

Exhibit 596-06

Invalidity of the '596 Patent by Gewirtz '227 and Other Prior Art

The asserted claims of U.S. Patent No. 8,924,596 are anticipated and/or rendered obvious in view of US8051227B1 (“Gewirtz '227”) alone or in combination with knowledge of one of skill in the art or other prior art.

This claim chart is based on Defendants’ present understanding of the asserted claims, Plaintiffs’ vague and overbroad infringement contentions (and corresponding, implicit claim constructions), and Defendants’ investigation to date. Defendants are not adopting Plaintiffs’ apparent constructions, nor are Defendants admitting to the correctness of any particular construction. Defendants reserve all rights to amend this invalidity claim chart in light of any claim construction developments or any amendments to Plaintiffs’ infringement contentions. Further, as discovery is ongoing and Defendants continue to seek discovery from third parties regarding the references identified in Defendants’ invalidity contentions as well as other potential prior art, Defendants reserve the right to amend or supplement this claim chart in connection with Defendants’ continuing investigation. At a minimum, Defendants intend to revise these invalidity contentions as appropriate based on discovery from third parties as discovery progresses.

The claim chart below identifies where each limitation of each asserted claim of the '596 patent can be found in Gewirtz '227. This analysis is based on Plaintiff’s interpretation of the claims as indicated in their Infringement Contentions. The citations provided below are exemplary, rather than exhaustive, and Defendants reserve the right to rely upon any other portion of the cited references.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
[1pre] A system for dividing and synchronizing a processing task across a plurality of processing elements comprising:	To the extent the preamble is considered a claim limitation, Gewirtz '227 teaches “[a] system for dividing and synchronizing a processing task across a plurality of processing elements,” including at least the following exemplary citations. For example, Gewirtz '227 teaches a system for dividing and synchronizing processing tasks by disclosing, as non-limiting examples, software agents in a multiprocessor system that divide and synchronize processing tasks. <i>See, e.g.:</i> “A command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around the pointer if the pointer reaches an end of the first queue based on the queue size.” Gewirtz '227 at Abstract.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	FIG. 1 Gewirtz '227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301 Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302 Atomically read a pointer from a hardware-based queue state register associated with the queue. 303 Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304 Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled 500, illustrates a system architecture. At the top, a block labeled 'Agent(s) 120' is connected via a bidirectional arrow to a vertical stack of 'Access Register' blocks. This stack is divided into two sections: section 505 contains 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, and 'Access Register for Qn'; section 506 contains 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. This entire stack is labeled 'Memory Mapped Registers 101'. To the right of this stack is a central block containing a 'Scanner 504' and an 'Address Decoder 104', connected by a bidirectional arrow. Below this central block is another bidirectional arrow connecting to a block containing a 'Configuration Register 105' and a 'Queue State Register 106'. This lower block is part of a larger structure labeled 'HW-Based Registers 102'. To the right of the central block is a vertical stack of 'Priority Group' blocks, labeled 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. This stack is labeled 'Memory 103'. Bidirectional arrows connect the central 'Scanner/Address Decoder' block to both the 'Memory Mapped Registers 101' and the 'Priority Group' stack. The entire diagram is labeled 'FIG. 5'. Gewirtz '227 at Figure 5.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically read data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6.

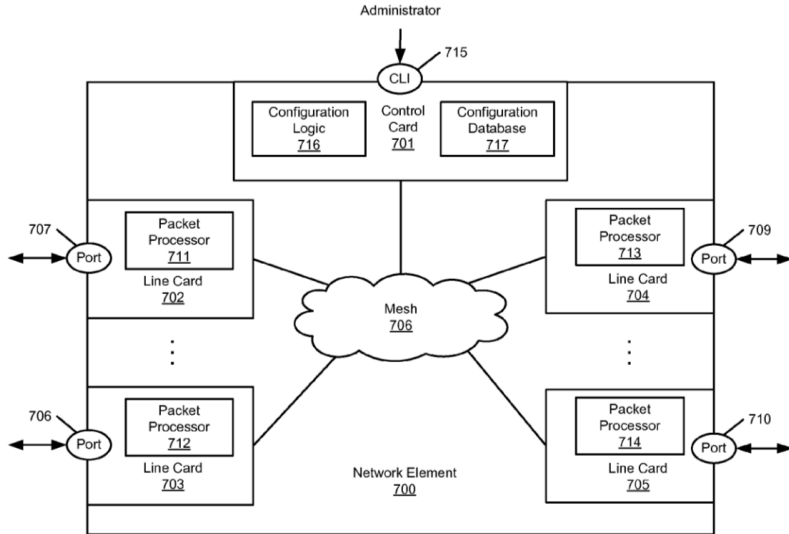
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 7 Gewirtz '227 at Figure 7. “According to one aspect of the invention, a command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register, where the first queue was allocated in a memory. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the first queue. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing, via a first increment/decrement hardware logic, the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around, via the first increment/decrement hardware logic, the pointer of the first hardware-based queue state register if the pointer reaches an end of the first queue based on the queue size. As a result, the pointer is obtained and updated atomically through hardware so that the software process represented by the first agent is not required to obtain an exclusive access control of the first queue from the Operating System.” Gewirtz '227 at col. 1, ll. 39–64.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“FIG. 1 is a block diagram illustrating a hardware-controlled queuing system according to one embodiment of the invention. Referring to FIG. 1, system 100 includes, but not limited to, one or more memory mapped registers 101, one or more hardware-based registers 102, and a memory or memory section 103 (e.g., part of main memory) defining one or more queues (e.g., logical queues) coupled to each other via an address decoder 104. Memory mapped registers 101 (also referred to as data access registers or simply access registers) include one or more access registers for each queue defined in memory 103.” Gewirtz ’227 at col. 3, ll. 8–18. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[1a] an input queue implemented in hardware;	Gewirtz ’227 teaches “an input queue implemented in hardware,” including at least the following exemplary citations. Plaintiffs have cited the use of an input queue implemented in memory through software as allegedly satisfying limitation [1a]. <i>See</i> Am. Compl. at ¶59; ’596 Infringement Contentions at 43–58. To the extent that Plaintiffs argue that Gewirtz ’227 ‘s input queue is in memory Gewirtz ’227 teaches or renders obvious claim limitation [1a] under Plaintiffs’ apparent interpretation of the claim. For example, Gewirtz ’227 teaches an input queue implemented in hardware, by disclosing, as non-limiting examples, an input queue that uses memory to store the queue’s contents. <i>See, e.g.:</i> “A command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer. The pointer stored in the first hardware-based queue state register is atomically

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	updated, including incrementing the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around the pointer if the pointer reaches an end of the first queue based on the queue size.” Gewirtz ’227 at Abstract. FIG. 1 Gewirtz ’227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	<pre> graph TD 301[Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element.] --> 302[Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory.] 302 --> 303[Atomically read a pointer from a hardware-based queue state register associated with the queue.] 303 --> 304[Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue.] 304 --> 305[Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size.] </pre> 300 FIG. 3 Gewirtz '227 at Figure 3.

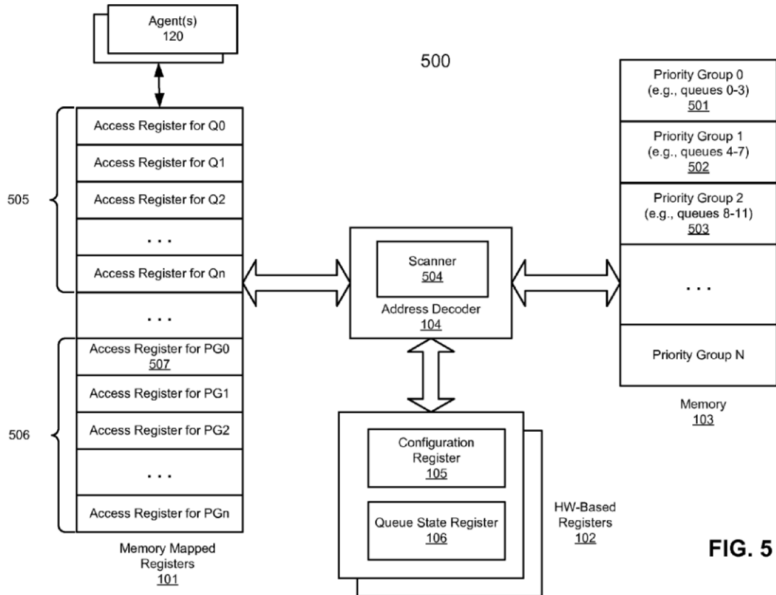
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 Gewirtz '227 at Figure 5. “FIG. 1 is a block diagram illustrating a hardware-controlled queuing system according to one embodiment of the invention. Referring to FIG. 1, system 100 includes, but not limited to, one or more memory mapped registers 101, one or more hardware-based registers 102, and a memory or memory section 103 (e.g., part of main memory) defining one or more queues (e.g., logical queues) coupled to each other via an address decoder 104. Memory mapped registers 101 (also referred to as data access registers or simply access registers) include one or more access registers for each queue defined in memory 103.” Gewirtz '227 at col. 3, ll. 8–18. “In one embodiment, a programmable queue includes a configuration register 105, a queue state register 106, storage that is allocated for the queue elements (e.g., corresponding memory segment of memory 103), and data access registers (e.g., corresponding memory mapped registers 101). Configuration software can set up a queue by writing the corresponding configuration register 105 and enabling the queue as well as defining its behavior. Once the queue is programmed, any other agent (e.g., agents 120) in the system can read or write (e.g., push or pop) from or to the queue by issuing standard reads and write commands (loads and stores) to the corresponding memory mapped registers 101.” Gewirtz '227 at col. 3, ll. 43–54.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“In one embodiment, configuration register 105 defines the operation of the queue including parameters, such as, for example, base memory address 107, element or entry size 108, queue depth 109, and options (e.g., flow control options, processor interrupt options, and prioritization options, etc.) Base memory address 107 is a memory pointer pointing to the beginning of the allocated queue storage. Entry size 108 and queue size 109 define the size of the queue. The flow control option enables a built-in credit counter for each queue. The processor interrupt options enable the queue hardware to generate processor interrupts to a programmable target whenever certain queue events are detected. For example, an interrupt can be generated when a queue goes from empty to non-empty; drains below a predetermined threshold; or fills above a predetermined threshold, etc. The priority option allows queues to be grouped into priority groups, which will be described in details further below.” Gewirtz ’227 at col. 3, l. 55 – col. 4, l. 4. “In one embodiment, state register 106 includes read pointer 111, write pointer 112, a number of occupied entries 113, and available entries for reservation. Read pointer 111 points to a head of a queue for reading a next queue element or data entry. Write pointer 112 points to a tail of the queue for writing new data into the queue. Number of occupied entries 113 includes data representing the number of entries currently stored in the queue. The reservation field can be used to specify the number of entries available for writing into the queue. All fields of state register 106 are maintained by hardware logic, for example, atomically.” Gewirtz ’227 at col. 4, ll. 5–15. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“According to one embodiment, when an agent such as agent 120 desires to write data to a specific queue, the agent may specific perform a write operation to an access register associated with the specific queue. For example, if agent 120 desires to write data to queue 0, it may write data to access register 115. In response, address decoder 104 may access a set of hardware-based registers associated with the queue being accessed (e.g., hardware-based registers associated with queue 0). In this example, decoder 104 may atomically obtain a write pointer 112 from queue state register 106 associated with the queue being accessed. In addition, decoder 104 may obtain base address 107 and entry size 108 from configuration register 105 associated with the queue being accessed. A memory address for the entry to be written is determined based on the write pointer 112, base address 107, and entry size 108. In one embodiment, the memory address of the entry being written can be determined as follows: [Equation] where (Write Pointer*Entry Size) represents an offset from the base address.” Gewirtz ’227 at col. 5, l. 50 – col. 6, l. 4. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[1b] an output queue implemented in hardware;	Gewirtz ’227 teaches “an output queue implemented in hardware,” including at least the following exemplary citations. Plaintiffs have cited the use of an output queue implemented in memory through software as allegedly satisfying limitation [1b]. <i>See</i> Am. Compl. at ¶59; ’596 Infringement Contentions at 58–72. To the extent that Plaintiffs argue that Gewirtz ’227’s output queue is in memory Gewirtz ’227 teaches or renders obvious claim limitation [1b] under Plaintiffs’ apparent interpretation of the claim. For example, Gewirtz ’227 teaches an output queue implemented in hardware, by disclosing, as non-limiting examples, an output queue that uses memory to store the queue’s contents. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	FIG. 1 Gewirtz '227 at Figure 1.

