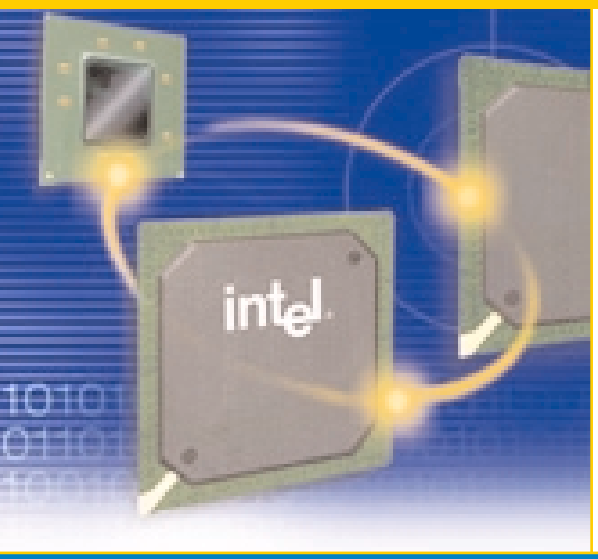
Volume 06 Issue 03 Published, August 15, 2002 ISSN 1535766X



Intel® Technology Journal

Network Processors

The Next Generation of Intel IXP Network Processors

A compiled version of all papers from this issue of the Intel Technology Journal can be found at:
http://developer.intel.com/technology/itj/index.htm

The Next Generation of Intel IXP Network Processors

Matthew Adiletta, Mark Rosenbluth, Debra Bernstein,
Gilbert Wolrich, Hugh Wilkinson
Intel Communications Group, Intel Corporation

Index words: network processors, IXP, communication architecture, routing, switching, Ethernet, ATM, multi-service switches, multi-processors, microprocessor architecture, multi-threading, 10Gb/s, OC-192, OC-48.

ABSTRACT

This paper describes the next generation of Intel Internet eXchange Processors (IXPs). The IXP family of network processors is growing with the addition of three new parts. This paper focuses on the high-end IXP2800. The IXP2800 is capable of 10Gb/s ATM, OC-192 POS, or Ethernet data processing. The IXP2400 is a sibling and is capable of sustained OC-48 or Quad Gigabit Ethernet data processing. The third new member of the family is the IXP440, which is a Customer Premise Equipment (CPE) class device. The IXP2800 and IXP2400 share the architectural chassis, major functional units and software programming model, as well as the same instruction set.

This paper covers system architecture, micro-architecture, and functional unit characteristics, and provides insights into the challenges of processing an incoming cell or packet every 35ns. Special attention is paid to the problem of enqueuing and dequeuing onto and from a linked list that is maintained in external memory. The challenge is that cell and packet arrival rates are approaching the external memory access latencies.

Finally, this paper concludes with future directions for the IXP family.

INTRODUCTION

The IXP1200 is the first member of the IXP network processor family. It has been designed into over 200 products at a wide range of companies and market segments. Introduced in 1999, it provided OC-12 or Gigabit Ethernet packet processing capability. The IXP2800 and IXP2400 leverage many learnings from the experiences of the IXP1200. In particular, refining the computational needs and memory access bandwidths required at different incoming line rates has led to providing both greater computational capability and memory bandwidth. The IXP2800 provides over 23,000 MIPs, the IXP2400 4800 MIPs, and the IXP1200 1200 MIPs.

There are many competing approaches to network processing. One approach is through dedicated hardware state machines with configurability, or minimal software programming capability. Another approach is through very high-performance microprocessors that are provided with a very flexible software programming capability. The IXP family employs the flexible software approach, with state-of-the-art compilers and debuggers. This allows the IXP to address many market segments and allows our customers to develop a base hardware platform that they can then use in different applications. Additionally, by providing a flexible software platform, customers can download features and capabilities to enhance product lifespans and product experiences.

The first section of this paper describes three system architectures using the IXP2800. It is interesting to note that the system architectures detailed may also be applied to the IXP2400, albeit at a lower incoming cell or packet rate.

The second section focuses on the internal architecture of the IXP2800. The chassis, or the interconnection between the different functional units, will be described, as well as the major functional units.

The third section provides details on the microengine, which is the processor arrayed in either 16 instantiations for the IXP2800 or eight instantiations for the IXP2400.

The challenges of packet or cell processing at 10 gigabits per second are then described along with the solution employed by the IXP2800. Then flexibility versus software complexity is discussed. The paper concludes with a discussion of the future directions of the IXP high-end processor.

IXP2800 SYSTEM EXAMPLES

Many system architectures are possible employing the IXP2800. This section details three configurations of IXP2800 processors supporting various switching applications.

Metro-LAN 10 Gigabit Ethernet Switching or OC-192 Packet over SONET Switching Blade

In this system architecture, two IXP2800 network processors are used (Figure 1). The top IXP2800 is used for ingress processing. Ingress processing tasks may include classification, metering, policing, congestion avoidance, statistics, segmentation, and traffic scheduling into a switching fabric. The bottom IXP2800 is used for egress processing. Egress processing tasks may similarly include reassembly, congestion avoidance, statistics, and traffic shaping. Both input and output buffering are supported using small DRAM buffers linked together by linked lists maintained in SRAM.

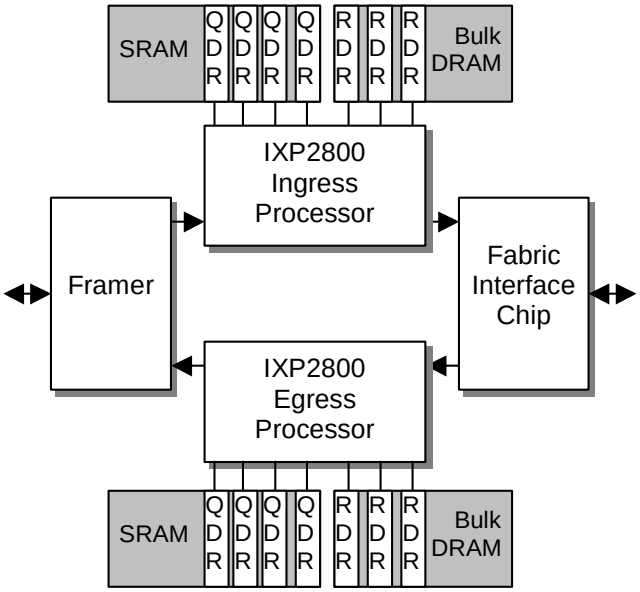


Figure 1: Metro-LAN 10 Gigabit Ethernet switching or OC-192 Packet over SONET configuration

The framer interfaces to the two IXP2800 network processors using the interface defined by the Optical Internetworking Forum SPI-4.2 Implementation Agreement. The fabric interfaces to the two IXP2800 network processors using the Common Switch Interface Specification-L1 (CSIX-L1) protocol implemented on top of the SPI-4.2 physical signaling. The ingress and egress network processors present a single full-duplex interface to the fabric, as if they were a single chip.

