank Notification Queues

Each stream includes four queues 1700 that are serviced using a weighted round robin discipline. The weighting is used to reflect the priority associated with a given queue. For example, the four queues for a given stream may be serviced in the following ratios: Q1 at 50%, Q2 at 25%, Q3 at 15% and Q4 at 10%.

The multi-function multiport maintains four cell pointers for each queue: start, end, head, and tail. In one implementation, each pointer is 23 bits long and can address the entire memory associated with the multi-function multiport. The start and end pointers mark the boundaries of the queue's region, while the head and tail pointers point to the next cell (notification) to read and next cell to write respectively. The head and tail pointers are restricted to align within the region defined by the start and end pointers, and standard wraparound arithmetic is performed when incrementing these pointers.

Given the description above, it should be clear that the region for a queue can be as small as one cell and as large as the entire memory bank. It is up to the software to configure the pointers at initialization time to define the sizes of the regions, and to ensure that regions are non-overlapping with each other and with the memory allocated to the global packet buffer.

Typically, the software is used to allocate memory to a stream proportional to the stream's bandwidth.

c) Read Request Generation

Output request processor 306 services the queues to extract notifications from the head regions of queues 1700. Output request processor generates a first read request based on the full address received from output switch 102. Thereafter subsequent read requests are generated for transmission to the input switch based on the offset information provided in the request (in the notification cell) or indirect cells (as will be described below). Read requests

include a stream identifier and a full address. Read requests are sent by the output request processor to an appropriate read request queue 305. One read request queue 305 is provided for each bank of memory.

In one implementation, if the packet length, as determined from the route information provided with the notification cell, is greater than five (5) cells, then the output request processor first requests the transfer (read from memory) of the first indirect cell associated with the packet. This is accomplished by computing the address of the indirect cell based on the full address and the offsets provided in the notification cell. After the indirect cell request is generated, the output request processor generates read requests for the remaining cells in the packet based on the full address and the offsets provided in the notification cell. Upon receipt of a indirect cell from the output switch 102, output request processor continues to generate read requests for the remaining cells in the packet based on the offset information contained within the indirect cell.

Subsequent indirect cells are retrieved in a similar fashion. That is, at the time for reading the next indirect cell, the address of the next indirect cell is computed based on the last offset stored in the previous indirect cell. The timing of retrieving the indirect cells is accomplished such that no delays in the output stream are incurred. Each subsequent indirect cell is retrieved prior to the end of the processing of the prior indirect cell. In this way, once the output stream is initialized, no buffering of data is required and no interruptions due to the latency associated with the retrieval process are experienced.

Output requests to an individual memory bank are processed strictly in order. That is, the multi-function multiport may track each request issued to a memory bank (through the read request queues) and is assured that the data received in response to a series of requests to the same memory bank will be strictly delivered according to the sequence or pattern in which they were issued. Output request processor 306 keeps track of requests generated for each memory bank through the use of reply queues (not shown). The request queue contains a stream number and a read address. When a request is issued to memory, the entry is removed from the request queue and the stream number portion is placed in an associated reply queue. When a reply is received, the entry at the head of the reply queue is removed and the reply is sent to the stream number (in stream output buffer 312) indicated by the stream number retrieved from the reply queue.

As cells are received back at the multi-function multiport 150 (responsive to the read requests), they are stored in an associated stream output buffer 312. Stream output buffer 312 includes a plurality of FIFOs, one for each stream. Each cell received for a stream is placed in the streams associated FIFO. For given packet, the multi-function multiport stores a fixed number of cells (in the FIFO) required to provide a streamed output prior to initializing the output of the stream to line output interface 308. In one implementation of the present invention,

twelve cells are stored prior to beginning output (stream data) from the output port. The selection of the number of cells for storage in output buffer 312 is based on the latency in the read process (number of clock cycles between a read request from an multi-function multiport and the arrival of the cell associated with the read request to the output port).

Output formatter 314 receives the cells from output buffer 312 and couples the data with media header information stored in memory 310. Each request (notification) received from output switch 102 includes a next hop index. The next hop index indicates the starting address in memory 310 of the media header information associated with a given type of transmission (derived from the destination of the packet). Media header information stored in memory 310 may be loaded upon initialization of the router and updated by the controller as required. Output formatter 314 couples the cell data returned from memory with the appropriate media header to generate a proper packet for transfer out of router 20 on the line output interface 308.

Packet Routing Overview

Referring now to FIG. 14, in a method of routing packets through a switch a packet is received at a multi-function multiport (1800). The multi-function multiport divides the packet into fixed length cells and transfers the cells to an input switch (1802). Input switch removes the key information from the first cell in a packet and stores it temporarily in a key buffer (1804). Thereafter the input switch routes the cells to memory banks resident in the multi-function multiports in a time division multiplexed manner (1806). The input switch stores the first address in memory where the first cell is stored and computes offsets for each additional cell associated with the offset in memory for the next contiguous memory bank into which the next cell is written (1808). The input switch creates indirect cells to store linking information for the packet if the packet length exceeds five cells (1810). If the number of cells exceeds the number of available offsets in an indirect cell, then the old indirect cell is stored in memory and a new indirect cell is created and loaded based on the offsets calculated for each new cell received at the input switch.

When the packet (and its indirect cells if any) have been stored in memory, then the key, full address of the first cell and offset information is transferred as a lookup request to a controller (1814). The controller performs a lookup operation that can include a plurality of chained lookup operations and generates a result. The result includes the destination port (multi-function multiport), address, offset information and next hop index (1816). A notification including the result is transferred to the output switch for transfer to the appropriate multi-function multiport (1818).

Upon receipt of a notification, the multi-function multiport generates read requests a cell at a time to the input switch for the data associated with the packet (1820). The input switch issues the read requests in a time division multiplexed fashion generating a single request to each

memory bank per cell slot (1822). When the memory bank receives the request from the input switch, cell data and a multi-function multiport identifier associated with the request are transferred to the output switch (1824). Again, at each cell slot, the output switch transfers a single cell to each of the multi-function multiports. Upon receipt, the multi-function multiport couples the cell data with media header information and streams the data to the destination (1826).

Alternative Implementations

The present invention has been described in terms of specific embodiments, which are illustrative of the invention and not to be construed as limiting.

For example, the system can be configured to run the same jtree over two different fields in the key (this is so the system can support a match which is true when one of a list of prefixes matches either the destination address or the source address in a packet).

Next hops can be designed to support many features in addition to packet forwarding. Additional packet processing features that are possible include but are not limited to accounting, sampling, quality of service, flow policing, and load balancing.

For accounting, fields in an intermediate or extended next hop can indicate that a specific counter should be incremented. For example, the routes in a jtree for longest prefix match lookup are next hops. If these next hops point to different counters, the counters can be used to count the number of packets taking each route. Note that more than one next hop can be set up to point to the same counter. For supporting sampling and quality of service, next hops can contain fields that modify sampling enable bits and quality of service values stored in the result buffer. A lookup step can be added to the chain of lookups to test certain fields in the key to select the next hop to make the desired modification.

For supporting flow policing, next hops can contain pointers to data structures storing policing parameters. A firewall filtering step can be inserted into a chain of lookups to select a set of policing parameters based on different fields in the key buffer.

For supporting load balancing, an intermediate next hop can be designed to point to more than one final next hop for selection.

The data structures (jtrees, index tables, filter programs) stored in memory 920 (of FIG. 9) to support the various lookup operations can be atomically updated so that updates to the data

structures can be performed at the same time lookup operations are being executed. Each of the data structures includes a pointer to a next operation. The pointers are of the form of a next hop. The next hop may be an intermediate next hop, or a final next hop. Modifications can be made to a chain (a series of lookups that are to be performed for a particular type of packet (e.g., incoming interface)) without requiring a current lookup operation to be terminated. That is, an entry in the chain can be removed by updating the pointer in the lookup specified one entry in the chain above the item to be removed to point to the entry in the chain that is one entry after the deleted item. The update is atomic, in that existing operations are not affected, and any subsequent results returned after the update will branch to the appropriate next entry in the lookup chain. All changes can be resolved to a single write operation. Similarly, a lookup can be added to the chain by adding the data structure to memory including designating the result that is returned by the added entry point to the entry in the chain at the point just after where the new entry is to be included in the chain. Finally, the pointer in the entry (the result returned) just above the point where the new entry in the chain is to be included is updated to point to the new entry. Again, existing operations are not required to be terminated and subsequent chain processing operations (that are executed after the change) will include the added lookup.

Similarly, within the data structures, updates to individual portions of the branching operations (e.g., branches in a tree, a branch in a filter program, or a result returned in an index search) can be performed atomically.

Other embodiments are within the scope of the following claims.

What is claimed is:

50483-4

CLAIMS:

1. A method for performing a lookup in a switching device of a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the method comprising:

determining a first lookup operation to be executed, the first lookup operation being selected from a group including at least a tree search, an index search, and a filter operation;

executing the first lookup operation including returning a result;

determining if the result includes a pointer to another lookup operation in the sequence of operations;

if so, executing the lookup operation indicated by the result; and

else, terminating the lookup,

wherein a lookup operation includes a function list that specifies one or more functions to execute during the execution of the lookup operation and wherein the function is selected from the group of a management function, accounting function and policing function,

wherein the method further comprises identifying when a lookup operation specifies a function and executing the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed, and

50483-4

wherein the execution of the function includes determining when a packet should be sampled for further processing and including in the result a designation that indicates the packet is to be sampled.

2. The method of claim 1, wherein the lookup operation indicated by the result is a different lookup operation from the group than the first lookup operation.

3. A route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the route lookup engine comprising:

a group of different lookup operation engines for executing lookup operations including returning a result, the group of different lookup operation engines including a tree search lookup engine, an index search engine, and a filter engine;

a lookup engine operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network; and

a memory configurable to store one or more tree data structures where the pointer returned for invoking the tree search engine includes an indicator point to a

particular tree data structure stored in the memory to be searched in the lookup operation.

4. The route lookup engine of claim 3 further comprising a memory configurable to store one or more index data structures and where the pointer returned for invoking the index search engine includes an indicator pointing to a particular index data structure stored in the memory to be searched in the lookup operation.

