

**MECHANISM FOR PROVIDING EARLY COHERENCY DETECTION TO
ENABLE HIGH PERFORMANCE MEMORY UPDATES IN A LATENCY
SENSITIVE MULTITHREADED ENVIRONMENT**

CROSS REFERENCE TO RELATED APPLICATIONS

5 This application claims priority from U.S. Provisional
Patent Application Ser. No. 60/315,144 (Attorney Docket No.
10559-579P01), filed August 27, 2001.

BACKGROUND

10 In a pipelined processing environment, work arrives at
a fixed rate. For example, in a network processor
application, network packets may arrive every "n" ns. Each
arriving packet requires access to information stored in
memory (e.g., SRAM). Because the memory access speed is
slower than the arrival rate, a pipeline is used to process
15 the packets. The exit rate must match the arrival rate.

Each packet is classified into a flow. Successively
arriving packets may be from different flows, or from the
same flow. In the case of the same flow, processing steps
must be performed for each packet in strict arrival order.

20 In prior pipelined network processor implementations,
data rates and memory access speeds for "same flow" packet
processing are in a ratio such that the memory read access
time is not greater than the packet arrival rate. Thus, the



network processor cannot rely on a full pipeline rate without requiring faster memory access speeds.

The invention is defined in the independent claims. Some optional features of the invention are defined in the
5 dependent claims.

DESCRIPTION OF DRAWINGS

FIG. 1 is a block diagram of a communication system employing a processor having multithreaded microengines to
10 support multiple threads of execution.

FIG. 2 is a block diagram of a programmable processor datapath (of the microengine from FIG. 1) that includes a CAM.

FIG. 3 is a diagram depicting the microengines as a
15 multi-stage, packet processing pipeline.

FIG. 4 is a block diagram of the CAM of FIG. 2.

FIG. 5A is a depiction of a queue and queue descriptor in SRAM memory.

FIG. 5B is a depiction of a cache of queue descriptors
20 and corresponding tag store implemented using the CAM (of FIG. 4).

FIG. 6 is a flow diagram illustrating an exemplary use of the CAM during a queue operation by one of the microengines programmed to perform queue management.

FIG. 7 is a flow diagram illustrating an exemplary use
5 of the CAM to support Cyclic Redundancy Check (CRC) processing by one of the pipeline microengines programmed to perform CRC processing.

DETAILED DESCRIPTION

Referring to FIG. 1, a communication system 10 includes a processor 12 coupled to one or more I/O devices, for example, network devices 14 and 16, as well as a memory system 18. The processor 12 is multi-threaded processor and, as such, is especially useful for tasks that can be broken into parallel subtasks or functions. In one embodiment, as shown in the figure, the processor 12 includes multiple microengines 20, each with multiple hardware controlled program threads that can be simultaneously active and independently work on a task. In the example shown, there are sixteen microengines 20, microengines 20a-20p (corresponding to microengines 0 through 15), and each of the microengines 20 is capable of processing multiple program threads, as will be described more fully below. The maximum number of context threads supported in the illustrated embodiment is eight, but other maximum amount could be provided. Each of the microengines 20 is connected to and can communicate with adjacent microengines via next neighbor lines 21, as shown. In the illustrated embodiment, the microengines 0-7 are organized as a first cluster (ME Cluster 0) 22a and the microengines 8-15 are organized as a second cluster (ME Cluster 1) 22b.

The processor 12 also includes a processor 24 that assists in loading microcode control for other resources of

the processor 12 and performs other general purpose computer type functions such as handling protocols and exceptions, as well as provides support for higher layer network processing tasks that cannot be handled by the microengines. In one
5 embodiment, the processor 24 is a StrongARM (ARM is a trademark of ARM Limited, United Kingdom) core based architecture. The processor (or core) 24 has an operating system through which the processor 24 can call functions to operate on the microengines 20. The processor 24 can use
10 any supported operating system, preferably a real-time operating system. Other processor architectures may be used.

The microengines 20 each operate with shared resources including the memory system 18, a PCI bus interface 26, an
15 I/O interface 28, a hash unit 30 and a scratchpad memory 32. The PCI bus interface 26 provides an interface to a PCI bus (not shown). The I/O interface 28 is responsible for controlling and interfacing the processor 12 to the network devices 14, 16. The memory system 18 includes a Dynamic
20 Random Access Memory (DRAM) 34, which is accessed using a DRAM controller 36 and a Static Random Access Memory (SRAM) 38, which is accessed using an SRAM controller 40. Although not shown, the processor 12 also would include a nonvolatile memory to support boot operations. The DRAM 34 and DRAM
25 Controller 36 are typically used for processing large

volumes of data, e.g., processing of payloads from network packets. The SRAM 38 and SRAM controller 40 are used in a networking implementation for low latency, fast access tasks, e.g., accessing look-up tables, memory for the processor 24, and so forth. The SRAM controller 40 includes a data structure (queue descriptor cache) and associated control logic to support efficient queue operations, as will be described in further detail later. The microengines 20a-20p can execute memory reference instructions to either the DRAM controller 36 or the SRAM controller 40.