Packets are streamed into the ingress IXP2800 at or above line rate. The processing of a packet begins upon receipt of the initial part. The parts of a packet are received, reassembled, processed, buffered into DRAM, and enqueued for transmission into the fabric. Subsequently, the packet is scheduled and transmitted into the fabric to be processed by an egress IXP2800. The egress IXP2800 reassembles the packet in DRAM and queues the packet for outgoing transmission. Subsequently, the packet is transmitted out the egress framer. At both the ingress IXP2800 and the egress IXP2800, packet data is written to and read from DRAM only a single time. The DRAM interface constists of three Rambus DRAM (RDRAM) channels operating at a clock rate of up to 533MHz, offering an aggregate peak bandwidth of 51Gb/s.

At a maximum packet rate of approximately 15 million packets per second for 10 Gigabit Ethernet, the IXP2800 supports a service time of 8.53 usec per packet for receive and transmit processing by distributing the processing across 128 different computation threads. The IXP2800 can support the execution of up to 1493 microengine instructions per packet (93 instructions per microengine * 16 microengines) at this packet rate and a clock rate of 1.4GHz.

At a maximum packet rate of approximately 28 million packets per second for OC-192 Packet over SONET, the IXP2800 supports a service time of 4.57 usec per packet for receive and transmit. The IXP2800 can support the execution of up to 800 microengine instructions per packet at this packet rate and a clock rate of 1.4GHz.

The IXP2800 supports four QDR II SRAM interfaces that may be clocked at up to 250MHz. Each interface supports an independent read and write port, providing an aggregate read rate of 32Gb/s and, simultaneously, an aggregate write rate of 32Gb/s. These interfaces may be used to access SRAM. Additionally, Ternary Content Adressable Memories (TCAM), which support the same interface, are becoming available.

Classification may be performed using TRIE data structures in SRAM, hashing and collision resolution in SRAM, and/or TCAM tables.

At 15 million packets per second, the IXP2800 provides for 64 read and 64 write SRAM references per packet. At 28 million packets per second, the IXP2800 provides for 32 read and 32 write SRAM references per packet. The references may be used for network address lookup, multi-tuple classification, policing, packet or buffer descriptors, queuing, statistics, and scheduling. The IXP2800 supports an aggregate rate in excess of 60 million queue operations per second on one or multiple queues. The number of queues that an IXP2800 supports is limited only by the available SRAM capacity, not by any on-chip resource limit. Tables 1 and 2 depict a possible allocation of SRAM bandwidth.

Table 1: Ingress possible allocation of SRAM references

Function	QDR Reads	QDR Writes
Destination address TRIE route lookup	7	
7-tuple TCAM rule lookup	1	5
Buffer descriptor	2	2
Queue and freelist linked-list operations	6	6
Metering	3	3
Congestion avoidance (WRED)	5	4
Per-rule statistics	2	2
Per (min-size) packet totals	26	22

Table 2: Egress possible allocation of SRAM references

Function	QDR Reads	QDR Writes
Reassembly context	2	2
7-tupple TCAM rule lookup	1	5
Buffer descriptor	2	2
Queue and freelist linked-list operations	6	6
Congestion avoidance (WRED)	5	4
Per-rule statistics	2	2
Per (min-size) packet totals	18	21

Varying product requirements will increase or decrease the allocation of SRAM references per packet. For instance, incorporating ATM segmentation and reassembly into the processing flow will add a couple of read and write references on ingress and egress. A balanced system design will try to balance the consumption of resources across ingress and egress processors.

This configuration represents a cost-effective and extremely flexible approach to basic packet processing at 10Gb/s rates.

10GB/S MULTI-SERVICE SWITCH BLADE

In this system architecture, three IXP2800 network processors are used (Figure 2). The ingress processing is distributed across two IXP2800 network processors.

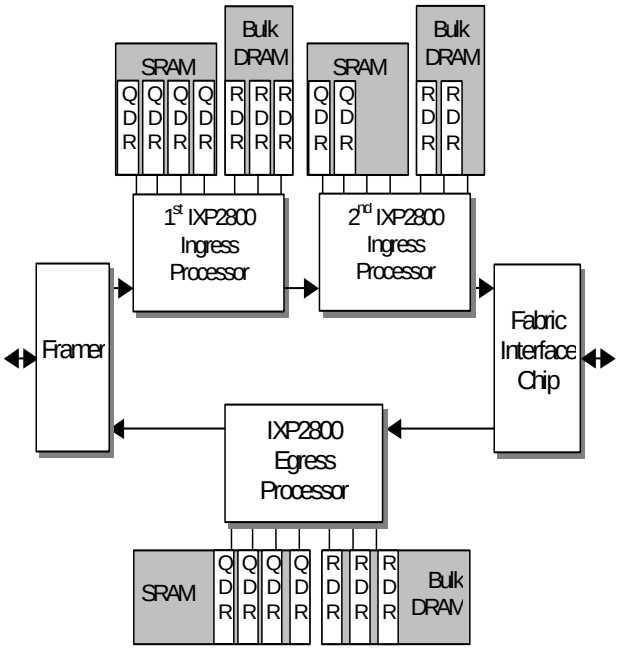


Figure 2: 10Gb/s multi-service configuration

The egress processing is accomplished with a single IXP2800, as in the prior configuration.

The distinguishing characteristic about this configuration is the division of labor between the 1st and 2nd ingress IXP2800 processors. The configuration is intended to support broadly varying rates of packet processing while maintaining expected aggregate throughput rates.

The 1st ingress IXP2800 is responsible for transferring received packet pieces into contiguous ring buffers in DRAM, as they are received, with minimal processing. Multiple rings may be supported, with the destination ring identified by minimal packet classification. These rings provide for an elasticity buffer to allow for varying rates of packet processing performed subsequent to the storage of the packets in DRAM. The size of the rings may vary, based upon the expected arrival rate of the packets and the elasticity requirements. Maintenance of the rings requires minimal SRAM accesses but does require static allocation of memory per ring. The smallest configuration of DRAM that supports the maximum bandwidth (3 DRAM components) supports 96 mega-bytes of storage.

Smaller rings in SRAM shadow the rings in DRAM. The entries in the SRAM rings provide status regarding the processing of the packet and information to pass on to the 2nd ingress IXP2800 for final processing. Upon completion of the processing of a packet in the 1st ingress IXP2800, the status is updated. Earlier packets may complete processing subsequently, but the ring insures in-order forwarding to the 2nd ingress IXP2800, as the packets are fetched from the ring in-order and only after they complete processing.