5. A route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the route lookup engine comprising:

a group of different lookup operation engines for executing lookup operations including returning a result, the group of different lookup operation engines including a tree search lookup engine, an index search engine, and a filter engine;

a lookup engine operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network; and

a memory configurable to store one or more filter data structures and where the pointer returned for invoking

the filter engine includes an indicator pointing to a particular filter data structure stored in the memory to be searched in the lookup operation.

6. The route lookup engine of claim 3, wherein a lookup operation includes a function list that specifies one or more functions to execute during the execution of the lookup operation and where the lookup engine is operable to read the function list and execute the one or more functions in the lookup.

7. The route lookup engine of claim 6 where the function is selected from the group of a management function, accounting function and policing function.

8. The route lookup engine of claim 3 wherein the lookup engine is operable to identify when a lookup operation specifies a function and execute the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed.

9. A route lookup engine for performing a lookup in a packet switched network where the lookup includes a plurality of distinct operations each of which returns a result that includes a pointer to a next operation in a sequence of operations for the lookup, the route lookup engine comprising:

a group of different lookup operation engines for executing lookup operations including returning a result; and

a lookup engine operable to determine a first lookup operation in the sequence to be executed, evaluate the result returned from the execution of the first lookup operation to determine if the result includes a pointer to

50483-4

another lookup operation in the sequence of operations, invoke a particular lookup operation engine from the group of lookup operation engines based on the pointer to execute a next lookup operation in the sequence of operations and terminate the lookup and return a result to be used in routing the packet through the packet switched network,

wherein the lookup engine is operable to identify when a lookup operation specifies a function and execute the function including returning a result that indicates a next lookup operation in the sequence of operations to be executed, and

where the execution of the function includes determining when a packet should be sampled for further processing and including in the result a designation that indicates the packet is to be sampled.

SMART & BIGGAR

OTTAWA, CANADA

PATENT AGENTS

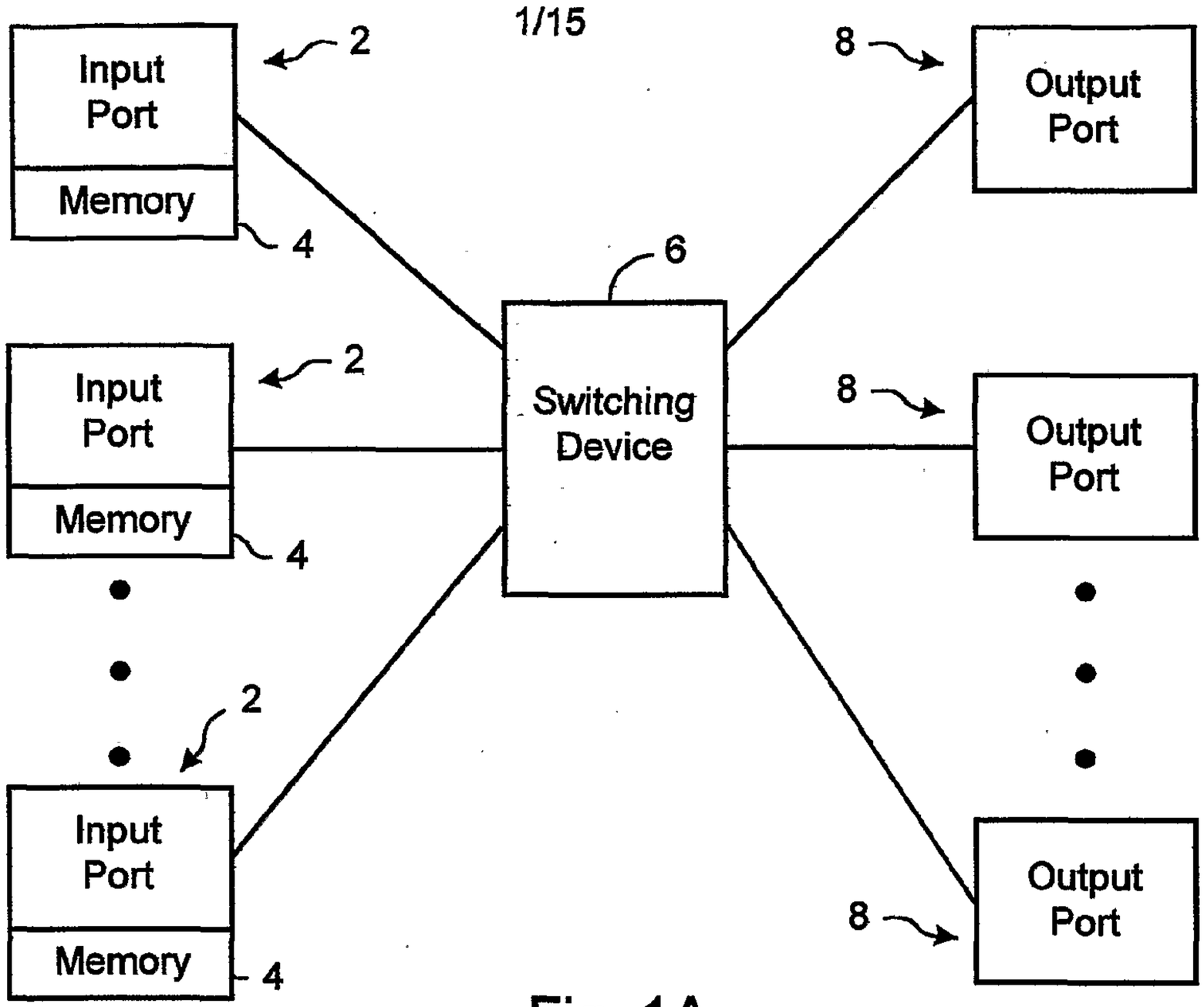


Fig. 1A
(Prior Art)

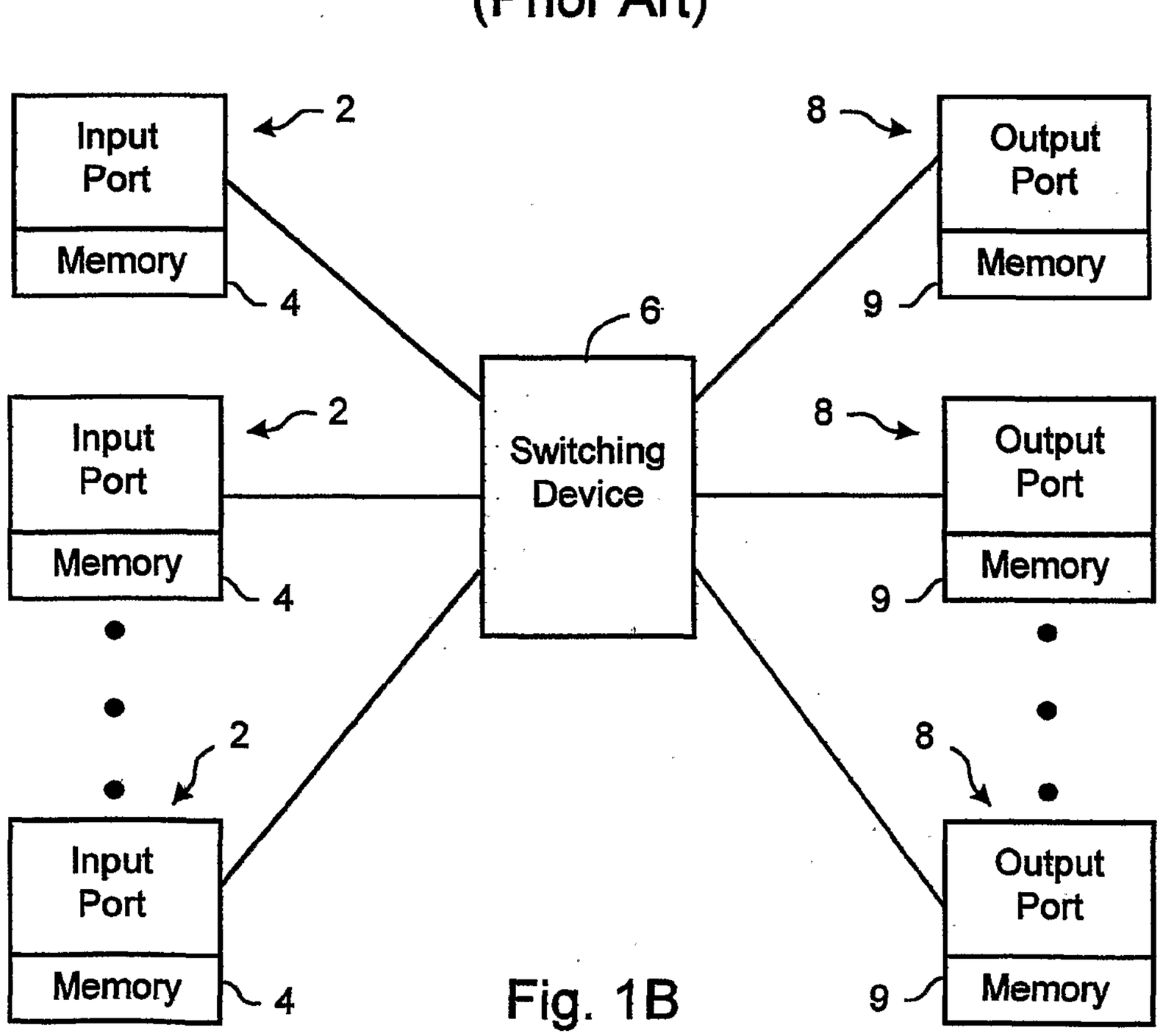


Fig. 1B
(Prior Art)