The devices 14 and 16 can be any network devices capable of transmitting and/or receiving network traffic data, such as framing/MAC devices, e.g., for connecting to 10/100BaseT Ethernet, Gigabit Ethernet, ATM or other types of networks, or devices for connecting to a switch fabric. For example, in one arrangement, the network device 14 could be an Ethernet MAC device (connected to an Ethernet network, not shown) that transmits packet data to the processor 12 and device 16 could be a switch fabric device that receives processed packet data from processor 12 for transmission onto a switch fabric. In such an implementation, that is, when handling traffic to be sent to a switch fabric, the processor 12 would be acting as an ingress network processor. Alternatively, the processor 12 could operate as an egress network processor, handling traffic that is

received from a switch fabric (via device 16) and destined for another network device such as network device 14, or network coupled to such device. Although the processor 12 can operate in a standalone mode, supporting both traffic directions, it will be understood that, to achieve higher performance, it may be desirable to use two dedicated processors, one as an ingress processor and the other as an egress processor. The two dedicated processors would each be coupled to the devices 14 and 16. In addition, each network device 14, 16 can include a plurality of ports to be serviced by the processor 12. The I/O interface 28 therefore supports one or more types of interfaces, such as an interface for packet and cell transfer between a PHY device and a higher protocol layer (e.g., link layer), or an interface between a traffic manager and a switch fabric for Asynchronous Transfer Mode (ATM), Internet Protocol (IP), Ethernet, and similar data communications applications. The I/O interface 28 includes separate receive and transmit blocks, each being separately configurable for a particular interface supported by the processor 12.

Other devices, such as a host computer and/or PCI peripherals (not shown), which may be coupled to a PCI bus controlled by the PC interface 26 are also serviced by the processor 12.

In general, as a network processor, the processor 12 can interface to any type of communication device or interface that receives/sends large amounts of data. The processor 12 functioning as a network processor could
5 receive units of packet data from a network device like network device 14 and process those units of packet data in a parallel manner, as will be described. The unit of packet data could include an entire network packet (e.g., Ethernet packet) or a portion of such a packet, e.g., a cell or
10 packet segment.

Each of the functional units of the processor 12 is coupled to an internal bus structure 42. Memory busses 44a, 44b couple the memory controllers 36 and 40, respectively, to respective memory units DRAM 34 and SRAM 38 of the memory
15 system 18. The I/O Interface 28 is coupled to the devices 14 and 16 via separate I/O bus lines 46a and 46b, respectively.

Referring to FIG. 2, an exemplary one of the microengines 20a is shown. The microengine (ME) 20a
20 includes a control store 50 for storing a microprogram. The microprogram is loadable by the processor 24.

The microengine 20a also includes an execution datapath 54 and at least one general purpose register (GPR) file 56 that are coupled to the control store 50. The datapath 54
25 includes several datapath elements, including an ALU 58, a

multiplier 59 and a Content Addressable Memory (CAM) 60.

The GPR file 56 provides operands to the various datapath processing elements including the CAM 60. Opcode bits in the instruction select which datapath element is to perform
5 the operation defined by the instruction.

The microengine 20a further includes a write transfer register file 62 and a read transfer register file 64. The write transfer register file 62 stores data to be written to a resource external to the microengine (for example, the
10 DRAM memory or SRAM memory). The read transfer register file 64 is used for storing return data from a resource external to the microengine 20a. Subsequent to or concurrent with the data arrival, an event signal from the respective shared resource, e.g., memory controllers 36, 40,
15 or core 24, can be provided to alert the thread that the data is available or has been sent. Both of the transfer register files 62, 64 are connected to the datapath 54, as well as the control store 50.

Also included in the microengine 20a is a local memory
20 66. The local memory 66 is addressed by registers 68a, 68b, which supplies operands to the datapath 54. The local memory 66 receives results from the datapath 54 as a destination. The microengine 20a also includes local control and status registers (CSRs) 70, coupled to the
25 transfer registers, for storing local inter-thread and

global event signaling information and other information,
and a CRC unit 72, coupled to the transfer registers, which
operates in parallel with the execution datapath 54 and
performs CRC computations for ATM cells. The microengine 20a
5 also includes next neighbor registers 74, coupled to the
control store 50 and the execution datapath 54, for storing
information received from a previous neighbor ME in pipeline
processing over a next neighbor input signal 21a, or from
the same ME, as controlled by information in the local CSRs
10 70.

In addition to providing an output to the write
transfer unit 62, the datapath can also provide an output to
the GPR file 56 over line 80. Thus, each of the datapath
elements, including the CAM 60 that can return a result
15 value from an executed. A next neighbor output signal 21b
to a next neighbor ME in the processing pipeline can be
provided under the control of the local CSRs 80.