As packet-processing threads become available, a scheduling decision is made in software regarding which rings should be serviced. Threads read sufficient parts of packets in DRAM to process the packets. Threads may take an arbitrary time to complete processing of the packet, subject to the elasticity provided by the DRAM buffering and the average packet arrival rate. Separately, as Transmit Buffer (TBUF) elements become available, the status of the rings is polled to forward packets to the 2nd ingress IXP2800.

The 1st ingress IXP2800 is best suited to performing multi-level, multi-protocol packet classification and editing. By design, most of the SRAM bandwidth is available for classification. Update of flow-specific or queue-specific state, including statistics, is deferred until the 2nd ingress IXP2800. A digest is forwarded with the packet that describes such state as needs updating. (The bandwidth available through the SPI-4.2 physical interface approaches 20Gb/s, accommodating the increased payload per packet.)

The 2nd ingress IXP2800 receives packets with no packet interleaving or limited packet interleaving, reducing the accesses to SRAM to reassemble the packets. All classification processing has been completed. The 2nd ingress IXP2800 is responsible for any remaining metering and policing, statistics, queuing and buffering, congestion avoidance, and transmit scheduling into the fabric.

By design, the division of labor between the 1st and 2nd ingress processor distributes the use of SRAM bandwidth across the two processors. The 1st ingress IXP2800 supports nearly arbitrary processing times, while maintaining the order of packets within categories (rings). The 2nd ingress IXP2800 updates shared state in-order. The egress IXP2800 operates exactly as described in the prior configuration that also uses a single egress IXP2800.

OC-48 (4 X OC-12 OR 16 X OC-3) SWITCHING BLADE

In this system architecture, a single IXP2800 network processor is used for both ingress and egress processing (Figure 3). External silicon components multiplex the data from the framer and fabric into the SPI-4.2/CSIX receiver and distribute the transmit data from the SPI-4.2/CSIX transmitter to the framer and fabric.

The IXP2800 supports the capability to simultaneously multiplex the SPI-4.2 and the CSIX-L1 protocols on the same interface. The switch chip allows interfacing to both an OC-48 framer (probably using SPI-3 or UTOPIA Level 3) and a fabric supporting a CSIX-L1 interface.

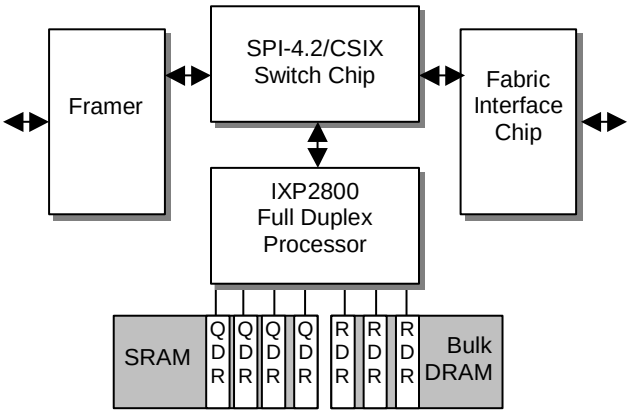


Figure 3: OC-48 (4 x OC-12 or 16 x OC-3) configuration

In this configuration, the IXP2800 is offered half of the aggregate load supported by the prior configurations. There is sufficient DRAM bandwidth to write packets to DRAM on receptions and read them back for processing as in the prior multi-service switching configuration, although the packets are stored using the linked-list organization of buffers. Rings of buffer descriptors are used to enforce in-order enqueuing of the packets to linked-list queues, just as in the prior configuration. The different code paths for ingress and egress processing may be handled on the same microengines or distributed across different microengines in order to optimize the utilization of the microcode stores. Finally, different microengines are allocated to updating shared state in-order and coherently.

THE IXP2800 MICROARCHITECTURE

The IXP2800 has 10 major internal units (Figure 4). The IXP2400 also has 10 major units; however, a few of the units have variations. The IXP2800 units and the variations for the IXP2400 are described below.

The Media-Switch-Fabric Interface

The Media and Switch Fabric (MSF) Interface is used to connect an IXP to a physical layer device (PHY) and/or a