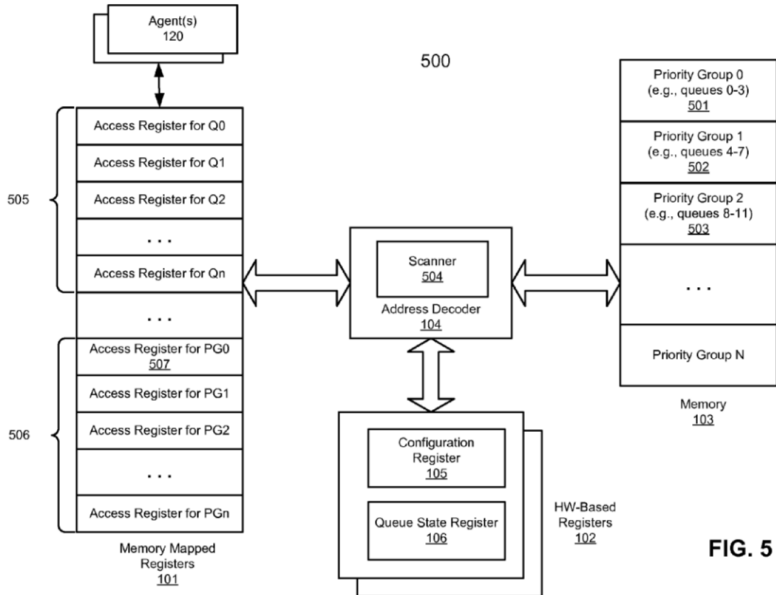
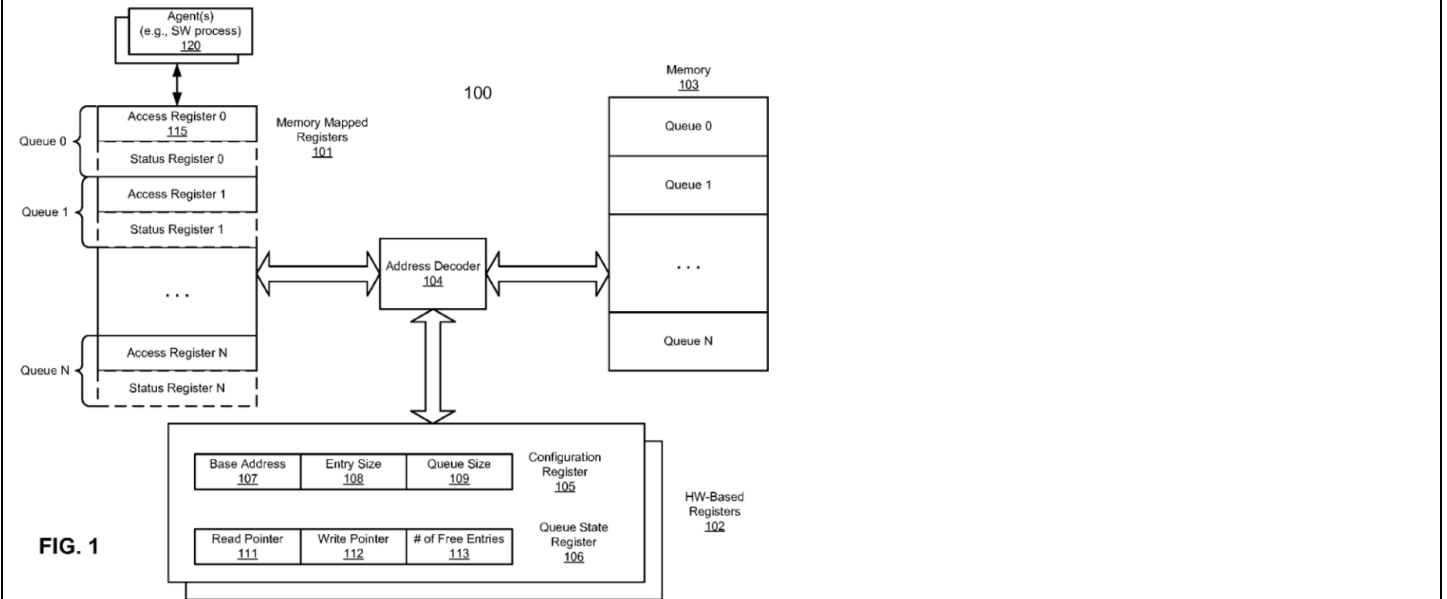
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 Gewirtz '227 at Figure 5. “FIG. 1 is a block diagram illustrating a hardware-controlled queuing system according to one embodiment of the invention. Referring to FIG. 1, system 100 includes, but not limited to, one or more memory mapped registers 101, one or more hardware-based registers 102, and a memory or memory section 103 (e.g., part of main memory) defining one or more queues (e.g., logical queues) coupled to each other via an address decoder 104. Memory mapped registers 101 (also referred to as data access registers or simply access registers) include one or more access registers for each queue defined in memory 103.” Gewirtz '227 at col. 3, ll. 8–18. “In one embodiment, a programmable queue includes a configuration register 105, a queue state register 106, storage that is allocated for the queue elements (e.g., corresponding memory segment of memory 103), and data access registers (e.g., corresponding memory mapped registers 101). Configuration software can set up a queue by writing the corresponding configuration register 105 and enabling the queue as well as defining its behavior. Once the queue is programmed, any other agent (e.g., agents 120) in the system can read or write (e.g., push or pop) from or to the queue by issuing standard reads and write commands (loads and stores) to the corresponding memory mapped registers 101.” Gewirtz '227 at col. 3, ll. 43–54.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“In one embodiment, configuration register 105 defines the operation of the queue including parameters, such as, for example, base memory address 107, element or entry size 108, queue depth 109, and options (e.g., flow control options, processor interrupt options, and prioritization options, etc.) Base memory address 107 is a memory pointer pointing to the beginning of the allocated queue storage. Entry size 108 and queue size 109 define the size of the queue. The flow control option enables a built-in credit counter for each queue. The processor interrupt options enable the queue hardware to generate processor interrupts to a programmable target whenever certain queue events are detected. For example, an interrupt can be generated when a queue goes from empty to non-empty; drains below a predetermined threshold; or fills above a predetermined threshold, etc. The priority option allows queues to be grouped into priority groups, which will be described in details further below.” Gewirtz ’227 at col. 3, l. 55 – col. 4, l. 4. “In one embodiment, state register 106 includes read pointer 111, write pointer 112, a number of occupied entries 113, and available entries for reservation. Read pointer 111 points to a head of a queue for reading a next queue element or data entry. Write pointer 112 points to a tail of the queue for writing new data into the queue. Number of occupied entries 113 includes data representing the number of entries currently stored in the queue. The reservation field can be used to specify the number of entries available for writing into the queue. All fields of state register 106 are maintained by hardware logic, for example, atomically.” Gewirtz ’227 at col. 4, ll. 5–15. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“The memory address is used to read data from the corresponding queue in memory 103. Thereafter, read pointer 111 may be atomically updated via hardware logic (e.g., decoder 104), for example, by incrementing or decrementing read pointer 111 dependent upon the specific implementation of the queue. When read pointer 111 is updated, in this example, being incremented, hardware logic may also determine whether read pointer 111 has reached the bottom or top of the queue, based on base address 107 and queue size 109 obtained from configuration register 105. If read pointer 111 does reach the top or bottom of the queue, read pointer 111 may be wrapped around or reset to a predetermined value such as zero. In addition, the number of free entries field 113 may also be updated by hardware logic, for example, by incrementing field 113, since an entry has been popped from the queue. These operations are performed atomically and controlled by hardware logic or logics. As a result, the agent does not have to worry about the exclusive access to the queues. It will be appreciated that certain firmware or low level software may also be utilized in conjunction with hardware logics.” Gewirtz ’227 at col. 5, ll. 29–49. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[1c] a first processing element having access to said input queue and said output queue;	Gewirtz ’227 teaches “a first processing element having access to said input queue and said output queue,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches a first processing element having access to said input queue and said output queue, by disclosing, as non-limiting examples, a software agent running on a processor. The software agent running on the processor has access to both the input queue and output queue. <i>See, e.g.:</i> “A command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	memory based on the pointer. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around the pointer if the pointer reaches an end of the first queue based on the queue size.” Gewirtz ’227 at Abstract.  FIG. 1 Gewirtz ’227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system architecture 500. At the top, an 'Agent(s) 120' is connected via a bidirectional arrow to a set of 'Memory Mapped Registers 101'. This set includes a group of registers 505 (Access Register for Q0, Q1, Q2, ..., Qn) and another group 506 (Access Register for PG0, PG1, PG2, ..., PGn). These registers are connected to a central 'Scanner 504' and 'Address Decoder 104' block. Below this central block is a stack of 'HW-Based Registers 102', which includes a 'Configuration Register 105' and a 'Queue State Register 106'. The central block is also connected to a 'Memory 103' structure on the right, which is organized into 'Priority Groups' (e.g., Priority Group 0, 1, 2, ..., N). The entire system is labeled 500. Gewirtz '227 at Figure 5.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6.

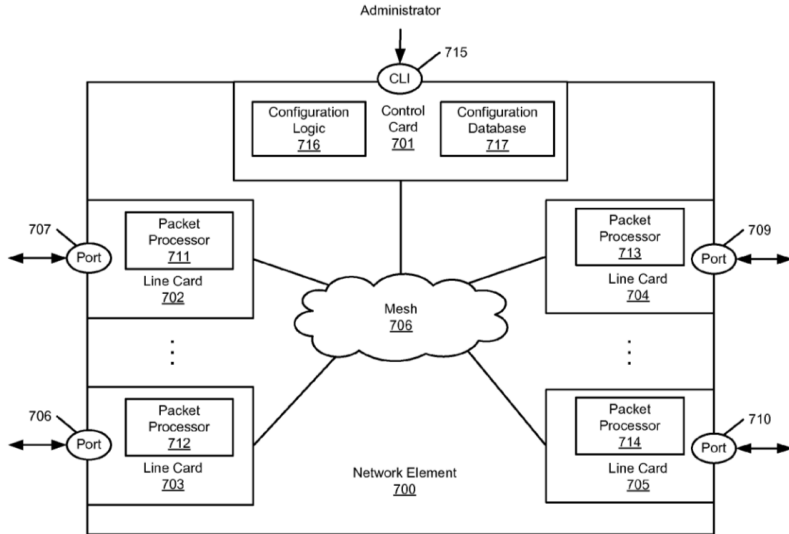
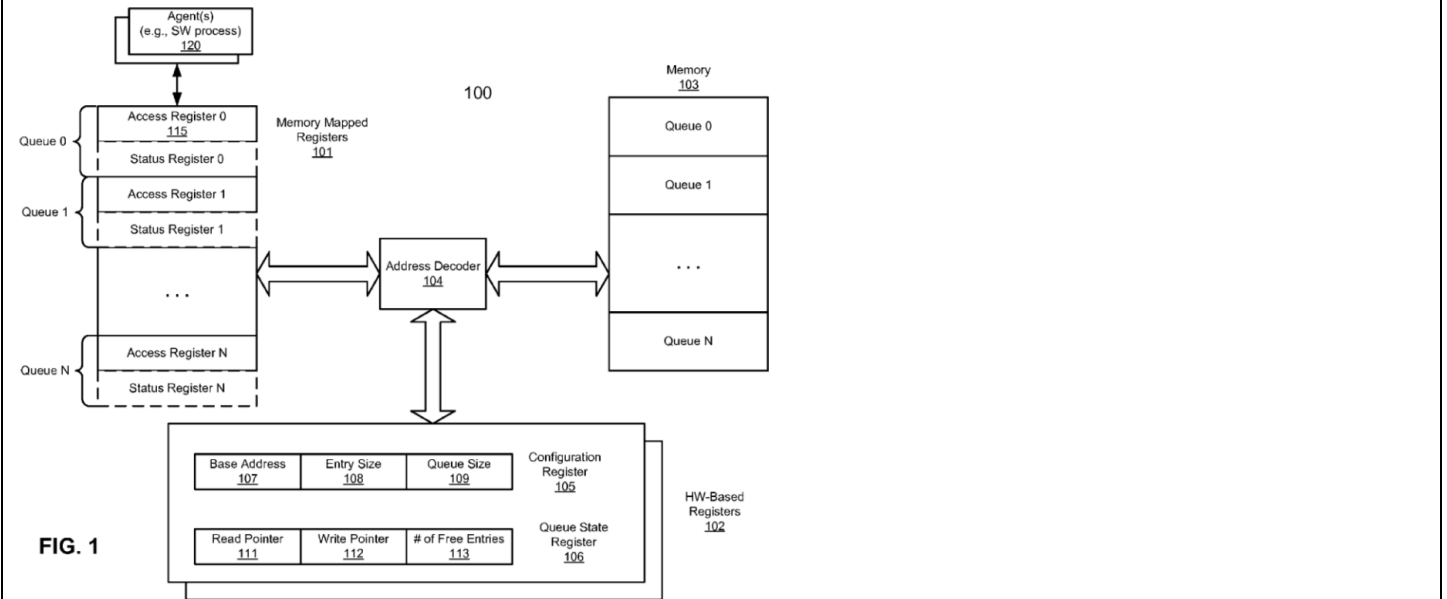
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 7 Gewirtz '227 at Figure 7. “In a multi-processor system, it is frequently desirable to build queues, or first-in-first-out (FIFO) buffers, in memory. A queue is a data structure with FIFO properties where agents push elements onto the queue (writes) and pop elements off of the queue (reads). This is normally accomplished by allocating storage for the queue elements, and separately allocating storage for the queue state which includes read and write pointers, and any other desired information.” Gewirtz '227 at col. 1, ll. 12–19. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz '227 at col. 4, ll. 16–33.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“According to one embodiment, when an agent such as agent 120 desires to write data to a specific queue, the agent may specific perform a write operation to an access register associated with the specific queue. For example, if agent 120 desires to write data to queue 0, it may write data to access register 115. In response, address decoder 104 may access a set of hardware-based registers associated with the queue being accessed (e.g., hardware-based registers associated with queue 0). In this example, decoder 104 may atomically obtain a write pointer 112 from queue state register 106 associated with the queue being accessed. In addition, decoder 104 may obtain base address 107 and entry size 108 from configuration register 105 associated with the queue being accessed. A memory address for the entry to be written is determined based on the write pointer 112, base address 107, and entry size 108. In one embodiment, the memory address of the entry being written can be determined as follows: [Equation] where (Write Pointer*Entry Size) represents an offset from the base address.” Gewirtz ’227 at col. 5, l. 50 – col. 6, l. 4. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[1d] at least one second processing element in communication with said first processing element;	Gewirtz ’227 teaches “at least one second processing element in communication with said first processing element,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches at least one second processing element in communication with said first processing element, by disclosing, as non-limiting examples, a second software agent running on a second processor. The second processor, the second processing element is in communication with the first processing element. <i>See, e.g.:</i> “A command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	memory based on the pointer. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around the pointer if the pointer reaches an end of the first queue based on the queue size.” Gewirtz ’227 at Abstract.  FIG. 1 Gewirtz ’227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	<pre> graph TD 301[Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element.] --> 302[Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory.] 302 --> 303[Atomically read a pointer from a hardware-based queue state register associated with the queue.] 303 --> 304[Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue.] 304 --> 305[Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size.] </pre> 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system 500. At the top, a block labeled 'Agent(s) 120' is connected via a bidirectional arrow to a vertical stack of 'Access Register' blocks. This stack is divided into two sections: section 505 contains 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, and 'Access Register for Qn'; section 506 contains 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. This entire stack is labeled 'Memory Mapped Registers 101'. To the right of this stack is a central block containing a 'Scanner 504' and an 'Address Decoder 104', connected by a bidirectional arrow. This central block is connected via bidirectional arrows to the 'Access Register' stack and to a vertical stack of 'Priority Group' blocks on the right. The 'Priority Group' stack includes 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. This stack is labeled 'Memory 103'. Below the central block is a stack of 'HW-Based Registers 102' containing a 'Configuration Register 105' and a 'Queue State Register 106', connected to the central block by a bidirectional arrow. The entire system is labeled 500. Gewirtz '227 at Figure 5.

FIG. 5

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically read data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6.

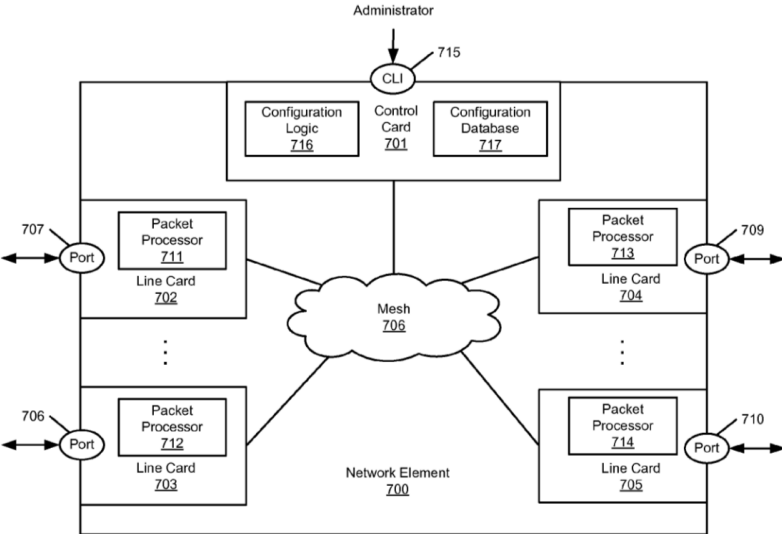
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 7 Gewirtz '227 at Figure 7. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz '227 at col. 4, ll. 16–33. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[1e] a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element;	Gewirtz ’227 teaches “a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element, by disclosing, as non-limiting examples, a queue state register. The queue state register stores a value of indicative space and is accessible by both the first and second processing elements discussed above. Defendants have proposed a construction that “register” means “set of flip-flops that store a set of related bits.” Gewirtz ’227 teaches a reservation register that is a set of flip-flops that store a set of related bits. <i>See e.g.</i>, Gewirtz ’227 at 2:59–64 (“Another register (also referred to as a state register or status register) holds the dynamic state of the queue, including a read pointer and a write pointer, queue occupancy, and/or number of free writeable entries for flow control purposes. This register may be updated by hardware logic that manages the queues.”). For example, the queue state register identified above is explained and described as a register and the description of its behavior is consistent with a set of flip-flops that store a set of related bits. <i>See id.</i> Gewirtz ’227 is a multiprocessor system that uses registers. A POSITA reading Gewirtz ’227 would have understood the description of these registers to be a set of flip-flops that store a set of related bits in view of the relevant disclosures, and further because that is how hardware systems and registers in particular had been designed and implemented for decades. <i>See, e.g.</i>: “A command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and

Claim Language

Exemplary Evidence of Anticipation and/or Obviousness

wrapping around the pointer if the pointer reaches an end of the first queue based on the queue size.” Gewirtz ’227 at Abstract.