Other details of the microengine have been omitted for
simplification. However, it will be appreciated that the
20 microengine would include (and the control store 50 would be
coupled to) appropriate control hardware, such as program
counters, instruction decode logic and context arbiter/event
logic, needed to support multiple execution threads.

Referring to FIG. 3, an exemplary ME task assignment
25 for a software pipeline model of the processor 12 is

illustrated in 90. The processor 12 supports two pipelines:
a receive pipeline and a transmit pipeline. The receive
pipeline includes the following stages: re-assembly pointer
search ("RPTR") 92, re-assembly information update ("RUPD")
5 94, receive packet processing (six stages) 96a-96f, metering
stages ME1 98 and ME2 100, congestion avoidance ("CA") 102,
statistics processing 104 and a queue manager ("QM") 106.
The receive pipeline begins with data arriving in a receive
block of the I/O interface 28 and ends with transmits queues
10 107 (stored in SRAM). The transmit pipeline stages include:
a TX scheduler 108, the QM 106, a Transmit Data stage 110
and the statistics processing 104.

The RPTR, RUPD and packet processing pipe stages work
together to re-assemble segmented frames back into complete
15 packets. The RPTR stage 92 finds the pointer to the
reassembly state information in the SRAM 38 and passes this
pointer to the RUPD 98. The RUPD 98 manages the reassembly
state, which involves allocating DRAM buffers, and
calculating offsets, byte counts and other variables, and
20 provides the packet processing stage 96 with a pointer to
the location in DRAM where the network data should be
assembled.

The threads of the packet processing stages 96 complete
the re-assembly process by writing the data (payload) to the
25 allocated DRAM buffer and also look at the L2 through L7

packet headers to process the packet. These stages are application dependent and can therefore vary from one application to another. For example, one application may support IP destination searches to determine destination
5 port, and a 7-tuple search to identify flows and support access lists.

To support ATM re-assembly, the RX pipeline requires a cyclic redundancy code (CRC) stage in addition to the pipe stages already described. CRC support can be provided by
10 replacing the first one of the packet processing stages (stage 96a, as shown) and including additional information in the re-assembly state table. The CRC 96a reads the re-assembly state to get the AAL type and CRC residue, verifies the Virtual Circuit (VC) is configured for AAL5, performs
15 CRC calculation over the cell, and updates the CRC residue in the re-assembly state.

Metering 98, 100 is used to monitor bandwidth of a flow. It checks whether each incoming packet is in profile or not. When a connection is made, a set of parameters are
20 negotiated, e.g., Committed Information Rate (CIR) and Committed Burst Size (CBS), which define the bandwidth used by the flow. The metering function can be implemented according to any one of a number of known schemes, such as token bucket.

Congestion avoidance 102 monitors network traffic loads in an effort to anticipate and avoid congestion at common network bottlenecks.

The QM 106 is responsible for performing enqueue and
5 dequeue operations on the transmit queues 107 for all packets, as will be described in further detail below.

The receive pipeline threads parse packet headers and perform lookups based on the packet header information. Once the packet has been processed, it is either sent as an
10 exception to be further processed by the core 24, or stored in the DRAM 34 and queued in a transmit queue by placing a packet link descriptor for it in a transmit queue associated with the transmit (forwarding port) indicated by the header/lookup. The transmit queue is stored in the SRAM 38.
15 The transmit pipeline schedules packets for transmit data processing, which then sends the packet out onto the forwarding port indicated by the header/lookup information during the receive processing.

Collectively, the stages 92, 94, and 96a-96f form a
20 functional pipeline. The functional pipeline uses 8 microengines (MEs) in parallel, and each of the eight threads (threads 0 through 7) in each ME is assigned a single packet for processing. Consequently, at any one time there are 64 packets in the pipeline. Each stage executes

at one packet arrival rate times execution period of eight threads.

The stages 98, 100, 102, 104, 106, 108 and 110 are context pipe-stages and, as such, are each handled by a single (different) ME. Each of the eight threads in each stage handles a different packet.

Some of the pipe stages, such as CRC 96a, RUPD 94, QM 106, for example, operate on a "critical section" of code, that is, a code section for which only one ME thread has exclusive modification privileges for a global resource at any one time. These privileges protect coherency during read-modify-write operations. Exclusive modification privileges between MEs are handled by allowing only one ME (one stage) to modify the section. Thus, the architecture is designed to ensure that an ME not transition into a critical section stage until a previous ME has completed its processing in the critical section. For example, the RUPD 98 is a critical section that requires mutual exclusivity to shared tables in external memory. Thus, when transitioning from RPTR 92 to RUPD 94, thread 0 of ME1 of the RUPD 94 will not begin until all threads on ME 0 have completed the previous RUPD pipe stage. In addition, strict thread order execution techniques are employed in the pipeline at critical section code points to ensure sequence management of packets being handled by the different threads.