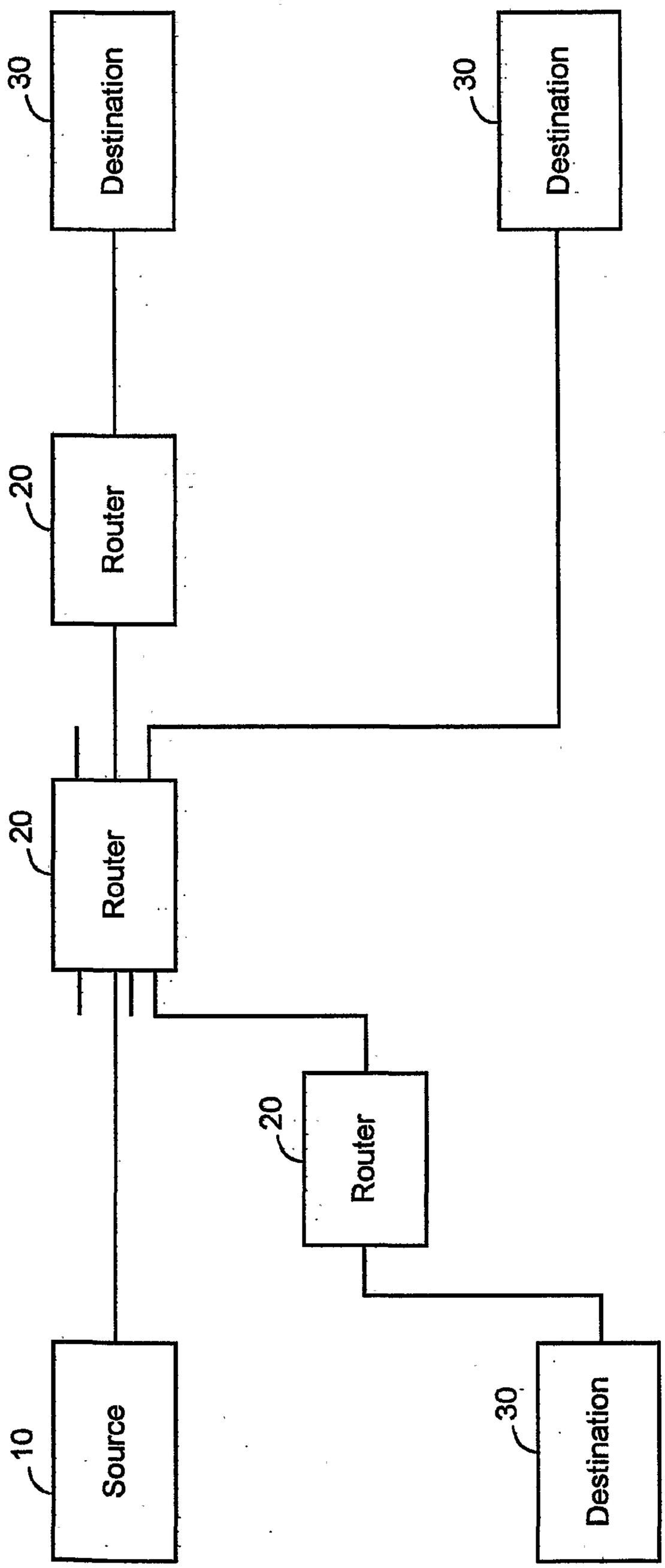

30
Destination
30
Destination
20
Router
20
Router
20
Router
10
Source
30
Destination

FIG. 2A

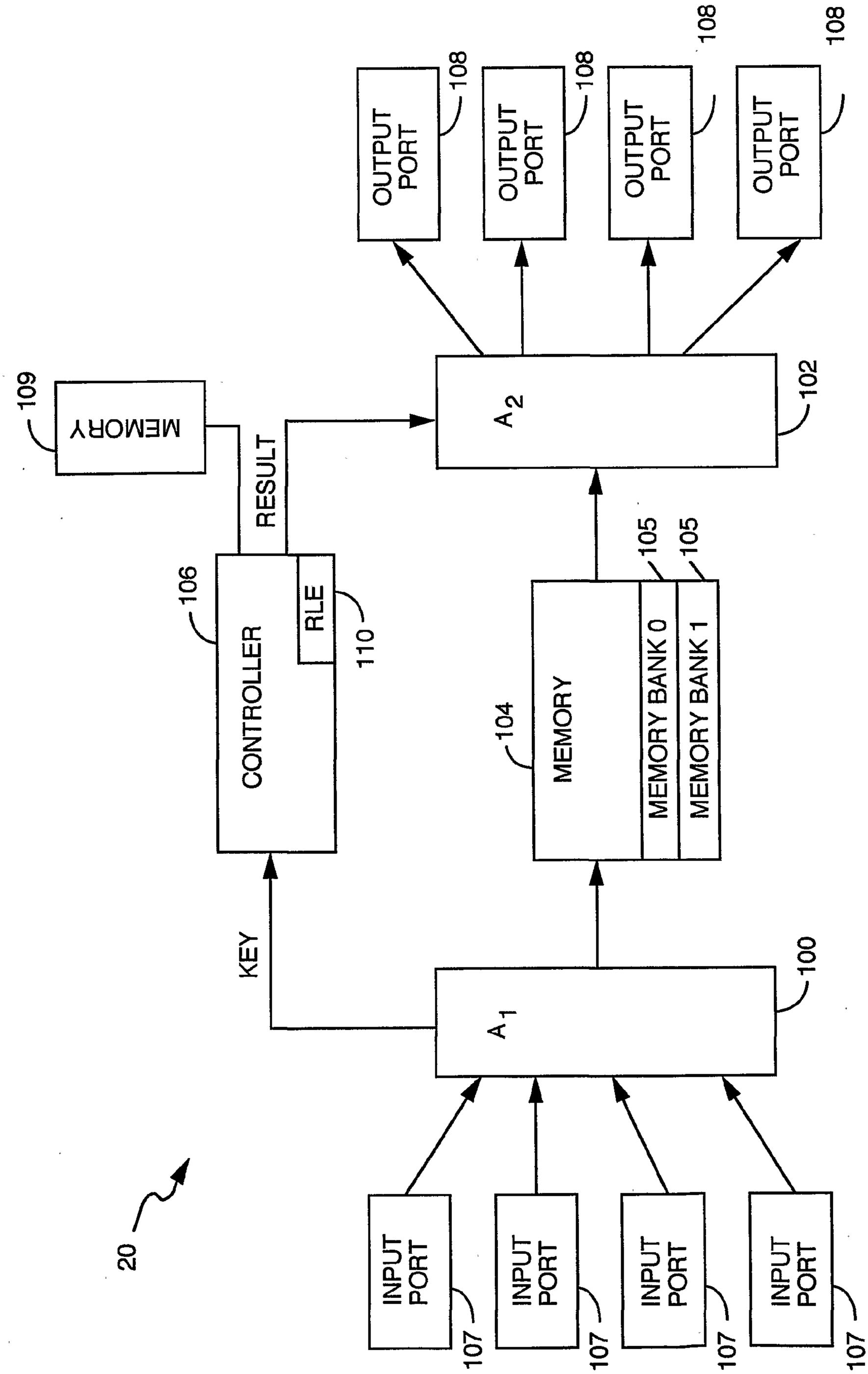

20
INPUT PORT
107
INPUT PORT
107
INPUT PORT
107
INPUT PORT
107
KEY
A1
100
MEMORY
104
MEMORY BANK 0
MEMORY BANK 1
105
105
CONTROLLER
106
RLE
110
RESULT
MEMORY
109
A2
102
OUTPUT PORT
108
OUTPUT PORT
108
OUTPUT PORT
108
OUTPUT PORT
108

FIG. 2B

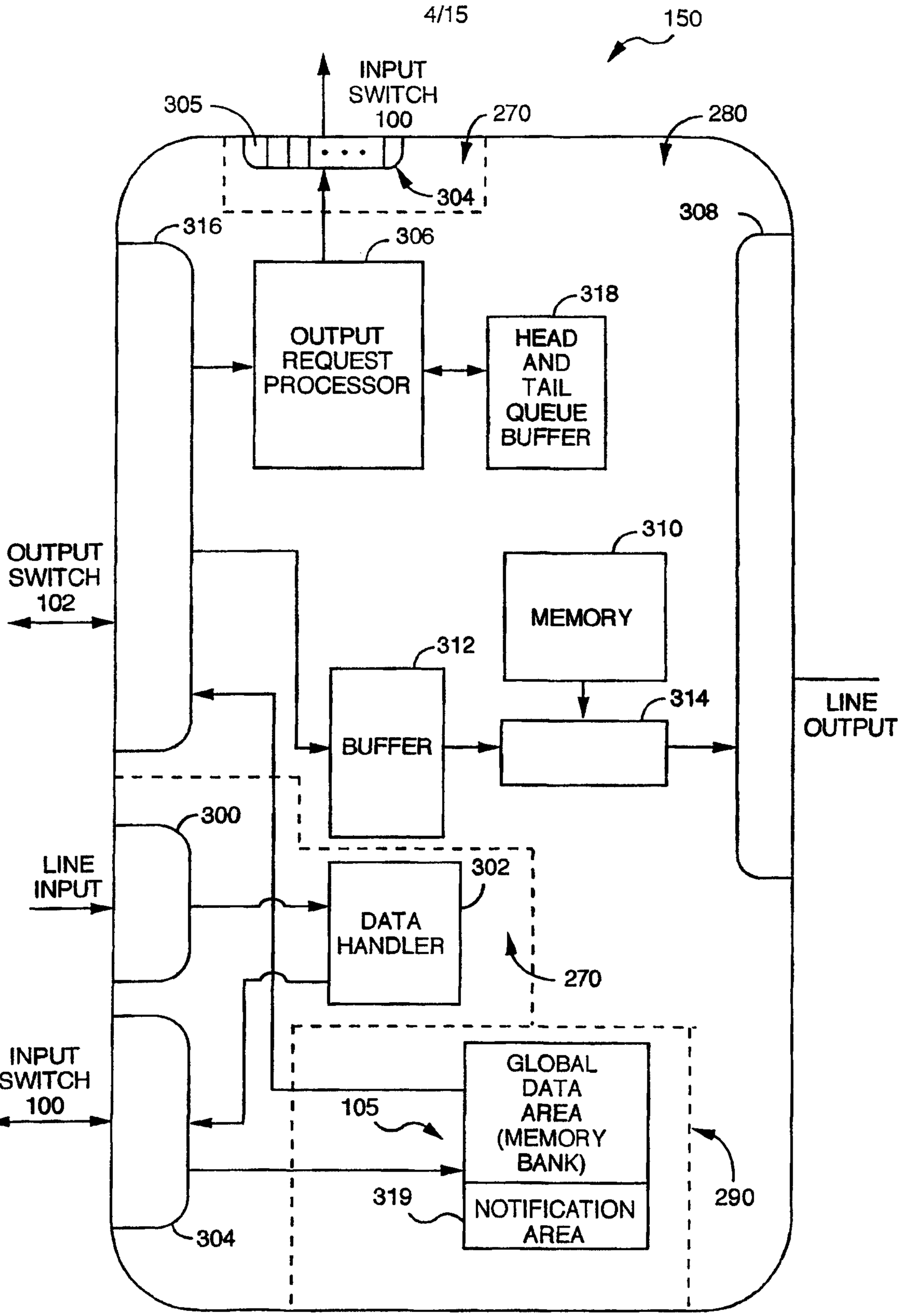

150
INPUT SWITCH 100
305
270
280
304
316
306
308
OUTPUT REQUEST PROCESSOR
318
HEAD AND TAIL QUEUE BUFFER
OUTPUT SWITCH 102
310
MEMORY
312
BUFFER
314
LINE OUTPUT
300
LINE INPUT
302
DATA HANDLER
270
INPUT SWITCH 100
105
GLOBAL DATA AREA (MEMORY BANK)
319
NOTIFICATION AREA
290
304

