Intel® IXP1200 Network Processor Family

Hardware Reference Manual

December 2001

Part Number: 278303-009

Revision History

Revision Date	Revision	Description
8/30/99	001	Beta 1 release.
10/29/99	002	Beta 3 release.
3/2/00	003	Beta 4 release.
6/5/00	004	Version 1.0 release.
9/27/00	005	Version 1.1 release.
12/20/00	006	Version 1.2 release.
6/1/01	007	Version 1.3 SDK release.
8/10/01	008	Version 2.0 SDK release. Miscellaneous changes.
12/07/01	009	Version 2.01 SDK release. Miscellaneous changes.

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The IXP1200 may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

Contents

Figures

Tables

Introduction 1

1.1 About this Document

This document serves as the hardware reference manual for the Intel® IXP1200 Network Processor Family. This includes the IXP1200, IXP1240 and IXP1250. This book is intended for use by Developers and is organized as follows:

Section 2, Technical Introduction contains a hardware overview.

Section 3, StrongARM* Core describes the operation of the embedded StrongARM* core.

Section 4, Microengines describes the operation of the Microengines.

Section 5, PCI Unit describes the operation of the PCI Unit.

Section 6, FBI Unit describes the operation of the IX Bus Interface Unit.

Section 7, SDRAM Unit describes the operation of the SDRAM Unit.

Section 8, SRAM Unit describes the operation of the SRAM Unit.

1.2 Related Documentation

Further information on the IXP1200 is available in the following documents:

IXP1200 Network Processor Datasheet - Contains summary information on the IXP1200 including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

IXP1240 Network Processor Datasheet - Contains summary information on the IXP1240 including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

IXP1250 Network Processor Datasheet - Contains summary information on the IXP1250 including a functional description, signal descriptions, electrical specifications, and mechanical specifications.

IXP1200 Network Processor Family Microcode Programmer's Reference Manual - Contains detailed programming information for designers.

IXP1200 Network Processor Development Tools User's Guide - Describes the IXP1200 Workbench and the development tools you can access through the use of the Workbench.

IXP1200 Network Processor Family Microcode Software Reference Manual - Contains detailed software technical information for designers.

ARM Architecture Reference Manual.* Available from ARM. Contact: ARM Limited, 985 University Ave., Suite 5, Los Gatos, CA 95030. Phone: 408-399-5199. FAX: 408-399-8854. Email: info@arm.com. WWW: http://www.arm.com.

1.3 Conventions

The following conventions are used in this manual:

1.3.1 Data Terminology

Table 1-1. Data Terminology

Term	Words	Bytes	Bits
Byte	½	1	8
Word	1	2	16
Longword	2	4	32
Quadword	4	8	64

1.3.2 Definitions

MPKT

The data read from a MAC device receive FIFO as the result of a single receive request to the receive state machine. Read in as 64 bytes.

Packet

The data framed between the assertion of an SOP signal and assertion of its associated EOP signal.

Technical Introduction 2

This chapter is a relatively detailed introduction to the IXP1200 microarchitecture, and it's key features. It serves as a base from which the user can better understand the more detailed chapters that follow.

2.1 Overview

The IXP1200 is an integrated Network Processor, comprised of a single StrongARM processor, six *Microengines*, standard memory interfaces, and high-speed bus interfaces. It is targeted at networking applications requiring a high degree of flexibility, programmability, scalability, performance, and low power consumption. The unique architecture of the IXP1200 affords the user a highly concurrent packet processing model, while keeping the programming model simple. This is accomplished by providing many features in hardware that simplify the programming model.

The IXP1200 allows the designer to implement in software, what was previously implemented in custom ASICs. This flexible, reprogrammable approach has many advantages, including faster development time, easier bug-fixes, the ability to add features to products after deployment in the field, and the ability to conform to standards that are not yet solidified.

The Microengines are custom processors implemented specifically for networking applications. They are especially well suited to high-speed data manipulation and movement. The Microengines being fully programmable processors, are able to examine packet contents at all levels of the networking stack. This makes them suitable not only for layer 2 and 3 switching/forwarding, but also for applications that require deeper inspection and manipulation of packet contents.

The IXP1200 has a number of noteworthy features that are described in more detail in Section 2.3, "Key Architectural Features:

- Multi-Processing - allows multiple network packets to be processed in parallel.
- Distributed Data Storage architecture - positions data close to where it's needed for faster access, - in the Microengines.
- Hardware Multi-Threading - allows each Microengine to process multiple packets with minimal context switching overhead.
- Active Memory Optimizations - executes a series of memory requests in the most efficient manner, thereby increasing effective memory bandwidth.
- Multi-level Concurrency - allows multiple packets to be processed simultaneously by overlapping the data accesses required to process one packet, with the compute cycles required to process another packet.
- Block Data Movement - efficient movement of large amounts of data.
- Scalability - allows the architecture to scale with increasing processing demands.