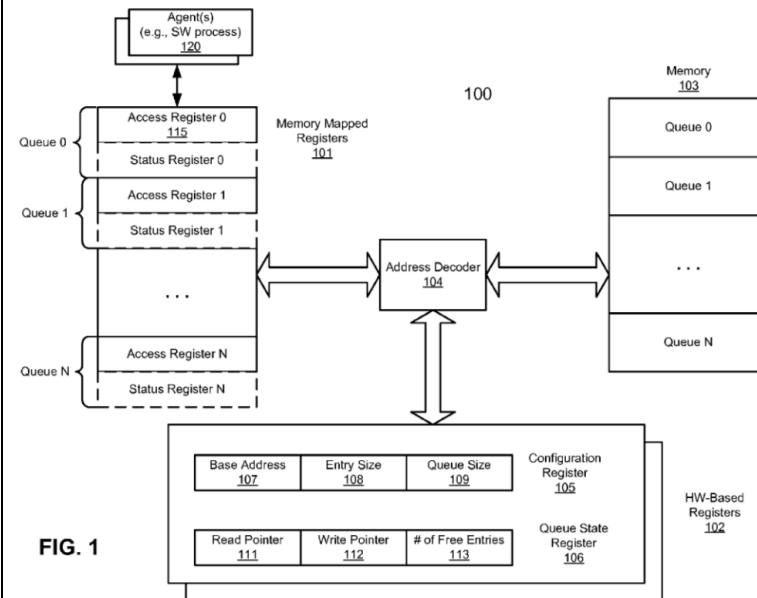
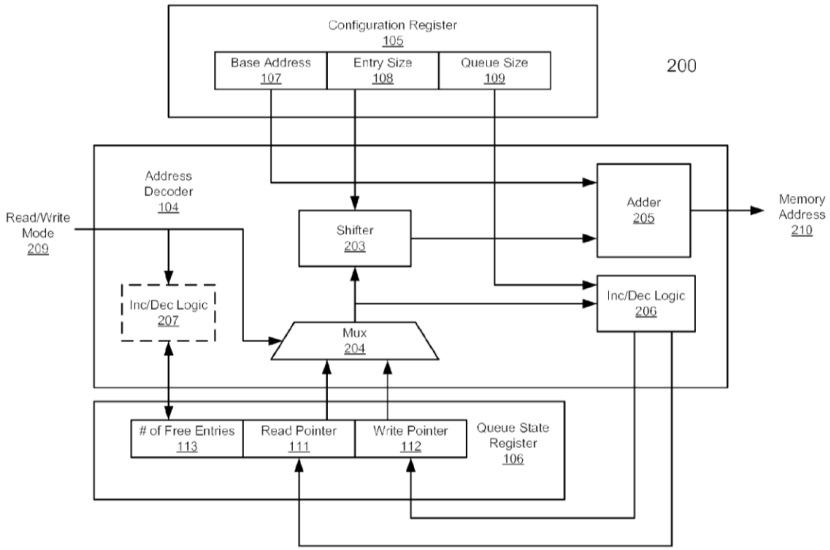


FIG. 1

Gewirtz '227 at Figure 1.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 The diagram, labeled FIG. 2, illustrates a system architecture for queue management. At the top, a Configuration Register 105 contains three fields: Base Address 107, Entry Size 108, and Queue Size 109. These fields are connected to a central processing block 200. Within block 200, an Address Decoder 104 receives a Read/Write Mode 209 input and outputs to an Inc/Dec Logic 207 (shown in a dashed box) and a Mux 204. The Inc/Dec Logic 207 also receives input from the Queue State Register 106. The Mux 204 receives inputs from the Inc/Dec Logic 207 and the Queue State Register 106, and its output goes to a Shifter 203. The Shifter 203 outputs to an Adder 205 and an Inc/Dec Logic 206. The Adder 205 also receives inputs from the Configuration Register 105 (specifically Base Address 107 and Entry Size 108) and the Inc/Dec Logic 206. The final output of the Adder 205 is the Memory Address 210. The Queue State Register 106 at the bottom contains three fields: # of Free Entries 113, Read Pointer 111, and Write Pointer 112. It is connected to the Inc/Dec Logic 207 and 206, and the Mux 204. FIG. 2 Gewirtz '227 at Figure 2.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301 Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302 Atomically read a pointer from a hardware-based queue state register associated with the queue. 303 Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304 Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system architecture. At the top, a block labeled 'Agent(s) 120' is connected via a bidirectional arrow to a vertical stack of 'Memory Mapped Registers 101'. This stack is divided into two sections: section 505 contains 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, and 'Access Register for Qn'; section 506 contains 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. To the right of the registers is a central block labeled '500' containing a 'Scanner 504' and an 'Address Decoder 104'. Below this central block is a stack of 'HW-Based Registers 102' containing a 'Configuration Register 105' and a 'Queue State Register 106'. To the right of the HW-based registers is a vertical stack of 'Memory 103' blocks, including 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. Bidirectional arrows connect the Memory Mapped Registers 101 to the Scanner 504, the Scanner 504 to the HW-Based Registers 102, and the HW-Based Registers 102 to the Memory 103. A bidirectional arrow also connects the Scanner 504 to the Address Decoder 104. Gewirtz '227 at Figure 5.

FIG. 5

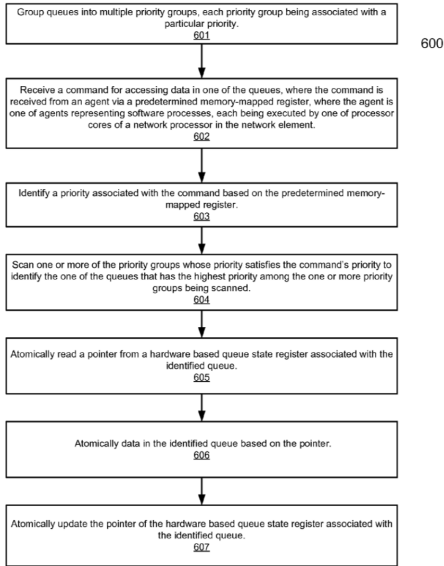
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 600 FIG. 6 Gewirtz '227 at Figure 6. “According to one aspect of the invention, a command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register, where the first queue was allocated in a memory. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the first queue. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing, via a first increment/decrement hardware logic, the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around, via the first increment/decrement hardware logic, the pointer of the first hardware-based queue state register if the pointer reaches an end of the first queue based on the queue size. As a result, the pointer is obtained and updated atomically through hardware so that the software process represented by the first agent is not required to obtain an exclusive access control of the first queue from the Operating System.” Gewirtz '227 at col. 1, ll. 39–64.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“According to some embodiments, a high performance hardware implementation is employed to manage logical queues with configuration registers which use queue element storage that is allocated in a normal memory. A queue is made up of one or more configuration registers and access registers. One register (also referred to as a configuration register) holds the configuration of the queue and defines its operation. This register may be written by configuration or system software to create a queue in memory, thus the properties of the queue are highly programmable. Once a queue is set up, it can be used concurrently by any number of agents in the system, both producers (e.g., agents to write to the queue) and consumers (e.g., agents to read from the queue).” Gewirtz ’227 at col. 2, ll. 46–58. “Another register (also referred to as a state register or status register) holds the dynamic state of the queue, including a read pointer and a write pointer, queue occupancy, and/or number of free writeable entries for flow control purposes. This register may be updated by hardware logic that manages the queues. Software can then perform queue operations by performing simple reads and writes to one or more data access registers (e.g., memory mapped registers). A write operation may cause an element to be pushed onto the tail of the queue and a read operation may cause an element to be popped from the head of the queue. The queue hardware (e.g., address decoder) performs the translation between read and write pointers and memory addresses. All maintenance of the queue state is done automatically in hardware, and will be atomically visible, along with the new queue entry (or the new head of the queue for a pop), to the next access.” Gewirtz ’227 at col. 2, l. 59 – col. 3, l. 7. “FIG. 1 is a block diagram illustrating a hardware-controlled queuing system according to one embodiment of the invention. Referring to FIG. 1, system 100 includes, but not limited to, one or more memory mapped registers 101, one or more hardware-based registers 102, and a memory or memory section 103 (e.g., part of main memory) defining one or more queues (e.g., logical queues) coupled to each other via an address decoder 104. Memory mapped registers 101 (also referred to as data access registers or simply access registers) include one or more access registers for each queue defined in memory 103.” Gewirtz ’227 at col. 3, ll. 8–18. “For each queue, according to one embodiment, there is an access register and a status register. An access register can be used to read and/or write data from and/or to the corresponding queue defined in memory 103. A status register can be used to obtain certain state information (e.g., current queue occupancy) from one or more of hardware-based registers 102 associated with the respective queue. Similarly, for each queue, a set of hardware-based registers are defined and allocated as part of hardware-based registers 102. Memory mapped registers of a particular queue may be mapped and/or translated by address decoder 104 to one or more of

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	hardware-based registers 102 and queue entries defined in memory 103 of the corresponding queue.” Gewirtz ’227 at col. 3, ll. 19–31. “According to certain embodiments, system 100 provides a software accessible hardware queuing structure in which all maintenance of the queue state is done automatically in hardware, for example, via address decoder 104 and hardware-based registers 102, as well as their underlying hardware logic. One of the challenges is to provide a simple efficient software access model, at the same time ensuring that all software visible state is atomically visible, including but not limited to new queue entry, empty, full state etc., while enabling coherency between potentially hundreds of independent software clients.” Gewirtz ’227 at col. 3, ll. 32–42. “In one embodiment, a programmable queue includes a configuration register 105, a queue state register 106, storage that is allocated for the queue elements (e.g., corresponding memory segment of memory 103), and data access registers (e.g., corresponding memory mapped registers 101). Configuration software can set up a queue by writing the corresponding configuration register 105 and enabling the queue as well as defining its behavior. Once the queue is programmed, any other agent (e.g., agents 120) in the system can read or write (e.g., push or pop) from or to the queue by issuing standard reads and write commands (loads and stores) to the corresponding memory mapped registers 101.” Gewirtz ’227 at col. 3, ll. 43–54. “In one embodiment, configuration register 105 defines the operation of the queue including parameters, such as, for example, base memory address 107, element or entry size 108, queue depth 109, and options (e.g., flow control options, processor interrupt options, and prioritization options, etc.) Base memory address 107 is a memory pointer pointing to the beginning of the allocated queue storage. Entry size 108 and queue size 109 define the size of the queue. The flow control option enables a built-in credit counter for each queue. The processor interrupt options enable the queue hardware to generate processor interrupts to a programmable target whenever certain queue events are detected. For example, an interrupt can be generated when a queue goes from empty to non-empty; drains below a predetermined threshold; or fills above a predetermined threshold, etc. The priority option allows queues to be grouped into priority groups, which will be described in details further below.” Gewirtz ’227 at col. 3, l. 55 – col. 4, l. 4. “In one embodiment, state register 106 includes read pointer 111, write pointer 112, a number of occupied entries 113, and available entries for reservation. Read pointer 111 points to a head of a queue for reading a next queue element or data entry. Write pointer 112 points to a tail of the queue for writing new data into the queue. Number of occupied entries 113 includes data representing the number of entries currently stored in the queue. The reservation field can be used to specify the number of entries available for writing into the

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	queue. All fields of state register 106 are maintained by hardware logic, for example, atomically.” Gewirtz ’227 at col. 4, ll. 5–15. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz ’227 at col. 4, ll. 16–33. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6. “The memory address is used to read data from the corresponding queue in memory 103. Thereafter, read pointer 111 may be atomically updated via hardware logic (e.g., decoder 104), for example, by incrementing or decrementing read pointer 111 dependent upon the specific implementation of the queue. When read pointer 111 is updated, in this example, being incremented, hardware logic may also determine whether read pointer 111 has reached the bottom or top of the queue, based on base address 107 and queue size 109 obtained from configuration register 105. If read pointer 111 does reach the top or bottom of the queue, read

Exhibit 596-06

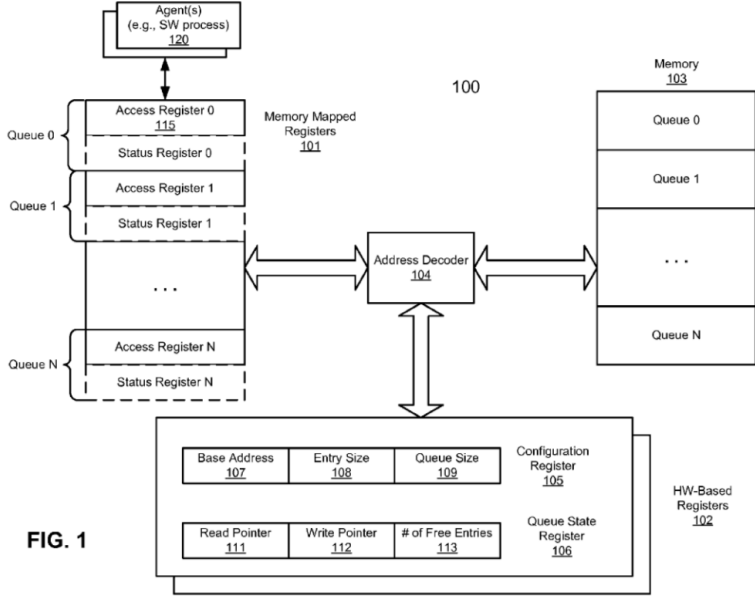
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	pointer 111 may be wrapped around or reset to a predetermined value such as zero. In addition, the number of free entries field 113 may also be updated by hardware logic, for example, by incrementing field 113, since an entry has been popped from the queue. These operations are performed atomically and controlled by hardware logic or logics. As a result, the agent does not have to worry about the exclusive access to the queues. It will be appreciated that certain firmware or low level software may also be utilized in conjunction with hardware logics.” Gewirtz ’227 at col. 5, ll. 29–49. “The memory address is used to write data to the corresponding queue in memory 103. Thereafter, write pointer 112 may be atomically updated via hardware logic (e.g., decoder 104), for example, by incrementing or decrementing write pointer 112 dependent upon the specific implementation of the queue. When write pointer 112 is updated, in this example, being incremented, hardware logic may also determine whether write pointer 112 has reached the bottom or top of the queue, based on base address 107 and queue size 109 obtained from configuration register 105. If write pointer 112 does reach the top or bottom of the queue, write pointer 112 may be wrapped around or reset to a predetermined value such as zero. In addition, the number of free entries field 113 may also be updated by hardware logic, for example, by decrementing field 113, since an entry has been pushed into the queue. These operations are performed atomically and controlled by hardware logic. As a result, the agent does not have to worry about exclusive access to the queues. Again, it will be appreciated that certain firmware or low level software may also be utilized in conjunction with hardware logic.” Gewirtz ’227 at col. 6, ll. 6–25. “FIG. 2 is a block diagram illustrating a hardware-based queue implementation according to one embodiment of the invention. For example, queue 200 may be implemented as part of system 100 of FIG. 1. Referring to FIG. 2, queue 200 includes configuration register 105 of all queues, which may be stored in a control random access memory (RAM) and queue state register 106 of all queues, which may be stored in a status RAM. In addition, system 200 includes the address decoder 104 having shifter 203, multiplexer 204, adder 205, and increment/decrement logics 206-207.” Gewirtz ’227 at col. 6, ll. 26–35. “Further, the number of free entries 113 of the queue is updated via increment/decrement logic 207. Either read pointer 111 or write pointer 112 may also be updated via increment/decrement logic 206 based on queue size 109, dependent upon whether the access is a read or write access which is obtained from the output of multiplexer 204.” Gewirtz ’227 at col. 6, ll. 53–58. “One of the advantages of the programmable queue as shown in FIGS. 1-2 is that, once defined, system agents can perform normal read and write commands that are processed atomically as pushes and pops from that queue. This reduces the latency seen when reading from the queue and, even more importantly,