FIG. 3

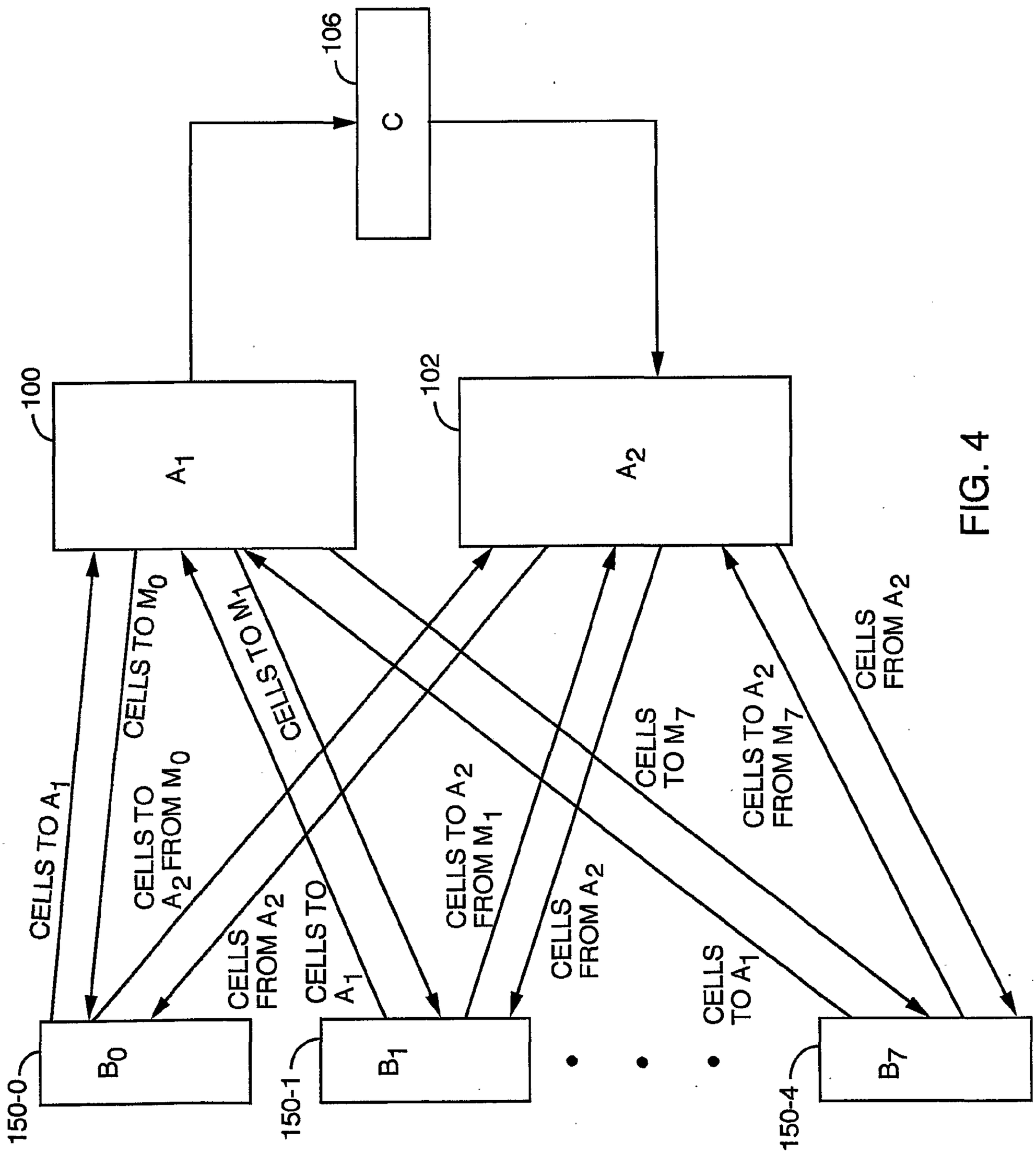

106
C
100
A_1
102
A_2
CELLS TO A_1
CELLS TO M_0
CELLS TO A_2 FROM M_0
CELLS TO M_1
CELLS FROM A_2
CELLS TO A_1
CELLS TO A_2 FROM M_1
CELLS FROM A_2
CELLS TO M_7
CELLS TO A_2 FROM M_7
CELLS FROM A_2
CELLS TO A_1
150-0
B_0
150-1
B_1
150-4
B_7

FIG. 4

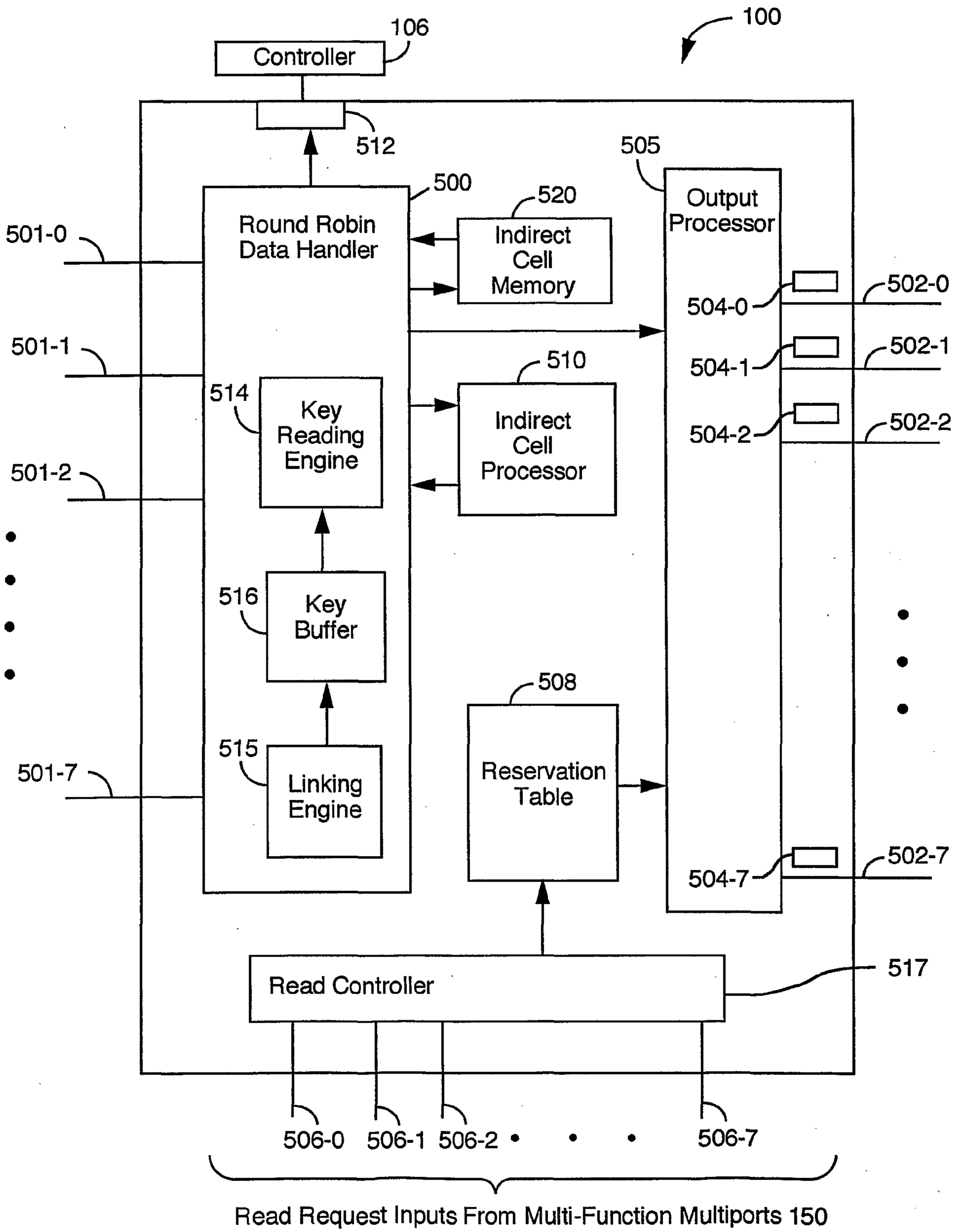

106
Controller
100
512
505
500
520
Output Processor
Round Robin Data Handler
Indirect Cell Memory
501-0
504-0
502-0
501-1
510
504-1
502-1
514
Key Reading Engine
Indirect Cell Processor
501-2
504-2
502-2
516
Key Buffer
508
515
Linking Engine
Reservation Table
501-7
504-7
502-7
517
Read Controller
506-0
506-1
506-2
506-7
Read Request Inputs From Multi-Function Multiports 150