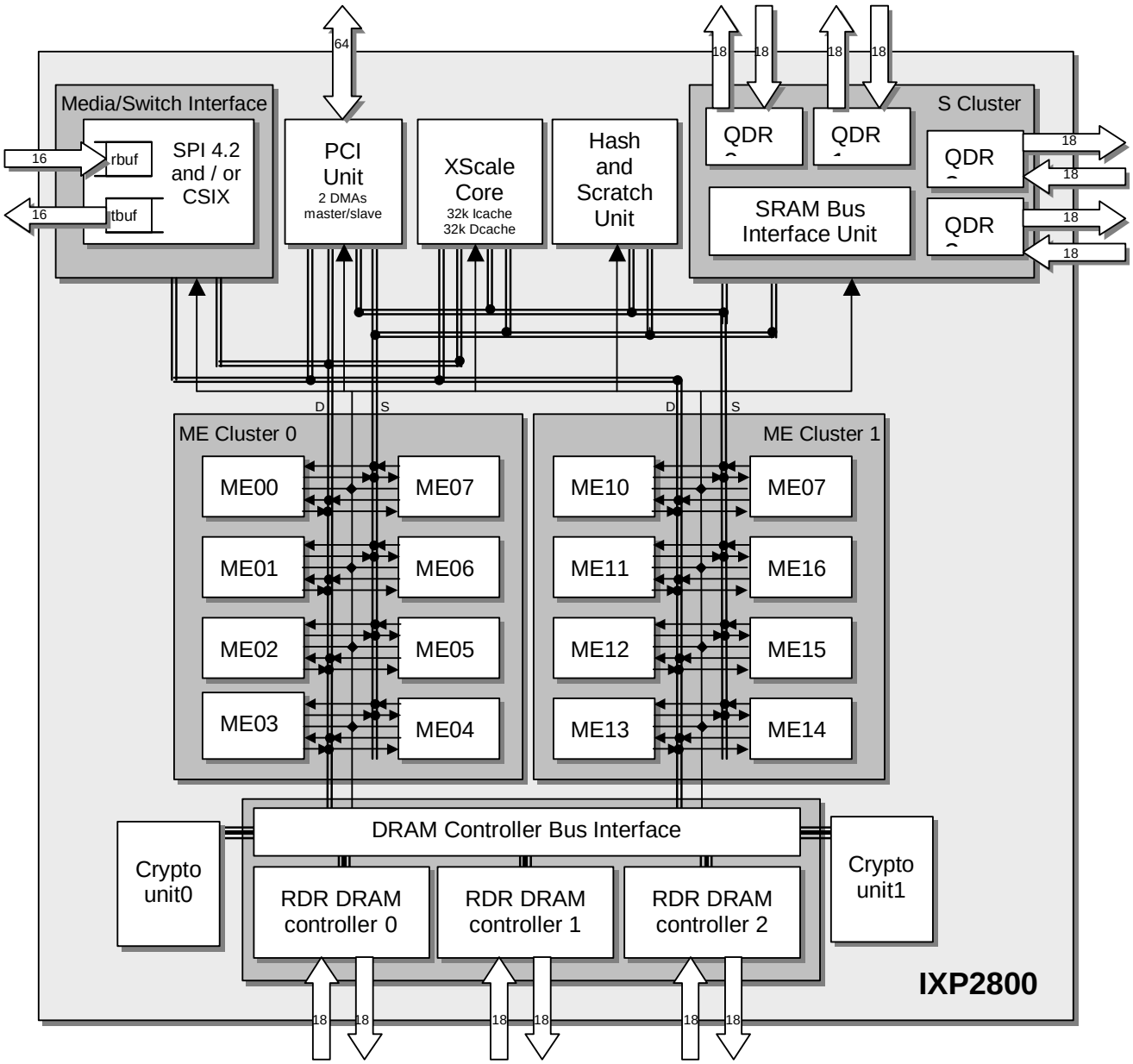


Figure 4: IXP2800 block diagram

switch fabric. The MSF consists of separate receive and transmit interfaces. Each of the receive and transmit interfaces can be separately configured on the IXP2800 for either SPI-4 Phase 2 (System Packet Interface) for PHY devices or CSIX-L1 (Common Switch Interface Specification, Layer 1) protocol for switch fabric interfaces. Additionally, configuration provides for multiplexing both protocols over the interface simultaneously. The IXP2400 is similar; however, instead of SPI-4 phase 2 signaling and protocol, the IXP2400 supports POS PHY Level 3 (dual 32-bit uni-directional 125MHz bus) and CSIX-L1 protocol.

The receive and transmit ports are unidirectional and independent of each other. Each IXP2800 port has 16 data signals, a clock, a control signal, and a parity signal, all of which use Low Voltage Differential Signaling (LVDS) and are sampled on both edges of clock. There is also a flow control port consisting of a clock, data, parity, and ready status bits, and it is used to communicate

between two IXP2800 chips, or an IXP2800 and a switch fabric interface. All the high-speed LVDS interfaces support dynamic deskew training. The IXP2800 supports 10Gb/s inbound traffic and 15Gb/s outbound or 15Gb/s inbound and 10Gb/s outbound. The overspeed (15 vs. 10Gb/s) is required by fabrics, which have inherent inefficiencies. The average bandwidth required by a fabric may be 10Gb/s; however, for extended moments they may burst 15Gb/s. The IXP can source or sink these extended burst rates.

Incoming packets are received into the Receive Buffer (RBUF). Outgoing packets are held in the Transmit Buffer (TBUF). The RBUF and TBUF are both RAMs and store data in sub-blocks (referred to as *elements*), and are accessed by either the microengines or XScale™.

The RBUF and TBUF each contain 8KB of data. The element size is programmable as either 64 bytes, 128 bytes, or 256 bytes per element. In addition, either buffer can be programmed to be split into one, two, or three partitions, depending on application. For SPI-4, one partition is used. For CSIX, two partitions are used (control and data c-frames). For both SPI-4 and CSIX, three partitions are used.

The microengine can read data from the RBUF to the microengine in_bound registers using the MSF[read] instruction. The microengine can promote data from RBUF to DRAM directly using the DRAM[rbuf_rd] instruction.

The microengine can promote data into the TBUF along with status via writes from the outbound_transfer registers using the MSF[write] instruction. The microengine can control movement of data from DRAM directly to the TBUF using the DRAM[tbuf_wr] instruction.

The IXP Chassis

The chassis is the bus system, which interconnects all the units within the IXP. The chassis employs uni-directional buses to implement a microengine-based distributed memory storage mechanism. The microengine has inbound and outbound transfer registers. The chassis is used to retrieve data from the outbound transfer registers and deliver data to the inbound registers. The chassis consists of data busses, which connect the microengine transfer registers to the various shared resources (i.e., SRAM, DRAM, hash, cryptography units). Additionally, the chassis has multiple instantiations of a command bus. This command bus runs ahead of the data buses. It notifies the shared resources that a microengine is requiring service and indicates the source and destination addresses, the function to be performed, and any other information required to complete the requested task. Additionally, the command bus has a field indicating the data length of the requested transfer.

The chassis operates at half the frequency of the microengine. This is up to 700MHz for the IXP2800 and up to 300MHz for the IXP2400.

THE MICROENGINE CLUSTERS

The IXP2800 has 16 microengines, configured as two clusters of eight identical microengines. The reason for this partitioning is to provide more communication capability between the microengine and the rest of the chip resources. Each cluster has its own copy of command and data busses. Thus each microengine shares the command bus with seven other microengines, rather than with 15 other microengines, as would be the case without the two-cluster configuration. More details about the capabilities and internal configuration of the microengine are presented later in this paper.

The SRAM cluster

The SRAM cluster consists of four independent SRAM controllers, each of which controls external Quad-Data-Rate (QDR) SRAMs. The reason for four channels is to provide sufficient control information bandwidth for 10Gb network applications. SRAMs are a good choice for control information, which tends to have many small data structures such as queue descriptors and linked lists. SRAMs, unlike DRAMs, allow for small access size and additionally allow access to any address sequence with no restrictions. Each SRAM controller, running at 200MHz, provides 800MB/s of read bandwidth and 800MB/s of write bandwidth.

In addition to the normal read and write access, the IXP2800 SRAM controllers provide three additional hardware functions.

1. *Atomic read-modify-write operations: increment, decrement, add, subtract, bit-set, bit-clear, and swap*. The atomic operations are useful for implementing software semaphores. They can also be used for multiple processes that modify a shared variable without using conventional mutex to obtain ownership, for example, update a network statistic via an atomic add operation. This is more efficient, since it eliminates the mutex operation altogether in this case.

2. *Linked-list queue operations*. This hardware accelerates enqueue and dequeue to linked-list operations by eliminating the read-to-write or read-to-read latency. For example, to do an enqueue, software must read the current list tail and then use it as an address to write the new link to memory. The SRAM controller keeps the tail address in on-chip registers and does the enqueue write locally; this saves the time that would have been spent by

the microengine to get the tail value and then simply use it as the address for the write.

