

CHAPTER 2

THE IXP12XX PROGRAMMING ENVIRONMENT

The goal of this chapter is to understand what the various components of the IXP12xx hardware and IXA SDK 2.0 software tools do for your software and how they impact the overall programming environment.

Programming the IXP12xx differs considerably from programming on general-purpose processors. It may be that when you first opened your IXP12xx system you were looking for a list of the ported operating systems and a working ANSI C/C++ compiler. What you discovered was that the microengines don't run any operating systems, and the C compiler, with its new `__declspec(...)` modifiers and slew of intrinsics, looks only vaguely familiar.

In traditional programming environments, operating systems, and compilers the details of thread scheduling, register counts, register sizes, and machine instruction sets can be ignored. When programming the IXP12xx microengines, ignoring these details makes it difficult to produce optimized code and let the hardware do what it was designed to do.

Understanding the IXP12xx programming environment goes beyond just learning yet another set of software programming interfaces.

Understanding the IXP12xx programming environment means, perhaps for the first time, understanding the processor architecture. But consider the upside: by taking advantage of the IXP12xx hardware, you can offload work to it. And one less thing to write and debug is always a win.

This chapter describes those components of the processor's architecture relevant to programmers. Typically such descriptions provide good reference material, but can be difficult to remember. So, in order to put the pieces of the processor architecture into perspective, the description is followed by a practical, and hopefully memorable, explanation of a day in the life of a packet in the IXP12xx hardware. We will finish the chapter

with a description about the software programming environment for the IXP12xx.

PROGRAMMABLE PROCESSING UNITS

As shown in Figure 2-1, the IXP12xx consists of seven programmable processors: one StrongARM* core{ XE "StrongARM core" } and six microengines all on the same die. The StrongARM core is an Advanced Reduced Instruction Set Computer (RISC){ XE "reduced instruction set computer (RISC)" } Machines (ARM) general-purpose processor. The availability of operating systems such as Linux{ XE "tools and platforms:Linux" }* and VxWorks{ XE "tools and platforms:VxWorks" }*, C/C++ cross-compilers, debuggers and integrated development environments (IDEs){ XE "integrated development environment (IDE)" }, make the core familiar territory for us programmers. Except for programming related to core and microengine interactions, we won't cover programming the StrongARM core.

The six microengines are RISC processors optimized for fast-path packet processing.

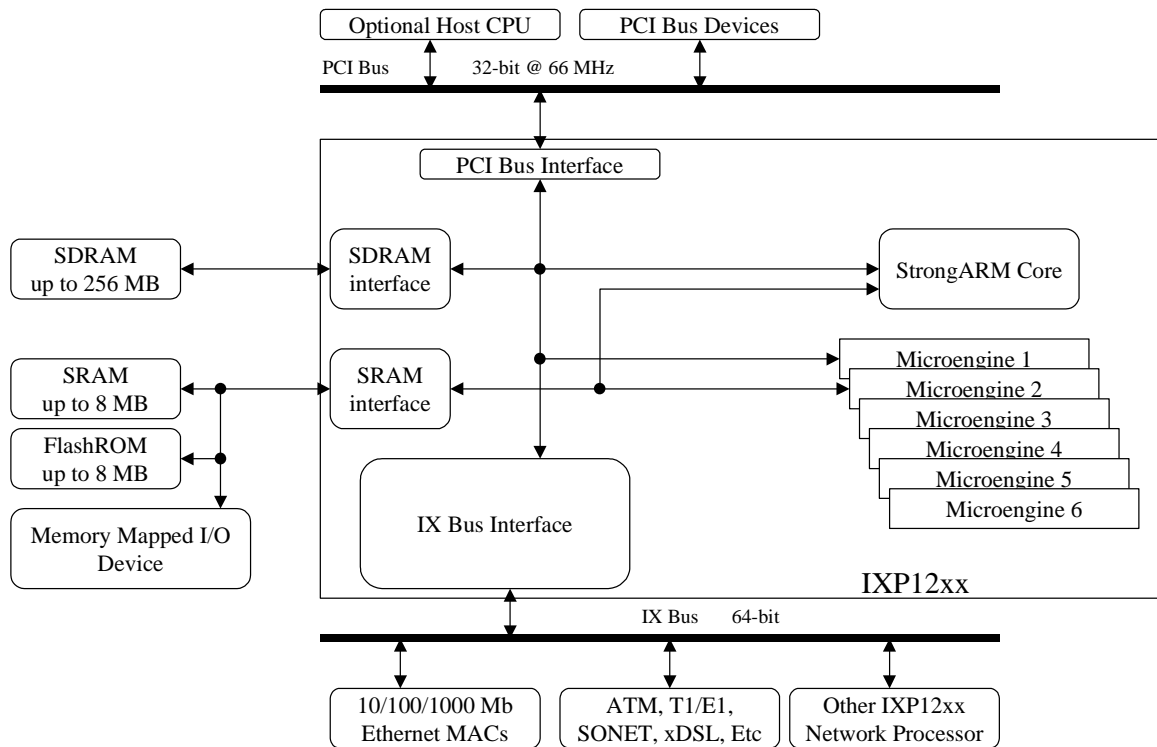


Figure 2-1 The functional units of the IXP12xx.

NextGen: The next generation processors will have 8 or 16 microengines and an Intel® XScale{ XE "tools and platforms:XScale" } core instead of the StrongARM core. Each microengine will have eight threads, twice the current number, and a wider range of instructions. Intel's XScale microarchitecture supports a new version of the ARM* instruction set (V5.0) and can support much higher clock frequencies than the StrongARM microarchitecture. The next generation processors will run at frequencies of up to 1.4GHz. Additionally, the IX Bus Interface is replaced with a Media Switch Interface that will support several standard physical interfaces to switch fabrics and framing hardware. See Chapter 13 for more detailed information about next generation IXP2xxx hardware.

StrongARM Core

The StrongARM core{ XE "StrongARM core" } on an IXP12xx series processor is based on the Intel SA-1 core. This core implements the 32-bit ARM V4 architecture as defined by ARM Limited. The StrongARM core offers an exceptional tradeoff between computational capability and power consumption.

Intel and many other companies support the StrongARM core with software like operating systems such as Linux and VxWorks as well as compiler tool chains from the GNU project. Programming the StrongARM core is not much different from programming any other embedded general-purpose processor. Since a wealth of information is available on this topic, it is not covered in any detail here.

Microengines

The microengines{ XE "microengines" } have an instruction set specifically tuned for processing network data. The instruction set has operations that operate at bit, byte, and long-word levels, in addition to arithmetic and logical operations combined with shift and rotate operations in single instructions. However, the microengines have no integer multiplication or divide, and no floating-point operations. Integer multiplication{ XE "integer multiplication" } is achieved with the sign condition code and an instruction that performs a conditional add operation based on the sign condition code. Integer multiplication can be accomplished through iterative, conditional add operations.

Note: New IXP12xx programmers are usually curious about the lack of a multiplication instruction in the microengines. Even though multiplication can be accomplished through iterative add operations, designing data structures and algorithms on the microengines to avoid multiplication is common. For example, indexing an array requires a multiplication of the array index with the array element size. This multiplication can be reduced to a single shift operation if the array element size is a power of 2 long-words. So by padding the size of an array element to a power of 2 long-words, we can remove the need for array indexing multiplication operations. Similarly, network applications that rely on multiplication can implement an iterative multiplication,

be restricted to run on a smaller range of inputs so all multiplications can be approximated by shifts, or be run on the core.

Figure 2-2 shows the functional blocks in each microengine.