FIG. 5

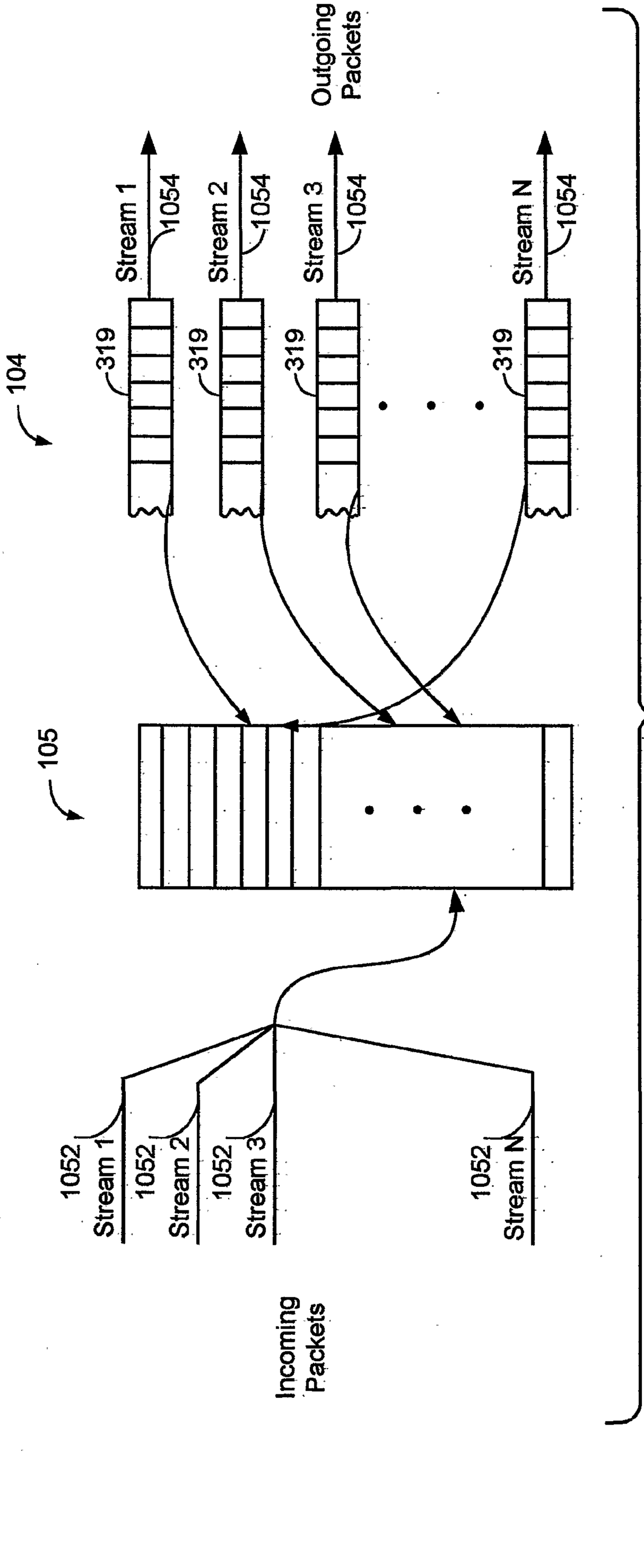

Outgoing Packets
Stream 1
Stream 2
Stream 3
Stream N
1054
319
104
105
1052
Incoming Packets

FIG. 6

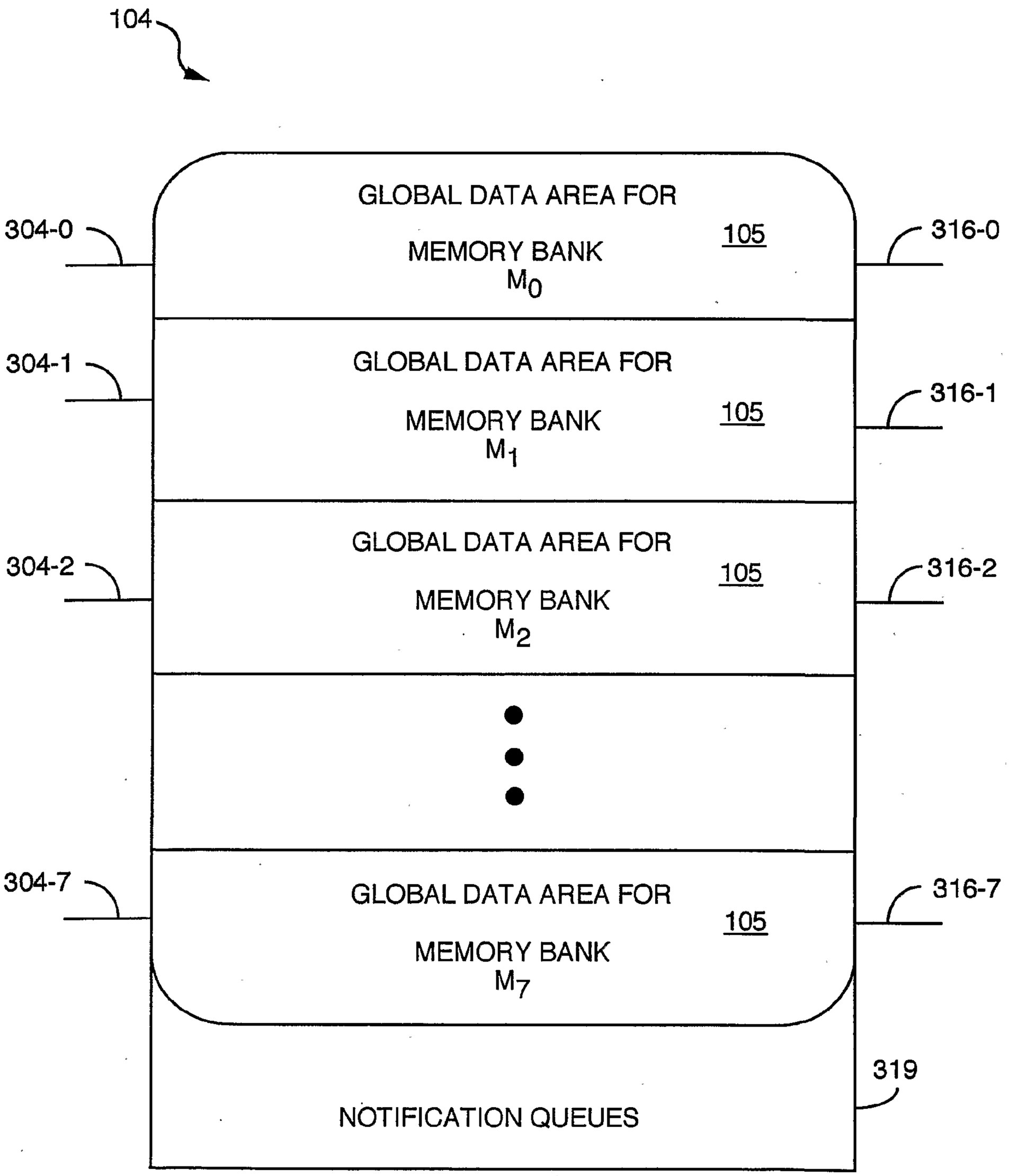

104
GLOBAL DATA AREA FOR
MEMORY BANK
M_0
105
304-0
316-0
GLOBAL DATA AREA FOR
MEMORY BANK
M_1
105
304-1
316-1
GLOBAL DATA AREA FOR
MEMORY BANK
M_2
105
304-2
316-2
304-7
GLOBAL DATA AREA FOR
MEMORY BANK
M_7
105
316-7
NOTIFICATION QUEUES
319