3. *Ring operations*. A ring is also sometimes called a *circular buffer*. It consists of a block of SRAM addresses, which are referenced through a head and tail pointer. Data is inserted at the tail of the ring (using the content of the tail pointer as the address) and removed from the head (using the content of the head pointer as the address). The SRAM controller keeps the head and tail pointers in on-chip registers and increments them as they are used. The advantage is that multiple processors can add data to and remove data from the rings without having to use a mutex to obtain ownership.

It is also possible to attach an external coprocessor, such as Ternary Content Addressable Memory (TCAM), or classification processors to the SRAM interface. The interface conforms to the Network Processor Forum's LA-1 (Look-Aside) interface specification.

The DRAM Cluster

The DRAM cluster provides three independent DRAM controllers, each of which controls external Rambus DRAMs (RDRAMs). The reason for three channels is to provide sufficient data buffering bandwidth for 10Gb network applications. DRAMs are a good choice for a data buffer because they offer excellent burst bandwidth and are much denser and cheaper per bit relative to SRAM. Each DRAM controller, running at 133MHz (note that this equates to 533MHz DDR, which is 1066 M transfers/sec on the data pins), provides 17Gb/s of bandwidth, shared between reads and writes.

The three DRAM controllers provide hardware interleaving of the DRAM address space (often referred to as *striping*). This is done to spread accesses evenly to prevent "hot spots" in the memory. If all accesses for a period of time were to address only one of the controllers, then only one-third of the bandwidth would be available. The way the interleaving works is that each controller simultaneously receives all access requests and compares the address to the range of addresses that fall within its range. It then claims either all, part, or none of the access request according to the result of the address compare. The entire process is done in hardware, completely transparent to the software.

The Cryptography Unit

The cryptography unit performs authentication and bulk encryption. It is believed that these two datapath tasks are critical strategic functions for the network processor. The crypto engines are innovative designs that have a very small footprint, yet the two engines provide 10Gb/s throughput performance. This unit is covered in detail in a subsequent article in this journal.

The Hash Unit

The hash unit can perform either 48-bit, 64-bit, or 128-bit polynomial division. The hash function implemented is an irreducible polynomial, which has the characteristic of a one-to-one mapping. This means that if there is a collision, checking the unused bits of the remainder against that entry's saved and unused remainder bits confirms or denies the collision. The multiplier to the hash function is programmable so that if a default multiplier is not performing efficiently, a new one may be calculated.

The motivation for the hash unit hardware is that performing a high-quality hash in software is cycle consuming. Layer 2 lookups for Ethernet employ a hash on the 48-bit source and destination addresses for bridging. The hash hardware acceleration is excellent for this lookup. Ipv6 employs 128-bit source and destination addresses, and the hash unit may be used for data reduction.

The basic idea behind the hash unit is to take correlated data and uniformly distribute it across a small set space. For example, the hash unit may be used to take the 48-bit Ethernet destination address and map it into a much smaller 16-bit addressed destination table. A good hash function will uniformly distribute entries in the smaller table to reduce the probability of a collision.

The Scratch Unit

The scratch unit contains an on-chip 16KB scratchpad memory, running at 700MHz. To a programmer, the scratchpad memory provides very similar capability to the SRAM described earlier. The main difference is that the capacity of the scratchpad is much smaller than the external SRAMs. However, the scratchpad has lower latency (running at 700MHz instead of 200MHz as the external SRAMs). The scratchpad provides the atomic read-modify-write and ring operations as described in the SRAM section.

The XScale™ Processor

The XScale processor is compliant with the ARM Version 5TE (Advanced Risc Machines), and runs at 700MHz. Normally, it is used as a system control plane processor, handling exception packets and doing management tasks. It contains independent 32KB instruction and data caches, and a full capability memory management unit. The XScale has uniform access to all system resources, so it can efficiently communicate with the microengine though data structures in shared memory.

The PCI Unit

The PCI Unit provides an interface to industry standard 64-bit 66MHz PCI Rev 2.2. It is typically used as a control plane interface, either to an external microprocessor, for example, a Pentium®, or as an external device interface, such as a public key accelerator. The PCI unit can act as a PCI bus master, allowing XScale or microengine access to external PCI targets, or as a PCI bus target, allowing external devices to transfer data to and from the IXP2800 external SRAM and DRAM memory spaces. The PCI Unit also contains DMA channels that can be programmed to do bulk data transfers between DRAM and external PCI targets.

THE IXP2XXX MICROENGINE

Several goals guided the specification of the ME:

- Efficient silicon implementation. The need for lots of compute capability in the network processor dictated the need for a large number of MEs.
- High frequency to allow for sufficient instructions per packet. The ME has a six-stage pipeline and runs at 1.4GHz in P861 (.13