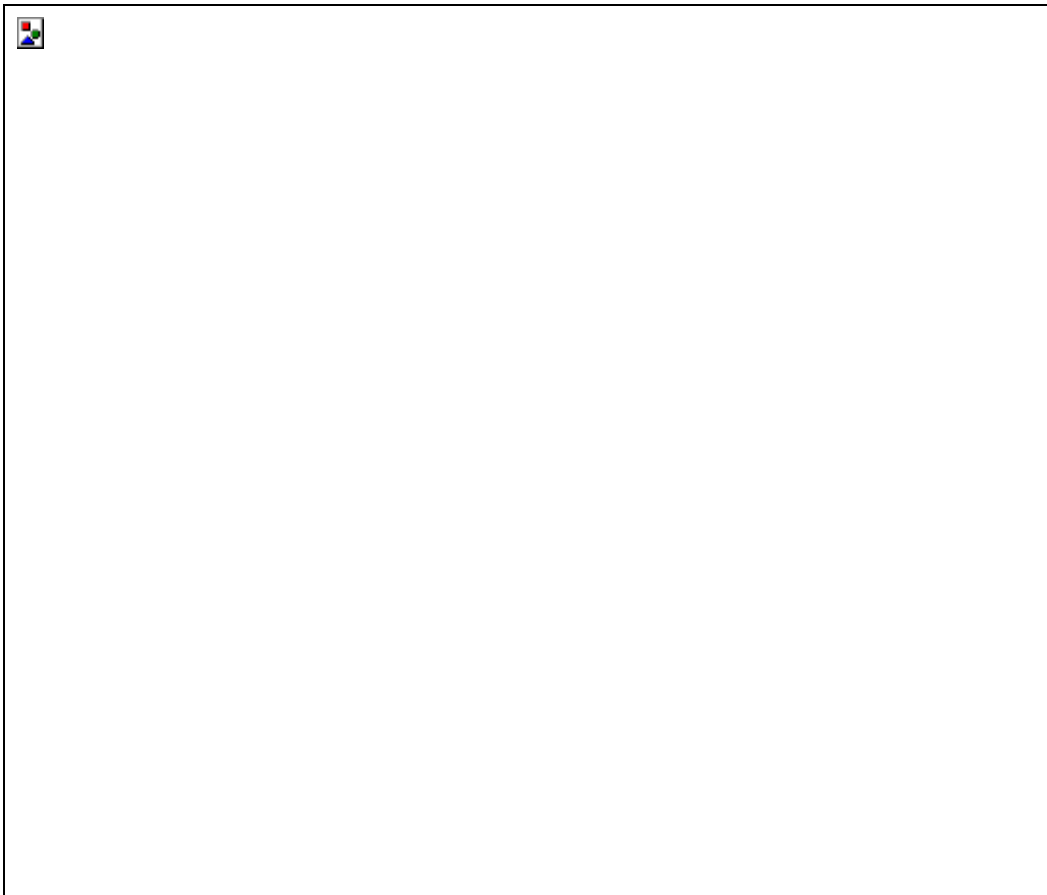


Figure 2-2: IXP12xx Microengine Block Diagram

Each instruction on the microengines takes up one long-word (32-bits) of control store space. Each microengine has an independent 8KB instruction store, enough for 2,048 instructions. Some older versions of the IXP12xx have 4-kilobyte instruction stores, which allow for 1,024 instructions. Code on the Intel StrongARM core{ XE "StrongARM core" } loads this instruction store before the microengines begin running. Once

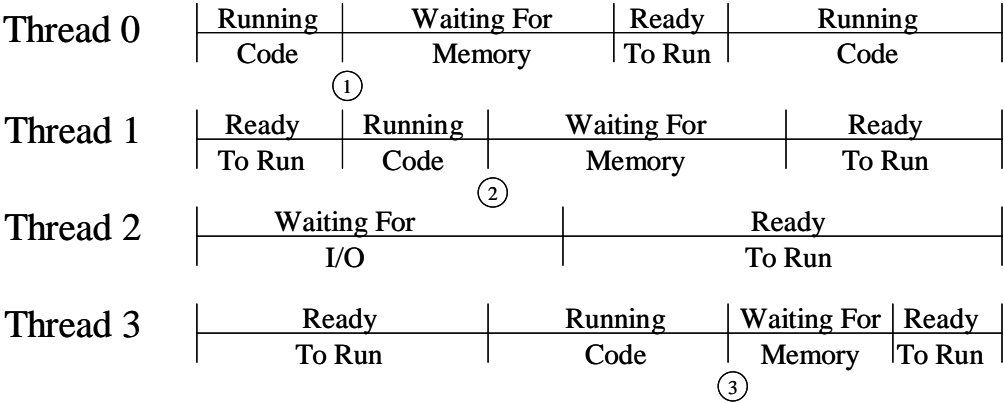
the microengines are running, the instructions are executed in a five-stage pipeline and take one cycle to execute when the pipeline is full. When instructions block on memory or device access, or when branch instructions force some instructions in the pipeline to be aborted, the average instruction execution time is longer than one cycle.

Each IXP12xx series microengine has four hardware-assisted threads of execution, and all threads in a particular microengine execute code from the single instruction store on the microengine. Unless this code contains instructions that branch based on the thread number, all of the threads end up executing the same code. To have one or more threads executing different code, the code must branch to the different code segments according to the thread number. Of course, branching based on thread number, reduces the effective size of code that each thread can execute since the total amount of code must still fit within the single instruction store on the microengine.

The registers of each microengine do not need to be flushed to memory when the control of the microengine switches from one thread to another, as is the case for most registers in general-purpose processors. Register flushing{ XE "register flushing" } is avoided by allocating an equal portion of the total register set to each microengine thread. Although it is possible for any thread to access any register on the microengine, by default each thread accesses its own equal portion of the registers. The latency experienced for a context switch – switching control of the microengine from one thread to another – is therefore the same as an instruction that causes the pipeline to abort the current execution, about three clock cycles.

The non-preemptive hardware thread arbiter swaps between threads in a microengine in round-robin order, only activating threads that are ready to run. The scheduler keeps track of which threads are ready to run and when one thread gives up control of the microengine, it searches in thread ID order for another thread in the microengine that is ready to run. That's what we mean by "round-robin." What about "non-preemptive?" Many operating systems have preemptive thread schedulers, which means that the developer cannot control or predict when a particular piece of code will be interrupted in order to let another piece of code run. This is not true with the microengine thread arbiter. The code must explicitly give up control of the microengine before another thread is allowed to run.

Often, threads release control of the microengine while waiting for memory or other hardware operations to complete. While code is accessing memory - an operation that can take tens of cycles – the thread can swap out and allow another thread to run while waiting for the memory access to complete. This maximizes the work the microengine is doing. Figure 2-3 is a timing diagram that illustrates this thread swapping{ XE "thread swapping" } mechanism.



This diagram shows a timeline of thread activity on one microengine. At the beginning of the timeline, thread 0 is running code and the other threads are either waiting for responses from other hardware units or ready to run. At the time marked with the number 1, thread 0 issues a memory read, and explicitly releases control of the microengine to wait for that memory access to complete. At the same time, the arbiter determines that the next thread, thread 1, is ready to run and starts it running. At the time marked with the number 2, thread 1 issues a memory access and explicitly releases control of the microengine to wait for that memory access to complete. The arbiter then tries to run thread 2, but it is waiting for an I/O access that hasn't been completed. So the scheduler runs thread 3, since it is ready to run. At the time marked with the number 3, thread 3 issues a memory reference and explicitly releases control of the microengine. The arbiter then runs thread 0 because it is the next ready thread.

Figure 2-3 Sample Timing Diagram for Microengine Threads

The non-preemptive thread arbiter{ XE "thread arbiter" } makes it possible for microengine threads to deal with memory asynchronously. While a microengine thread can choose to explicitly release control of the microengine while, say, waiting for a memory operation, it can also choose not to release control.