FIG. 7

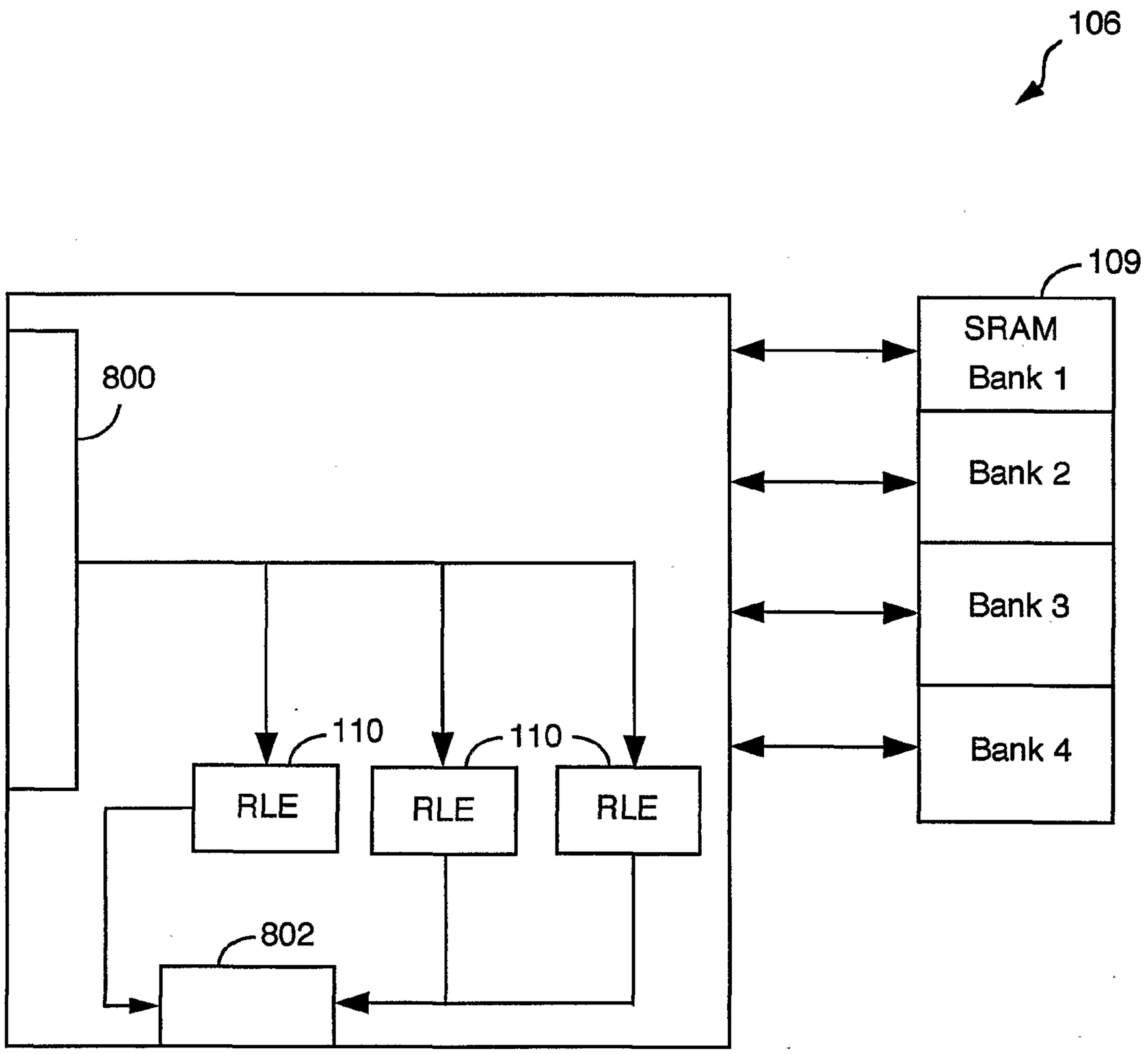

106
109
SRAM
Bank 1
Bank 2
Bank 3
Bank 4
800
110
110
RLE
RLE
RLE
802

FIG. 8

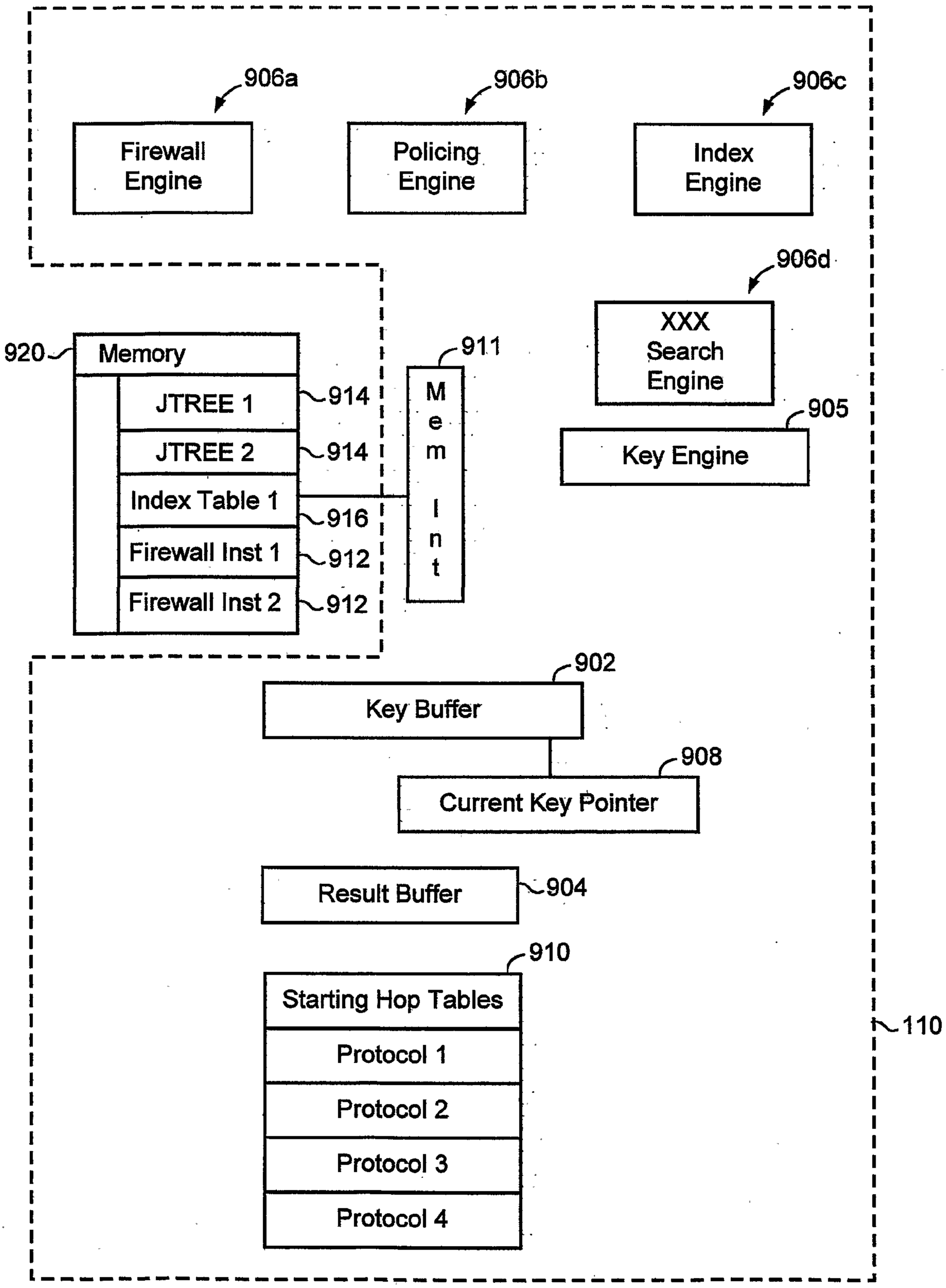

906a
906b
906c
Firewall Engine
Policing Engine
Index Engine
906d
XXX Search Engine
905
Key Engine
920
Memory
JTREE 1
914
JTREE 2
914
Index Table 1
916
Firewall Inst 1
912
Firewall Inst 2
912
911
Mem Int
902
Key Buffer
908
Current Key Pointer
Result Buffer
904
910
Starting Hop Tables
Protocol 1
Protocol 2
Protocol 3
Protocol 4
110

FIG. 9

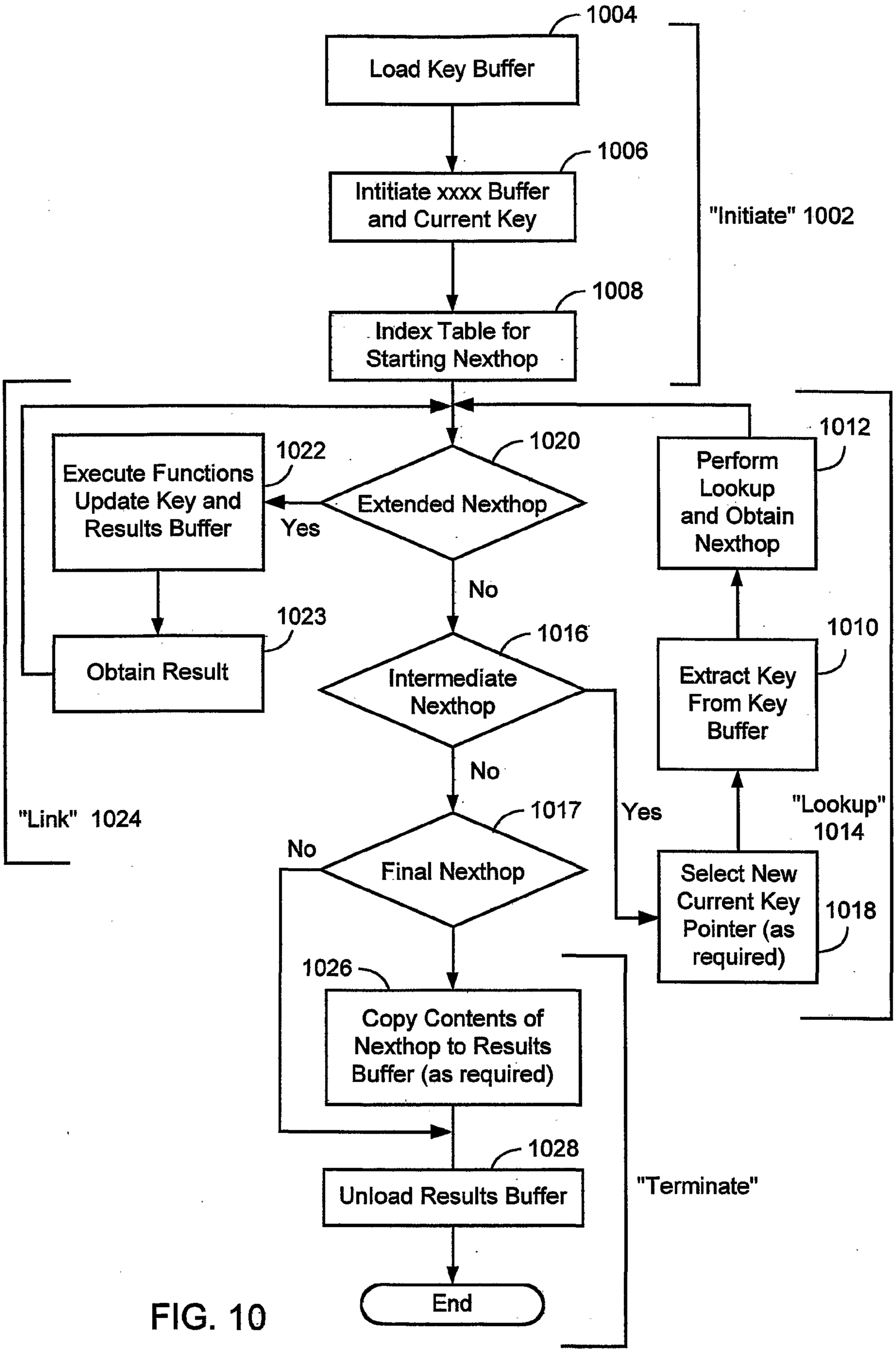

1004
Load Key Buffer
1006
Intitiate xxxx Buffer and Current Key
"Initiate" 1002
1008
Index Table for Starting Nexthop
1012
Perform Lookup and Obtain Nexthop
1022
Execute Functions Update Key and Results Buffer
1020
Extended Nexthop
Yes
No
1023
Obtain Result
1016
Intermediate Nexthop
1010
Extract Key From Key Buffer
No
Yes
"Lookup" 1014
"Link" 1024
1017
No
Final Nexthop
Select New Current Key Pointer (as required)
1018
1026
Copy Contents of Nexthop to Results Buffer (as required)
1028
Unload Results Buffer
"Terminate"
End