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	eliminates the need for a locking access to gain exclusive ownership of the memory holding all of the queue data. This avoids contention over the lock. Memory contention over shared memory space is always detrimental to multi-processor system performance. In addition to the atomicity and locking benefits, the queues are extremely flexible, and can include a variety of optional features. They can be defined to have a large range of depths and element sizes. They can have built-in credit counters (e.g., represented by the number of free entries of a queue state register) to assist with system flow control. They can generate processor interrupts when their state changes to assist with message passing and resource management. They can be accessed with blocking and non-blocking reads depending on the desired behavior when an agent tries to read an empty queue. Finally, they can be organized into priority groups to assist in the processing of certain kinds of workloads.” Gewirtz ’227 at col. 6, l. 66 – col. 7, l. 20. “FIG. 3 is a flow diagram illustrating a method for access a queue controlled by hardware logic according to one embodiment of the invention. Note that method 300 may be performed by processing logic which may include hardware, firmware, software, or a combination there of. For example, method 300 may be performed by systems as shown in FIGS. 1 and/or 2. Referring to FIG. 3, at block 301, a command (e.g., a memory access command, such as a read command or a write command) is received from an agent via a predetermined memory mapped register (e.g., data access register). The agent may be one of the software processes, each being executed by one of multiple processor cores of a network processor of a network element. At block 302, a queue is identified based on the predetermined memory mapped register, where the queue has been allocated in a memory. At block 303, a pointer of the queue is atomically read from a hardware-based queue state register associated with the queue. At block 304, data is accessed (e.g., read or write) at a memory location of a memory based on the pointer, where the memory location is allocated as a part of the queue. In one embodiment, a base address is obtained from a hardware-based configuration register of the queue, where the configuration register is configured by system software prior to being used. A memory address is generated via hardware logic based on the pointer and the base address. The data is accessed at the memory location based on the memory address. Thereafter, at block 305, the pointer of the HW-based queue state register of the queue is updated atomically, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. In addition, the number of free entries of the queue may also be updated.” Gewirtz ’227 at col. 7, ll. 21–53. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31. “reading a queue size of the queue from a hardware-based configuration register associated with the queue, and” Gewirtz ’227 at col. 12, ll. 30–31. “4. The method of claim 1, wherein the step of atomically updating the pointer of the hardware-based queue state register further comprises atomically updating, via a second increment/decrement hardware logic, a number of free entries of the queue in the hardware-based queue state register.” Gewirtz ’227 at col. 12, ll. 59–63. “5. The method of claim 4, further comprising: receiving a second command from the agent to query a status of the queue via a second predetermined memory-mapped register; and in response to the second command, returning to the agent the status of the queue including the number of free entries of the queue from the hardware-based queue state register.” Gewirtz ’227 at col. 12, l. 64 – col. 13, l. 4. “a hardware-based queue state register associated with the queue;” Gewirtz ’227 at col. 13, ll. 27–28. “9. The system of claim 8, wherein the hardware address decoder further comprises a second increment/decrement hardware logic to increment a number of free entries of the queue in the hardware-based queue state register in response to the read command.” Gewirtz ’227 at col. 13, ll. 63–67. “13. The system of claim 12, wherein the second increment/decrement hardware logic is adapted to decrement the number of free entries of the queue in the hardware-based queue state register in response to the write command.” Gewirtz ’227 at col. 14, ll. 24–27. “14. The system of claim 13, further comprising a second memory-mapped register to receive a query command to retrieve a status of the queue including the number of free entries of the queue.” Gewirtz ’227 at col. 14, ll. 28–31.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
[1f] computer storage storing instructions, which when executed by said at least one second processing element:	Gewirtz '227 teaches computer storage storing instructions executed by said at least one second processing element, including at least the following exemplary citations. For example, Gewirtz '227 teaches computer storage storing instructions that are executed by said at least one second processing element, by disclosing, as non-limiting examples, a processor that runs software processes, e.g., agents. The processor executes those instructions stored in the computer storage. <i>See, e.g.</i>, Gewirtz '227 at 11:40–48 (“Embodiments of the invention also relate to an apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a general-purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable medium. A machine-readable medium includes any mechanism for storing information in a form readable by a machine (e.g., a computer).”). <i>See, e.g.</i>:  FIG. 1 Gewirtz '227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301 Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302 Atomically read a pointer from a hardware-based queue state register associated with the queue. 303 Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304 Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Atomically read a base address and an entry size from a HW-based configuration register associated with a queue being accessed. <u>321</u> ↓ Generate a memory address based on the base address, the entry size, and the pointer, where the data is accessed at the memory location identified by the memory address. <u>322</u> 304 FIG. 4A Gewirtz '227 at Figure 4A.

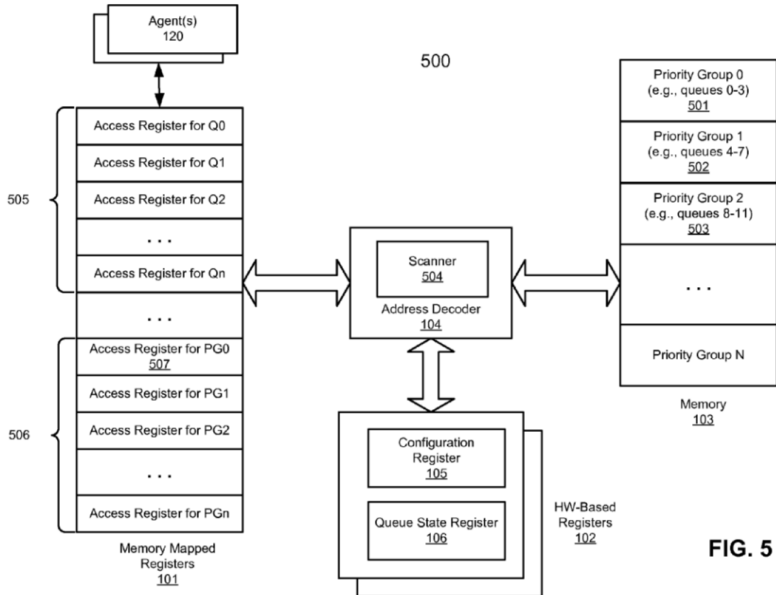
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 The diagram, labeled FIG. 5, illustrates a system 500. At the top, a block labeled 'Agent(s) 120' is connected via a bidirectional arrow to a vertical stack of 'Memory Mapped Registers 101'. This stack includes 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, 'Access Register for Qn', another ellipsis, 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. To the right of this stack is a block labeled 'Scanner 504' and 'Address Decoder 104'. A bidirectional arrow connects the 'Access Register for Qn' to the 'Scanner 504'. Below the scanner is a block labeled 'Configuration Register 105' and 'Queue State Register 106', collectively labeled 'HW-Based Registers 102'. A bidirectional arrow connects the 'Scanner 504' to the 'Configuration Register 105'. To the right of the scanner is a vertical stack of 'Priority Groups' labeled 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. A bidirectional arrow connects the 'Scanner 504' to this stack. Below the priority groups is a block labeled 'Memory 103'. A bidirectional arrow connects the 'Address Decoder 104' to the 'Memory 103'. The entire system is labeled 500. Gewirtz '227 at Figure 5. “According to one aspect of the invention, a command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register, where the first queue was allocated in a memory. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the first queue. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing, via a first increment/decrement hardware logic, the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around, via the first increment/decrement hardware logic, the pointer of the first hardware-based queue state register if the pointer reaches an end of the first queue based on the queue size. As a result, the pointer is obtained and updated atomically through hardware so that the software process represented by the first agent is not required to obtain an exclusive access control of the first queue from the Operating System.” Gewirtz '227 at col. 1, ll. 39–64.

Exhibit 596-06

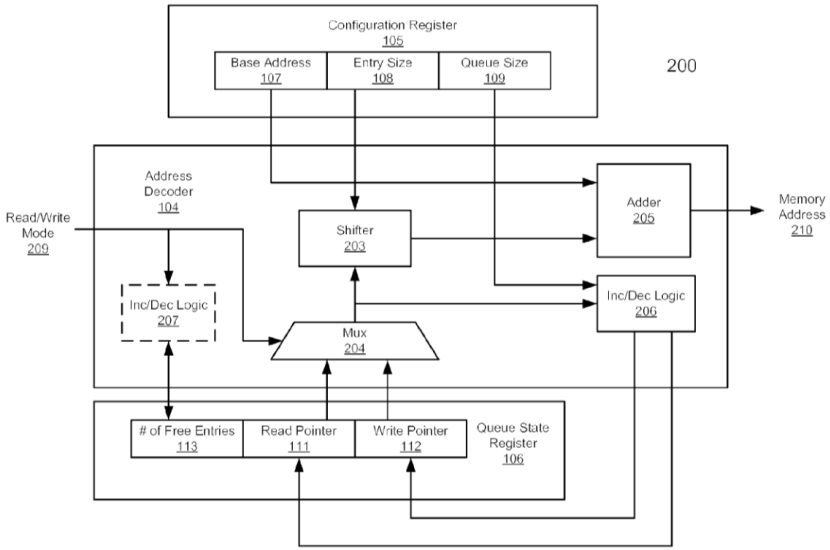
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz ’227 at col. 4, ll. 16–33. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6. “One of the advantages of the programmable queue as shown in FIGS. 1-2 is that, once defined, system agents can perform normal read and write commands that are processed atomically as pushes and pops from that queue. This reduces the latency seen when reading from the queue and, even more importantly, eliminates the need for a locking access to gain exclusive ownership of the memory holding all of the queue data. This avoids contention over the lock. Memory contention over shared memory space is always detrimental to multi-processor system performance. In addition to the atomicity and locking benefits, the queues are extremely flexible, and can include a variety of optional features. They can be defined to have a large range of depths and element sizes. They can have built-in credit counters (e.g., represented by the

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	number of free entries of a queue state register) to assist with system flow control. They can generate processor interrupts when their state changes to assist with message passing and resource management. They can be accessed with blocking and non-blocking reads depending on the desired behavior when an agent tries to read an empty queue. Finally, they can be organized into priority groups to assist in the processing of certain kinds of workloads.” Gewirtz ’227 at col. 6, l. 66 – col. 7, l. 20. “Embodiments of the invention also relate to an apparatus for performing the operations herein. This apparatus may be specially constructed for the required purposes, or it may comprise a general-purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable medium. A machine-readable medium includes any mechanism for storing information in a form readable by a machine (e.g., a computer). For example, a machine-readable (e.g., computer-readable) medium includes a machine (e.g., a computer) readable storage medium (e.g., read only memory (“ROM”), random access memory (“RAM”), magnetic disk storage media, optical storage media, flash memory devices, etc.), etc.” Gewirtz ’227 at col. 11, ll. 40–53. “5. The method of claim 4, further comprising: receiving a second command from the agent to query a status of the queue via a second predetermined memory-mapped register; and in response to the second command, returning to the agent the status of the queue including the number of free entries of the queue from the hardware-based queue state register.” Gewirtz ’227 at col. 12, l. 64 – col. 13, l. 4.
[1g] accesses said reservation register and reads said stored value;	Gewirtz ’227 teaches “accesses said reservation register and reads said stored value,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches that the second processing element executes instructions that causes it to access said reservation register and reads said stored value, by disclosing, as non-limiting examples, accessing a queue state register that contains the number of free entries in the associated queue. The second processor executing a software-agent accesses and reads the stored value in the reservation register. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	FIG. 1 Gewirtz '227 at Figure 1.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 The diagram, labeled FIG. 2, illustrates a system architecture for queue management. At the top, a Configuration Register 105 contains three fields: Base Address 107, Entry Size 108, and Queue Size 109. These fields provide input to an Address Decoder 104 and a Shifter 203. The Address Decoder 104 also receives a Read/Write Mode 209 signal. Its output goes to an Inc/Dec Logic 207 (shown in a dashed box) and a Mux 204. The Inc/Dec Logic 207 also receives input from the Queue State Register 106. The Shifter 203 receives input from the Base Address 107 and the Queue State Register 106. Its output goes to an Adder 205 and the Inc/Dec Logic 206. The Adder 205 also receives input from the Queue State Register 106 and produces the Memory Address 210. The Inc/Dec Logic 206 receives input from the Shifter 203 and the Queue State Register 106. The Queue State Register 106 contains three fields: # of Free Entries 113, Read Pointer 111, and Write Pointer 112. The Read Pointer 111 and Write Pointer 112 are connected to the Mux 204, which outputs to the Inc/Dec Logic 207. The # of Free Entries 113 is connected to the Inc/Dec Logic 207. The Inc/Dec Logic 207 outputs to the Queue State Register 106. The entire system is labeled 200. FIG. 2 Gewirtz '227 at Figure 2.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301 Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302 Atomically read a pointer from a hardware-based queue state register associated with the queue. 303 Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304 Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	304 Atomically read a base address and an entry size from a HW-based configuration register associated with a queue being accessed. <u>321</u> ↓ Generate a memory address based on the base address, the entry size, and the pointer, where the data is accessed at the memory location identified by the memory address. <u>322</u> FIG. 4A Gewirtz '227 at Figure 4A.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system architecture. At the top, a block labeled 'Agent(s) 120' is connected via a bidirectional arrow to a vertical stack of 'Memory Mapped Registers 101'. This stack includes 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, 'Access Register for Qn', another ellipsis, 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. A bracket labeled '505' groups the Q registers, and a bracket labeled '506' groups the PG registers. To the right of the registers is a central block labeled '500' containing a 'Scanner 504' and an 'Address Decoder 104'. Below this is a block labeled 'HW-Based Registers 102' containing a 'Configuration Register 105' and a 'Queue State Register 106'. Bidirectional arrows connect the 'Memory Mapped Registers 101' to the 'Scanner 504', and the 'HW-Based Registers 102' to the 'Address Decoder 104'. To the right of the central block is a 'Memory 103' structure containing 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. Bidirectional arrows connect the 'Scanner 504' to the 'Memory 103'. Gewirtz '227 at Figure 5.

FIG. 5

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	<pre> graph TD 601[Group queues into multiple priority groups, each priority group being associated with a particular priority. 601] --> 602[Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602] 602 --> 603[Identify a priority associated with the command based on the predetermined memory-mapped register. 603] 603 --> 604[Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604] 604 --> 605[Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605] 605 --> 606[Atomically read data in the identified queue based on the pointer. 606] 606 --> 607[Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607] </pre> FIG. 6 Gewirtz '227 at Figure 6. “When software wants to read an element from the queue, or write an element to the queue, it first must ensure that it has exclusive and atomic control of the queue element and queue state storage. Then it must update the queue state and access the queue element to perform its operation. Finally, the memory that controls the ownership of the shared memory block must be updated to indicate that the operation is complete. Since there are usually multiple processors using a queue, this requires some forms of data locking of the queue states to ensure that the access to the queue states and the access to the queue element are atomic and consistent. Data locking, and particularly contention over the lock state memory locations, is often a performance-limiting operation in multi-processor systems. In particular, if there are both multiple producers and multiple consumers of queue data, the performance implications can be severe.” Gewirtz '227 at col. 1, ll. 20–35. “According to one aspect of the invention, a command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register, where the first queue was allocated in a memory. A pointer is atomically read from a first hardware-based queue

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the first queue. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing, via a first increment/decrement hardware logic, the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around, via the first increment/decrement hardware logic, the pointer of the first hardware-based queue state register if the pointer reaches an end of the first queue based on the queue size. As a result, the pointer is obtained and updated atomically through hardware so that the software process represented by the first agent is not required to obtain an exclusive access control of the first queue from the Operating System.” Gewirtz ’227 at col. 1, ll. 39–64. “Another register (also referred to as a state register or status register) holds the dynamic state of the queue, including a read pointer and a write pointer, queue occupancy, and/or number of free writeable entries for flow control purposes. This register may be updated by hardware logic that manages the queues. Software can then perform queue operations by performing simple reads and writes to one or more data access registers (e.g., memory mapped registers). A write operation may cause an element to be pushed onto the tail of the queue and a read operation may cause an element to be popped from the head of the queue. The queue hardware (e.g., address decoder) performs the translation between read and write pointers and memory addresses. All maintenance of the queue state is done automatically in hardware, and will be atomically visible, along with the new queue entry (or the new head of the queue for a pop), to the next access.” Gewirtz ’227 at col. 2, l. 59 – col. 3, l. 7. “In one embodiment, configuration register 105 defines the operation of the queue including parameters, such as, for example, base memory address 107, element or entry size 108, queue depth 109, and options (e.g., flow control options, processor interrupt options, and prioritization options, etc.) Base memory address 107 is a memory pointer pointing to the beginning of the allocated queue storage. Entry size 108 and queue size 109 define the size of the queue. The flow control option enables a built-in credit counter for each queue. The processor interrupt options enable the queue hardware to generate processor interrupts to a programmable target whenever certain queue events are detected. For example, an interrupt can be generated when a queue goes from empty to non-empty; drains below a predetermined threshold; or fills above a predetermined threshold, etc. The priority option allows queues to be grouped into priority groups, which will be described in details further below.” Gewirtz ’227 at col. 3, l. 55 – col. 4, l. 4. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be