The processor 12 also supports the use of caching mechanisms to reduce packet processing times and improve the speed at which the processor 12 operates with respect to incoming traffic. For example, the SRAM controller 40 (FIG. 1) maintains a cache of most recently used queue descriptors (stored in the SRAM 38), as will be further described. Also, the local memory 66 (FIG. 2) caches CRC information, such as CRC residue (also stored in the SRAM) 38, used by the CRC 96a. If more than one thread in a pipe stage such as the QM 106 is required to modify the same critical data, a latency penalty is incurred if each thread reads the data from external memory (that is, SRAM), modifies it and writes the data back to external memory. To reduce the latency penalty associated with the read and write, the ME threads can use the ME CAM 60 (FIG. 2) to fold these operations into a single read, multiple modifications and, depending on the cache eviction policy, either one or more write operations, as will be described.

FIG. 4 shows an exemplary embodiment of the CAM 60. The CAM 60 includes a plurality of entries 120. In the illustrated embodiment, there are 16 entries. Each entry 120 has an identifier value (or tag) 122, e.g., a queue number or memory address that can be compared against an input lookup value. As will be discussed later, each identifier value is associated with a stored unit of

information that is related to and used during packet processing, e.g., a queue descriptor, re-assembly state data, and so forth. Each entry also includes an entry number 124 and state information 126 associated with the identifier 122 in that same entry. Compare results 128 are provided to a Status and LRU logic unit 130, which produces a lookup result 132. The lookup result 132 includes a hit/miss indicator 134, state information 136 and an entry number 138. Collectively, the fields 134 and 136 provide status 140.

The width of the identifiers 122 is the same as the source registers being used to provide load the CAM entries or provide lookup values, e.g., the registers of the GPR file 56 (FIG. 3). In the embodiment shown, the state information 126 is implemented as a state bit. The width and format of the state information, and the number of identifiers are based on design considerations.

During a CAM lookup operation, the value presented from a source such as the GPR file 56 is compared, in parallel, to each identifier 122 with a resulting Match signal 142 per identifier. The values of each identifier were previously loaded by a CAM load operation. During that load operation, the values from the register file 56 specified which of the identifiers and the values of the identifiers to be loaded.

The state information is also loaded into the CAM during the CAM load operation.

The identifier 122 is compared against the lookup value in a source operand provided by an instruction, e.g.,

5 Lookup[dest_reg, src_reg].

The source operand specified by the parameter "src_reg" holds the lookup value to be applied to the CAM 60 for lookup. The destination register specified by parameter "dest_reg" is the register that receives the result of the
10 CAM lookup 60.

All entries 120 are compared in parallel. In one embodiment, the lookup result 132 is a 6-bit value which is written into the specified destination register in bits 8:3, with the other bits of the register set to zero. The
15 destination register can be a register in the GPR file 56. Optionally, the lookup result 132 can also be written into either of the LM_ADDR registers 68a, 68b (FIG. 2) of the ME 22.

For a hit (that is, when the hit/miss indicator 134 of
20 the result 132 indicates a hit), the entry number 138 is the entry number of the entry that matched. When a miss occurs and the hit/miss indicator 134 thus indicates a miss, the entry number 138 is the entry number of the Least Recently-Used (LRU) entry in the CAM array. The state information

136 is only useful for a hit and includes the value in the state field 126 for the entry that hit.

The LRU logic 130 maintains a time-ordered list of CAM entry usage. When an entry is loaded, or matches on a
5 lookup, it is moved to a position of Most Recently Used (MRU), a lookup that misses does not modify the LRU list.

All applications can use the hit/miss indication 134. The entry number 138 and state information 136 provide additional information that may be used by some
10 applications. On a miss, for example, the LRU entry number can be used as a hint for cache eviction. The software is not required to use the hint. The state information 136 is information produced and used only by software. It can
differentiate different meanings for a hit, such as
15 unmodified versus modified data. The software can use the information for branch decisions, as offset into data tables, among other uses.

Other instructions that use and manage the CAM can include:

20 Write [entry, src_reg], opt_tok;
 Write_State (state_value, entry) ;
 Read_Tag (dest_reg, entry);
 Read_State (dest_reg, entry); and
 Clear.

The Write instruction writes an identifier value in the src_reg to the specified CAM entry. An option token can be used to specify state information. The Read_Tag and Read_State instructions are used for diagnostics, but can also be used in normal functions. The tag value and state for the specified entry are written into the destination register. Reading the tag is useful in the case where an entry needs to be evicted to make room for a new value—that is, the lookup of the new value results in a miss, with the LRU entry number returned as a result of the miss. The read instruction can then be used to find the value that is stored in that entry. The Read_Tag instruction eliminates the need to keep the identifier value corresponding to the LRU entry number in another register. The Clear instruction is used to flush all information out of the CAM.

When the CAM is used as a cache tag store, and each entry is associated with a block of data in Local Memory 66, the result of the lookup can be used to branch on the hit/miss indicator 134 and use the entry number 138 as a base pointer into the block in Local Memory 66.