FIG. 10

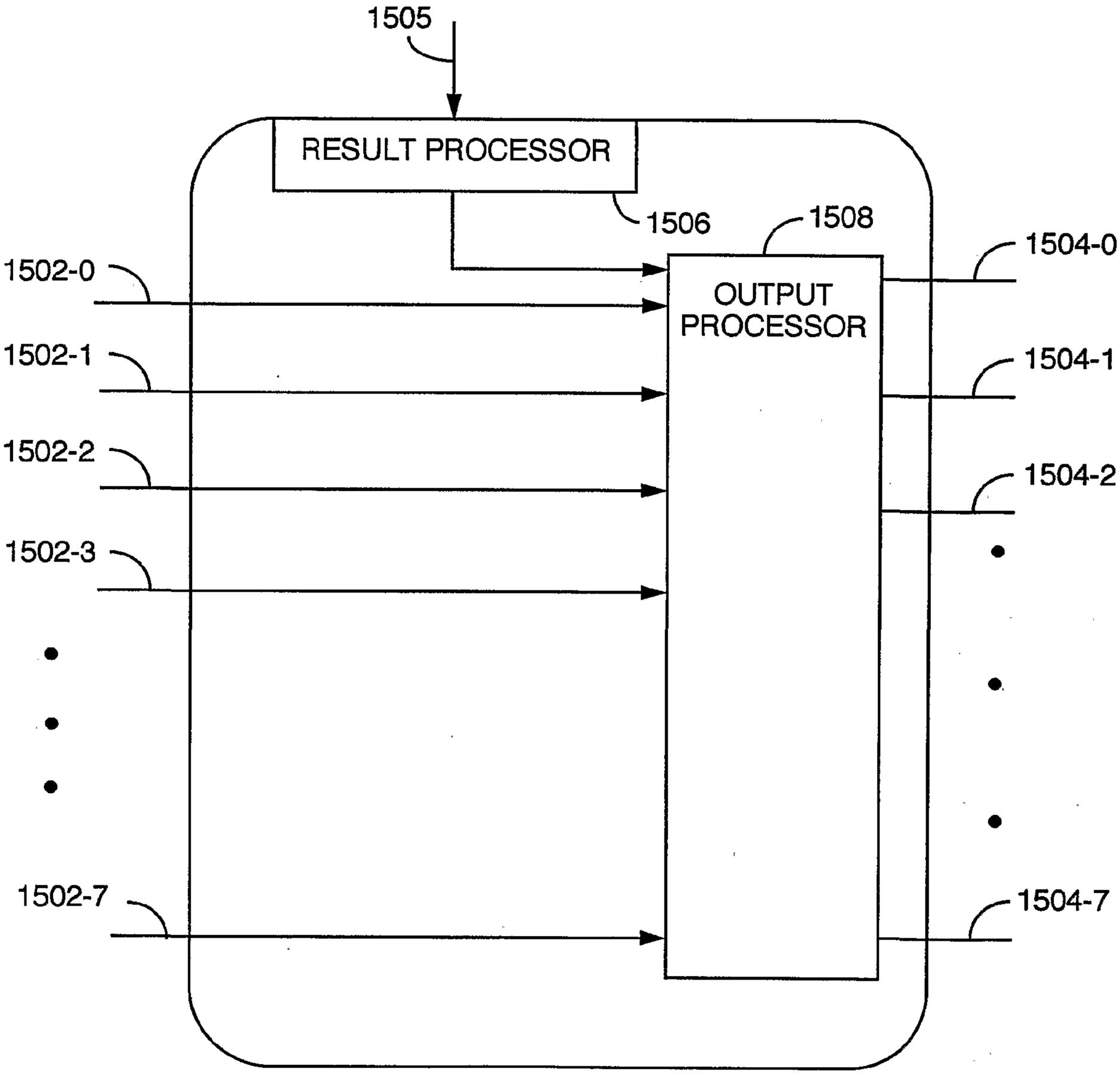

1505
RESULT PROCESSOR
1506
1508
OUTPUT PROCESSOR
1502-0
1502-1
1502-2
1502-3
1502-7
1504-0
1504-1
1504-2
1504-7

FIG. 11

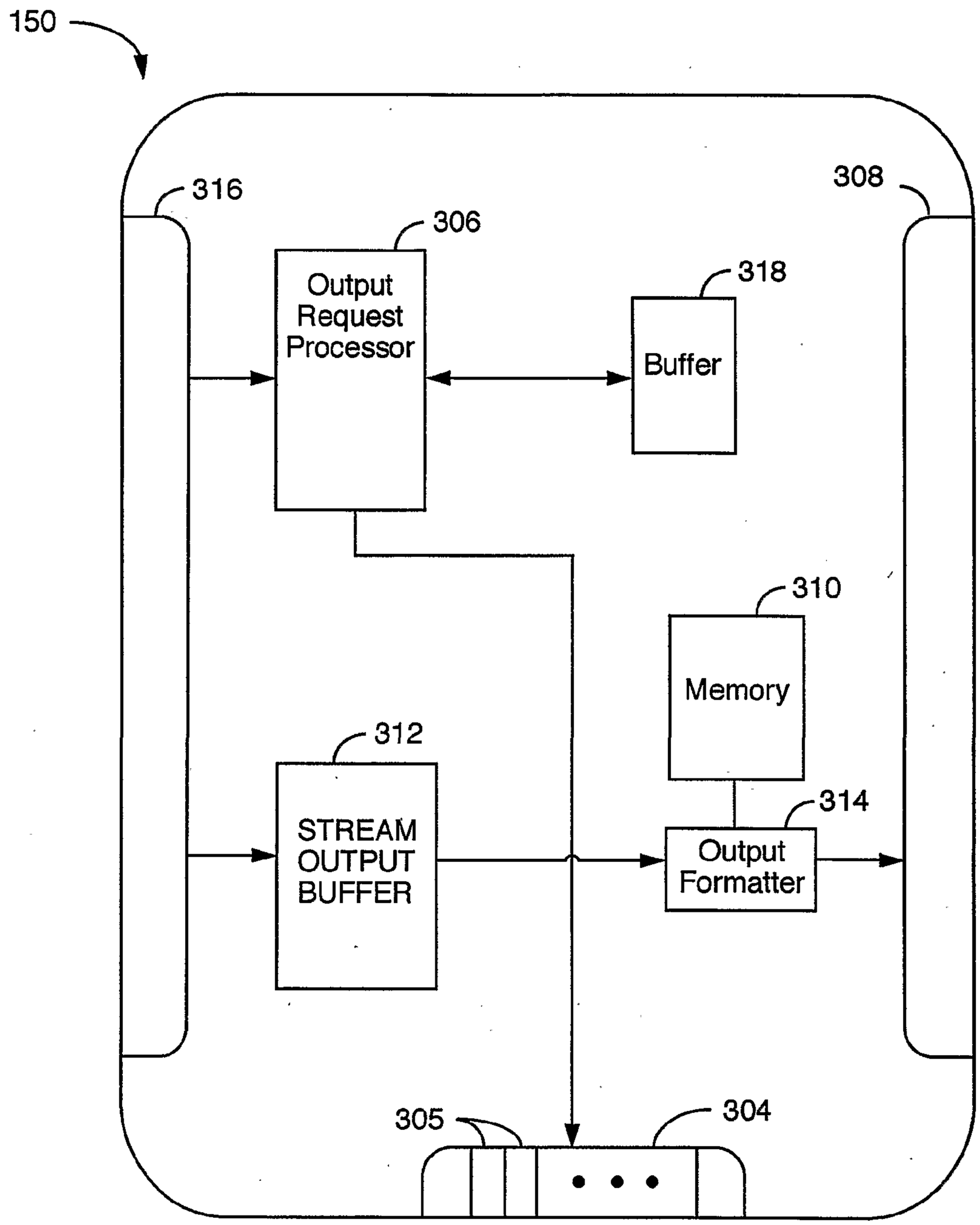

150
316
308
306
Output Request Processor
318
Buffer
310
Memory
312
STREAM OUTPUT BUFFER
314
Output Formatter
305
304