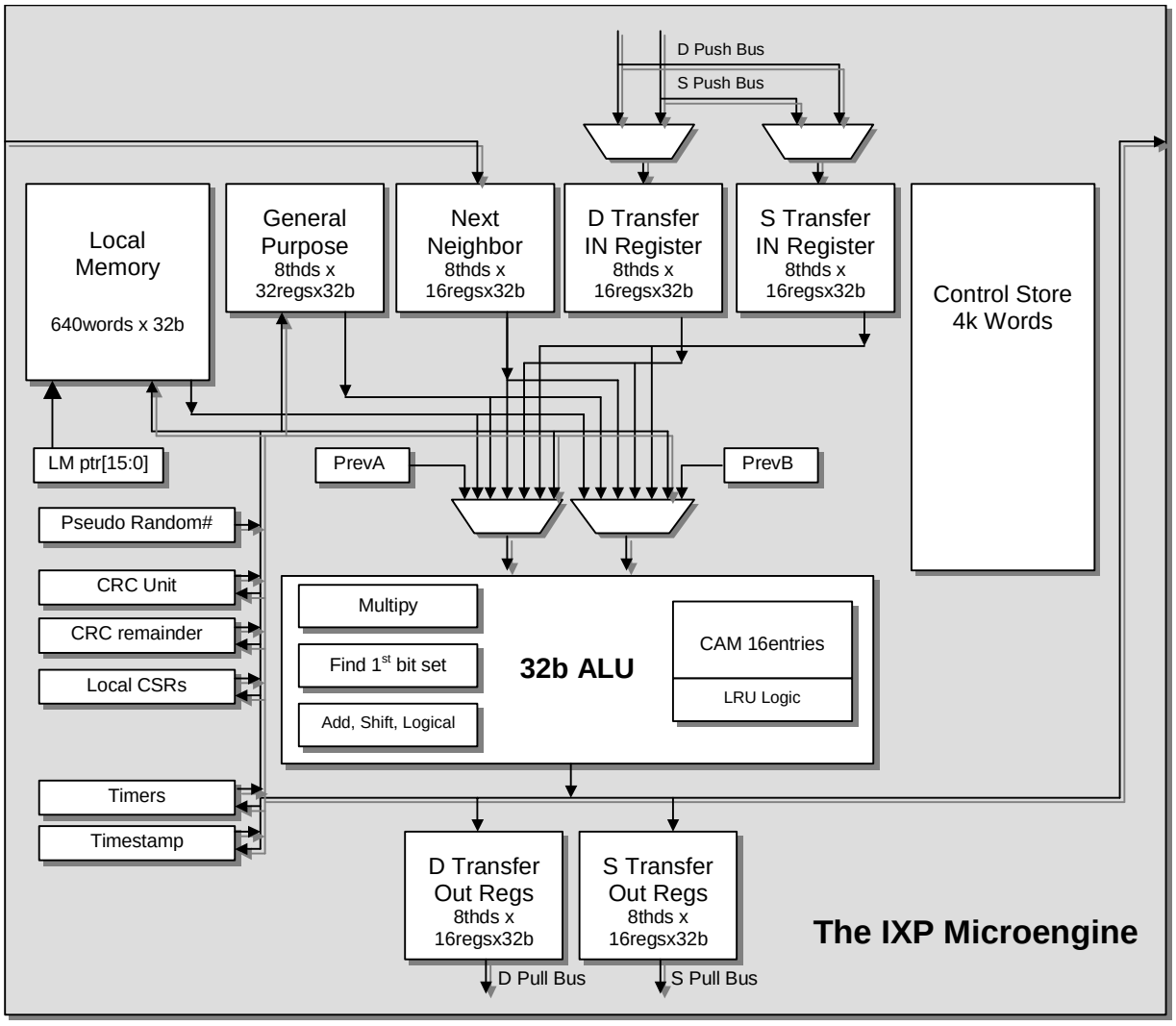


Figure 5: IXP microengine block diagram

micron).

- Large register set. Having many registers minimizes the need to shuffle program variables back and forth between registers and memory. Having to shuffle uses valuable cycles without accomplishing useful work.
- Low-latency local memory in the ME. This is addressable memory, in addition to the registers. It can be used in any way the application chooses, for example, to hold packet data or state related to ports, etc.
- Efficient intra-ME communication capability. This is useful in the applications described earlier in this article.
- Multiple threads. Given the disparity in processor cycle times vs. external memory times, a single thread of execution often blocks waiting for external memory operations to complete. Having multiple threads available allows for threads to interleave operation—there is often at least one thread ready to run while others are blocked. This makes more productive use of the other ME resources, which would otherwise be idle.

There are eight hardware threads available in the ME. To allow for efficient thread swapping, each thread has its own register set, program counter, and thread-specific local registers. Having a copy per thread eliminates the need to move thread-specific information to/from shared memory and ME registers for each swap. Fast thread swapping allows a thread to do computation while other threads wait for IO (typically, external memory accesses) to complete, or for a signal from another thread or hardware unit. (Note that a swap is similar to a taken branch in timing.)

Each of the eight threads will always be in one of four states.

- Inactive—Some applications may not require all eight threads. Unused threads can be kept in an inactive state by setting the appropriate value in a configuration register.
- Executing—The executing thread is the one in control of the ME. Its PC is used to fetch the instructions that are executed. A thread will stay in this state until it executes an instruction that causes it to go to sleep state (there is no hardware interrupt or pre-emption; thread swapping is completely under software control). At most, one thread can be in executing state at any time.
- Ready—In this state, a thread is ready to execute but is not because a different thread is executing. When the executing thread goes to sleep state, the MEs thread arbiter selects the next thread to go to the executing state from among all the threads in the ready state. The arbitration is round robin.
- Sleep—In this state, the thread is waiting for some external event(s) to occur (typically, but not limited to, an IO access). In this state the thread does not arbitrate to enter the executing state.

At most, one thread can be in executing state at a time; any number of threads can be in any of the other states.

Registers

As shown in the block diagram in Figure 5, each ME contains four types of 32-bit datapath registers:

1. 256 general-purpose registers
2. 512 transfer registers
3. 128 next neighbor registers
4. 640 32-bit words of local memory

Each of the first three types is partitioned per thread. The local memory is shared among all threads.

GPRs are used for general programming purposes. They are read and written exclusively under program control. GPRs, when used as a source in an instruction, supply operands to the execution datapath. When used as a destination in an instruction, they are written with the result of the execution datapath.

Transfer registers are used for transferring data to and from the ME and locations external to the ME (for example, DRAMs, SRAMs, etc).

Next Neighbor (NN) registers are used as an efficient method to pass data from one ME to the next, for example, when implementing a data-processing pipeline. The NN registers can supply instruction source operands; when NN register is the destination of an instruction, that value is written in the next ME.

The NN registers can also be configured to act as a circular ring instead of addressable registers. In this mode the source operands are “popped” from the head of the ring, and destination results are “pushed” to the tail of the ring. The head and tail pointers are maintained in hardware in the ME.

For applications that don’t need to use the NN registers for intra-ME communications, the ME can be put into a mode where an instruction with NN as destination will write the NN register in the same ME. This increases the

number of registers available to an application. The choice of this mode is independent of the use of ring mode; all combinations are supported.

Local Memory (LM) is addressable storage located in the ME. LM is read and written exclusively under program control (i.e., it is private to the ME). The distinction between LM and the registers described above is that the LM address is computed by the program at run-time, whereas the register addresses are determined at compile time and bound in the instruction. Each thread has two LM address registers, which are written by special instructions. The specific LM location selected is based on the value in one of the LM address registers, which is specified in the instruction.

All of the registers described above, including LM, are built using two-ported register files: one read port and one write port. The area efficiency of two-ported registers relative to multiport registers is important in allowing the large number of registers to fit in the allocated silicon area. Of course, the use of two-port registers places some restrictions on which combinations of registers can source operands for each instruction. The restrictions are managed by the register allocator in the compiler and assembler, and in practice there are no limitations found in normal programs.

Instructions

The instruction set of the ME is similar to that of many RISC microprocessors, with some additional features tailored to the network processor task.