2.2 IXP1200 Functional Units

Figure 2-1. IXP1200 Block Diagram

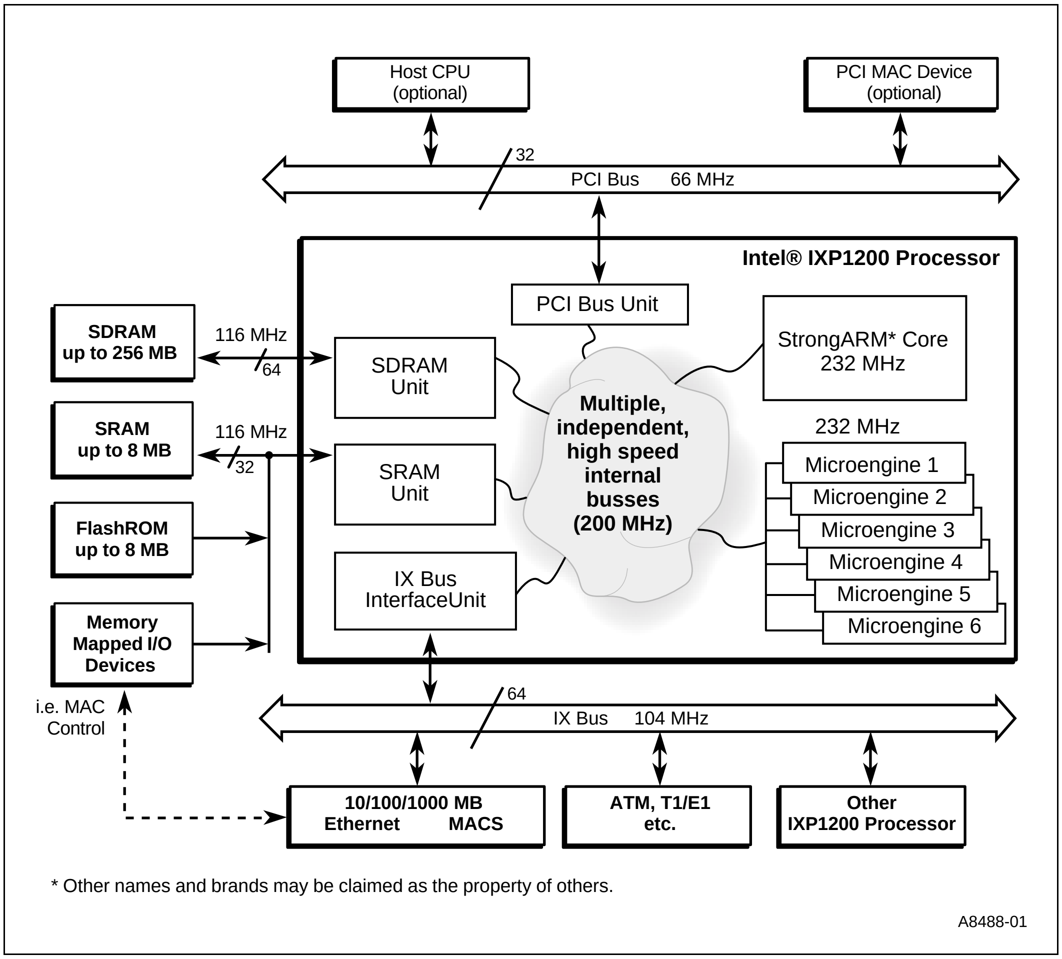


Figure 2-1 is a simplified block diagram of the IXP1200 which shows the six main functional units. These functional units are described briefly below, and are described in more detail in the following sections. The internal busses are shown as a “cloud” here for the sake of simplicity. The many internal busses that connect the various functional units are described in more detail in Section 2.7, “Internal Architecture.

- StrongARM* core - A full 32-bit RISC processor core with integrated caches that can be used for management functions, running routing protocols, exception handling, and other tasks.
- Six Microengines - These compact, efficient RISC engines can be used for any function requiring high-speed packet inspection, data manipulation, and data transfer. These are fully programmable 32-bit engines with a 5-stage execution pipeline and a large (256) register set. Hardware multithreading, and context-sensitive register windows enable very fast context switching.
- SDRAM Unit - A shared, intelligent memory interface that can be accessed by the StrongARM* core, the Microengines, and devices on the PCI bus. A glue-less interface to

standard, low-cost SDRAMs. Capable of moving blocks of data between the SDRAM and the Microengines or the IX Bus Unit or devices on the PCI bus.

- SRAM Unit - A shared, intelligent memory interface that is accessible by the StrongARM and the Microengines. A glue-less interface to standard pipelined and flowthrough SRAMs. Capable of moving blocks of data between the SRAM and the Microengines
- PCI Bus Interface Unit - A standard interface that may be used to interface to other PCI devices, or another host processor.
- IX Bus Unit - An intelligent data movement engine controlled by the Microengines. Capable of transferring blocks of data between the IXP1200 and networking devices such as MACs and SARs. While the IX Bus Unit can perform many sophisticated data transfer operations internal to the IXP1200, the (external) IX Bus Interface itself is simple, and it is straightforward to design peripherals for it. The IX Bus is described in more detail in Section 2.7.6.

Other functions contained within the IXP1200 include:

- A hardware hashing unit (48 or 64 bit)
- An on-chip 4KByte Scratchpad RAM for storing globally accessible data, and passing messages between processors and between *threads*.
- On-chip Transmit and Receive FIFOs for buffering data on the IX Bus.
- Connecting the different functional units within the IXP1200 are a series of high speed busses that transfer data between the units. These are parallel data paths that operate independently of each other and increase the data throughput within the network processor.
- The IXP1200 also contains a UART for debug purposes; a number of counters that can be used for time-stamping, etc; and a number of features for inter-processor and inter-thread communication.