FIG. 12

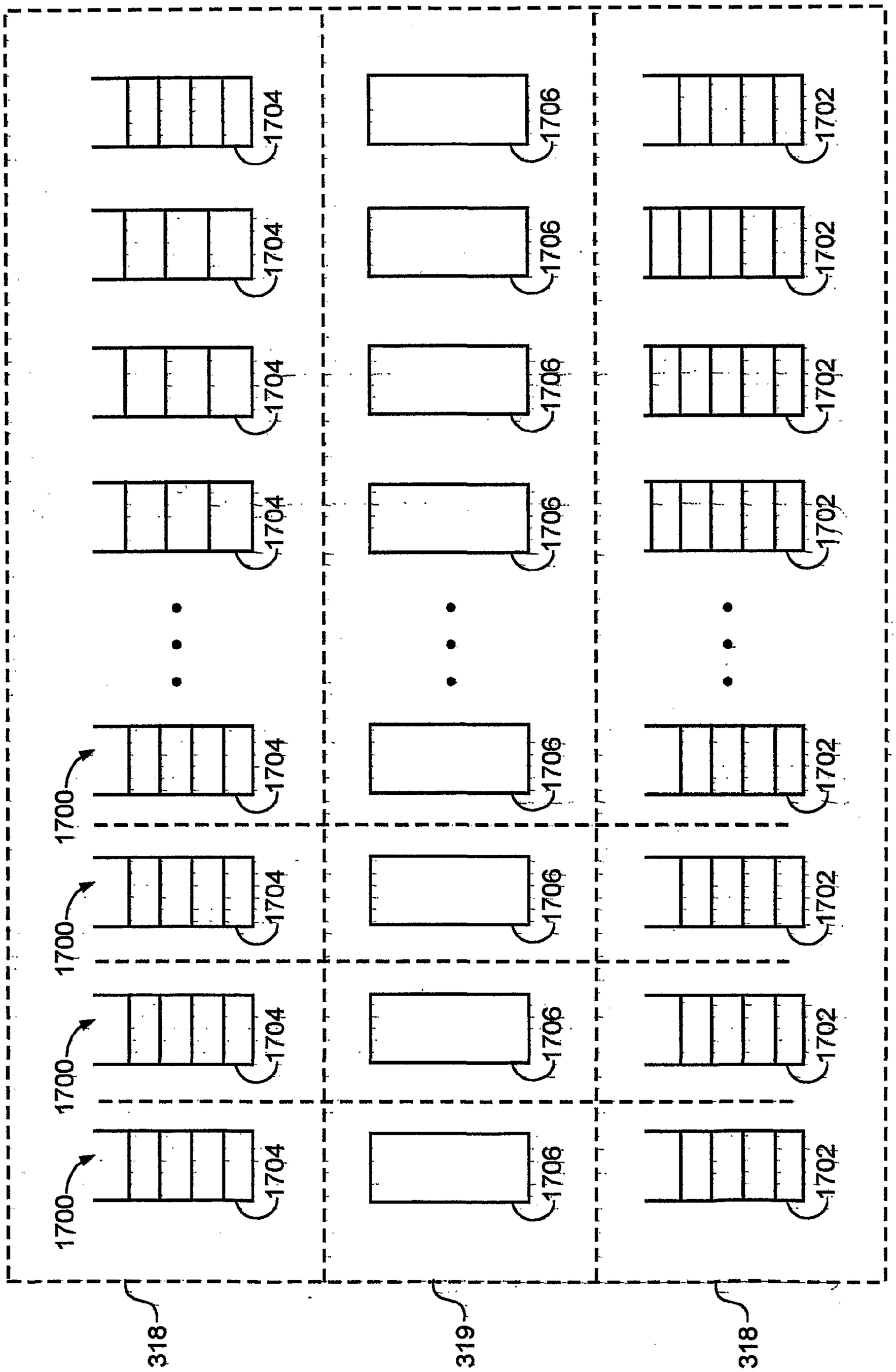


FIG. 13

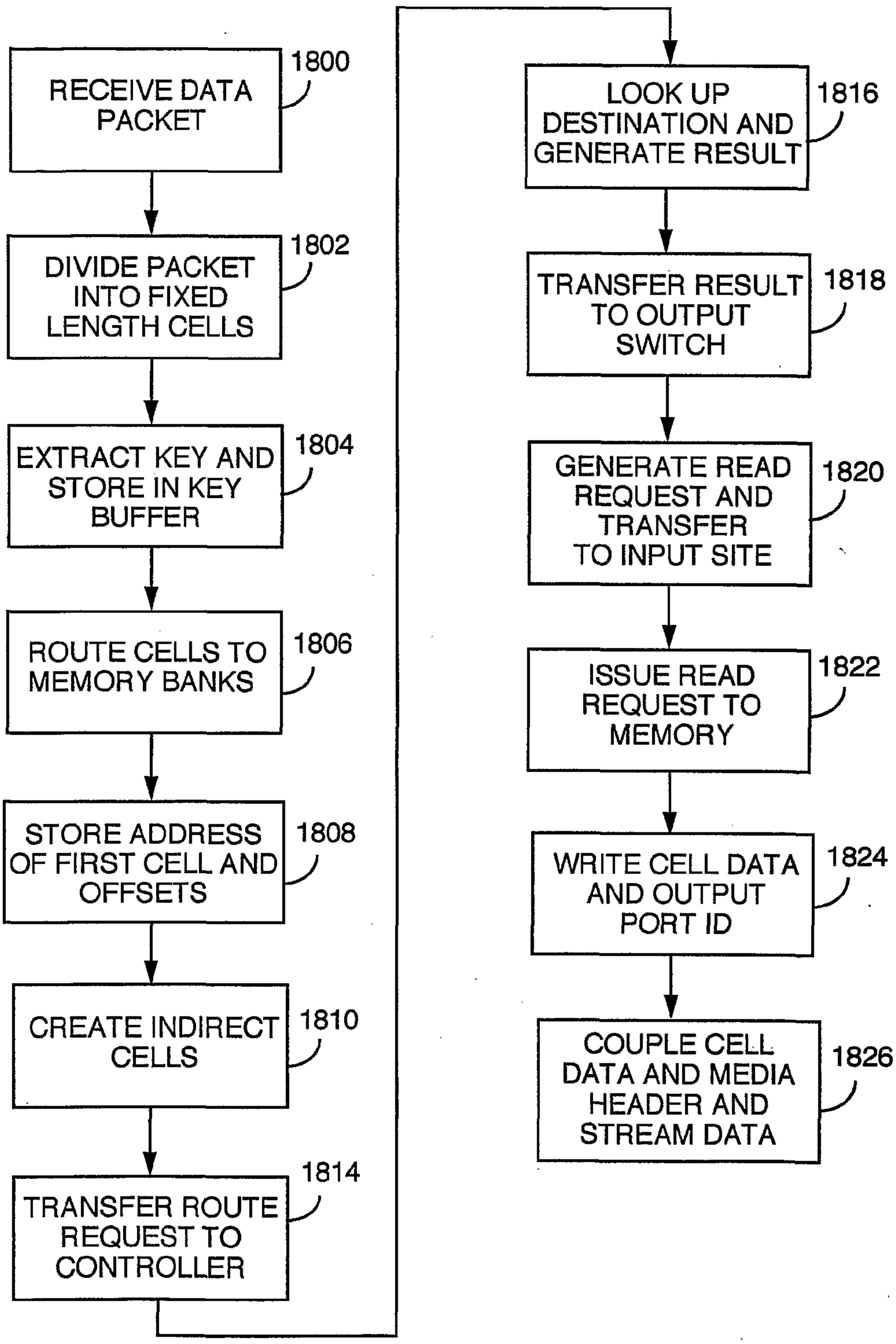
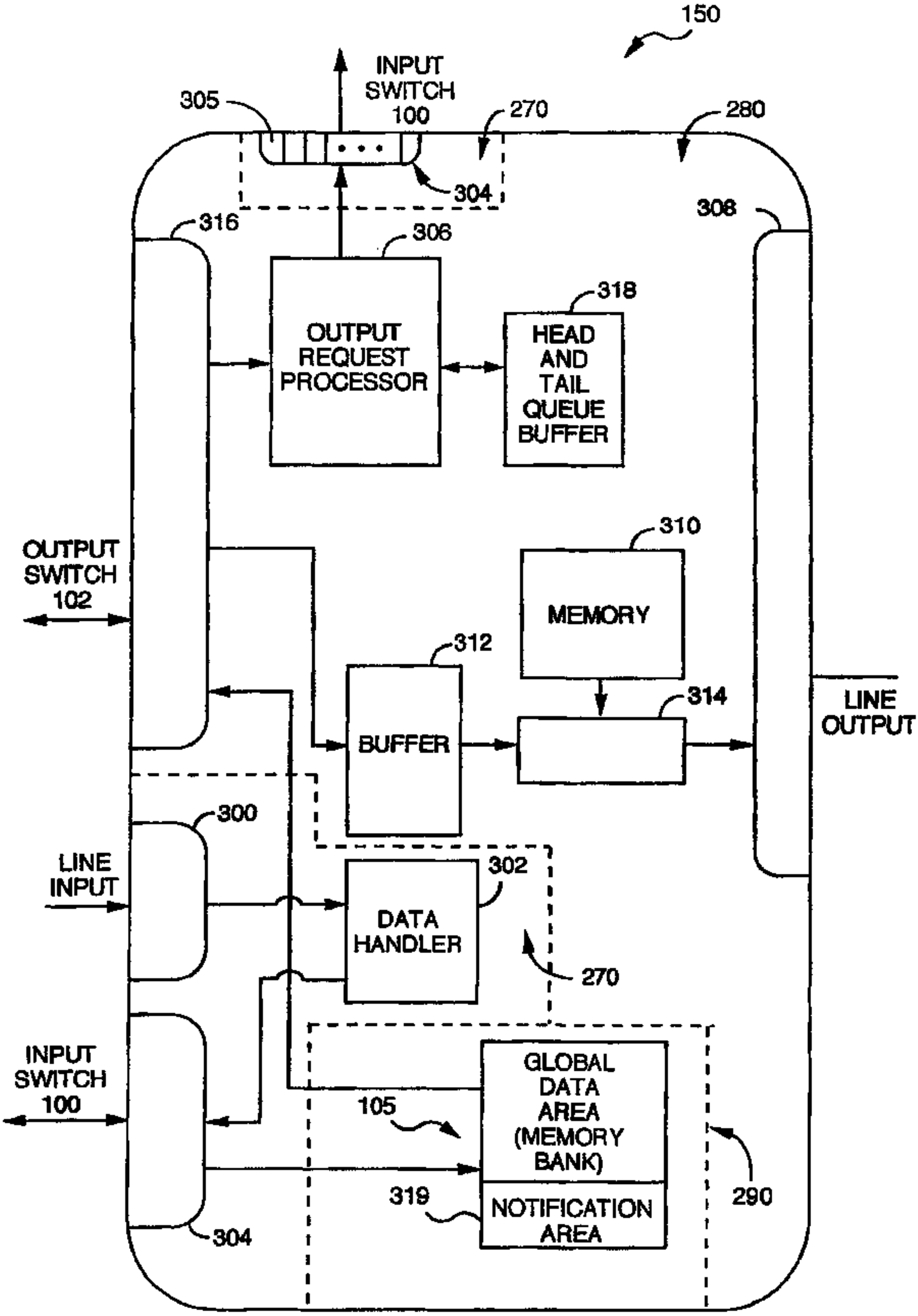

RECEIVE DATA PACKET
1800
DIVIDE PACKET INTO FIXED LENGTH CELLS
1802
EXTRACT KEY AND STORE IN KEY BUFFER
1804
ROUTE CELLS TO MEMORY BANKS
1806
STORE ADDRESS OF FIRST CELL AND OFFSETS
1808
CREATE INDIRECT CELLS
1810
TRANSFER ROUTE REQUEST TO CONTROLLER
1814
LOOK UP DESTINATION AND GENERATE RESULT
1816
TRANSFER RESULT TO OUTPUT SWITCH
1818
GENERATE READ REQUEST AND TRANSFER TO INPUT SITE
1820
ISSUE READ REQUEST TO MEMORY
1822
WRITE CELL DATA AND OUTPUT PORT ID
1824
COUPLE CELL DATA AND MEDIA HEADER AND STREAM DATA
1826

FIG. 14


150
INPUT SWITCH 100
305
270
280
304
316
306
308
318
OUTPUT REQUEST PROCESSOR
HEAD AND TAIL QUEUE BUFFER
310
OUTPUT SWITCH 102
MEMORY
312
314
BUFFER
LINE OUTPUT
300
LINE INPUT
302
DATA HANDLER
270
INPUT SWITCH 100
GLOBAL DATA AREA (MEMORY BANK)
105
319
NOTIFICATION AREA
290
304