For example, a microengine thread issues a memory read request and then continues processing. The completion of the memory read request could then be asynchronously reported back to the microengine thread. Asynchronous memory access is a key differentiator between microengines and most general-purpose processors, and we will be seeing a lot of this idea as we go along.

The non-preemptive nature of the thread arbiter simplifies synchronization within a microengine. To do computation on a register and maintain mutual-exclusion, simply avoid instructions that give up control of the microengine. The hardware then maintains mutual exclusion for you. Just make sure your code releases control once in a while, or else no other threads will get to run.

The microengines have three different types of registers: general purpose, Synchronous Random Access Memory (SRAM) { XE "synchronous random access memory (SRAM)" } transfer, and Synchronous Dynamic Random Access Memory (SDRAM) { XE "synchronous dynamic random access memory (SDRAM)" } transfer. These registers are described here:

- General purpose: Each microengine has 128, 32-bit general-purpose registers (GPRs), allocated into two banks of 64 registers. The two banks are called the A and B banks. The distinction between which bank a register is in is not important until you inadvertently require a particular variable to be in both banks. This can occur because any instruction that allows two GPR's as input requires one of the GPRs to be from the A bank and the other from the B bank. Thus, the following code is not valid:

```
/* x, y, z are GPRs */  
op(x, y); /* Requires x and y to be in opposite banks */  
op(x, z); /* Requires x and z to be in opposite banks */
```

```
op(y, z); /* Requires y and z to be in opposite banks
          IMPOSSIBLE */
```

In this example, there are three GPRs named, x , y , and z . If x and y are used together in an instruction, x and y must be in different banks. Which bank each is in is not important. If then x and z are together in an instruction, x and z must be in opposite banks. Taken together with the first instruction means that y and z must be in the same bank. Thus, any instruction using y and z in the same instruction results in code that cannot compile on the IXP12xx. In practice, this problem is rare and easy to work around by copying one of the variables into a temporary variable before use.

The 128 GPRs per-microengine can be accessed in thread-local or absolute mode. In thread-local mode, each thread accesses a unique set of 32 GPRs, or 16 A bank GPRs and 16 B bank GPRs.

GPRs registers can also be allocated as “absolute{ XE "absolute registers" }” (also called “global{ XE "global registers" \t "See absolute registers" }”), making them accessible by any thread on the microengine. Absolute registers are useful for inter-thread communication{ XE "inter-thread communication" } within a microengine.

- SRAM transfer: Each microengine also has 64 SRAM transfer registers. Like the GPRs, when these transfer registers are accessed in a thread-local manner, each thread accesses an equal, unique, set of these registers. SRAM transfer registers can also be addressed globally, but that is rarely done.

SRAM transfer registers are used to read from and write to all functional units on the IXP12xx except for the SDRAM unit. This means SRAM transfer registers are used to read and write data to and from the SRAM unit as well as the IX Bus and Peripheral Components Interconnect (PCI){ XE "peripheral components interconnect (PCI)" } interfaces.

SRAM transfer registers, and transfer registers in general, are the primary mechanism for dealing with asynchronous memory operations. When data is read from these other functional units, it is placed in SRAM transfer registers, and when microengine code writes data to these units, it must first be placed in transfer registers. Half of these registers are write-only, and the other half are read-only. For example, in order to write to SRAM, the microengine code must put data in a write-only SRAM transfer register, and to read data from

SRAM, the code must read from a read-only SRAM transfer register. Thankfully, the microcode assembler and the microengine C compiler prohibit the programmer from doing this incorrectly. However it is still possible to get confused when dealing with the read-only and write-only distinction of transfer registers. When reading microengine assembly code, be sure to remember that read and write transfer registers share the same names. Writing to the register name places the data into the write-only transfer register; reading from the register gets the data from the read-only transfer register.

- SDRAM transfer: Each microengine has 64 SDRAM transfer registers divided equally into read-only and write-only. SDRAM transfer registers are used exclusively for communication between the microengines and the SDRAM unit.

The advantage of having separate transfer and general-purpose registers is that the microengine can continue processing with GPRs while other functional units of the IXP12xx read and write the transfer registers. Chapter 8 explores this capability in more detail.

The IXP12xx series microengines also have access to many control status registers (CSRs). These CSRs are used for a wide variety of control and configuration, so we'll cover their usage and meaning as we go.

OTHER FUNCTIONAL UNITS

In addition to the microengines and the Intel StrongARM core{ XE "StrongARM core" }, Figure 2-1 shows four other functional units of the IXP12xx, each primarily focused on providing access to external devices. The SRAM and SDRAM units provide access to different memory types, providing better memory optimization. The Peripheral Components Interconnect (PCI){ XE "peripheral components interconnect (PCI)" } unit provides an interface to an external industry-standard PCI bus, like those found in all personal computers today. The IX Bus Interface Unit is historically referred to as the First In First Out (FIFO){ XE "First In First Out (FIFO)" } Bus Interface (FBI){ XE "FIFO Bus Interface (FBI)" }. The FBI provides access to a high-speed external data bus where all packets will come and go. Additionally, the FBI Unit contains a small amount of on-chip SRAM memory called scratchpad memory{ XE "memory:scratchpad memory" }, chip-wide control status registers (CSRs){ XE "control status

register (CSR" }, a hash generator and, a little programmable processor called the ready-bus sequencer{ XE "ready-bus sequencer" }.

Memory Interfaces

The IXP12xx provides three different memory interfaces: scratchpad, SRAM and SDRAM. Table 2-1 shows the tradeoffs of each type of interface in terms of size, latency, minimum addressable unit, and special operations. The scratchpad memory interface is the smallest size, with the lowest latency memory available to the IXP12xx. SDRAM is the largest memory interface, with the highest latency memory available to the IXP12xx, and is optimized for bulk sequential accesses. The SRAM{ XE "synchronous random access memory (SRAM)" } memory interface takes the middle ground between scratchpad and SDRAM in both size and latency.

Table 2-1. The properties of the three IXP12xx memory interfaces

Memory interface	Minimum addressable unit (bytes)	Size ¹ (in minium addr. units)	Size (bytes)	Approx. unloaded latency (clks)	Special Operations
Scratchpad	4	1K	4K (on-chip)	12 - 14	Atomic increment, bit test-and-set, bit test-and-clear
SRAM	4	2M	8M (addressable)	16 – 20	CAM lock, bit test-and-set, bit test-and-clear, push-pop queues
SDRAM	8	32M	256M (addressable)	33 – 40	Direct path to and from the FBI which allows data to be moved between the two without first going through one of the processors.

¹ In minimum addressable units, which is the smallest unit of directly addressable data. For example, both scratchpad and SRAM have minimum addressable units of 4 bytes, whereas SDRAM has an 8 byte minimum addressable unit. So reading 4 bytes of data from SRAM address 1 will return bytes 4 through 7 in memory. Reading 8 bytes of data from SDRAM address 1 will return bytes 8 through 15 in memory.

NextGen: The microengine version 2 (MEv2{ XE "Microengine Version 2 (MEv2)" }) architecture adds a 640KB memory bank for each microengine. This new memory bank is called local memory and represents the lowest latency memory available to a microengine. Additionally, the next generation processors will have 16KB of scratchpad memory, 1GB of addressable SRAM, 2GB of addressable SDRAM, and more special operations on each of these memory types.

Each memory type implements a unique set of special operations summarized in the final column of Table 2-1. Most of the special operations are targeted at managing multithreading in the microengines. Thus, like any multithreaded programming, synchronization primitives and atomic operations are key to programming the microengines. Synchronization mechanisms, such as the Content Addressable Memory (CAM){ XE "content addressable memory (CAM)" } locks provided by the SRAM unit and the scratchpad increment operation, allow multiple threads to coordinate access to shared data structures.