The IX Bus Interface is **the** high speed data flow interface to the IXP1200. Whereas the PCI interface is fast, and runs at up to 66 MHz, it is limited by the PCI bus protocol, and by the bus width. This does not preclude the PCI bus from being used as the main data path into and out of the IXP1200, but it should be remembered that there is no direct connection (internally) between the PCI Unit and the Microengines.

The IXP1200 has three types of memory resources available to the programmer: SRAM, SDRAM, and Scratchpad RAM. These memories vary in latency, size, and bandwidth. The advantage of having these three types of memories is that each of the memories operate in parallel, and the programmer may make use of each of these in areas suited to their characteristics.

For example, SDRAM memory is typically used for packet data storage and very large tables, whereas SRAM memory is used for table lookups where low latency is very important. Scratchpad RAM being internal to the IXP1200 is smaller, but with very low latency and is most often used for interprocess communication, and shared semaphores or counters.

2.3 Key Architectural Features

- **Multi-Processing** - The six Microengines, plus the StrongARM make up the compute resources of the IXP1200. They share a common set of resources: SDRAM, SRAM, PCI, IX Bus, Scratchpad RAM, and a number of other functions (described later).
- **Distributed Data Storage Architecture** - Each of the Microengines has its own local register file of 256 (32-bit) registers. 128 of these registers are *Transfer Registers*. The Microengines

have a load-store architecture. A Microengine cannot access data external to it. Instead, it must bring the data into its Transfer Registers, operate on it, and then write the data from the Transfer Registers out to its destination (i.e. SDRAM). Once the data is in its Transfer Registers, a Microengine has single-cycle access to it, thus greatly speeding up its processing capabilities. When data is not in its Transfer Registers, a Microengine can issue a Reference Command to fetch the required data, then "go to sleep" while waiting for the data to come back. This is where hardware multi-threading comes into play.

- **Hardware Multi-Threading** - Each of the Microengines actually has four Program Counters, and is designed to be able to support up to four Threads (also called Contexts). When one of the Threads is waiting for some data to be fetched, it can "go to sleep" and allow another Thread within the same Microengine to run. This allows four program threads to run on the same Microengine, thus utilizing the Microengine compute resources in a much more efficient fashion.

  This leads to better silicon utilization, and lower power than an approach in which more Microengines are implemented, but without multi-threading. In a non-multi-threading design, Microengines end up spending a lot of time just waiting for the data to come back. Multi-threading in the IXP1200 is implemented in hardware, but controlled by software. Performing multi-threading in hardware makes zero overhead context switching possible, as described in Section 2.7.2, “Microengines.
- **Active Memory Optimization** - Both the SDRAM and SRAM Units have a number of queues and optimization logic built in. This allows the six Microengines (actually the 24 Microengine threads) as well as the StrongARM to request read or write transfers from the memory units, and allows the memory units to intelligently determine the optimal order in which to carry out these requests. Though these Active Memory Optimizations are implemented in hardware, they are controlled by software on an instruction by instruction basis.
- **Concurrency** - Due to the several independent data and control busses internal to the IXP1200, data can be moved simultaneously:
  - between the SDRAM Unit and the Microengines or IX Bus Unit
    (in both directions: read and write, - simultaneously)
  - between the SRAM Unit and the Microengines
    (in both directions: read and write, - simultaneously)
  - between the SDRAM Unit and the PCI Unit
  - between the IX Bus Unit and the Microengines

  All these can occur simultaneously, and independently of each other. These are under software control, but implemented in hardware, so the programmer need not worry about the details.

  There are additional forms of concurrency in the IXP1200, which are described in more detail in Section 2.4.6.
- **Block Transfers** - Due to the large register set on each of the Microengines, a single instruction can cause up to 64 bytes of data to flow from one functional unit to another, or up to 128 bytes of data to flow across the IX Bus. This makes better use of the Microengine compute resources, as well as reducing code size.
- **Scalability** - The IXP1200 architecture implements scalability in a number of different dimensions. Firstly, the architecture is such, that future members of the IXP1200 family can add additional Microengines, while maintaining the same programming model. Secondly, future IXP network processors can increase the instruction store within the Microengines. Thirdly, multiple IXP1200s can be connected in a single design, allowing increased processing power or data bandwidth.

Connecting the six functional units are a number of very high-speed internal busses. These busses operate at the full core frequency (currently 232 MHz). The busses are detailed in the descriptions of the individual functional units in Section 2.7, “Internal Architecture, however, it should be noted here that there are separate Read and Write busses between some of the functional units allowing simultaneous transfers across these busses. Also, some of the functional units have separate, unshared busses connecting them (i.e. PCI Unit to SDRAM Unit). In addition to these internal busses which significantly improve the performance of the IXP1200, several of the functional units have their own DMA engines (called Push-Pull Engines). These Push-Pull Engines are transparent to the programmer, but serve as hardware DMA Controllers to move blocks of data between the functional units. The combination of multiple independent busses, the Microengines, and the Push-Pull Engines serve to move large amounts of data within the IXP1200 concurrently. Thus, internal data-path bottlenecks are removed. More detail on the functioning of the internal busses, as well as the Push-Pull Engines follows in a later section.

2.4 Some Architectural Concepts

The IXP1200 architecture introduces some new concepts, and also makes use of some existing concepts but uses them in novel ways. This section gives a broad overview of these concepts and how they work together to form the IXP1200 architecture. Understanding these concepts is key to understanding the IXP1200 architecture.