In another embodiment, the state 126 can be implemented as a single lock bit and the result 132 can be implemented to include a status code (instead of the separate indicator and state fields) along with the entry number 138. For example, the code could be defined as a two-bit code, with

possible results to include a "miss" (code '01'), "hit" (code '10') and "locked" (code '11'). A return of the miss code would indicate that the lookup value is not in the CAM, and the entry number of the result value is the Least Recently Used (LRU) entry. As discussed above, this value could be used as a suggested entry to be replaced with the lookup value. A hit code would indicate that the lookup value is in the CAM and the lock bit is clear, with the entry number in the result being the entry number of the entry that has matched the lookup value. A locked code would indicate that the lookup value is in the CAM and the locked bit 126 is set, with the entry number that is provided in the result again being the entry number of the entry that matched the lookup value.

The lock bit 126 is a bit of data associated with the entry. The lock bit could be set or cleared by software, e.g., using a LOCK or UNLOCK instruction, at the time the entry is loaded, or changed in an already loaded entry. The lock bit 126 can be used to differentiate cases where the data associated with the CAM entry is in flight, or pending a change, as will be discussed in further detail later.

As mentioned earlier, a context pipe stage that uses critical data is the only ME that uses that critical data. Therefore, the replacement policy for the CAM entries is to replace the LRU only on CAM misses. On the other hand, a

functional pipeline (like the pipeline 114 of FIG. 3) performs the same function on multiple MEs. In a functional pipeline, therefore, a given ME is required to evict all critical data to external memory before it exits a stage
5 that uses critical data and also must ensure that the CAM is cleared prior to any threads using the CAM.

Before a thread uses the critical data, it searches the CAM using a critical data identifier such as a memory address as a lookup value. As described earlier, the search
10 results in one of three possibilities: a "miss", a "hit" or a "lock". If a miss is returned, then data is not saved locally. The thread reads the data from external memory (that is, from the SRAM 38) to replace the LRU data. It evicts LRU data from local memory (SRAM controller cache, or
15 local memory 66) back to external memory, optionally locks the CAM entry and issues a read to get the new critical data from external memory. In certain applications, as will be described later, the lock is asserted to indicate to other threads that the data is in the process of being read into
20 local memory, or to indicate to the same thread (the thread that initiated the read) that the memory read is still in progress. Once the critical data is returned, the thread awaiting the data processes the data, makes any modifications to the data, writes it to local memory,

updates the entry from which LRU data was evicted with the new data and unlocks the CAM entry.

If the result is a lock, the thread assumes that another ME thread is in the process of reading critical data and that it should not attempt to read the data. Instead,
5 it tests the CAM at a later time and used the data when the lock is removed. When the result is a hit, then the critical data resides in local memory. Specific examples of CAM use will now be described with reference to FIGS. 5
10 through 8.

As discussed above, and as shown in FIG. 3, the processor 12 can be programmed to use one of the microengines 20 as the QM 106. The CAM 60 in the QM 106 serves as a tag store holding the tags of queue descriptors
15 that are cached by the SRAM controller 40.

The QM 106 receives enqueue requests from the set of microengines functioning as the receive functional pipeline 114. The receive pipeline 114 is programmed to process and classify data packets received by one of the network devices
20 14, 16 (FIG. 1), e.g., the physical layer device 14. The enqueue requests specify which output queue an arriving packet should be sent to. The transmit scheduler 108 sends dequeue requests to the QM 106. The dequeue requests specify the output queue from which a packet is to be

removed for transmittal to a destination via one of the network devices, 14, 16, e.g., the switch fabric 16.

An enqueue operation adds information that arrived in a data packet to one of the output queues and updates the
5 corresponding queue descriptor. A dequeue operation removes information from one of the output queues and updates the corresponding queue descriptor, thereby allowing the network device 16 to transmit the information to the appropriate destination.

10 Referring to FIG. 5A, an example of "n" transmit queues 150 and their corresponding queue descriptors 152 residing in external memory (SRAM 38) is shown. Each output queue 150 includes a linked list of elements 154, each of which has a pointer with the address of the next element in the
15 queue. Each element 154 also includes a pointer that points to information that is stored elsewhere and that the element represents. Typically, the pointer of the last element in the queue 150 contains a null value. The queue descriptor 152 includes an end of pointer EOP indicator 156, a segment
20 count 158, a head pointer 160, a tail pointer 162 and a frame count 164. The descriptor 152 may also include other queue parameters (not shown). The head pointer 160 points to the first element of the transmit queue 150, and the tail pointer 30 points to the last element of the transmit queue

150. The segment count 158 identifies the number of elements in the transmit queue 150.

Referring now to FIG. 5B, executing enqueue and dequeue operations for a large number of transmit queues 150 in the SRAM memory 38 at high-bandwidth line rates can be accomplished by storing some of the queue descriptors 152 in a cache 170 in the SRAM controller 40. The ME 20 executing as the queue manager 106 uses the identifiers 122 of the entries 120 in its CAM 60 to identify the memory addresses of the sixteen queue descriptors 152 most-recently-used in enqueue or dequeue operations, that is, the cached queue descriptors. The cache 170 stores the corresponding queue descriptors 152 (the EOP value 156, the segment count 158, the head pointer 160, tail pointer 162 and the frame count 164) stored at the addresses identified in the tag store (CAM 60).