Scratchpad

Scratchpad memory{ XE "memory:scratchpad memory" } is 1K long-word (4KB) of on-chip memory. Scratchpad provides a small, low-latency memory interface to all of the microengines. The scratchpad is physically located within the FBI, but that fact is typically only interesting to people that also worry about transistor counts and power consumption.

ICON Sometimes, the fact that scratchpad physically resides in the FBI might be an issue to a programmer. Since the FBI is responsible for many other tasks, adding a large number of scratchpad accesses to the FBI's workload might deteriorate overall system performance.

In addition to random access reads and writes, scratchpad also provides atomic bit-test-and-set, bit-test-and-clear, and increment operations. The atomic increment operation is unique to scratchpad memory. This operation is ideal for keeping simple counters across multiple microengine threads{ XE "microengine threads" }.

SRAM

Unlike scratchpad memory, SRAM is off-chip. The IXP12xx only provides an interface to SRAM memory. This interface is embodied in the SRAM unit. The SRAM unit provides an interface for up to 2M long-words (8MB) of medium-latency memory.

Note: The name of the SRAM unit is a bit misleading. Multiple memory types can be externally attached to the SRAM unit. In fact, the hardware interface provides facilities for certain “slow port” memories such as boot ROM, or even special-purpose devices that offload computations from the IXP12xx.

For example, classification coprocessors exist for the IXP12xx that can offload tasks such as the longest-prefix match lookups done in IP routers. Such devices could all be memory mapped into the SRAM address space and accessed through the SRAM unit.

Like scratchpad, the SRAM unit provides atomic bit-test-and-set and bit-test-and-clear operations. In addition, the SRAM unit supports eight Last In First Out (LIFO){ XE "Last In First Out (LIFO)" } linked lists. Push and pop operations on these lists are atomic. Meaning, you can have eight stacks in memory and the SRAM unit makes it easy for multiple threads to push and pop from these stacks. See Chapter 4 for information on properly using these stacks for buffer memory management

The SRAM unit also provides a generic synchronization primitive for locking and unlocking memory. When a microengine thread attempts to lock a previously locked memory location, the hardware will block the thread until the memory is unlocked. Locking memory can be combined with a read operation, and unlocking memory can be combined with a write operation.

ICON Use SRAM locks like a critical section for momentary synchronization and you have a new best friend. Use these locks like a condition variable for event signaling and you discover a new meaning for the word frustration. See Chapter 7 for a discussion of the trouble attributed to a quirk in the implementation of the locks that arises with inter-microengine programming.

SDRAM

Like SRAM memory, SDRAM memory is external to the IXP12xx. The SDRAM unit provides an interface for up to 32M quad-words (256MB) of high-throughput memory. The SDRAM unit does not accommodate the atomic bit operations like scratchpad and SRAM. Instead, the SDRAM unit's unique functionality lies in the ability to move data to and from the FBI unit without the data going through the microengines. For those of us writing software, this means one less thing to write and debug, one less bus transfer, and more time for the microengines to be performing other useful work.

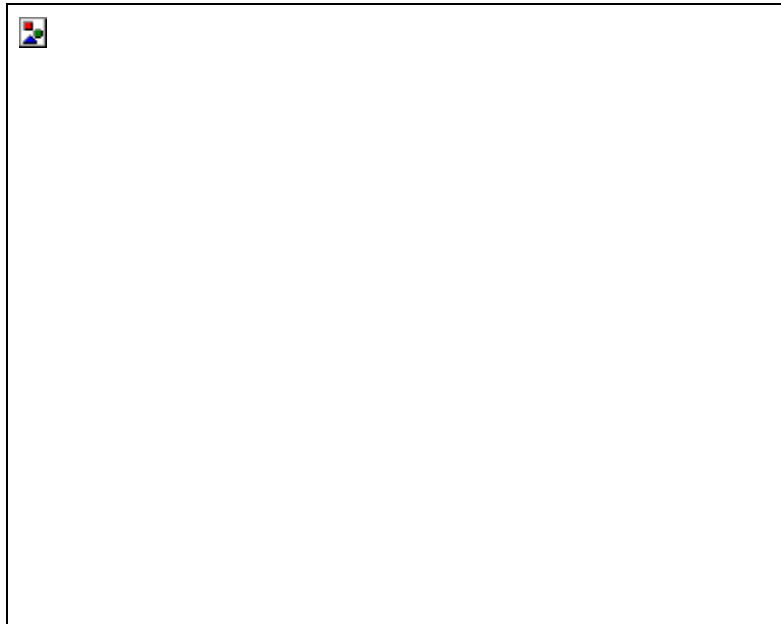
Minimum Addressable Unit

Note the description of the scratchpad as 1K long-word instead of 4K bytes. Likewise, SRAM is described as 2M long-words and SDRAM as 32M quad-words instead of the familiar 8MB and 256MB respectively. This phrasing is deliberate. None of the memory interfaces are byte-addressable from the microengines. Each scratchpad and SRAM address represents 32 bits, or 4 bytes, of data and each SDRAM address represents 64 bits, or 8 bytes, of data. Thus two, 32-bit transfer registers are required to read (write) data from (to) a single SDRAM address. When you read or write SRAM or scratchpad memory, you must provide long-word addresses. When you read or write SDRAM memory, you must provide quad-word addresses.

NextGen: While the next generation hardware still accesses SRAM and scratchpad memory in long-words, and SDRAM memory in quad-words, these memory types are byte-addressable.

Being unable to address individual bytes inevitably has implications on the design of data structures and code. Consider trying to read the destination Internet protocol{ XE "destination internet protocol (DIP)" } (IP) address (DIP) of an IP packet encapsulated in an Ethernet II frame. Remember that SDRAM has a direct connection to the FBI, so packet data is placed in SDRAM memory. Figure 2-4 shows such a frame in SDRAM memory along with the address boundaries of the memory. Note that the DIP, a 4-byte quantity, spans an address boundary. This means two SDRAM addresses will be read to get the DIP: one read from address 3 and a second from address 4. Each read requires two registers, for a total of four, 32-bit registers needed to get the 32-bits desired.

On the positive side, these two memory reads can be combined into one instruction, and extracting the final destination IP address from the 4 transfer registers can be accomplished in two `alu` instructions. Chapter 8 shows how to design data structures and code that deals with the non-byte addressability of the IXP12xx memory.



To read the destination IP address from an Ethernet II frame carrying an IP packet in SDRAM would require 2, 8-byte reads because of the non-byte addressability of the memory.

Figure 2-4 The effects of non-byte addressable memory on microcode

Memory Command Queues

As discussed earlier, the microengine threads run with a non-preemptive arbiter. The active thread on any microengine is responsible for releasing control of the microengine. This control means software can be written that allows hiding memory latencies by issuing read, write, and other special commands on the memory interfaces and then continuing to perform work while the memory interface services the command. Now this memory latency-hiding thing is cool, but it gets even better. Each of the three memory interfaces queues incoming command requests and

nothing prevents a single microengine thread from having multiple requests outstanding in one of these queues.

For example, microcode can be written that initiates a packet transfer from the FBI into SDRAM, reads that same packet data from the FBI into registers in the microengine so the packet header can be processed, and increments a global packet counter in scratchpad memory, all at the same time. Now imagine performing these operations in parallel across many microengine threads and some of the unique programming opportunities of the IXP12xx start to emerge.

The SRAM and SDRAM units provide multiple command queues and allow the software to place any given command on the most appropriate queue. As Figure 2-5 illustrates, both the SRAM and SDRAM units provide an ordered queue, a priority queue, and one or more optimize memory queues.

The default queue for all commands is the ordered queue, which maintains the order of the commands. While just using the ordered queue is probably all that's needed, knowing about the other queues in the SRAM and SDRAM units could prove useful.