2.4.1 References

Standard microprocessors (such as the integrated StrongARM processor) access memory directly by issuing the memory address, and then waiting for the data to be returned. If the data is not returned immediately, wait states are generated and compute cycles are wasted.

In comparison, the Microengines utilize a form of addressing called *References*. To access data external to a Microengine, a Microengine executes a Reference Command, but does not have to wait for the data to come back. Instead it can do some other useful work by letting another program thread within the Microengine run.

While the Microengines have a set of local registers just like any other microprocessor, they do not have direct access to SDRAM, SRAM, and other resources that are not within the Microengine. A Microengine accesses its local registers in one clock cycle, however, to access resources that are not local to it, the Microengine issues Reference Commands (to bring data into the local registers).

Reference Commands are used to:

- request a read or write from/to SDRAM
- request a read or write from/to SRAM
- request a read or write from/to on-chip Scratchpad RAM
- request a hashing operation to be performed on data in the transfer registers
- request data to be transferred over the IX Bus interface (to/from a device on the IX Bus)
- request data to be moved between SDRAM and the IX Bus Unit
- request data to be transferred between Microengine transfer registers and the IX Bus Unit

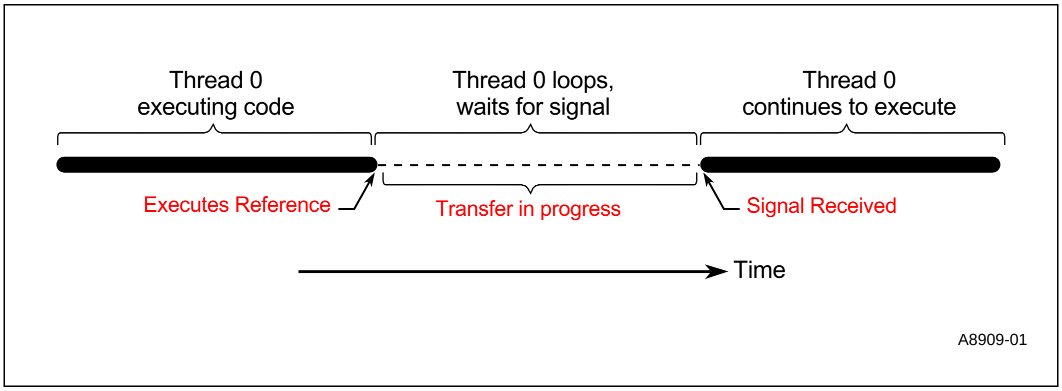
In general, anything that takes time to complete, is implemented through a Reference Command, and the Microengine need not wait for the transfer to complete.

2.4.2 Signals and Synchronization

Since a Microengine can issue a Reference Command but does not need to wait for the requested action to be completed, some means must be provided so that the Microengine may know when the requested task has been completed. To accomplish this, the different functional units (i.e. SRAM Unit, SDRAM Unit, etc.) have the ability to send signals back to the requesting Microengine indicating when the requested task has been completed.

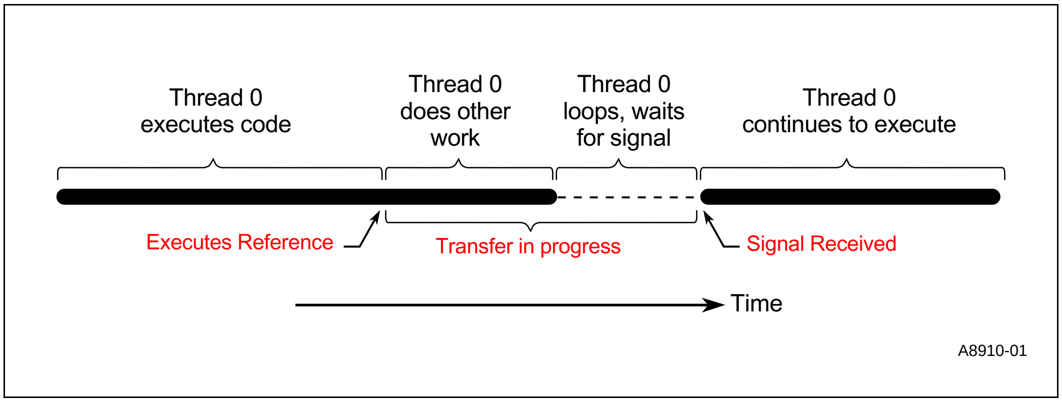
A Microengine can issue a Reference Command, and simply wait for the signal to be returned as shown in Figure 2-2.

Figure 2-2. Thread waiting for signal



Or a Microengine can continue to execute instructions while the transfer is in progress and do some other useful work, and then later check to see if the signal has been received (shown in Figure 2-3).

Figure 2-3. Thread synchronizing with signal



Alternatively, the Microengine *Thread* can issue a Reference Command, and then *swap out* as described in the next section, and illustrated in Figure 2-4 “Context Swapping".

Signals are the primary method provided within the architecture to allow Microengine Threads to gain notification that events external to the Microengine have occurred. Signals are used not only to signify the completion of Reference Commands, but also for inter-processor and inter-thread communication. For example, threads may signal each other to indicate an event, or the StrongARM* core may signal any of the Microengine threads.