Exhibit 596-06

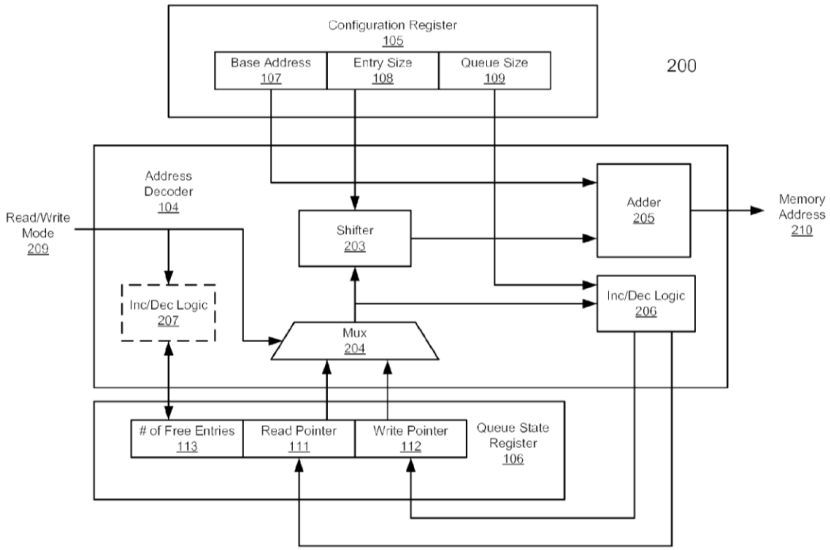
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz ’227 at col. 4, ll. 16–33. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6. “One of the advantages of the programmable queue as shown in FIGS. 1-2 is that, once defined, system agents can perform normal read and write commands that are processed atomically as pushes and pops from that queue. This reduces the latency seen when reading from the queue and, even more importantly, eliminates the need for a locking access to gain exclusive ownership of the memory holding all of the queue data. This avoids contention over the lock. Memory contention over shared memory space is always detrimental to multi-processor system performance. In addition to the atomicity and locking benefits, the queues are extremely flexible, and can include a variety of optional features. They can be defined to have a large range of depths and element sizes. They can have built-in credit counters (e.g., represented by the number of free entries of a queue state register) to assist with system flow control. They can generate processor interrupts when their state changes to assist with message passing and resource management. They

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	can be accessed with blocking and non-blocking reads depending on the desired behavior when an agent tries to read an empty queue. Finally, they can be organized into priority groups to assist in the processing of certain kinds of workloads.” Gewirtz ’227 at col. 6, l. 66 – col. 7, l. 20. “5. The method of claim 4, further comprising: receiving a second command from the agent to query a status of the queue via a second predetermined memory-mapped register; and in response to the second command, returning to the agent the status of the queue including the number of free entries of the queue from the hardware-based queue state register.” Gewirtz ’227 at col. 12, l. 64 – col. 13, l. 4. “14. The system of claim 13, further comprising a second memory-mapped register to receive a query command to retrieve a status of the queue including the number of free entries of the queue.” Gewirtz ’227 at col. 14, ll. 28–31.
[1h] determines when said read value indicates available space in said input queue for said first processing element to issue a command;	Gewirtz ’227 teaches “determines when said read value indicates available space in said input queue for said first processing element to issue a command,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches that the second processing element executes instructions that causes it to determine when said read value indicates available space in said input queue for said first processing element to issue a command, by disclosing, as non-limiting examples, a second processor executing a software-agent reading the queue status register and checking for available space in the associated queue. The second processor accesses and reads the stored value in the reservation register. The second processor determines if the stored value indicates available space in the input queue for the first processing element to issue a command. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	FIG. 1 Gewirtz '227 at Figure 1.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 The diagram, labeled FIG. 2, illustrates a system architecture for queue management. At the top, a Configuration Register 105 contains three fields: Base Address 107, Entry Size 108, and Queue Size 109. These fields are connected to a central processing block 200. Within block 200, an Address Decoder 104 receives a Read/Write Mode 209 input and outputs to an Inc/Dec Logic 207 (shown in a dashed box) and a Mux 204. The Inc/Dec Logic 207 also receives input from the Queue State Register 106. The Mux 204 receives inputs from the Inc/Dec Logic 207 and the Queue State Register 106, and its output goes to a Shifter 203. The Shifter 203 outputs to an Adder 205 and an Inc/Dec Logic 206. The Adder 205 also receives inputs from the Base Address 107 and the Entry Size 108 of the Configuration Register 105. The Inc/Dec Logic 206 receives inputs from the Queue Size 109 and the Queue State Register 106. The output of the Adder 205 is the Memory Address 210. The Queue State Register 106 at the bottom contains three fields: # of Free Entries 113, Read Pointer 111, and Write Pointer 112. Arrows indicate the data flow between these components. FIG. 2 Gewirtz '227 at Figure 2.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	<pre> graph TD 301[Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element.] --> 302[Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory.] 302 --> 303[Atomically read a pointer from a hardware-based queue state register associated with the queue.] 303 --> 304[Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue.] 304 --> 305[Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size.] </pre> 300 FIG. 3 Gewirtz '227 at Figure 3.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Atomically read a base address and an entry size from a HW-based configuration register associated with a queue being accessed. <u>321</u> ↓ Generate a memory address based on the base address, the entry size, and the pointer, where the data is accessed at the memory location identified by the memory address. <u>322</u> 304 FIG. 4A Gewirtz '227 at Figure 4A.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system 500. At the top, an 'Agent(s) 120' is connected via a bidirectional arrow to a set of 'Memory Mapped Registers 101'. This set includes 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, 'Access Register for Qn', another ellipsis, 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. To the right of these registers is a 'Scanner 504' and an 'Address Decoder 104', which are connected to the registers by a bidirectional arrow. Below the scanner and address decoder is a block labeled 'HW-Based Registers 102', containing a 'Configuration Register 105' and a 'Queue State Register 106', connected to the scanner and address decoder by a bidirectional arrow. To the right of the HW-based registers is a 'Memory 103' containing 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. A bidirectional arrow connects the scanner and address decoder to the memory. The entire system is labeled 500. The diagram is captioned 'FIG. 5'. Gewirtz '227 at Figure 5.

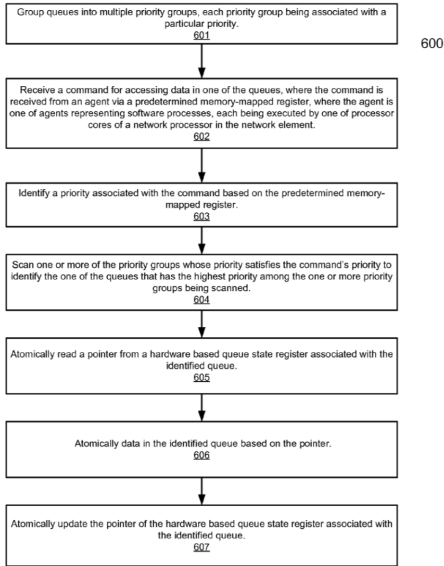
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 6 Gewirtz '227 at Figure 6. “When software wants to read an element from the queue, or write an element to the queue, it first must ensure that it has exclusive and atomic control of the queue element and queue state storage. Then it must update the queue state and access the queue element to perform its operation. Finally, the memory that controls the ownership of the shared memory block must be updated to indicate that the operation is complete. Since there are usually multiple processors using a queue, this requires some forms of data locking of the queue states to ensure that the access to the queue states and the access to the queue element are atomic and consistent. Data locking, and particularly contention over the lock state memory locations, is often a performance-limiting operation in multi-processor systems. In particular, if there are both multiple producers and multiple consumers of queue data, the performance implications can be severe.” Gewirtz '227 at col. 1, ll. 20–35. “According to one aspect of the invention, a command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register, where the first queue was allocated in a memory. A pointer is atomically read from a first hardware-based queue

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the first queue. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing, via a first increment/decrement hardware logic, the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around, via the first increment/decrement hardware logic, the pointer of the first hardware-based queue state register if the pointer reaches an end of the first queue based on the queue size. As a result, the pointer is obtained and updated atomically through hardware so that the software process represented by the first agent is not required to obtain an exclusive access control of the first queue from the Operating System.” Gewirtz ’227 at col. 1, ll. 39–64. “Another register (also referred to as a state register or status register) holds the dynamic state of the queue, including a read pointer and a write pointer, queue occupancy, and/or number of free writeable entries for flow control purposes. This register may be updated by hardware logic that manages the queues. Software can then perform queue operations by performing simple reads and writes to one or more data access registers (e.g., memory mapped registers). A write operation may cause an element to be pushed onto the tail of the queue and a read operation may cause an element to be popped from the head of the queue. The queue hardware (e.g., address decoder) performs the translation between read and write pointers and memory addresses. All maintenance of the queue state is done automatically in hardware, and will be atomically visible, along with the new queue entry (or the new head of the queue for a pop), to the next access.” Gewirtz ’227 at col. 2, l. 59 – col. 3, l. 7. “In one embodiment, configuration register 105 defines the operation of the queue including parameters, such as, for example, base memory address 107, element or entry size 108, queue depth 109, and options (e.g., flow control options, processor interrupt options, and prioritization options, etc.) Base memory address 107 is a memory pointer pointing to the beginning of the allocated queue storage. Entry size 108 and queue size 109 define the size of the queue. The flow control option enables a built-in credit counter for each queue. The processor interrupt options enable the queue hardware to generate processor interrupts to a programmable target whenever certain queue events are detected. For example, an interrupt can be generated when a queue goes from empty to non-empty; drains below a predetermined threshold; or fills above a predetermined threshold, etc. The priority option allows queues to be grouped into priority groups, which will be described in details further below.” Gewirtz ’227 at col. 3, l. 55 – col. 4, l. 4. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz ’227 at col. 4, ll. 16–33. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6. “One of the advantages of the programmable queue as shown in FIGS. 1-2 is that, once defined, system agents can perform normal read and write commands that are processed atomically as pushes and pops from that queue. This reduces the latency seen when reading from the queue and, even more importantly, eliminates the need for a locking access to gain exclusive ownership of the memory holding all of the queue data. This avoids contention over the lock. Memory contention over shared memory space is always detrimental to multi-processor system performance. In addition to the atomicity and locking benefits, the queues are extremely flexible, and can include a variety of optional features. They can be defined to have a large range of depths and element sizes. They can have built-in credit counters (e.g., represented by the number of free entries of a queue state register) to assist with system flow control. They can generate processor interrupts when their state changes to assist with message passing and resource management. They

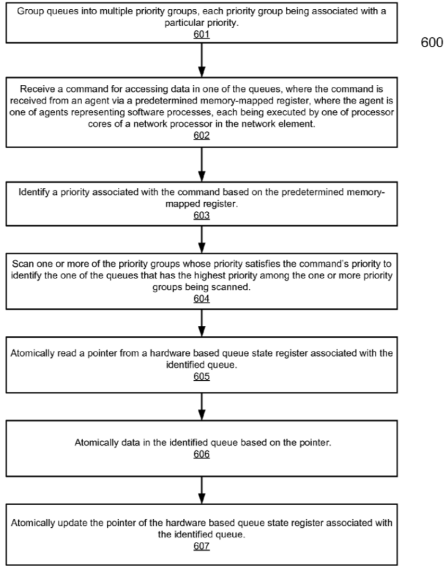
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	can be accessed with blocking and non-blocking reads depending on the desired behavior when an agent tries to read an empty queue. Finally, they can be organized into priority groups to assist in the processing of certain kinds of workloads.” Gewirtz ’227 at col. 6, l. 66 – col. 7, l. 20.
[1i] notifies said first processing element to issue said command to said input queue; and	Gewirtz ’227 teaches “notifies said first processing element to issue said command to said input queue,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches that the second processing element executes instructions that causes it to notify said first processing element to issue said command to said input queue, by disclosing, as non-limiting examples, processor interrupts. The second processor notifies the first processing element to issue the command to the input queue based on a processor interrupt. <i>See, e.g.:</i>  <pre> graph TD 601[Group queues into multiple priority groups, each priority group being associated with a particular priority. 601] --> 602[Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602] 602 --> 603[Identify a priority associated with the command based on the predetermined memory-mapped register. 603] 603 --> 604[Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604] 604 --> 605[Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605] 605 --> 606[Atomically data in the identified queue based on the pointer. 606] 606 --> 607[Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607] </pre> FIG. 6 Gewirtz ’227 at Figure 6. “When software wants to read an element from the queue, or write an element to the queue, it first must ensure that it has exclusive and atomic control of the queue element and queue state storage. Then it must update the queue state and access the queue element to perform its operation. Finally, the memory that controls the ownership of the shared memory block must be updated to indicate that the operation is

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	complete. Since there are usually multiple processors using a queue, this requires some forms of data locking of the queue states to ensure that the access to the queue states and the access to the queue element are atomic and consistent. Data locking, and particularly contention over the lock state memory locations, is often a performance-limiting operation in multi-processor systems. In particular, if there are both multiple producers and multiple consumers of queue data, the performance implications can be severe.” Gewirtz ’227 at col. 1, ll. 20–35. “In one embodiment, configuration register 105 defines the operation of the queue including parameters, such as, for example, base memory address 107, element or entry size 108, queue depth 109, and options (e.g., flow control options, processor interrupt options, and prioritization options, etc.) Base memory address 107 is a memory pointer pointing to the beginning of the allocated queue storage. Entry size 108 and queue size 109 define the size of the queue. The flow control option enables a built-in credit counter for each queue. The processor interrupt options enable the queue hardware to generate processor interrupts to a programmable target whenever certain queue events are detected. For example, an interrupt can be generated when a queue goes from empty to non-empty; drains below a predetermined threshold; or fills above a predetermined threshold, etc. The priority option allows queues to be grouped into priority groups, which will be described in details further below.” Gewirtz ’227 at col. 3, l. 55 – col. 4, l. 4. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6. “One of the advantages of the programmable queue as shown in FIGS. 1-2 is that, once defined, system agents can perform normal read and write commands that are processed atomically as pushes and pops from

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	that queue. This reduces the latency seen when reading from the queue and, even more importantly, eliminates the need for a locking access to gain exclusive ownership of the memory holding all of the queue data. This avoids contention over the lock. Memory contention over shared memory space is always detrimental to multi-processor system performance. In addition to the atomicity and locking benefits, the queues are extremely flexible, and can include a variety of optional features. They can be defined to have a large range of depths and element sizes. They can have built-in credit counters (e.g., represented by the number of free entries of a queue state register) to assist with system flow control. They can generate processor interrupts when their state changes to assist with message passing and resource management. They can be accessed with blocking and non-blocking reads depending on the desired behavior when an agent tries to read an empty queue. Finally, they can be organized into priority groups to assist in the processing of certain kinds of workloads.” Gewirtz ’227 at col. 6, l. 66 – col. 7, l. 20. “FIG. 3 is a flow diagram illustrating a method for access a queue controlled by hardware logic according to one embodiment of the invention. Note that method 300 may be performed by processing logic which may include hardware, firmware, software, or a combination there of. For example, method 300 may be performed by systems as shown in FIGS. 1 and/or 2. Referring to FIG. 3, at block 301, a command (e.g., a memory access command, such as a read command or a write command) is received from an agent via a predetermined memory mapped register (e.g., data access register). The agent may be one of the software processes, each being executed by one of multiple processor cores of a network processor of a network element. At block 302, a queue is identified based on the predetermined memory mapped register, where the queue has been allocated in a memory. At block 303, a pointer of the queue is atomically read from a hardware-based queue state register associated with the queue. At block 304, data is accessed (e.g., read or write) at a memory location of a memory based on the pointer, where the memory location is allocated as a part of the queue. In one embodiment, a base address is obtained from a hardware-based configuration register of the queue, where the configuration register is configured by system software prior to being used. A memory address is generated via hardware logic based on the pointer and the base address. The data is accessed at the memory location based on the memory address. Thereafter, at block 305, the pointer of the HW-based queue state register of the queue is updated atomically, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. In addition, the number of free entries of the queue may also be updated.” Gewirtz ’227 at col. 7, ll. 21–53. “FIG. 6 is a flow diagram illustrating a method for accessing a queue according to another embodiment of the invention. Note that method 600 may be performed by processing logic which may include hardware,