Figure 2-5 The command queues internal to the SRAM and SDRAM units

The software can elect to place any command (read or write) into the priority queue. Entries in the priority queue are always serviced before entries in the ordered or optimize queues, even if this means starving these other queues indefinitely.

The software can also elect to 'optimize' any command. For both SRAM and SDRAM, optimizing a set of commands instructs the hardware to rearrange the order of the commands to take advantage of the physical

bus characteristics. For the SRAM unit, optimizing commands means separating the read and write commands and performing multiple reads followed by multiple writes. For the SDRAM unit, optimizing commands means separating the commands by memory address and issuing commands against different physical banks of memory. By using the priority and optimize queues, throughput and latency{ XE "latency" } are improved by taking advantage of bus timings.

Rule of Thumb: It can be difficult to tell when to use the priority and optimize queues. Since these options do not necessarily maintain the order of commands, be judicious about using them in conjunction with memory latency hiding schemes. Figuring out how to avoid every read-after-write and write-after-write hazard isn't worth it. However, if there is a read-only or write-only piece of data, or the code waits for each read and write to complete before performing more memory commands, using the priority and optimize queues is an easy way to speed things up and impress your friends. Try adding priority options to commands that unlock some shared resource. Optimized commands are great for groups of sequential reads or writes like traversing a data structure or writing updates to a packet.

The FBI

The FBI{ XE "FIFO Bus Interface (FBI)" } is the proverbial kitchen sink of the IXP12xx. The FBI holds the on-chip scratchpad memory{ XE "memory:scratchpad memory" }, contains a set of control and status registers (CSRs){ XE "control and status register (CSR)" }, a hash generation unit{ XE "hash generation unit" }, and the interface to the IX bus. The details of every CSR aren't important here, so each CSR that is important will be covered when needed. The hash function of the FBI generates 48-bit and 64-bit hashes of the same sized data. The actual hash function is parameterized by several CSRs.

The real purpose of the FBI is to provide an interface for receiving and transmitting packets. Packet reception and transmission on the IXP12xx is a unique, complex act of reassembling and segmenting small partial-packet data chunks. The reassembly and segmentation aspects of packet movement will likely dictate the overall threading model developed for your microcode. So, understanding the receive and transmit state

machines that control the basic processes of receiving and transmitting packets on the IXP12xx is important.

The Receive State Machine

Receiving a packet on the IXP12xx is accomplished by a series of handshakes between the microengines, the FBI, and the external hardware. All of these handshakes are coordinated by the microengines. The external data bus on the IXP12xx, called the IX bus, transfers 64-byte chunks of data at a time. These 64-byte chunks are called mpackets{ XE "mpackets" }. For physical media that have frames larger than 64-bytes, the microengines have to coordinate reassembling the individual mpackets into one complete frame.

To simplify the discussion of the receive state machine, let's just consider the process of receiving one mpacket. All of the software details related to the reassembly process are in Chapter 4.

Figure 2-6 shows the five components of the FBI primarily responsible for the reception of a single mpacket. The microengine first determines whether any mpacket is present by checking the `rcv_rdy_lo`{ XE "rcv_rdy_lo" } or `rcv_rdy_hi`{ XE "rcv_rdy_hi" } CSRs. The `rcv_rdy_lo` and `rcv_rdy_hi` registers represent a bitmask{ XE "bitmask" } where each bit represents mpacket availability on a physical port.

For example, if bit 5 is set, port 5 has an mpacket available. The `rcv_rdy_lo` register represents ports 0 through 31, while the `rcv_rdy_hi` register represents port 32 through 55. Collectively, we refer to these two CSRs as the receive-ready bits. Checking the receive-ready bits can either be done by polling the associated CSRs, or by having receive state machine "autopush{ XE "autopush" }" the receive-ready bits into the transfer registers of a specified microengine.

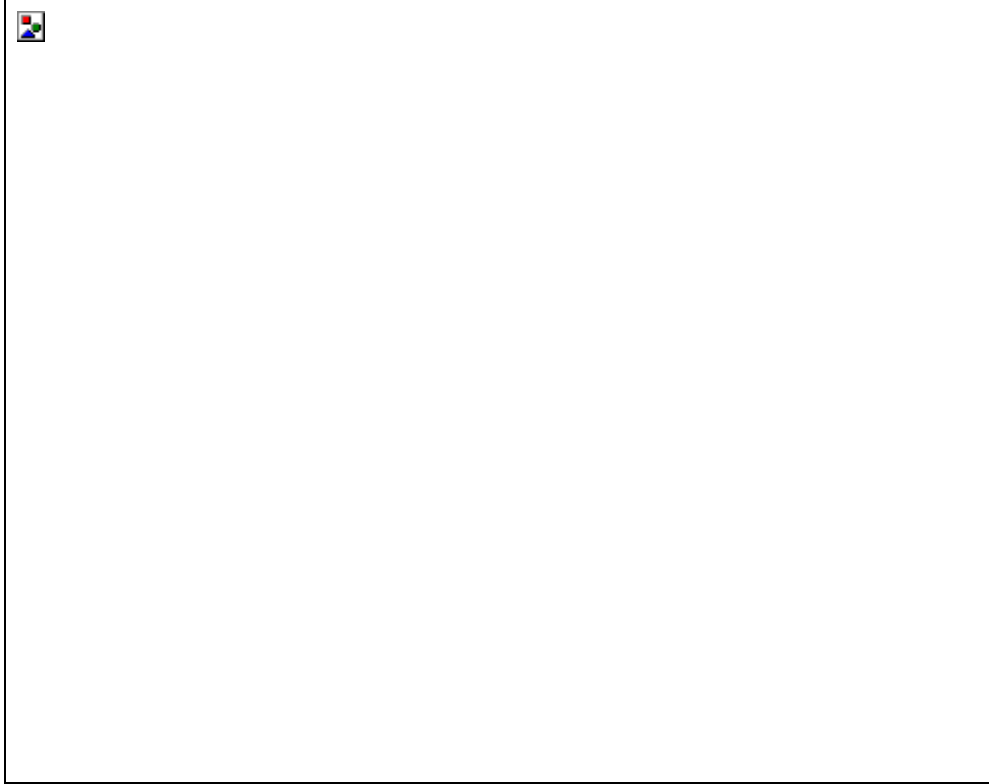


Figure 2-6 The components of the receive state machine

Once an mpacket is available on a port, a microengine initiates a transfer of this data, called a receive request, from the physical hardware into the FBI. A receive request is issued by writing the `rcv_req{ XE "rcv_req" }` CSR in the FBI.

A receive request instructs the FBI to move one or two mpackets from the physical hardware into a Receive FIFO (RFIFO){ XE "Receive First In First Out (RFIFO)" } element. The FBI contains 16 RFIFO elements, each one holding one mpacket. No RFIFO element is better than another and how they are used is entirely under the control of the software.²

² The RFIFO is a misleading name. Receive buffer, RBUF, array would have been a much better name and indeed that is precisely the name chosen for the next-generation hardware.

When the receive-request mpacket transfer is complete, the FBI signals the microengine and updates the receive-control (`rcv_cntrl`) CSR. The `rcv_cntrl` register contains status information about the mpacket received. Actually, the `rcv_cntrl` register is a queue of four registers. From a programmer's viewpoint, reading the `rcv_cntrl` register represents a dequeue operation, but it is sufficient to think of the `rcv_cntrl` register as just that, a register. This status information consists of several failure notifications, reassembly state, and the size of data in the mpacket.

Except for all of the details, which are covered later, that's the process of receiving an mpacket. For now, remember that the data moves from the IX Bus devices into RFIFO elements, initiated by the microengines issuing a receive request.