The queue manager 106 issues commands to return queue descriptors 152 to memory 38 and fetch new queue descriptors 152 from memory such that the queue descriptors stored in the cache 170 remain coherent with the addresses in the tag store 60. The queue manager 106 also issues commands to the SRAM controller 38 to indicate which queue descriptor 152 in the cache 170 should be used to execute the command. The commands that reference the head pointer 160 or tail pointer

162 of a queue descriptor 152 in the cache 170 are executed in the order in which they arrive at the SRAM controller 38.

Locating the cache 170 of queue descriptors 152 at the memory controller 40 allows for low latency access to and from the cache 170 and the memory 38. Also, having the control structure for queue operations in a programming engine can allow for flexible high performance while using existing micro-engine hardware.

The threads associated with the QM 106 execute in strict order. The threads use local inter-thread signaling to maintain strict order. To ensure that the QM 106 keeps up with in an incoming line rate, each thread performs one enqueue and one dequeue operation in a time slot equal to the minimum frame arrive time.

FIG. 6 illustrates an exemplary queue operation 180 (representing either an enqueue or dequeue operation) performed by the QM 106. The QM 106 receives 182 a request for a queue operation 182. The request is received from the CA content pipestage ME when it is an enqueue request and is received from the TX scheduler content pipe-stage ME when it is request for a dequeue operation. The QM 106 reads 184 a queue number from the request.

The QM 106 then uses its CAM to detect temporal dependencies between the queue specified in the request and the last 16 queues to which the QM 106 performed such an

operation. Thus, the QM 106 performs a CAM lookup 186 based on the queue number identified in the request. If there is a dependency, i.e., the QM thread detects 188 a CAM hit, the latency of reading a queue descriptor is eliminated because the CAM hit indicates that the descriptor corresponding to the queue number is currently maintained in the queue descriptor cache 170 (FIG. 5B). In the event that a hit occurs, the QM 106 proceeds to execute an instruction 190 that commands the SRAM controller 40 to perform the requested operation.

If, at 188, it is determined that the CAM search results in a miss, the entry number of the least recently used CAM entry is returned to the QM 106. There is a direct mapping between the CAM entry and a cache entry (queue descriptor). In other words, an LRU CAM entry "n" indicates that the cache entry "n" should be evicted. Therefore, the QM 106 evicts 192 from the cache the queue descriptor corresponding to the queue number stored in the LRU CAM entry. Once the cache entry is evicted, the QM 106 reads 194 the "new" queue descriptor (that is, the queue descriptor of the queue number in the request) into the cache from the SRAM. The new queue descriptor includes the linked list head pointer (for dequeue) and tail pointer (for enqueue), and a count that indicates the number of frames or buffers on the queue (as shown in FIGS. 5A-5B). The QM 106

also stores 196 the queue number of the new queue descriptor in the CAM entry that had been identified as the LRU entry to replace the number of the evicted queue descriptor. The QM 106 executes an instruction 190 that commands the SRAM
5 controller 40 to perform the requested operation.

The SRAM controller 40 performs the linked list operation for enqueue or dequeue.

When an operation of either type (enqueue or dequeue) is performed, the QM 106 sends a message to the TX scheduler
10 108. After a dequeue operation, the QM 106 passes a transmit request to the TX data context pipe-stage 110.

Another stage that uses the CAM 60 is the CRC processing pipe stage 96a. The ME 20 in this stage of the receive functional pipeline 114 uses its internal CAM 60 to
15 maintain coherency of the CRC residue (in the re-assembly state table) between the eight threads executing the CRC processing pipe stage 96a.

Referring now to FIG. 7, a CRC pipe-stage program flow 200, including the use of the CAM 60 in support of the
20 function, is shown. The CRC stage 96a is entered only when the previous ME has indicated (via the next neighbor line 21a (FIG. 2)) that it has exited the stage. This ensures that the ME will access the most recent critical data (CRC residue). It is also critical that, throughout this pipe-
25 stage, all threads execute in strict order to ensure that

the CRC is calculated correctly. Because the CRC stage 96a uses the CAM 60, it firsts clears 202 the CAM of any data still in the CAM from a previous pipe-stage. It reads 204 the port type and determines 206 if it has been assigned an ATM cell. If the cell is not an ATM cell (that is, it is some other type, such as Ethernet or POS), the ME performing the CRC stage passes 208 the cell through without any processing. If the cell is an ATM cell, the ME 20 performs the CRC processing.

10 The processing includes the following activities:
reading the CRC residue, ATM type and SOP/EOP state in SRAM;
determining if the cell is carrying an SOP, body or EOP;
validating that the VC is carrying AAL5 cells and, if so,
performing the CRC computation; and updating CRC residue and
15 EOP-SOP status in SRAM.