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	firmware, software, or a combination thereof. For example, method 600 may be performed by system 500 of FIG. 5. Referring to FIG. 6, at block 601, multiple queues are grouped into one or more priority groups, each having a particular (e.g., different) priority. For example, the queues are grouped by system software when configuring the queues, for example, by an option field (not shown) in the corresponding hardware-based queue configuration register 105 or hardware-based queue state register 106. At block 602, a command or request is received from an agent for accessing data in one of the queues via a predetermined memory-mapped register. The agent may be one of multiple agents, each being executed by one of processor cores of a network processor of a network element. At block 603, a priority associated with the command is identified based on the predetermined memory-mapped register. At block 604, processing logic scans one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. At block 605, a pointer is atomically read from a hardware based queue state register associated with the identified queue. At block 606, data is accessed in the identified queue based on the pointer. Thereafter, at block 607, the pointer of the hardware-based queue state register associated with the identified queue is atomically updated. In addition, the number of free entries may also be updated.” Gewirtz ’227 at col. 8, l. 59 – col. 9, l. 21. “receiving a command for accessing data in one of the plurality of queues, wherein the command is received from an agent via a predetermined memory mapped register, wherein the agent is one of a plurality of agents representing a plurality of software processes, each being executed by one of a plurality of processor cores of a network processor in the network element;” Gewirtz ’227 at col. 14, ll. 37–43.
[1j] wherein said first processing element receives notification from said at least one second processing element regarding issuing the command, issues said command to said input queue, and receives a response corresponding to said command from said output queue.	Gewirtz ’227 teaches “wherein said first processing element receives notification from said at least one second processing element regarding issuing the command, issues said command to said input queue, and receives a response corresponding to said command from said output queue,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches wherein said first processing element receives notification from said at least one second processing element regarding issuing the command, issues said command to said input queue, and receives a response corresponding to said command from said output queue, by disclosing, as non-limiting examples, the first processor running a software-agent receives the notification, issues the command, and receives a corresponding response from the output queue. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically read data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6. “According to one embodiment, when an agent such as agent 120 desires to write data to a specific queue, the agent may specific perform a write operation to an access register associated with the specific queue. For example, if agent 120 desires to write data to queue 0, it may write data to access register 115. In response, address decoder 104 may access a set of hardware-based registers associated with the queue being accessed (e.g., hardware-based registers associated with queue 0). In this example, decoder 104 may atomically obtain a write pointer 112 from queue state register 106 associated with the queue being accessed. In addition, decoder 104 may obtain base address 107 and entry size 108 from configuration register 105 associated with the queue being accessed. A memory address for the entry to be written is determined based on the write pointer 112, base address 107, and entry size 108. In one embodiment, the memory address of the entry being written can be determined as follows: [Equation] where (Write Pointer*Entry Size) represents an offset from the base address.” Gewirtz '227 at col. 5, l. 50 – col. 6, l. 4. “FIG. 3 is a flow diagram illustrating a method for access a queue controlled by hardware logic according to one embodiment of the invention. Note that method 300 may be performed by processing logic which may include hardware, firmware, software, or a combination there of. For example, method 300 may be performed by systems as shown in FIGS. 1 and/or 2. Referring to FIG. 3, at block 301, a command (e.g., a memory

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	access command, such as a read command or a write command) is received from an agent via a predetermined memory mapped register (e.g., data access register). The agent may be one of the software processes, each being executed by one of multiple processor cores of a network processor of a network element. At block 302, a queue is identified based on the predetermined memory mapped register, where the queue has been allocated in a memory. At block 303, a pointer of the queue is atomically read from a hardware-based queue state register associated with the queue. At block 304, data is accessed (e.g., read or write) at a memory location of a memory based on the pointer, where the memory location is allocated as a part of the queue. In one embodiment, a base address is obtained from a hardware-based configuration register of the queue, where the configuration register is configured by system software prior to being used. A memory address is generated via hardware logic based on the pointer and the base address. The data is accessed at the memory location based on the memory address. Thereafter, at block 305, the pointer of the HW-based queue state register of the queue is updated atomically, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. In addition, the number of free entries of the queue may also be updated.” Gewirtz ’227 at col. 7, ll. 21–53. “FIG. 6 is a flow diagram illustrating a method for accessing a queue according to another embodiment of the invention. Note that method 600 may be performed by processing logic which may include hardware, firmware, software, or a combination thereof. For example, method 600 may be performed by system 500 of FIG. 5. Referring to FIG. 6, at block 601, multiple queues are grouped into one or more priority groups, each having a particular (e.g., different) priority. For example, the queues are grouped by system software when configuring the queues, for example, by an option field (not shown) in the corresponding hardware-based queue configuration register 105 or hardware-based queue state register 106. At block 602, a command or request is received from an agent for accessing data in one of the queues via a predetermined memory-mapped register. The agent may be one of multiple agents, each being executed by one of processor cores of a network processor of a network element. At block 603, a priority associated with the command is identified based on the predetermined memory-mapped register. At block 604, processing logic scans one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. At block 605, a pointer is atomically read from a hardware based queue state register associated with the identified queue. At block 606, data is accessed in the identified queue based on the pointer. Thereafter, at block 607, the pointer of the hardware-based queue state register associated with the identified queue is atomically updated. In addition, the number of free entries may also be updated.” Gewirtz ’227 at col. 8, l. 59 – col. 9, l. 21.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“receiving a command from an agent via a first predetermined memory-mapped register, the agent being one of a plurality of agents representing a plurality of software processes, each being executed by one of a plurality of processor cores of a network processor in the network element;” Gewirtz ’227 at col. 12, ll. 12–17. “receiving a command for accessing data in one of the plurality of queues, wherein the command is received from an agent via a predetermined memory mapped register, wherein the agent is one of a plurality of agents representing a plurality of software processes, each being executed by one of a plurality of processor cores of a network processor in the network element;” Gewirtz ’227 at col. 14, ll. 37–43.
2. The system of claim 1, wherein said input queue is a first-in-first-out (FIFO) buffer.	Gewirtz ’227 teaches “[t]he system of claim 1, wherein said input queue is a first-in-first-out (FIFO) buffer,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches wherein said input queue is a first-in-first-out (FIFO) buffer, by disclosing, as non-limiting examples, a FIFO buffer. <i>See, e.g.:</i> “In a multi-processor system, it is frequently desirable to build queues, or first-in-first-out (FIFO) buffers, in memory. A queue is a data structure with FIFO properties where agents push elements onto the queue (writes) and pop elements off of the queue (reads). This is normally accomplished by allocating storage for the queue elements, and separately allocating storage for the queue state which includes read and write pointers, and any other desired information.” Gewirtz ’227 at col. 1, ll. 12–19.
3. The system of claim 1, wherein said system is part of a storage area network (SAN) controller.	Gewirtz ’227 teaches “[t]he system of claim 1, wherein said system is part of a storage area network (SAN) controller,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches wherein said system is part of a storage area network (SAN) controller, by disclosing, as non-limiting examples, a network element that uses the multiprocessor system described in Gewirtz ’227. Network elements are used in a SAN controller. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram illustrates a network element 700. At the top, an Administrator is connected to a CLI (715) on a Control Card (701). The Control Card is part of a top section that also includes Configuration Logic (716) and a Configuration Database (717). Below this section is a central Mesh (706). The Mesh is connected to four Line Cards (702, 703, 704, 705). Each Line Card contains a Packet Processor (711, 712, 713, 714) and is connected to an external Port (707, 708, 709, 710). The entire structure is labeled as Network Element 700. FIG. 7 Gewirtz '227 at Figure 7. “FIG. 7 is a block diagram illustrating a network element according to one embodiment of the invention. Network element 700 may be implemented as any network element having queue systems as shown in FIGS. 1-2 and 5. For example, at least one of the line cards 702-705 may include a queue system as described above. Referring to FIG. 7, network element 700 includes, but is not limited to, a control card 701 (also referred to as a control plane) communicatively coupled to one or more line cards 702-705 (also referred to as interface cards or user planes) over a mesh 706, which may be a mesh network, an interconnect, a bus, or a combination thereof. A line card is also referred to as a data plane (sometimes referred to as a forwarding plane or a media plane). Each of the line cards 702-705 is associated with one or more interfaces (also referred to as ports), such as interfaces 707-710 respectively. Each line card includes a packet processor, routing functional block or logic (e.g., blocks 711-714) to route and/or forward packets via the corresponding interface according to a configuration (e.g., routing table) configured by control card 701, which may be configured by an administrator via an interface 715 (e.g., a command line interface or CLI). According to one embodiment, control card 701 includes, but is not limited to, configuration logic 716 and database 717 for storing information configured by configuration logic 716.” Gewirtz '227 at col. 9, ll. 22–45.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“Referring back to FIG. 7, in the case that network element 700 is a router (or is implementing routing functionality), control plane 701 typically determines how data (e.g., packets) is to be routed (e.g., the next hop for the data and the outgoing port for that data), and the data plane (e.g., lines cards 702-703) is in charge of forwarding that data. For example, control plane 701 typically includes one or more routing protocols (e.g., Border Gateway Protocol (BGP), Interior Gateway Protocol(s) (IGP) (e.g., Open Shortest Path First (OSPF), Routing Information Protocol (RIP), Intermediate System to Intermediate System (IS-IS), etc.), Label Distribution Protocol (LDP), Resource Reservation Protocol (RSVP), etc.) that communicate with other network elements to exchange routes and select those routes based on one or more routing metrics.” Gewirtz ’227 at col. 9, ll. 46–60.
4. The system of claim 1, wherein said system is part of a network attached storage (NAS) controller.	Gewirtz ’227 teaches “[t]he system of claim 1, wherein said system is part of a network attached storage (NAS) controller,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches wherein said system is part of a network attached storage (NAS) controller, by disclosing, as non-limiting examples, a network element that uses the multiprocessor system described in Gewirtz ’227. Network elements are used in a NAS controller. <i>See, e.g.:</i>

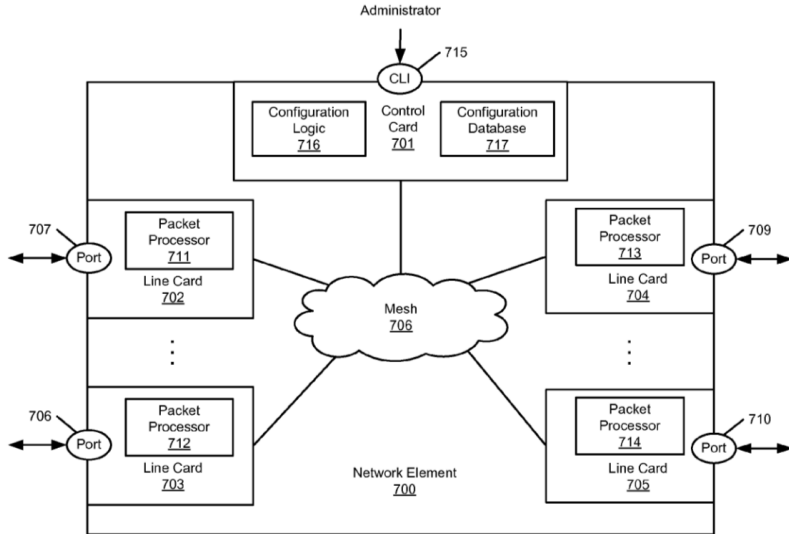
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 7 Gewirtz '227 at Figure 7. “FIG. 7 is a block diagram illustrating a network element according to one embodiment of the invention. Network element 700 may be implemented as any network element having queue systems as shown in FIGS. 1-2 and 5. For example, at least one of the line cards 702-705 may include a queue system as described above. Referring to FIG. 7, network element 700 includes, but is not limited to, a control card 701 (also referred to as a control plane) communicatively coupled to one or more line cards 702-705 (also referred to as interface cards or user planes) over a mesh 706, which may be a mesh network, an interconnect, a bus, or a combination thereof. A line card is also referred to as a data plane (sometimes referred to as a forwarding plane or a media plane). Each of the line cards 702-705 is associated with one or more interfaces (also referred to as ports), such as interfaces 707-710 respectively. Each line card includes a packet processor, routing functional block or logic (e.g., blocks 711-714) to route and/or forward packets via the corresponding interface according to a configuration (e.g., routing table) configured by control card 701, which may be configured by an administrator via an interface 715 (e.g., a command line interface or CLI). According to one embodiment, control card 701 includes, but is not limited to, configuration logic 716 and database 717 for storing information configured by configuration logic 716.” Gewirtz '227 at col. 9, ll. 22–45.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“Referring back to FIG. 7, in the case that network element 700 is a router (or is implementing routing functionality), control plane 701 typically determines how data (e.g., packets) is to be routed (e.g., the next hop for the data and the outgoing port for that data), and the data plane (e.g., lines cards 702-703) is in charge of forwarding that data. For example, control plane 701 typically includes one or more routing protocols (e.g., Border Gateway Protocol (BGP), Interior Gateway Protocol(s) (IGP) (e.g., Open Shortest Path First (OSPF), Routing Information Protocol (RIP), Intermediate System to Intermediate System (IS-IS), etc.), Label Distribution Protocol (LDP), Resource Reservation Protocol (RSVP), etc.) that communicate with other network elements to exchange routes and select those routes based on one or more routing metrics.” Gewirtz ’227 at col. 9, ll. 46–60.
5. The system of claim 1, wherein said stored value is decremented by a predefined value prior to returning said stored value to said at least one second processing element.	Gewirtz ’227 teaches “[t]he system of claim 1, wherein said stored value is decremented by a predefined value prior to returning said stored value to said at least one second processing element,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches wherein said stored value is decremented by a predefined value prior to returning said stored value to said at least one second processing element by disclosing, as non-limiting examples, decrementing the number of free entries upon a successful reservation read of the queue state register. The queue state register decrements the stored value by a predefined value. <i>See, e.g.:</i>

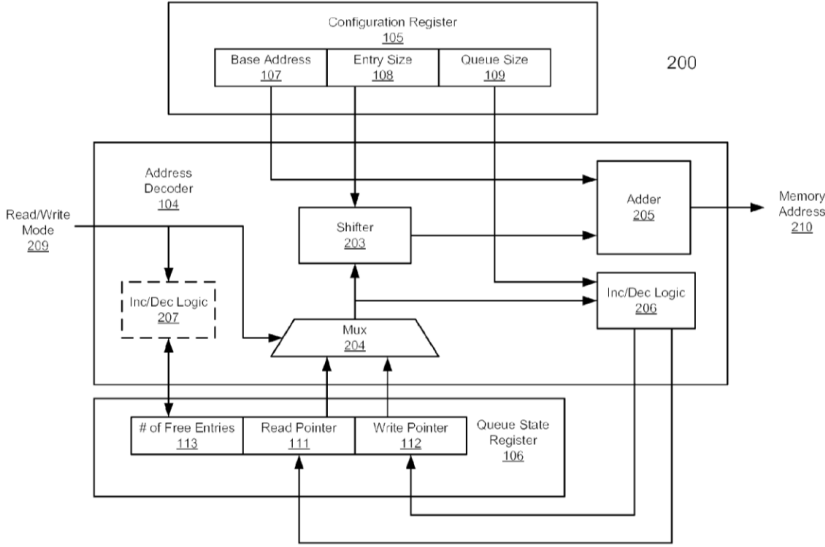
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 2 Gewirtz '227 at Figure 2. “In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz '227 at col. 4, l. 51 – col. 5, l. 6.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“The memory address is used to write data to the corresponding queue in memory 103. Thereafter, write pointer 112 may be atomically updated via hardware logic (e.g., decoder 104), for example, by incrementing or decrementing write pointer 112 dependent upon the specific implementation of the queue. When write pointer 112 is updated, in this example, being incremented, hardware logic may also determine whether write pointer 112 has reached the bottom or top of the queue, based on base address 107 and queue size 109 obtained from configuration register 105. If write pointer 112 does reach the top or bottom of the queue, write pointer 112 may be wrapped around or reset to a predetermined value such as zero. In addition, the number of free entries field 113 may also be updated by hardware logic, for example, by decrementing field 113, since an entry has been pushed into the queue. These operations are performed atomically and controlled by hardware logic. As a result, the agent does not have to worry about exclusive access to the queues. Again, it will be appreciated that certain firmware or low level software may also be utilized in conjunction with hardware logic.” Gewirtz ’227 at col. 6, ll. 6–25. “Further, the number of free entries 113 of the queue is updated via increment/decrement logic 207. Either read pointer 111 or write pointer 112 may also be updated via increment/decrement logic 206 based on queue size 109, dependent upon whether the access is a read or write access which is obtained from the output of multiplexer 204.” Gewirtz ’227 at col. 6, ll. 53–58. “Note that for purpose of illustration, system 200 represents one queue. It will be appreciated that multiple queues, each having a separate set of configurations register 105 and queue state register 106, may also be applied, while certain components such as shifter 203, multiplexer 204, adder 205, and/or increment/decrement logic 206-207 may or may not be shared among multiple queues.” Gewirtz ’227 at col. 6, ll. 59–65. “4. The method of claim 1, wherein the step of atomically updating the pointer of the hardware-based queue state register further comprises atomically updating, via a second increment/decrement hardware logic, a number of free entries of the queue in the hardware-based queue state register.” Gewirtz ’227 at col. 12, ll. 59–63. “9. The system of claim 8, wherein the hardware address decoder further comprises a second increment/decrement hardware logic to increment a number of free entries of the queue in the hardware-based queue state register in response to the read command.” Gewirtz ’227 at col. 13, ll. 63–67. “12. The system of claim 11, wherein the first increment/decrement hardware logic is adapted to obtain the queue size from the hardware-based configuration register and to wrap around the write pointer of the