The Ready-bus Sequencer

How do the receive-ready bits get set in the first step of the receive process? The FBI contains a tiny processor called the ready-bus sequencer{ XE "ready-bus sequencer" }. Typically, the ready-bus sequencer is programmed to probe the physical devices on the external data bus to find out which ones have available data. Based on the information discovered, the sequencer updates the receive-ready bits.

The ready-bus sequencer contains a 12-entry instruction store and is programmable using a small instruction set. The basic instructions are:

- Poll the Media Access Controllers (MACs){ XE "media access controller (MAC)" } for ready bits
- Push the results of the ready bits into transfer registers
- Send and receive information between two or more IXP12xx's sharing a single IX Bus.

Programming the ready-bus sequencer is generally a one-time task depending on the exact hardware configuration, so the discussion about it is covered in Chapter 10, as an advanced programming topic.

The nature of the ready-bus sequencer causes a delay between the time that the IX Bus device actually indicates data availability (or unavailability) and the time that the receive-ready bits get set or cleared respectively. This can lead to the receive-ready bits sometimes having stale data. In a few instances, the microcode can actually run fast enough

to make this delay a factor in correct operation of the device. Chapters 6 and 10 provide details on dealing with this delay.

The Transmit State Machine

Figure 2-7 shows the hardware components in the FBI used by the microengines to transmit packets. One of these components is the ready-bus sequencer. The ready-bus sequencer does the same thing for transmit ready bits that it does for receive-ready bits. It collects a state bit from each port and puts it in a CSR called `xmit_rdy`{ XE "xmit_rdy" } for the microengine code to read.

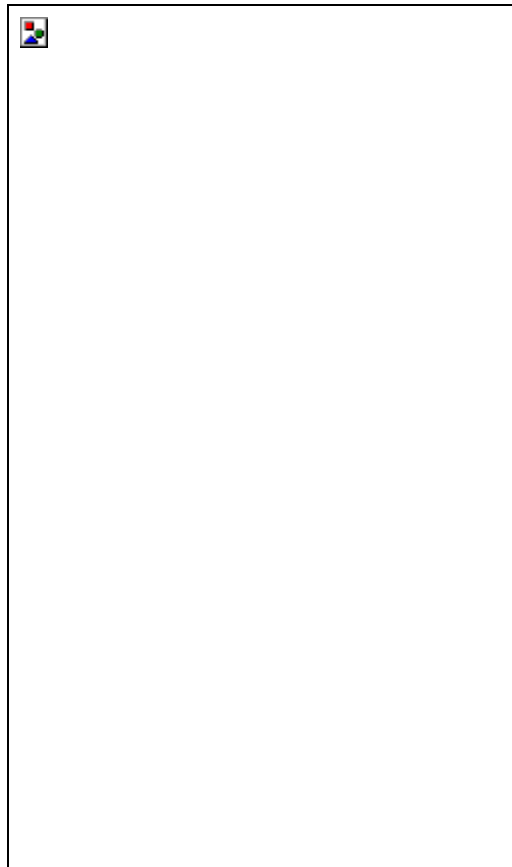


Figure 2-7 The components of the FBI used in transmitting packets

The most important component of the FBI used for transmitting packets is the array of Transmit FIFO (TFIFO){ XE "Transmit First In First Out (TFIFO)" } elements. Each TFIFO element has the following fields:

- The data field holds 64 bytes of buffer data to be sent to a MAC device.
- 8 bytes of control information telling the transmit state machine which port to use to send out the data as well as where the data fits in a potentially larger packet.
- The valid field indicates if the data and control sections of the structure are complete. The transmit state machine does not operate on a TFIFO element without the valid bit turned on.

Unlike the RFIFO elements, TFIFO elements are treated by the hardware as a circular buffer. The transmit state machine maintains a TFIFO element pointer into the array of TFIFO elements. If the pointer is on a TFIFO element with the valid bit on, the transmit state machine processes the mpacket, clears the valid bit, and advances the TFIFO element pointer.

Actually, transmitting packets is not quite this simple, but Chapter 9 goes into greater detail.

THE IXP12XX FAMILY

The IXP12xx designation represents a family of four processors: the IXP1200, IXP1240, IXP1250, and the extended temperature IXP1250. All of the information contained in this chapter applies to every processor in the family, however the IXP1240 and IXP1250 contain extra hardware support for Cyclic Redundancy Check (CRC){ XE "cyclic redundancy check (CRC)" } calculations and Error Correction Code (ECC){ XE "error correction code (ECC)" } memory interfaces. ECC support deals with the hardware interface to SDRAM and is transparent to the programmer. This book does not deal with ECC.

CRC support enables the programmer to compute three different cyclic redundancy checks on data in SDRAM. Offloading the compute-intensive task of calculating CRCs to the hardware enables applications that require CRC results to see significant performance improvements. See Chapter 10, for information on using the CRC feature of the IXP1240 and IXP1250.

Table 2-2 summarizes the capabilities of the members of the IXP12xx family of processors.

Table 2-2. The properties of the processors in the IXP12xx family

Feature	IXP1200	IXP1240	IXP1250	Ext. Temp. IXP1250
Core clock speed (MHz)	166, 200, 232	166, 200, 232	166, 200, 232	166
CRC support	No	Yes	Yes	Yes
ECC support	No	No	Yes	Yes
Operating temperature (°C)	0° to 70°	0° to 70°	0° to 70°	-45° to 85°

SideBar A day in the life of a packet

So now that you have read through the description of the IXP12xx hardware, if you still can't explain how a packet makes it into, then out of, the IXP12xx, don't sweat it. It was an embarrassingly long time after we started working with the IXP12xx before we could explain just how a packet traveled through the IXP12xx hardware.

From a programmer's perspective, the best way to learn about the IXP12xx hardware is through an example of the "big picture." This example of a day in the life of a packet examines how a single packet makes its way through the various functional units of the IXP12xx. Nearly every portion of the IXP12xx hardware that programmers care about is covered by following a packet through the hardware.

Figure 2-8 shows the functional units of the IXP12xx with numbers corresponding to the units visited during the packet reception, packet processing, and packet transmission.