- Computation instructions can take one or two operands, perform an operation, and optionally write back a result. The sources and destinations can be GPRs, transfer registers, next neighbor registers, and local memory. The operations are shifts, add/subtract, logical, multiply, byte align, and find first one bit. There is also a Content-Addressable-Memory (CAM), described below.
- Logical operations can be performed along with shifting one of the operands in a single instruction. This can often be used to collapse two operations into one, for example, in masking fields of a header.
- IO instructions are used to read and write various memory units in the NPU, such as receive buffer, transmit buffer, DRAM, and SRAM. There are also a number of higher-level operations available in the IO units, such as ring operations, atomic read-modify-write, and linked-list queue operations.
- Special instructions are provided for inserting bytes into registers. These are useful for packet header modification.
- Branches can be done, based on comparing a byte within a register to a literal value. This can be used to efficiently test for values in a header. Branches can also be done on individual bits set or clear within a register. This is useful for efficiently testing status flags. The above are in addition to the normal suite of branches on numerical results, such as greater than, less than, etc.
- Instructions can be placed into branch defer slots to minimize the number of cycles lost due to taken branches redirecting the ME pipeline. The compiler is able to move instructions that are executed, regardless of branch outcome into those slots.
- Hardware support is provided for integer multiply. Each instruction cycle can retire 8 bits of operand. Taking this approach vs. providing a full, autonomous multiply was a trade off of performance vs. silicon area. One advantage of this approach is that for small numbers, for example, 8 bits or 16 bits, the compiler can insert just enough cycles to complete the multiply.
- Hardware support is also provided for CRC operations for several industry standard polynomial values. The hardware can do a CRC over 32 bits every other cycle. This is equivalent to 22.4Gb/s at a ME frequency of 1.4GHz.

CAM

The CAM is a unique function that has a number of uses. The CAM has 16 entries; and each entry stores a 32-bit value. This allows a source operand to be compared against 16 values in a single instruction. All entries are compared in parallel, and the result of the lookup is written into the destination register. There are two outcomes (the lookup result is indicated by the value in a destination register bit, which a branch instruction can test in one cycle):

- A *miss* indicates that the lookup value was not found in the CAM. The result also contains the entry number of the least recently used entry (which can be used as a suggested entry to replace).
- A *hit* indicates that the lookup value was found in the CAM. The result also contains the entry number that holds the lookup value. In addition,

the result holds an additional 4 bits of state that the program can define and use.

The CAM can be used to accelerate multi-way compares. It can also be used to act as the tag store of a cache; in this case, the entry number of a matching value can be used as an index to data associated with the value (and stored, for example, in SRAM or LM). Because the CAM does not store any of the associated data, the hardware places no limitation on the amount of data stored for each cached entry. It could be as little as a few bits or as much as needed, limited only by SRAM memory capacity. The state bits can be used to store additional information about a cache entry, for example, if it has been modified or how many threads are making use of it.

Event Signals

The ME supports the concept of event signals. These are signals that a thread can use to indicate the occurrence of some event external to the ME; the thread can block (go to sleep state) waiting on the event. Typical use of events includes completion of IO and signals from other threads, for example, to indicate that some data has arrived and is ready for processing. Each thread has 15 event signals. These can be allocated and scheduled by the compiler in much the same way as registers are allocated. They allow for a large number of outstanding events and, therefore, concurrent processing of non-dependent tasks. For example, the thread could start an IO to read packet data from the receive buffer, start another IO to allocate a buffer from a freelist, and start a third IO to read the next task from a work list (on a ring). All of the IOs execute in parallel. Many microprocessors can also schedule multiple outstanding IOs; normally, that is handled in a hardware-based scoreboard. By using event signals, the ME places much of the burden on the compiler, which simplifies the hardware.

Other microengine features useful to the network processor task are the following:

- *Timestamp*–a 64-bit timestamp register that can be used for real-time tasks. The timestamp is guaranteed to be monotonically increasing for the lifetime of an application; it will not wrap around.
- *Pseudo-random number*–used for some algorithms that need random numbers. Note that this is pseudo-random and not suitable for security applications.

CHALLENGES AT 10GB/S

For high-speed networking systems an extremely efficient means for handling successive enqueue and dequeue requests to the same linked list queue structure is required to support a large number of queues (linked lists for memory efficiency) at line rate (packet/cell arrivals at ~40ns). Consecutive enqueue operations to the same linked list queue are latency constrained since the first enqueue must create the link to a list tail pointer before a subsequent entry can be linked on to that new tail. Likewise, for consecutive dequeue operations, the head pointer of the queue must be read to determine the new head pointer for the list before a subsequent dequeue operation is done. A control structure that can manage requests to a large number of queues as well as successive requests to only a few queues or to a single queue, plus a memory controller data path capable of back-to-back enqueue or dequeue to the same queue at the packet or cell arrival rate are required.

A single microengine designated the queue manager receives enqueue requests from the set of microengines that are programmed to perform receive processing and classification. The enqueue request specifies to which output queue an arriving packet or cell should be added. A microengine that functions as the transmit scheduler sends dequeue requests to the queue manager microengine that specifies the output queue from which a packet or cell is to be taken and then transmitted to an output interface (see Figure 6).

Each microengine contains a 16-entry Content Addressable Memory (CAM) that tracks which entry is the Least Recently Used (LRU). The queue manager microengine uses the CAM to implement a software-controlled cache containing the last 16 queue descriptors used to enqueue and/or dequeue packets or cells. While the CAM serves as the "tag store" holding the addresses of the queue descriptors that are being cached, the "data store" associated with each CAM entry is implemented in the SRAM controller logic. The data store for each queue descriptor contains the head pointer (address of the first entry of a queue), the tail pointer (address of the last entry of a queue), and a count entry (present "length" of the queue). Locating the data store for the cache of queue descriptors at the memory controller allows for low-latency access to and from the queue descriptor data cache and memory.

The queue manager microengine issues commands to return queue descriptors to memory and fetch new queue descriptors from memory such that the queue descriptor data store located at the memory controller remains coherent with the CAM tag store of queue descriptor addresses. The queue manager issues enqueue and dequeue commands indicating which of the 16 queue descriptor data store locations to use for the memory controller to perform the command.

All enqueue and dequeue commands are initiated in the order in which they arrived at the memory controller, and

these reference 1 of 16 data store tail or head pointers. An enqueue writes the address of the pointer to be added to the queue to the address of the cached tail pointer, and then updates the cached tail pointer to the address just added. Since enqueue requires only a write, the data store is updated in 2 cycles, and a subsequent enqueue even to the same queue can then be initiated. For dequeue, the address of the head pointer in the data store is returned to the queue manager microengine (this is the address locator for the buffer or cell to be transmitted), and a read of the contents of the head pointer is initiated. When the read data returns, it is loaded into the head pointer for specified data store entry. A subsequent dequeue request to a different queue can be initiated on the next cycle. However, a dequeue request to a queue where a read of the head pointer location is in progress must be held up until the data store location for that entry's head pointer is updated. An enqueue to a queue with a dequeue in progress can proceed since the tail pointer is not affected by the dequeue.

Having the control structure for queueing in a microengine allows for flexible high performance while using the existing hardware of the microengine. Distributing the data store part of the cache of queue descriptors allows for the low-latency memory operations required for successive enqueue and dequeue operations at high line rates.