Signals are not preemptive. They serve only to notify a thread of an event, or to enable a thread that is waiting for a signal.

2.4.3 Context Swapping and Threads

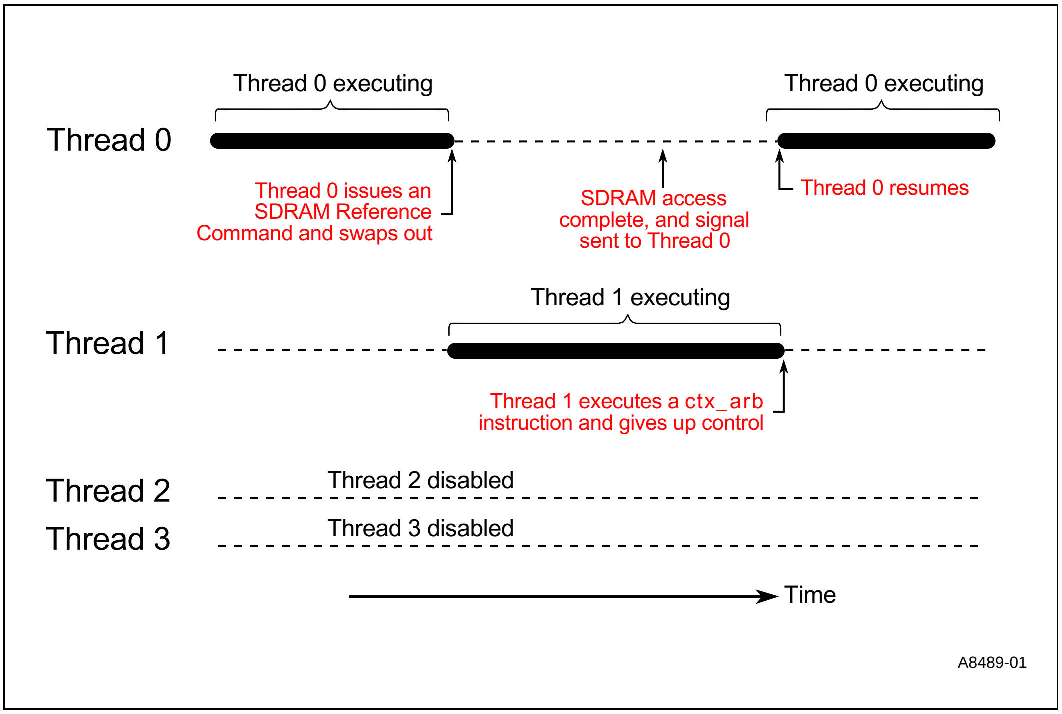
The Microengines utilize a feature called *Hardware Multithreading*. Each Microengine has four Program Counters, and supports four *Threads*. Each thread has its own Thread ID, and the IXP1200 hardware (i.e. the various functional units) are ‘thread-aware’. When a Microengine Thread issues a Reference Command, the receiving functional unit (i.e. SDRAM Unit) recognizes which thread issued that command, and can send a completion signal specifically back to that thread. In this way, multiple threads within a Microengine can issue Reference Commands, and the completion signals are sent back to the specific threads that issued the commands in the first place. This is all done by the hardware, so the programmer need not worry about keeping track of Reference Commands, and the IXP1200 programming model remains simple, and easy to use.

As mentioned above, a Microengine (actually, a Microengine thread) can issue a Reference Command, and then swap out, allowing another thread (within the same Microengine) to run. In this way, while one Microengine thread is waiting for data, or some operation to complete, another thread is allowed to run and complete some useful work (Figure 2-4). Since the functional units are thread aware, when a particular Reference Command is complete, the specific thread that issued the command is signalled, enabling it to ‘wake up’ and continue to run, since the data it was waiting for is now available

It should be noted that a thread does not simply “wake up” whenever it receives a signal. The Microengine architecture is not interrupt driven, and so signals *do not preempt* a thread that is already executing. Context switching occurs only when an executing thread *explicitly* gives up control. The ctx_arb Microengine instruction allows the context arbiter to determine which thread runs next. In addition, the ctx_arb option on Reference Commands issues the Reference Command and then allows context arbitration to take place. Context arbitration does not occur at any other time.

A thread that is swapped out and waiting for a signal is temporarily disabled (for arbitration) until the signal is received. The Microengine Context Arbiter is described in more detail in Microengine Section.

Figure 2-4. Context Swapping


Thread 0 executing
Thread 0 executing
Thread 0
Thread 0 issues an SDRAM Reference Command and swaps out
SDRAM access complete, and signal sent to Thread 0
Thread 0 resumes
Thread 1 executing
Thread 1
Thread 1 executes a ctx_arb instruction and gives up control
Thread 2
Thread 2 disabled
Thread 3
Thread 3 disabled
Time
A8489-01