 The CRC computation is performed using the CRC unit 72 (FIG. 2) in the ME 20. The CRC computation must be performed in strict order to ensure that the CRC for cells that belong to the same VC are computed with the correct CRC
20 residue.

 The CRC processing is divided into a read phase and a modify/write phase. The CAM 60 is used in both phases. In the first phase, the CAM 60 is used to decide whether a thread should read the residue/type fields from SRAM 38 or
25 use the result from a previous thread stored in the Local

Memory 66 (FIG. 2). The first phase begins with a given thread searching the CAM 210 using the pointer to the re-assembly state. If the thread detects 212 a CAM miss, the thread writes 214 a CAM entry with the re-assembly pointer and state information to lock the entry, and issues a read to obtain the CRC residue and AAL type from SRAM memory 38. If, at 212, the thread detects a hit, it does not issue a read.

When the thread receives 216 the appropriate event signaling, that is, an event signal indicating that the previous thread has completed processing, the thread wakes and begins phase 2 processing. It searches 218 the CAM using the same re-assembly pointer. If the thread had issued a read and determines 220 a locked status for a matched CAM entry, the thread moves 222 the read result in the transfer registers to the local memory. The thread that moves the result also unlocks the entry, thereby ensuring a hit for future CAM lookups for that particular pointer. Otherwise, if the CAM entry is not locked, then a hit has occurred, and the thread simply reads 224 the corresponding information, that is, the residue and type, from the Local Memory.

After the second phase CAM search, each thread validates that the VC is carrying AAL5 by examining the type field from the VC table. For an AAL5 type, the thread

computes 226 the CRC over the cell. If the type is not AAL5, the cell is handed off to an exception handler, or discarded, depending on the implementation.

If the thread determines 228 that the PTI bits in the
5 ATM header indicate that the cell is an EOP cell, the thread updates 230 the re-assembly state by setting the CRC residue to all zeroes and setting the SOP bit to a one. If the cell is not an EOP cell, the thread updates 232 the state with the new residue and sets SOP to zero. It saves 235 the
10 updated CRC residue and SOP in the Local Memory for use by other threads and, according to its writeback cache policy, also writes the CRC residue and SOP back to the re-assembly state in the SRAM 38. The thread passes 236 the SOP, EOP and body status to the next (packet processing) stage.

15 It is important that other stages in the RX pipeline know if the ATM cell contains an EOP, SOP or body. For ATM, the settings of the SOP and EOP bit indicate whether an entire cell was received (as opposed to an entire packet), so the CRC threads must use the EOP bit status provided in
20 the header PTI field. The PTI bits only support EOP, so when an EOP is detected, the CRC thread sets an SOP bit in its section of the re-assembly state table indicating to the next thread that it has an SOP. Each time the CRC thread reads the re-assembly state, it reads the SOP bit, and if it

is set, and the PTI bits in the ATM header indicate no EOP, it clears the SOP bit.

Because other stages do not read the CRC threads re-assembly state area, the CRC thread also passes the EOP/SOP status down the pipeline. Once the CRC threads have completed the CRC calculation and the re-assembly state table is updated, the threads are ready to move onto the next pipe-stage.

When a thread completes its CRC calculation and issues its SRAM write of the residue/type, it also signals the thread of the next ME indicating that it can start its CRC pipe-stage. It is important that the signaling ensures that the next ME is not provided a signal until it can be assured that any pending residues will be written before the next ME issues its residue reads.

It will be understood that, while the implementation described thus far uses the CAM 60 to reduce the number of read accesses (via "folding", as discussed earlier), the strict sequential ordering of the execution of context threads in a given stage is maintained not through the use of CAM, but instead by using local inter-thread signaling and by ensuring that read reference and modification activity completes before that same data is needed by successive threads.

It will be appreciated, however, that the CAM 60 could be used to maintain coherency and correct packet processing sequence as well. For example, say threads are handling two successive packets that are in the same flow (or are
5 associated with the same queue number) and access the same SRAM location. Because packet arrival rates are faster than SRAM access speeds, the thread handling the second packet will be ready to access the data before the SRAM read and modify activities of the thread handling the first (earlier)
10 packet have completed. In this situation, the software-controlled CAM cache implementation can be used to recognize the dependency and to ensure that the most current information is always used. Thus, each thread uses the CAM 60 to do multiple compares in parallel using the CAM Lookup
15 instruction, with a source register providing the flow number or queue number as the lookup value, as described earlier.

If a miss results, the thread commences the SRAM read and allocates a CAM entry into which the thread places the
20 flow number. If the flow is already in the CAM, a hit indicator is returned along with a unique pointer value (for example, which entry number in the CAM matched). The thread that gets a hit in the CAM can obtain the latest copy of the data from local memory (cache in SRAM controller 40, or ME
25 Local Memory 66) without having to do an SRAM read.