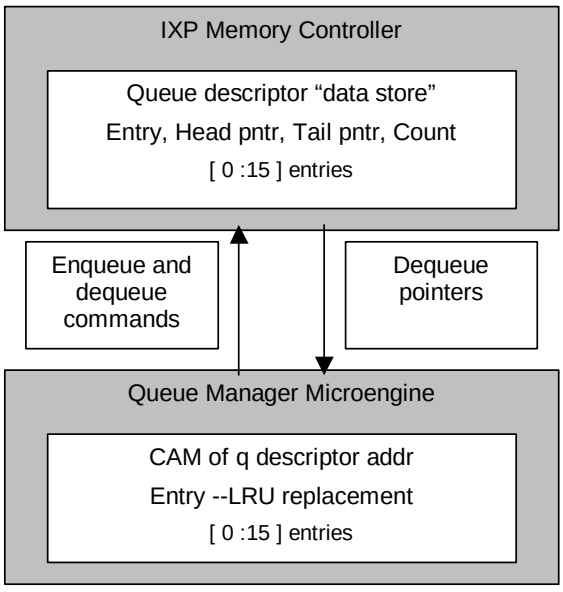


Figure 6: Enqueue dequeue memory controller

DISCUSSION

There is a trade off between programmable flexibility and software complexity. Flexibility provides great product advantages for feature enhancements and future upgrade capabilities. However, it also makes evaluation for performance against customer requirements and subsequent customer product development more challenging. The IXP family is addressing these challenges with tools and leadership silicon performance.

The IXP workbench is a state-of-the-art integrated development environment. Users write their code, compile (C language) or assemble (IXP macro language) their code with advanced error reporting, then debug the code on a very high-performance-cycle accurate simulator (>500 cycles per second simulation performance). This simulation environment provides advanced visualization tools and debugging facilities for rapid code maturation. The workbench environment can then be used to exercise the IXP silicon with the developed code. Advances to the workbench include rapid prototyping and static performance evaluation, given simple user heuristics.

Providing leadership silicon performance requires less software tuning to achieve given product goals. The IXP2800 with 16 parallel processors at 1.4GHz delivers on the promise of network processors. This promise includes providing a multi-application hardware-based platform for communication companies to leverage across multiple market segments. Additionally, it promises network processor customers differentiation by software. Within a given company, the promise of common software routines or functions to be leveraged by different product groups is also now possible for IXP customers.

CONCLUSIONS

The IXP family provides a very powerful, flexible hardware platform for a wide range of software-based network processing applications. The range of applications is widening and is identifying the opportunity for certain IXP variations tuned to specific applications. Recognizing this possibility, the design and implementation methods for the IXP family have been optimized for rapid future variations.

This is enabling a roadmap vision that is two-pronged. One prong is providing greater performance through the use of additional hardware multi-threading and additional microengines, while also including new strategic hardware acceleration engines such as the IXP2800 did with advanced dataplane cryptography acceleration.

The second prong is leveraging the Intel Communication Group's silicon portfolio for greater system integration. Integration is important when it can reduce system power, cost, and board area. This prong can provide current IXP customers with a product cost-reduction path.

Both of these prongs will leverage the silicon capabilities afforded by 90nm and, subsequently, 65nm high-performance CMOS.

Performance, integration, advanced tools, rapid software prototyping, advanced strategic hardware acceleration, extreme customer support: this is the roadmap vision for the IXP family.

ACKNOWLEDGMENTS

The authors acknowledge the contributions of Sanjeev Jain, David Romano, John Cyr, Jim Guilford, Bob Kushlis, Jose Niell, Milo Sprague, Kin-Yip Liu, Yim Pun, John Wishneusky, Donald Hooper, Bill Wheeler and the VMOD development team, John Sweeney, the IXP verification teams, and the IXP implementation teams led by John Beck (IXP2800) and Ahmad Zaidi (IXP2400).

REFERENCES

[1] Matthew Adiletta, et. al, "Packet over SONET: An Overview of the Packet Processing Flow of a 10 Gigabit/sec Datastream Mapped to an IXP2800," *Intel Technology Journal*, Vol. 6 Issue 3, August 2002.

[2] Internet Network Working Group, RFC 2697, September 1999.

[3] S. Floyd and V. Jacobson, "Random Early Detection Gateways for Congestion Avoidance," *IEEE/ACM* Transactions on Networking, V.1 N.4, August 1993, pp. 397-413.

[4] Internet Network Working Group, RFC 2863, April 1998.

[5] E. Johnson and A. Kunze, *IXP1200 Programming*, Intel Press, ISBN 0-9702846-7-5, 2002.

AUTHORS' BIOGRAPHIES

Matthew Adiletta is an Intel Fellow and Director of Communication Processor Architecture. He led the architectural development and implementation of the IXP2800 and is driving the IXP roadmap. He is interested in processor architecture and advanced implementation techniques for rapid silicon development. He is also intrigued with network security and classification. Adiletta has been responsible for 12 previous silicon chips, including silicon for VAXes, alphas, video, graphics, and communication. The IX2800 is the lucky 13th. Adiletta received his B.S. degree in electrical engineering, with Honors, at the University of Connecticut. He resides in Bolton, Massachusetts. His e-mail address is matthew.Adiletta@intel.com.

Debra Bernstein is an architect for Intel's Network Processor Division. She worked on the architecture of the IXP2000 series and the IXP1200. For the 2000 series, Deb has been particularly focused on the queuing problem. Previously, she worked on microprocessors in the VAX and Alpha family at Digital Equipment Corporation. She is a 1982 graduate from the University of Massachusetts at Amherst. Her e-mail address is debra.bernstein@intel.com.

Mark Rosenbluth is an architect in the Network Processor Division. He has been at Intel for four years and prior to that worked at Digital Equipment Corporation, where he was architect for PCI Bridges and also worked on VAX and Alpha microprocessors. He received a B.S.E.E. degree from Rutgers University. He resides in Uxbridge, Massachusetts, and can be reached via e-mail at mark.rosenbluth@intel.com.

Hugh Wilkinson is a systems architect in the Network Processor Division at Intel. Hugh's technical interests include switching fabrics, protocol design, software decomposition, and high-speed signaling. He received his B.S. degree in Computer Science from Boston University. He works in Hudson, Massachusetts, and can be reached at Hugh.Wilkinson@intel.com.

Gilbert Wolrich is a senior architect in the Network Processor Group in Hudson. He has contributed to the definition of both the IXP1200 and IXP2000 solutions. Gil has worked on high-performance network and general-purpose processors, numerous floating point units, and is interested in network security. Gil received a B.S. degree from R.P.I. and an M.S. degree from Northeastern University in Electrical Engineering. He resides in Framingham, Massachusetts, and can be reached via e-mail at gilbert.wolrich@intel.com.

For further information visit:

developer.intel.com/technology/itj/index.htm