2.4.4 Some Examples

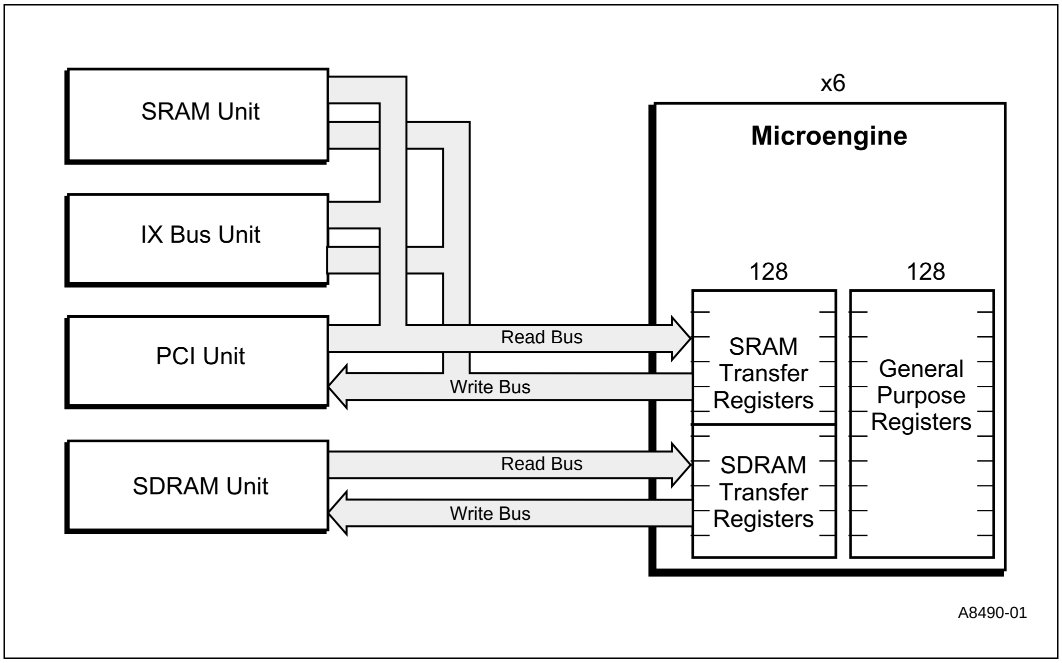
As an example, say Thread A (Microengine #1, Thread 2) is processing a packet header by doing a lookup in SRAM. Instead of waiting for the data to come back from SRAM, Thread A can continue to execute instructions, and begin to store the packet payload data to SDRAM. Later, when necessary, it can check to see if a signal has been received from the SRAM Unit so that it knows that the SRAM operation has been completed, and can then continue to process the header.

Alternatively, as an example that shows context swapping, say Thread A is doing a lookup in SRAM, and due to the nature of the algorithm, cannot continue processing till the data is available. In this case, Thread A would issue the SRAM Reference Command and swap out (all in one instruction). At this point, Thread B (Microengine #1, Thread 3) would execute its program, probably working on a different packet than Thread A. Later, when the SRAM Reference Command issued by Thread A completes, the SRAM Unit would send Thread A a signal, enabling it to ‘wake up’ and continue to execute its code where it left off. It should be noted that Thread A cannot begin executing code again until Thread B relinquishes control (swaps out explicitly).

Again, this is all accomplished in the hardware so that the programmer does not have to keep track of all the signals and Reference Commands that have been issued. The programmer treats each thread as a logically separate entity, simplifying the IXP1200 programming model. The result, however, is that the six Microengines, each with four logically separate threads, can be effectively executing 24 different programs. It is even possible (though not likely from a practical, implementation point of view) that each of the 24 threads can be working on a separate packet.

2.4.5 Local Data Storage and Block Transfers

Figure 2-5. Microengine Transfer Registers and internal data paths



As shown in Figure 2-5 each of the Microengines contains 128 *Transfer Registers*. 64 of these are for transferring data between the Microengine and SDRAM (called SDRAM Transfer Registers), while the other 64 are for transferring data between the Microengine and the other functional units (called SRAM Transfer Registers). Though the SRAM Transfer Registers are named so, they transfer data between the Microengine and all the functional units (except SDRAM).

Since the Microengines have this large local data transfer register area, one Reference Command can cause a block of data to be loaded into one portion of the transfer register area, while another Reference Command (say from a different thread in the same Microengine) can cause a block of data to be loaded into a different portion of the transfer register area, - without overwriting each other. As shown in Figure 2-5, these data transfers can happen simultaneously because there are four separate and independent data busses that connect the Microengines to the different functional units. These bus structures are described in more detail below. Another feature of this architecture is the ability to move large blocks of data with a single instruction. For example, a single Reference Command instruction from a Microengine can cause a block of data to (up to 64 bytes) to be read from, or written to the SRAM or SDRAM.

This large data transfer register area is shared by all four of the Microengine threads. Just as the IXP1200 functional units are thread-aware, so too are the transfer registers. In this way, when different threads within the same Microengine issue Reference Commands, the data requested by one thread does not overwrite the data requested by another thread. Each thread (context) has it’s own window of registers (one fourth of the total registers). This helps to keep the IXP1200 programming model simple and easy to use.