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	hardware-based queue state register if the write pointer reaches the end of the queue based on the queue size.” Gewirtz ’227 at col. 14, ll. 18–23.
6. The system of claim 1, wherein at least one third processing element executes said command and notifies said output queue to issue said response.	Gewirtz ’227 teaches “[t]he system of claim 1, wherein at least one third processing element executes said command and notifies said output queue to issue said response,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches wherein at least one third processing element executes said command and notifies said output queue to issue said response, by disclosing, as non-limiting examples, a third processor in the multiprocessor system. The third processor executes the command and notifies the output queue to issue a response. <i>See, e.g.:</i> “FIG. 3 is a flow diagram illustrating a method for access a queue controlled by hardware logic according to one embodiment of the invention. Note that method 300 may be performed by processing logic which may include hardware, firmware, software, or a combination there of. For example, method 300 may be performed by systems as shown in FIGS. 1 and/or 2. Referring to FIG. 3, at block 301, a command (e.g., a memory access command, such as a read command or a write command) is received from an agent via a predetermined memory mapped register (e.g., data access register). The agent may be one of the software processes, each being executed by one of multiple processor cores of a network processor of a network element. At block 302, a queue is identified based on the predetermined memory mapped register, where the queue has been allocated in a memory. At block 303, a pointer of the queue is atomically read from a hardware-based queue state register associated with the queue. At block 304, data is accessed (e.g., read or write) at a memory location of a memory based on the pointer, where the memory location is allocated as a part of the queue. In one embodiment, a base address is obtained from a hardware-based configuration register of the queue, where the configuration register is configured by system software prior to being used. A memory address is generated via hardware logic based on the pointer and the base address. The data is accessed at the memory location based on the memory address. Thereafter, at block 305, the pointer of the HW-based queue state register of the queue is updated atomically, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. In addition, the number of free entries of the queue may also be updated.” Gewirtz ’227 at col. 7, ll. 21–53.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
[7pre] A system for dividing and synchronizing a processing task across a plurality of processing elements comprising:	Gewirtz '227 teaches “[a] system for dividing and synchronizing a processing task across a plurality of processing elements,” including at least the following exemplary citations. <i>See</i> above at row [1pre].
[7a] an input queue implemented in hardware;	Gewirtz '227 teaches “an input queue implemented in hardware,” including at least the following exemplary citations. <i>See</i> above at row [1a].
[7b] an output queue implemented in hardware;	Gewirtz '227 teaches “an output queue implemented in hardware,” including at least the following exemplary citations. <i>See</i> above at row [1b].
[7c] a first processing element having access to said input queue and said output queue;	Gewirtz '227 teaches “a first processing element having access to said input queue and said output queue,” including at least the following exemplary citations. <i>See</i> above at row [1c].
[7d] at least one second processing element in communication with said input queue;	Gewirtz '227 teaches “at least one second processing element in communication with said input queue,” including at least the following exemplary citations. For example, Gewirtz '227 teaches at least one second processing element in communication with said input queue, by disclosing, as non-limiting examples, a second processor in the multiprocessor system that executes software agents. The second processor, the second processing element is in communication with the input queue. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	FIG. 1 Gewirtz '227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301 ↓ Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302 ↓ Atomically read a pointer from a hardware-based queue state register associated with the queue. 303 ↓ Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304 ↓ Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system 500. At the top, an 'Agent(s) 120' is connected via a bidirectional arrow to a set of 'Memory Mapped Registers 101'. This set includes a group of 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', and 'Access Register for Qn' (labeled 505), and another group of 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', and 'Access Register for PGn' (labeled 506). The Memory Mapped Registers 101 are connected to a central 'Scanner 504' and 'Address Decoder 104' block. This central block is also connected to a set of 'HW-Based Registers 102' which includes a 'Configuration Register 105' and a 'Queue State Register 106'. The central block is further connected to a 'Memory 103' structure containing 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', and 'Priority Group N'. Gewirtz '227 at Figure 5.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6.

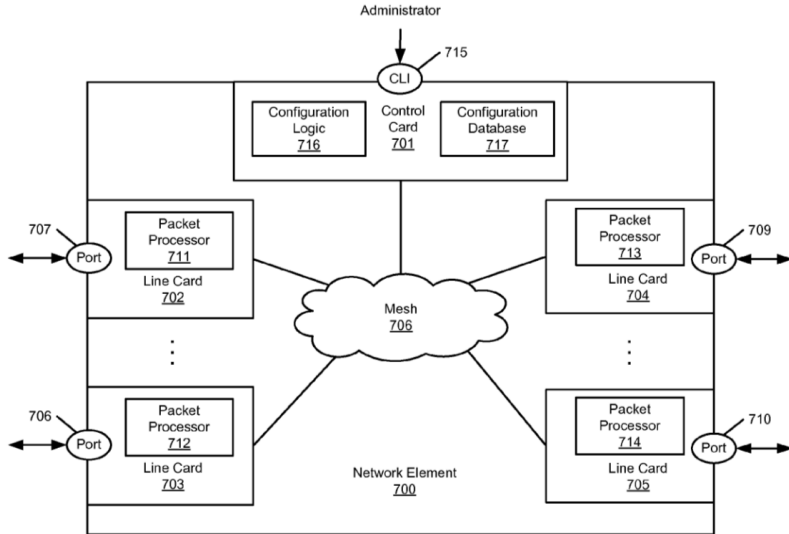
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 7 Gewirtz '227 at Figure 7. “In a multi-processor system, it is frequently desirable to build queues, or first-in-first-out (FIFO) buffers, in memory. A queue is a data structure with FIFO properties where agents push elements onto the queue (writes) and pop elements off of the queue (reads). This is normally accomplished by allocating storage for the queue elements, and separately allocating storage for the queue state which includes read and write pointers, and any other desired information.” Gewirtz '227 at col. 1, ll. 12–19. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz '227 at col. 4, ll. 16–33.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“According to one embodiment, when an agent such as agent 120 desires to write data to a specific queue, the agent may specific perform a write operation to an access register associated with the specific queue. For example, if agent 120 desires to write data to queue 0, it may write data to access register 115. In response, address decoder 104 may access a set of hardware-based registers associated with the queue being accessed (e.g., hardware-based registers associated with queue 0). In this example, decoder 104 may atomically obtain a write pointer 112 from queue state register 106 associated with the queue being accessed. In addition, decoder 104 may obtain base address 107 and entry size 108 from configuration register 105 associated with the queue being accessed. A memory address for the entry to be written is determined based on the write pointer 112, base address 107, and entry size 108. In one embodiment, the memory address of the entry being written can be determined as follows: [Equation] where (Write Pointer*Entry Size) represents an offset from the base address.” Gewirtz ’227 at col. 5, l. 50 – col. 6, l. 4. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[7e] a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element;	Gewirtz ’227 teaches “a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element,” including at least the following exemplary citations. <i>See above at row [1e].</i>
[7f] computer storage storing instructions, which when executed	Gewirtz ’227 teaches computer storage storing instructions executed by said at least one second processing element, including at least the following exemplary citations.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
by said at least one second processing element:	<i>See</i> above at row [1f].
[7g] receives a command from said first processing element;	Gewirtz '227 teaches “receives a command from said first processing element,” including at least the following exemplary citations. For example, Gewirtz '227 teaches that the second processing element executes instructions that causes it to receive a command from said first processing element, by disclosing, as non-limiting examples, a second processor executing a software agent that receives a command from a first processor via inter-processor communications. The second processor, the second processing element receives a command from the first processing element. <i>See, e.g.:</i> <pre> graph TD 301[Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301] --> 302[Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302] 302 --> 303[Atomically read a pointer from a hardware-based queue state register associated with the queue. 303] 303 --> 304[Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304] 304 --> 305[Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305] </pre> 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	<pre> graph TD 601[Group queues into multiple priority groups, each priority group being associated with a particular priority. 601] --> 602[Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602] 602 --> 603[Identify a priority associated with the command based on the predetermined memory-mapped register. 603] 603 --> 604[Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604] 604 --> 605[Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605] 605 --> 606[Atomically data in the identified queue based on the pointer. 606] 606 --> 607[Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607] </pre> FIG. 6 Gewirtz '227 at Figure 6. “According to one aspect of the invention, a command is received from a first agent via a first predetermined memory-mapped register, the first agent being one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. A first queue associated with the command is identified based on the first predetermined memory-mapped register, where the first queue was allocated in a memory. A pointer is atomically read from a first hardware-based queue state register associated with the first queue. Data is atomically accessed at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the first queue. The pointer stored in the first hardware-based queue state register is atomically updated, including incrementing, via a first increment/decrement hardware logic, the pointer of the first hardware-based queue state register, reading a queue size of the queue from a first hardware-based configuration register associated with the first queue, and wrapping around, via the first increment/decrement hardware logic, the pointer of the first hardware-based queue state register if the pointer reaches an end of the first queue based on the queue size. As a result, the pointer is obtained and updated atomically through hardware so that the software process represented by the first agent is not required to obtain an exclusive access control of the first queue from the Operating System.” Gewirtz '227 at col. 1, ll. 39–64.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“In one embodiment, state register 106 includes an optional built-in credit counter represented by the number of free entries 113 which can be accessed via a status register of memory mapped registers 101 corresponding to the queue. The credit counter originally contains the number of entries in the queue. When an agent wants to write an element into the queue, it first reads the reservation register to see if an entry is available. An agent may be one of multiple agents representing software processes, each being executed by one of multiple processor cores of a network processor in a network element. The queue hardware can return a pass/fail value to the agent in response to the read, depending on whether there is a reservation available, which may be specified via a reserved field (not shown) of state register 106. When an agent receives a pass value, it can perform a write to one of the data access registers. If it receives a fail value, it must attempt to obtain a reservation again. If the agent decides not to write an element on to the queue, the agent may be required to write the reservation register to free the credit for subsequent use. The available reservation count is maintained by hardware: it is decremented when a successful reservation read request is processed, and incremented when either a reservation register write is seen, or when an element is popped off of the queue.” Gewirtz ’227 at col. 4, l. 51 – col. 5, l. 6. “One of the advantages of the programmable queue as shown in FIGS. 1-2 is that, once defined, system agents can perform normal read and write commands that are processed atomically as pushes and pops from that queue. This reduces the latency seen when reading from the queue and, even more importantly, eliminates the need for a locking access to gain exclusive ownership of the memory holding all of the queue data. This avoids contention over the lock. Memory contention over shared memory space is always detrimental to multi-processor system performance. In addition to the atomicity and locking benefits, the queues are extremely flexible, and can include a variety of optional features. They can be defined to have a large range of depths and element sizes. They can have built-in credit counters (e.g., represented by the number of free entries of a queue state register) to assist with system flow control. They can generate processor interrupts when their state changes to assist with message passing and resource management. They can be accessed with blocking and non-blocking reads depending on the desired behavior when an agent tries to read an empty queue. Finally, they can be organized into priority groups to assist in the processing of certain kinds of workloads.” Gewirtz ’227 at col. 6, l. 66 – col. 7, l. 20. “FIG. 3 is a flow diagram illustrating a method for access a queue controlled by hardware logic according to one embodiment of the invention. Note that method 300 may be performed by processing logic which may include hardware, firmware, software, or a combination there of. For example, method 300 may be performed by systems as shown in FIGS. 1 and/or 2. Referring to FIG. 3, at block 301, a command (e.g., a memory access command, such as a read command or a write command) is received from an agent via a

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	predetermined memory mapped register (e.g., data access register). The agent may be one of the software processes, each being executed by one of multiple processor cores of a network processor of a network element. At block 302, a queue is identified based on the predetermined memory mapped register, where the queue has been allocated in a memory. At block 303, a pointer of the queue is atomically read from a hardware-based queue state register associated with the queue. At block 304, data is accessed (e.g., read or write) at a memory location of a memory based on the pointer, where the memory location is allocated as a part of the queue. In one embodiment, a base address is obtained from a hardware-based configuration register of the queue, where the configuration register is configured by system software prior to being used. A memory address is generated via hardware logic based on the pointer and the base address. The data is accessed at the memory location based on the memory address. Thereafter, at block 305, the pointer of the HW-based queue state register of the queue is updated atomically, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. In addition, the number of free entries of the queue may also be updated.” Gewirtz ’227 at col. 7, ll. 21–53. “FIG. 6 is a flow diagram illustrating a method for accessing a queue according to another embodiment of the invention. Note that method 600 may be performed by processing logic which may include hardware, firmware, software, or a combination thereof. For example, method 600 may be performed by system 500 of FIG. 5. Referring to FIG. 6, at block 601, multiple queues are grouped into one or more priority groups, each having a particular (e.g., different) priority. For example, the queues are grouped by system software when configuring the queues, for example, by an option field (not shown) in the corresponding hardware-based queue configuration register 105 or hardware-based queue state register 106. At block 602, a command or request is received from an agent for accessing data in one of the queues via a predetermined memory-mapped register. The agent may be one of multiple agents, each being executed by one of processor cores of a network processor of a network element. At block 603, a priority associated with the command is identified based on the predetermined memory-mapped register. At block 604, processing logic scans one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. At block 605, a pointer is atomically read from a hardware based queue state register associated with the identified queue. At block 606, data is accessed in the identified queue based on the pointer. Thereafter, at block 607, the pointer of the hardware-based queue state register associated with the identified queue is atomically updated. In addition, the number of free entries may also be updated.” Gewirtz ’227 at col. 8, l. 59 – col. 9, l. 21.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“receiving a command from an agent via a first predetermined memory-mapped register, the agent being one of a plurality of agents representing a plurality of software processes, each being executed by one of a plurality of processor cores of a network processor in the network element;” Gewirtz ’227 at col. 12, ll. 12–17. “5. The method of claim 4, further comprising: receiving a second command from the agent to query a status of the queue via a second predetermined memory-mapped register; and in response to the second command, returning to the agent the status of the queue including the number of free entries of the queue from the hardware-based queue state register.” Gewirtz ’227 at col. 12, l. 64 – col. 13, l. 4.
[7h] accesses said reservation register and reads said stored value;	Gewirtz ’227 teaches “accesses said reservation register and reads said stored value,” including at least the following exemplary citations. <i>See</i> above at row [1g].
[7i] determines when said read value indicates available space in said input queue to issue said command;	Gewirtz ’227 teaches “determines when said read value indicates available space in said input queue to issue said command,” including at least the following exemplary citations. <i>See</i> above at row [1h].
[7j] issues said command to said input queue on behalf of said first processing element; and	Gewirtz ’227 teaches “issues said command to said input queue on behalf of said first processing element,” including at least the following exemplary citations. For example, Gewirtz ’227 teaches that the second processing element executes instructions that causes it to issue said command to said input queue on behalf of said first processing element, by disclosing, as non-limiting examples, a second processor executing a software agent. The second processor issues the command to the input queue on behalf of the first processing element. <i>See, e.g.:</i>

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically read data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6. “According to one embodiment, when an agent such as agent 120 desires to write data to a specific queue, the agent may specific perform a write operation to an access register associated with the specific queue. For example, if agent 120 desires to write data to queue 0, it may write data to access register 115. In response, address decoder 104 may access a set of hardware-based registers associated with the queue being accessed (e.g., hardware-based registers associated with queue 0). In this example, decoder 104 may atomically obtain a write pointer 112 from queue state register 106 associated with the queue being accessed. In addition, decoder 104 may obtain base address 107 and entry size 108 from configuration register 105 associated with the queue being accessed. A memory address for the entry to be written is determined based on the write pointer 112, base address 107, and entry size 108. In one embodiment, the memory address of the entry being written can be determined as follows: [Equation] where (Write Pointer*Entry Size) represents an offset from the base address.” Gewirtz '227 at col. 5, l. 50 – col. 6, l. 4. “FIG. 3 is a flow diagram illustrating a method for access a queue controlled by hardware logic according to one embodiment of the invention. Note that method 300 may be performed by processing logic which may include hardware, firmware, software, or a combination there of. For example, method 300 may be performed by systems as shown in FIGS. 1 and/or 2. Referring to FIG. 3, at block 301, a command (e.g., a memory