When a thread loads a flow number into a CAM entry, it also stores state information in the entry to enable subsequent thread lookups to determine that either a) the SRAM read has been started, but is not yet completed (it is
5 "in-flight"); or b) the SRAM read has been completed, and the data is valid. If the "in-flight" status is determined, the subsequent thread knows that it should not start a read, but that it cannot yet use the read data. It can continue to test the status of the entry until it determines that the
10 status has been changed to reflect valid data.

Other embodiments are within the scope of the following claims.

What is claimed is:

1. A method of accessing shared-access information stored in external memory, the method comprising:

5 storing copies of information units related to packet processing from the external memory to local memory, each information unit having an associated identifier;

maintaining each identifier as an entry in a Content Addressable Memory (CAM), each entry including status
10 information associated with the information unit with which the identifier is associated; and

using the status information to determine validity of the information unit with which the status information is associated and to protect coherency during read-modify-write
15 operations by enforcing exclusive modification privileges between microengines;

wherein the status information comprises a lock status to indicate that the information unit with which the status information is associated is in process of being read from
20 the external memory to the local memory or in process of being modified in the local memory.

2. The method of claim 1 wherein storing copies of the information units comprises storing copies of the
25 information units in an external memory controller cache, the cached information units collectively corresponding to a portion of all such information units stored in the external memory.

30 3. The method of claim 2 wherein the information units are queue descriptors.

4. The method of claim 3 wherein the associated identifiers are queue numbers.

5 5. The method of claim 1 wherein the information units correspond to information in state tables.

6. The method of claim 5 wherein the associated identifiers are packet flow identifiers.

10

7. The method of claim 1, further comprising:

performing a lookup in the CAM of a selected one of the information units stored in the external memory based on the associated identifier; and

15 receiving from the CAM a lookup result that indicates if a match was found, a match indicating that the selected one of the information units is stored in the local memory.

8. The method of claim 7 wherein the lookup result
20 includes the status information of the matched identifier.

9. The method of claim 7 wherein the CAM maintains a Least Recently Used (LRU) list of the identifiers in the CAM and, if no match is found, the result providing an index to an
25 identifier from the LRU list.

10. The method of claim 9, further comprising using the LRU identifier to replace a least recently used information unit in the local memory with the selected one of the information
30 units from the external memory.

11. The method of claim 10, further comprising:

replacing the LRU identifier in the CAM with the identifier associated with the selected one of the information units.

5 12. The method of claim 7, further comprising:

setting the lock status of the identifier associated with the selected one of the information units to indicate invalid status;

10 modifying the selected one of the information units in the local memory; and

upon completion of the modification, changing the lock status to indicate valid status.

13. A computer program product residing on a computer-readable medium comprising instructions to cause a computer to:

store copies of information units related to packet processing from external memory to local memory, each information unit having an associated identifier;

20 maintain each identifier as an entry in a Content Addressable Memory (CAM), each entry including status information associated with the information unit with which the identifier is associated; and

25 use the status information to determine validity of the information unit with which the status information is associated and to protect coherency during read-modify-write operations by enforcing exclusive modification privileges between microengines;

30 wherein the status information comprises a lock status to indicate that the information unit with which the status information is associated is in process of being read from

the external memory to the local memory or of being modified in the local memory.

14. The computer program product of claim 13, wherein the
5 information units stored in the local memory comprise information units stored in an external memory controller cache, the cached information units collectively corresponding to a portion of all such information units stored in the external memory.

10

15. An apparatus comprising:

an external memory controller;

a local memory; and

a Content Addressable Memory (CAM) configured to
15 maintain entries including identifiers and associated status information, the identifiers being associated with units of information related to packet processing that are read from external memory to the local memory, and the apparatus being configured to cause the status information to indicate that
20 a unit of information, which has been associated with an entry of the CAM, is currently invalid when the unit of information is in process of being read from external memory to the local memory, and the apparatus being configured to protect coherency during read-modify-write operations by
25 enforcing exclusive modification privileges between microengines using the status information.

16. The apparatus of claim 15, wherein the external memory controller comprises an external memory controller cache,
30 and the local memory comprises the external memory controller cache.

17. The apparatus of claim 15, wherein the status information comprises a lock status to indicate that the information unit with which the status information is associated is in process of being read from the external memory to the local memory or of being modified in the local memory.

18. A system comprising:
multiple network devices;
10 external memory; and
a processor comprising multiple microengines, an external memory controller, a local memory, and a Content Addressable Memory (CAM) configured to maintain entries including identifiers and associated status information, the
15 identifiers being associated with units of information related to packet processing that are read from the external memory to the local memory, and the processor being configured to cause the status information to indicate that
20 a unit of information, which has been associated with an entry of the CAM, is currently invalid when the unit of information is in process of being read from external memory to the local memory.

19. The system of claim 18, wherein the external memory
25 controller comprises an external memory controller cache, and the local memory comprises the external memory controller cache.

20. The system of claim 18, wherein the status information
30 comprises a lock status to indicate that the information unit with which the status information is associated is in

process of being read from the external memory to the local memory or of being modified in the local memory.