2.4.6 Concurrency

Concurrency within the IXP1200 architecture takes a number of different forms:

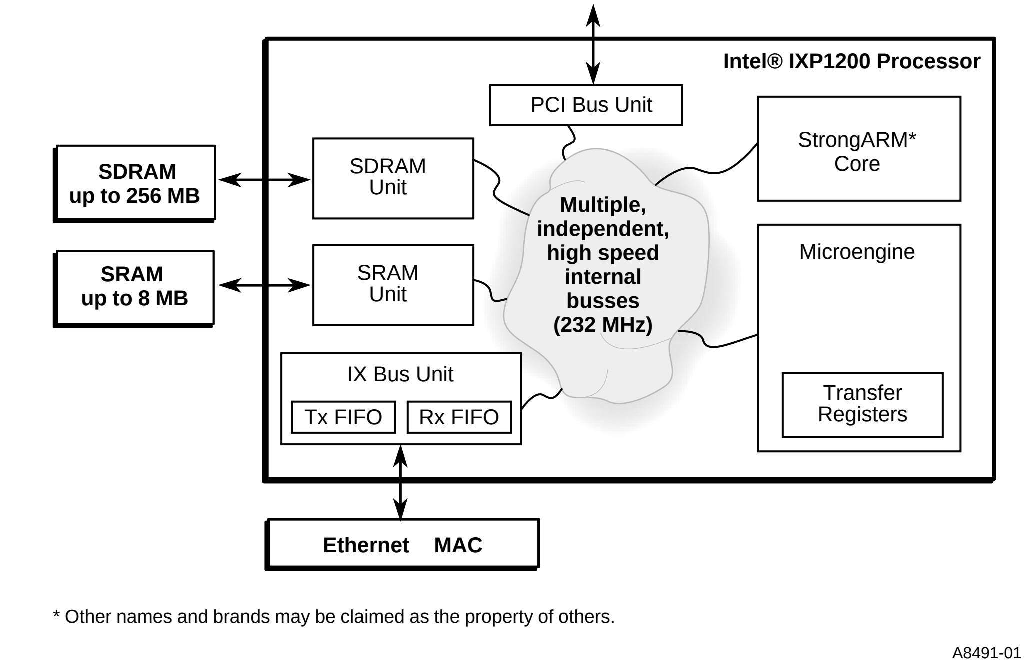
Of course, there is the concurrency of seven processors within the IXP1200. But beyond that, the architecture is such that there is a lot of concurrency at the individual Microengine level as well. While one thread within a Microengine is executing code, data transfers on behalf of any (or all) four threads can be occurring. In addition, since internal to the IXP1200 there are separate busses to the SDRAM and the SRAM, data transfers to both devices can be occurring in parallel. Furthermore, there are separate Read and Write busses for the SRAM and SDRAM as can be seen in Figure 2-5. This allows the Microengine Transfer Registers to transfer data to the SRAM and SDRAM in both directions (read and write) simultaneously.

Similarly, it is also possible for the other functional units to be executing tasks on behalf of the Microengines simultaneously. This can be happening while a particular thread is ‘sleeping’. A good example of this would be when a Microengine thread executes a Hashing Reference Command and then swaps out. When the swap occurs, another thread within the same Microengine can execute code, probably processing a separate packet. Meanwhile, at the same time the Hashing Unit will be executing the hash on behalf of the sleeping thread.

2.5 Typical Packet Data Flow

What follows is a simplified description of how a packet would be processed in a ‘typical’ application. The IXP1200 architecture is discussed in much more detail in further sections, but the description immediately below serves as a general introduction to acquaint the designer with the general flow of data through the IXP1200. An Ethernet application is used for illustration.

Figure 2-6. Typical Packet Data Flow in the IXP1200 (Simplified)



Generally, packet data moves through the IXP1200 in the following steps:

1. The Ethernet MAC receives data.
2. A MicroEngine instructs the IX Bus Unit (via a Reference Command) to get the data from the MAC and store it in the Receive FIFO (RFIFO). The IX Bus unit takes care of the actual transfer, independent of the Microengine.
3. The IX Bus Unit signals a Microengine thread that the data has been transferred.
4. The Microengine instructs the SDRAM Unit to transfer the data from the Receive FIFO to SDRAM (this actually takes place in the background). The SDRAM unit takes care of the actual transfer, independent of the Microengine.
5. The Microengine transfers the first few bytes (header) into its Transfer Registers (via a Reference Command to the IX Bus Unit).
6. It processes the header (or however much of the packet needs to be inspected), and determines what to do. It can make use of tables stored in SRAM (or SDRAM) to do lookups.
7. It modifies the packet header, if necessary, and writes out the new header to SDRAM.
8. It instructs the SDRAM Unit to write the packet data out to the Transmit FIFO. The SDRAM unit takes care of the actual transfer, independent of the Microengine. When the transfer is complete, the SDRAM Unit notifies the Microengine thread.
9. The Microengine instructs the IX Bus Unit to transfer the data to the appropriate MAC.

The IXP1200 is a *store-and-forward* packet processing architecture. When a device on the IX Bus (say, an Ethernet MAC) has data it needs to transfer to the IXP1200, it asserts a pin. One or more of the Microengines monitor these pins on a continuous basis. When it is noticed that a device on the IX Bus has data, a Microengine issues a Reference Command requesting the IX Bus Unit to transfer data from that device into the Receive FIFOs within the IX Bus Unit (Figure 2-6).

The IX Bus Transmit and Receive FIFOs serve as temporary buffer areas where IX Bus data is stored. When the data transfer is complete and the received data from the MAC now resides in the Receive FIFO, the IX Bus unit sends a signal to a Microengine thread. The thread may then transfer a number of bytes from the Receive FIFO (presumably the packet header) into its local Transfer Registers. It can also send a separate request to the IX Bus Unit to send the entire “chunk” of received data to SDRAM. While the IX Bus Unit is transferring the data to SDRAM, the Microengine can be processing the packet header. It can do table lookups in SRAM, perform hashing (using the hardware hashing unit) on the header fields, etc.