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	access command, such as a read command or a write command) is received from an agent via a predetermined memory mapped register (e.g., data access register). The agent may be one of the software processes, each being executed by one of multiple processor cores of a network processor of a network element. At block 302, a queue is identified based on the predetermined memory mapped register, where the queue has been allocated in a memory. At block 303, a pointer of the queue is atomically read from a hardware-based queue state register associated with the queue. At block 304, data is accessed (e.g., read or write) at a memory location of a memory based on the pointer, where the memory location is allocated as a part of the queue. In one embodiment, a base address is obtained from a hardware-based configuration register of the queue, where the configuration register is configured by system software prior to being used. A memory address is generated via hardware logic based on the pointer and the base address. The data is accessed at the memory location based on the memory address. Thereafter, at block 305, the pointer of the HW-based queue state register of the queue is updated atomically, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. In addition, the number of free entries of the queue may also be updated.” Gewirtz ’227 at col. 7, ll. 21–53. “FIG. 6 is a flow diagram illustrating a method for accessing a queue according to another embodiment of the invention. Note that method 600 may be performed by processing logic which may include hardware, firmware, software, or a combination thereof. For example, method 600 may be performed by system 500 of FIG. 5. Referring to FIG. 6, at block 601, multiple queues are grouped into one or more priority groups, each having a particular (e.g., different) priority. For example, the queues are grouped by system software when configuring the queues, for example, by an option field (not shown) in the corresponding hardware-based queue configuration register 105 or hardware-based queue state register 106. At block 602, a command or request is received from an agent for accessing data in one of the queues via a predetermined memory-mapped register. The agent may be one of multiple agents, each being executed by one of processor cores of a network processor of a network element. At block 603, a priority associated with the command is identified based on the predetermined memory-mapped register. At block 604, processing logic scans one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. At block 605, a pointer is atomically read from a hardware based queue state register associated with the identified queue. At block 606, data is accessed in the identified queue based on the pointer. Thereafter, at block 607, the pointer of the hardware-based queue state register associated with the identified queue is atomically updated. In addition, the number of free entries may also be updated.” Gewirtz ’227 at col. 8, l. 59 – col. 9, l. 21.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“receiving a command from an agent via a first predetermined memory-mapped register, the agent being one of a plurality of agents representing a plurality of software processes, each being executed by one of a plurality of processor cores of a network processor in the network element;” Gewirtz ’227 at col. 12, ll. 12–17. “receiving a command for accessing data in one of the plurality of queues, wherein the command is received from an agent via a predetermined memory mapped register, wherein the agent is one of a plurality of agents representing a plurality of software processes, each being executed by one of a plurality of processor cores of a network processor in the network element;” Gewirtz ’227 at col. 14, ll. 37–43.
[7k] wherein said first processing element receives a response corresponding to said command from said output queue.	
8. The system of claim 7, wherein said input queue is a first-in-first-out (FIFO) buffer.	Gewirtz ’227 teaches “[t]he system of claim 7, wherein said input queue is a first-in-first-out (FIFO) buffer,” including at least the following exemplary citations. <i>See above at row 2.</i>
9. The system of claim 7, wherein said system is part of a storage area network (SAN) controller.	Gewirtz ’227 teaches “[t]he system of claim 7, wherein said system is part of a storage area network (SAN) controller,” including at least the following exemplary citations. <i>See above at row 3.</i>
10. The system of claim 7, wherein said system is part of a network attached storage (NAS) controller.	Gewirtz ’227 teaches “[t]he system of claim 7, wherein said system is part of a network attached storage (NAS) controller,” including at least the following exemplary citations. <i>See above at row 4.</i>
11. The system of claim 7, wherein said stored value is decremented by a predefined value prior to returning said stored value to said	Gewirtz ’227 teaches “[t]he system of claim 7, wherein said stored value is decremented by a predefined value prior to returning said stored value to said at least one second processing element,” including at least the following exemplary citations. <i>See above at row 5.</i>

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
at least one second processing element.	
12. The system of claim 7, wherein at least one third processing element executes said command and notifies said output queue to issue said response.	Gewirtz '227 teaches “[t]he system of claim 7, wherein at least one third processing element executes said command and notifies said output queue to issue said response,” including at least the following exemplary citations. <i>See above at row 6.</i>
[13pre] A system for dividing and synchronizing a processing task across a plurality of processing elements comprising:	Gewirtz '227 teaches “[a] system for dividing and synchronizing a processing task across a plurality of processing elements,” including at least the following exemplary citations. <i>See above at row [1pre].</i>
[13a] an input queue implemented in hardware;	Gewirtz '227 teaches “an input queue implemented in hardware,” including at least the following exemplary citations. <i>See above at row [1a].</i>
[13b] an output queue implemented in hardware;	Gewirtz '227 teaches “an output queue implemented in hardware,” including at least the following exemplary citations. <i>See above at row [1b].</i>
[13c] a first processing element having access to said input queue and said output queue;	Gewirtz '227 teaches “a first processing element having access to said input queue and said output queue,” including at least the following exemplary citations. <i>See above at row [1c].</i>
[13d] at least one second processing element in communication with said input queue and said output queue;	Gewirtz '227 teaches “at least one second processing element in communication with said input queue and said output queue,” including at least the following exemplary citations. For example, Gewirtz '227 teaches at least one second processing element in communication with said input queue and said output queue, by disclosing, as non-limiting examples, a second processor executing a

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	software agent. The second processor, the second processing element is in communication with the input and output queue. <i>See, e.g.:</i> The diagram illustrates a system 100. At the top, an 'Agent(s) (e.g., SW process) 120' is connected via a bidirectional arrow to 'Access Register 0 115' and 'Status Register 0'. These registers are part of 'Queue 0'. Below Queue 0 is 'Queue 1' with 'Access Register 1' and 'Status Register 1'. Further down are 'Access Register N' and 'Status Register N' for 'Queue N'. These registers are collectively labeled as 'Memory Mapped Registers 101'. To the right of the registers is an 'Address Decoder 104'. A bidirectional arrow connects the 'Memory Mapped Registers 101' to the 'Address Decoder 104'. Another bidirectional arrow connects the 'Address Decoder 104' to 'Memory 103', which contains 'Queue 0', 'Queue 1', and 'Queue N'. Below the address decoder is a large block labeled 'HW-Based Registers 102'. This block contains two sub-sections: 'Configuration Register 105' with fields 'Base Address 107', 'Entry Size 108', and 'Queue Size 109'; and 'Queue State Register 106' with fields 'Read Pointer 111', 'Write Pointer 112', and '# of Free Entries 113'. A bidirectional arrow connects the 'Address Decoder 104' to the 'HW-Based Registers 102'. FIG. 1 Gewirtz '227 at Figure 1.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Receive a command from an agent via a first predetermined memory-mapped register, the agent being one of the agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 301 Identify a queue associated with the command based on the first predetermined memory-mapped register, wherein the queue was allocated in a memory. 302 Atomically read a pointer from a hardware-based queue state register associated with the queue. 303 Atomically access data at a memory location of the memory based on the pointer, the memory location having been allocated as a part of the queue. 304 Atomically update the pointer of the HW-based queue state register of the queue, including incrementing the pointer, reading a queue size from a HW-based configuration register, and wrapping around the pointer if the pointer reaches the end of the queue based on the queue size. 305 300 FIG. 3 Gewirtz '227 at Figure 3.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	The diagram, labeled FIG. 5, illustrates a system 500. At the top, an 'Agent(s) 120' is connected via a bidirectional arrow to a set of 'Memory Mapped Registers 101'. This set includes 'Access Register for Q0', 'Access Register for Q1', 'Access Register for Q2', an ellipsis, 'Access Register for Qn', another ellipsis, 'Access Register for PG0 507', 'Access Register for PG1', 'Access Register for PG2', an ellipsis, and 'Access Register for PGn'. A bracket labeled '505' groups the Q registers, and a bracket labeled '506' groups the PG registers. Below these is the label 'Memory Mapped Registers 101'. To the right, 'HW-Based Registers 102' include a 'Configuration Register 105' and a 'Queue State Register 106'. A bidirectional arrow connects the HW-based registers to a 'Scanner 504' and 'Address Decoder 104' block. This block is also connected via bidirectional arrows to the 'Memory Mapped Registers 101' and to a 'Memory 103' block. The memory block contains 'Priority Group 0 (e.g., queues 0-3) 501', 'Priority Group 1 (e.g., queues 4-7) 502', 'Priority Group 2 (e.g., queues 8-11) 503', an ellipsis, and 'Priority Group N'. The label 'FIG. 5' is positioned to the right of the diagram. Gewirtz '227 at Figure 5.

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	Group queues into multiple priority groups, each priority group being associated with a particular priority. 601 ↓ Receive a command for accessing data in one of the queues, where the command is received from an agent via a predetermined memory-mapped register, where the agent is one of agents representing software processes, each being executed by one of processor cores of a network processor in the network element. 602 ↓ Identify a priority associated with the command based on the predetermined memory-mapped register. 603 ↓ Scan one or more of the priority groups whose priority satisfies the command's priority to identify the one of the queues that has the highest priority among the one or more priority groups being scanned. 604 ↓ Atomically read a pointer from a hardware based queue state register associated with the identified queue. 605 ↓ Atomically data in the identified queue based on the pointer. 606 ↓ Atomically update the pointer of the hardware based queue state register associated with the identified queue. 607 600 FIG. 6 Gewirtz '227 at Figure 6.

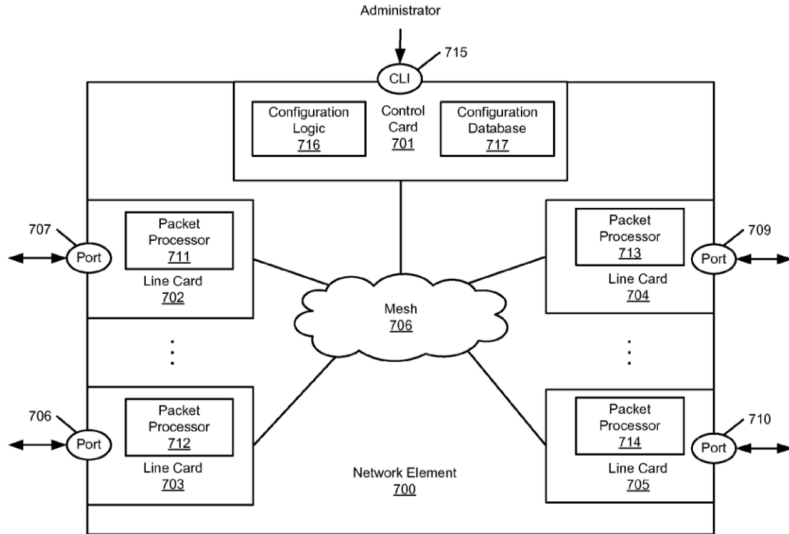
Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	 FIG. 7 Gewirtz '227 at Figure 7. “In a multi-processor system, it is frequently desirable to build queues, or first-in-first-out (FIFO) buffers, in memory. A queue is a data structure with FIFO properties where agents push elements onto the queue (writes) and pop elements off of the queue (reads). This is normally accomplished by allocating storage for the queue elements, and separately allocating storage for the queue state which includes read and write pointers, and any other desired information.” Gewirtz '227 at col. 1, ll. 12–19. “When a programmable queue is enabled and defined, according to one embodiment, agents 120 can perform reads and writes to the corresponding data access registers 101, which may cause queue operations to be performed atomically. These accesses may cause the appropriate queue element to be read or written, and the queue state to be updated accordingly by hardware logic. These two operations may appear to be atomic to all agents in the system. For reads, there can be two kinds of data access registers: blocking and non-blocking which have different behaviors when the queue is empty. Non-blocking reads may return a fail value to the requesting agent, whereas blocking reads may be queued in the queue element storage space (e.g., memory 103) in the order that they are received. When an element is subsequently written to the queue, the oldest waiting read return information (i.e. where to return the data) is read from the queue storage and the write data is sent to the agent that made that read.” Gewirtz '227 at col. 4, ll. 16–33.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
	“According to one embodiment, when an agent such as agent 120 desires to write data to a specific queue, the agent may specific perform a write operation to an access register associated with the specific queue. For example, if agent 120 desires to write data to queue 0, it may write data to access register 115. In response, address decoder 104 may access a set of hardware-based registers associated with the queue being accessed (e.g., hardware-based registers associated with queue 0). In this example, decoder 104 may atomically obtain a write pointer 112 from queue state register 106 associated with the queue being accessed. In addition, decoder 104 may obtain base address 107 and entry size 108 from configuration register 105 associated with the queue being accessed. A memory address for the entry to be written is determined based on the write pointer 112, base address 107, and entry size 108. In one embodiment, the memory address of the entry being written can be determined as follows: [Equation] where (Write Pointer*Entry Size) represents an offset from the base address.” Gewirtz ’227 at col. 5, l. 50 – col. 6, l. 4. “FIG. 5 is a block diagram illustrating a queue system according to another embodiment of the invention. Referring to FIG. 5, similar to system 100 of FIG. 1, system 500 includes a set of memory mapped registers 101 as part of data access registers, one or more sets of hardware-based registers 102 each having a configuration register 105 and queue state register 106 which corresponds to a queue defined in memory 103, and address decoder 104 for decode or interpret memory address based on information received from data access registers 101 and hardware-based registers 102 for accessing one or more queues defined in memory 103, as described above. For illustration purpose, certain reference numbers are maintained identical for certain components having similar or identical functionalities with respect to those in FIG. 1.” Gewirtz ’227 at col. 8, ll. 18–31.
[13e] a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element;	Gewirtz ’227 teaches “a reservation register implemented in hardware storing a value indicative of available space in said input queue, said reservation register accessible by both said first processing element and said at least one second processing element,” including at least the following exemplary citations. <i>See above at row [1e].</i>
[13f] computer storage storing instructions, which when executed	Gewirtz ’227 teaches computer storage storing instructions that are executed by said at least one second processing element including at least the following exemplary citations.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
by said at least one second processing element;	<i>See</i> above at row [1f].
[13g] receives a command from said first processing element;	Gewirtz '227 teaches “receives a command from said first processing element,” including at least the following exemplary citations. <i>See</i> above at row [7g].
[13h] accesses said reservation register and reads said stored value;	Gewirtz '227 teaches “accesses said reservation register and reads said stored value,” including at least the following exemplary citations. <i>See</i> above at row [1g].
[13i] determines when said read value indicates available space in said input queue to issue said command;	Gewirtz '227 teaches “determines when said read value indicates available space in said input queue to issue said command,” including at least the following exemplary citations. <i>See</i> above at row [1h].
[13j] issues said command to said input queue on behalf of said first processing element; and	Gewirtz '227 teaches “issues said command to said input queue on behalf of said first processing element,” including at least the following exemplary citations. <i>See</i> above at row [7j].
[13k] receives a response corresponding to said command from said output queue.	Gewirtz '227 teaches “receives a response corresponding to said command from said output queue,” including at least the following exemplary citations.
14. The system of claim 13, wherein said input queue is a first-in-first-out (FIFO) buffer.	Gewirtz '227 teaches “[t]he system of claim 13, wherein said input queue is a first-in-first-out (FIFO) buffer,” including at least the following exemplary citations. <i>See</i> above at row 2.
15. The system of claim 13, wherein said system is either part	Gewirtz '227 teaches “[t]he system of claim 13, wherein said system is either part of a storage area network (SAN) controller,” including at least the following exemplary citations.

Exhibit 596-06

Claim Language	Exemplary Evidence of Anticipation and/or Obviousness
of a storage area network (SAN) controller.	<i>See above at row 3.</i>
16. The system of claim 13, wherein said system is part of a network attached storage (NAS) controller.	Gewirtz '227 teaches “[t]he system of claim 13, wherein said system is part of a network attached storage (NAS) controller,” including at least the following exemplary citations. <i>See above at row 4.</i>
17. The system of claim 13, wherein said stored value is decremented by a predefined value prior to returning said stored value to said at least one second processing element.	Gewirtz '227 teaches “[t]he system of claim 13, wherein said stored value is decremented by a predefined value prior to returning said stored value to said at least one second processing element,” including at least the following exemplary citations. <i>See above at row 5.</i>
18. The system of claim 13, wherein at least one third processing element executes said command and notifies said output queue to issue said response.	Gewirtz '227 teaches “[t]he system of claim 13, wherein at least one third processing element executes said command and notifies said output queue to issue said response,” including at least the following exemplary citations. <i>See above at row 6.</i>