In the morning: packet reception

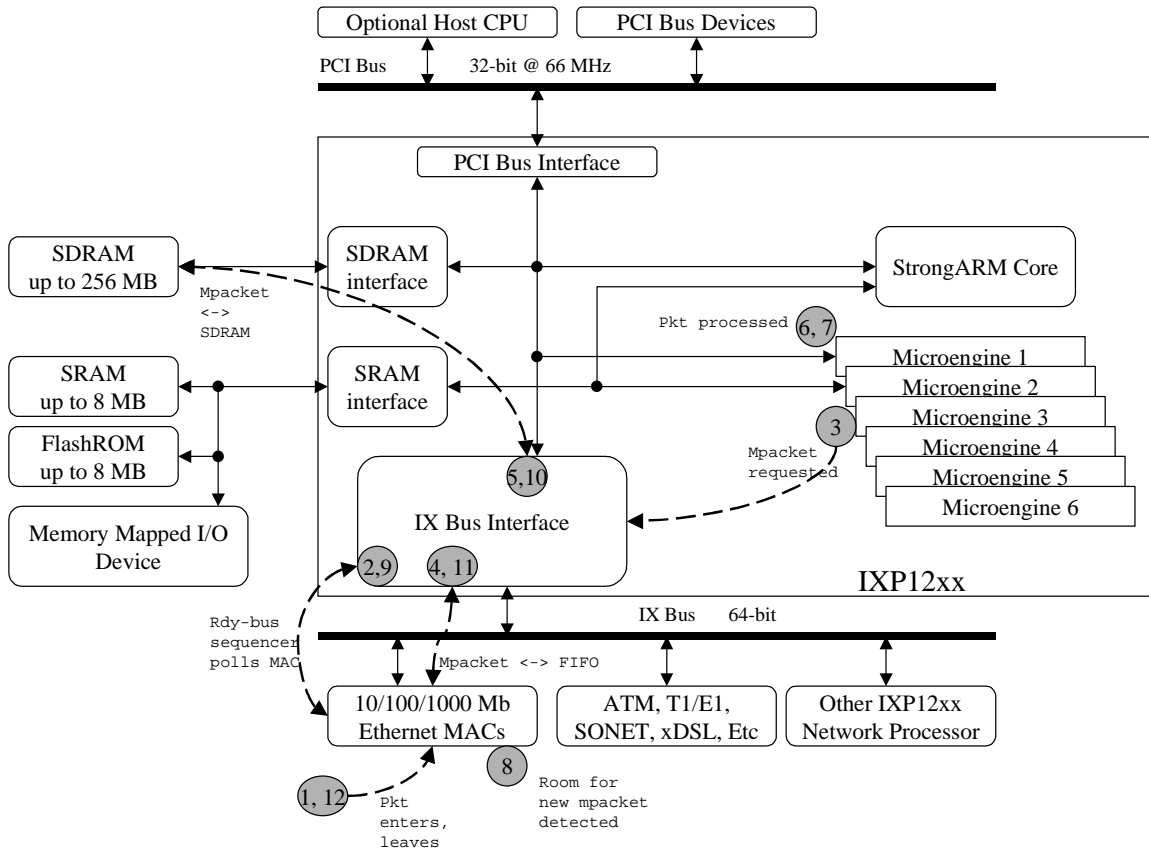


Figure 2-8 The functional units of the IXP12xx. The numbers correspond to the path of a packet through the hardware during packet reception, processing, and transmit.

1. The packet's day begins when it is received by one of the MAC devices attached to the IX Bus. A MAC is a piece of hardware, external to the IXP12xx that converts some physical media type to IX Bus ready-bus control signals and mpackets. The physical media types could be anything including Ethernet{ XE "Ethernet" }, Asynchronous Transfer Mode (ATM){ XE "asynchronous transfer mode (ATM)" }, and Synchronous Optical Network (SONET){ XE "Synchronous Optical Network (SONET)" }.
2. The ready-bus sequencer polls the MAC for mpacket{ XE "mpacket" } availability and when at least one mpacket's worth of data arrives at the MAC the ready-bus sequencer updates the receive-ready bits.

3. The microengines burst into action. After checking the receive-ready bits, the microengines --presuming they are programmed to do so, instruct the IX Bus Interface to transfer the first mpacket into a specific RFIFO element.
4. The IX Bus Interface transfers the mpacket from the MAC across the IX Bus and into the given RFIFO element. Once the mpacket has been completely transferred into the RFIFO element, the IX Bus Interface notifies the microengines.
5. The microengines move the mpacket directly from the RFIFO element into SDRAM memory.

This five-step process repeats for each mpacket in the packet until done.

The afternoon: packet processing

After the packet has been reassembled into SDRAM memory, the microengines, or even the StrongARM core{ XE "StrongARM core" }, can do something with it. The packet processing typically consists of some level of packet classification, followed by packet modification.

6. The packet classification stage begins reading various portions of the packet from SDRAM, or RFIFO elements, into the microengine. From there, the classification typically makes use of SRAM memory to store tables of moderate size that can be quickly searched.
7. The packet is modified by overwriting the packet data already in SDRAM memory.

For more details, see Chapter 8 for a look at Ethernet bridging as an example of packet processing.

The evening: packet transmission

To wrap up the packet's busy day, it must be transmitted out of the IXP12xx. From a high-level perspective, the transmission process is just the reverse of the reception process. The important technical differences between the two processes are covered in later chapters.

Before the packet transmission process begins, the assumption is that the packet-processing step provided the packet's outgoing port number.

8. The MAC checks its internal buffers for enough space to hold at least one outgoing mpacket.

9. The ready-bus sequencer eventually polls the MAC for availability of outgoing mpacket buffer space. The MAC indicates it has room for at least one mpacket and the ready-bus sequencer updates the transmit-ready bits.
10. The microengines under programmatic control check the transmit-ready bits for buffer space availability in the outgoing MAC. Once the microengines notice transmit buffer availability for the outgoing MAC, the first mpacket is transferred directly from SDRAM into a TFIFO element.
11. The IX Bus Interface uses the IX Bus to move the mpacket and control information into the MAC.
12. The MAC begins transmitting the mpacket with the appropriate media-specific framing.
Like the receive process, the transmission process must repeat itself for each mpacket in the outgoing packet. The MAC detects the end of the packet through information contained in the control information supplied by the microengines.

End sidebar

THE SOFTWARE

Intel and other software vendors provide software tools and frameworks to improve the developer's experience in working with the IXP12xx. These tools fall into three basic categories: programming languages, libraries, and development environments. In addition, there are two basic software frameworks provided by Intel. All of the tools and frameworks provided by Intel are part of the Internet eXchange Architecture (IXA){ XE "Internet eXchange Architecture (IXA)" } software development kit (SDK){ XE "software development kit (SDK)" } version 2.0.

Programming Languages

Microcode{ XE "microcode" } and microengine C{ XE "microengine C" } are the two languages that have tool chains available from Intel. Microcode is analogous to assembly on a general-purpose processor, and microengine C is analogous to a 3rd generation language like C.

Microcode allows fine-grained access to register allocation, so SRAM and SDRAM transfer registers can be explicitly used. It has no notion of pointers or functions, but does allow modularization of code through the use of inline-macros.

The following example illustrates some of the differences in complexity between programming in microcode and microengine C.

This piece of microcode takes data from SRAM, increments the data, and writes the result to SDRAM. For a crash course on reading microcode, see Appendix A.

```
// Allocate registers
.local address $sramData $$sdramData

// Load the address register with the SRAM address
immed[address, 0x20]

// Read SRAM
sram[read, $sramData, address, 0, 1], ctx_swap

// Increment data
alu[$$sdramData, $sramData, +, 1]

// Load the address register with the SDRAM address
immed[address, 0x50]

// Write SDRAM
sdram[write, $$sdramData, address, 0, 1], ctx_swap

// Release registers
.endlocal
```

Microengine C is very similar to the classic C language. It offers type safety, pointers to memory, and functions. Since the IXP12xx microengines don't have hardware assistance for a stack, the C language does not provide reentrant functions or function pointers. A stack could be implemented in software, but it would be terribly slow.

This piece of microengine C code performs the same functions as the microcode above:

```
// Declare a pointer into sram memory, sram word address of
// 0x20
__declspec(sram) long* sramData = (long*) 0x20;

// Declare a pointer into sdram memory, sdram quad-word
// address of 0x50
__declspec(sdram) long long* sdramData = (long long*) 0x50;
```

```
// Read the value at sram location 0x20, add one, and write
// back to sdram address 0x50.
*s dramData = (long long)(*sramData + 1);
```

For most of this book, the coding examples are in microengine C. There are some examples in microcode, but in most instances, microengine C coding results in much more readable code with a negligible performance penalty. This does not mean the book is not suitable for readers writing microcode. Indeed, this book is not meant to be a microcode *or* microengine C reference. You can find the *IXP1200 Programmer's Reference Manual* and *Microengine C Compiler Language Support* documents on the Intel web site and on the CD-ROM that comes with this book if you need microcode and microengine C references respectively. The examples are primarily presented in microengine C for readability, but the concepts apply equally well to a developer writing microcode.

Libraries

Intel provides a set of software libraries for the IXP12xx series microengines for both microcode and microengine C developers. The microcode version is called the IXP Blocks{ XE "IXP Blocks" } and the microengine C version is called the Portable MicroC Library{ XE "Portable MicroC Library" }.

These libraries have the same two primary purposes:

- To aid in portability- The IXP12xx series microengine instruction set is not exactly the same across the different product offerings. The interfaces into the IXP Blocks and the Portable MicroC Library were designed with portability in mind, so that code could be written for the IXP12xx and still compile and run on successive generations of the microengines.
- To help perform common programming tasks - For example, calculating the checksum of an IP header, an essential function of processing IP packets, can be done with either an IXP Block or a MicroC Library routine.

The IXP Blocks are broken up into several functional areas:

- Instruction Simplification – provides simplified interfaces to microcode instructions
- Operating System Emulation – provides services like mailboxes and critical sections

- Bus I/O – eases use of IX Bus instructions
- Utilities – provides buffer manipulation, packet queues, and hash tables
- Link Layer – helps with Ethernet field extraction
- IP Stack – helps with IP packet processing

Again, this book is not meant to be a definitive reference. Refer to the *IXP1200 Macro Library Reference Manual* on the accompanying CD-ROM for more information.

The Portable MicroC Library is broken up into the same functional areas except for Instruction Simplification, which is not necessary in microengine C. Also, the Link Layer and IP Stack areas are combined into a Protocol functional area. Refer to the MicroC Library Reference Manual for more information.

Development Environments

The integrated development environment provided in the IXA SDK 2.0 is the IXP12xx Developer's Workbench. This development tool allows development and debugging of microcode or microengine C code in a visual environment in Microsoft* Windows*. The Developer's Workbench comes with a cycle-accurate simulator that is an excellent tool for prototyping and debugging software without hardware. In addition, the Developer's Workbench contains a syntax-highlighting editor for both microcode and microengine C, as well as integration with the microcode assembler, microengine C compiler, and microcode linker. The Developer's Workbench also contains a source-level debugger for both microcode and microengine C. This book makes extensive use of the Developer's Workbench, beginning with a getting started guide in the next chapter.

Programming Frameworks

The IXA SDK 2.0 ships with two programming frameworks: a set of reference designs and the IXA Active Computing Engine (ACE){ XE "active computing engine (ACE)" } model.

The reference designs represent complete microcode and microengine C source for several typical network applications, including IP version 4 routing design, Ethernet bridging, and ATM adaptation layer 5 (AAL5)

segmentation and reassembly (SAR). The reference designs are extended by directly modifying the source code to include any value-added features desired. Each reference design is highly optimized and tuned to the particular reference application that it represents.

In the second programming framework, ACEs are units of packet processing functionality. Examples of ACEs include IPv4 unicast routing, Ethernet bridging, and network address translation (NAT){ XE "network address translation (NAT)" }. Intel also provides some System ACEs for receiving and transmitting packets. Developers chain ACEs together to form an application, and can also create ACEs of their own. For example, the following graph in Figure 2-9 shows a simple NAT application:



This sample IX-SDK application graph shows five ACEs connected together to perform basic NAT capabilities.

Figure 2-9 A sample NAT application with the IX-SDK

On the IXP12xx series network processors, two types of ACEs are available: regular ACEs and MicroACE{ XE "MicroACE" }s.

Regular ACEs are ACEs that run solely as C or C++ code in the StrongARM core{ XE "StrongARM core" } of the processor.

MicroACEs are ACEs that run with the help of the microengines. The successful MicroACE designs have most of the fast-path packets being processed completely in the microengines, with exception packets, table management, and configuration handled in the StrongARM core portion of the MicroACE. This separation is illustrated in Figure 2-10:



This figure shows the sample NAT application implemented using microACEs. In this example, most of the packets are processed in the microengines, with occasional control packets or packets that need more complicated processing going to the core portion of the microACE. Each packet is processed in a single microengine until right before it is transmitted. Then it is queued for processing by the transmit microACE.

Figure 2-10 A sample NAT application built from microACEs

Using IX-SDK on the IXP12xx decreases the time-to-market for products through a couple of methods. Components from multiple sources can be easily combined into an application like taking the Intel-supplied Ethernet Ingress, Ethernet Egress, and IPv4 Unicast Routing ACEs and adding a custom built load-balancing ACE to build a load-balancing application with Ethernet ports. In addition, the extensive core infrastructure offered by IX-SDK MicroACE framework means utilities are available to help with resource allocation, configuration, and inter-ACE communication. This additional infrastructure can significantly speed development.

SUMMARY

Each processor in the IXP12xx family of processors has seven processing units: the StrongARM core and six microengines. The core can be programmed with common tools, standard languages, and can run standard operating systems. The microengines are special-purpose RISC processors with an instruction set tuned to bit, byte, and long-word manipulations of data. On the IXP1240 and IXP1250, the instruction set also includes support for calculating CRCs.

Each microengine{ XE "microengine" } supports four threads of execution. The context switch time between threads on a microengine is minimal because the hardware maintains an equal portion of the register pool for each thread. In addition, the register set is partitioned into GPRs and transfer registers. Separate transfer registers allow the microengines to continue to compute with GPRs while memory and other functional units are accessed through transfer registers.

The IXP12xx provides access to external SRAM memory, SDRAM memory and the external PCI bus through the SRAM, SDRAM, and PCI functional units, respectively. The core and microengines can access external memory by issuing read and write commands to the SRAM and SDRAM units. SRAM memory is smaller and has lower latency than SDRAM memory. Additionally, the SRAM unit supports special commands, such as locks, atomic push and pop linked lists, and atomic bit-test-and-set. The SDRAM unit supports direct RFIFO and TFIFO data transfers. The PCI interface allows the core and microengines to initiate DMA transfers across the external PCI bus.

The FBI houses a hashing unit, scratchpad memory, system-wide CSRs, and the ready-bus sequencer. But, the FBI primarily provides an interface to the external data bus, the IX bus. The IX bus carries the high-speed packet data from external MAC devices from and to the IXP12xx. The IX bus requires packet data to be segmented into 64-byte chunks called mpackets. When receiving a packet, individual mpackets are moved from the IX bus MAC's into RFIFOs within the FBI. From there, the RFIFO data can be moved into transfer registers in the microengines, or directly into SDRAM memory. When transmitting packets, the microengines place mpackets into TFIFOs and validate the TFIFO. Subsequently, the FBI's transmit state machine will move the validated TFIFO data to the external MAC device via the IX bus.

The software tools provided in the IXA SDK 2.0 include the microengine C compiler, microcode assembler, the Developer's Workbench{ XE "tools and platforms:Developer's Workbench" } IDE, a set of reference designs for various network applications, and the ACE framework. The Developer's Workbench allows you to create, edit, simulate, and debug microengine code. The microengine C compiler and microcode assembler are integrated with the Developer's Workbench.

The IXA SDK 2.0 includes two software frameworks: a set of reference designs and the ACE model. The reference designs are working examples of typical network applications. Reference designs are intended for direct modification. The ACE model decomposes network applications in a pipeline of computational stages. Each stage, or ACE, can be chained together to create a complete network packet processing application. The ACE framework provides a model in which customers can add new ACEs and have these interact with Intel's supplied ACEs.

FURTHER READING

If you feel that you need to know more about the hardware as it relates to software, or if you just can't get enough information about the number of clocks it takes to flush the pipeline in an aborted class 3 conditional branch, then get yourself the *IXP1200 Network Processor Hardware Reference Manual* available on the accompanying CD-ROM. The HRM is a reference for all of the hardware details described in this book and a lot more, but is not necessary to understand the rest of this book.

Additionally the *IXP1200 Programmer's Reference Manual*, also available on the accompanying CD-ROM, is a great programming reference. Chapter 3 of the *IXP1200 Programmer's Reference Manual* contains the complete instruction set for the microengines and Chapter 4 details all of the register descriptions.