The IX Bus deals with data in chunks of 64 bytes called *MPKTS* (“m-packets”). For larger packets (larger than 64 bytes), as additional portions of the packet are transferred from the MAC into the Receive FIFO, the Microengine issues additional Reference Commands to have the data moved from the Receive FIFO directly to SDRAM, thereby buffering the packet in SDRAM. The Microengine needs to issue only one or two instructions to cause a whole block (1 or 2 MPKTs) of data to move directly between the IX Bus Unit and SDRAM. The Microengine is in complete control of each of these operations.

When the Microengine has finished processing the packet header, it writes the new packet header into SDRAM. Then, it monitors the devices on the Transmit (destination) side, and when the destination device is able to receive data from the IXP1200, the Microengine issues a Reference Command to the SDRAM Unit requesting that a chunk of data be moved from SDRAM to the Transmit FIFO. Now that the data to be transmitted is in the Transmit FIFO, a separate Reference command from the Microengine requests the IX Bus Unit to transmit the data to the appropriate device on the IX Bus.

In Figure 2-6 it may be noticed that data is not shown flowing into and out of SRAM. It is possible to use SRAM as a data buffer area, however the architecture is such that the SDRAM has greater bandwidth. Also there is a direct data path between the SDRAM and the IX Bus Unit. There is no direct path between the IX Bus and the SRAM Unit. The SRAM is intended more for packet descriptor tables and other lookup tables where low latency access is required.

While in the above description, it would seem as if the *same* Microengine (or Microengine thread) does all of these tasks, it is not necessarily so. Different Microengines or different threads could be sharing these tasks, depending on how the system is architected. Also, in the above description it may seem as if the Microengine (or Microengine thread) has to wait for each of the operations to complete, it is not necessarily so. When a Reference Command is issued, the Microengine thread does not need to wait for completion. It can either continue to do other useful work, or swap out and let another thread run. This tends to make for much higher utilization of the compute resources within the Microengine.

2.6 External Interfaces

The SDRAM interface is a glueless interface supporting up to 256 Mbytes of standard SDRAM with a 64-bit data path. The intent is for the SDRAM to hold any large data tables, such as routing tables. The SRAM interface supports either pipelined, or flow-through SRAMs with a 32-bit data path, and up to 8 Mbytes of memory. The intent is for the SRAM to hold any buffer management (i.e. table pointers, etc.) data that require low latency accesses. The SRAM and SDRAM interfaces run concurrently, and independently of each other, thereby allowing the designer to spread the burden of memory accesses between both types of devices, so neither one becomes a bottleneck.

Both SDRAM and SRAM interfaces run at half the core clock frequency, so while the part internally runs at 232 MHz, the SRAM and SDRAM run at 116 MHz. The IXP1200 was architected to use standard SRAM and SDRAM to reduce system cost, while at the same time optimizing these interfaces to get maximum utilization out of these memories. These optimizations are described in further detail in Section 2.7.3, “SRAM Unit, and Section 2.7.4, “SDRAM Unit. At 116 MHz, the SDRAM interface has a maximum bandwidth of 928 MBytes/sec, while the SRAM interface has a maximum bandwidth of 464 MBytes/sec. Whereas the SDRAM interface offers a higher bandwidth, the SRAM interface offers a significantly lower latency. A properly architected design will take advantage of the features of both types of memory.

The PCI interface is compliant to the PCI 2.1 Specification. This is a 32-bit PCI interface that can run at 33 MHz with the standard number of PCI loads, or at up to 66 MHz in a point-to-point configuration. Additional loading of the PCI bus is possible by de-rating of the PCI clock speed. The PCI interface is intended to be used for communication between the IXP1200 and any PCI peripheral (such as a local Ethernet MAC), or a separate host processor. If used with a host processor, it is intended for the IXP1200 to perform all data movement/processing tasks, while the host processor is responsible for management tasks. The PCI interface allows maximum data rates of 264 MBytes/sec at 66 MHz. Whereas the IXP1200 PCI bus does not have as high a bandwidth as the IX Bus, it can still be used as the major data path in less demanding applications.

The IX Bus is an Intel proprietary bus, and is the main data path to and from the IXP1200. While the bus is proprietary, no licensing is required, and Intel encourages its customers and third party silicon vendors to design devices that interface to this bus. An IX Bus Design Guide, IX Bus Functional Model, and IBIS models are available. The IX Bus is very similar to a bus employed by Seeq* and Lucent* network devices (MACs) as well. Currently, Intel produces two devices that hookup to the IX Bus without any additional glue logic. These is an octal Fast Ethernet (10/100 Mbps) MAC device, and a dual Gigabit Ethernet MAC. Future IX Bus devices are currently in design. More detailed information on the IX Bus can be found in this Manual, and in the *IX Bus Design Guide*.

The IX Data Bus is a 64-bit data bus that runs at up to 104 MHz. At 104 MHz, the IX Bus has a maximum bandwidth of 832 MBytes/sec (6.6 Gbits/sec). The IX Bus can be configured to run in a number of different modes and speeds. These include a 64-bit bidirectional mode in which data can be transferred in only one direction at a time, and a 32-bit unidirectional mode in which data can be simultaneously transferred in both directions, but with half the data bus width.

2.7 Internal Architecture

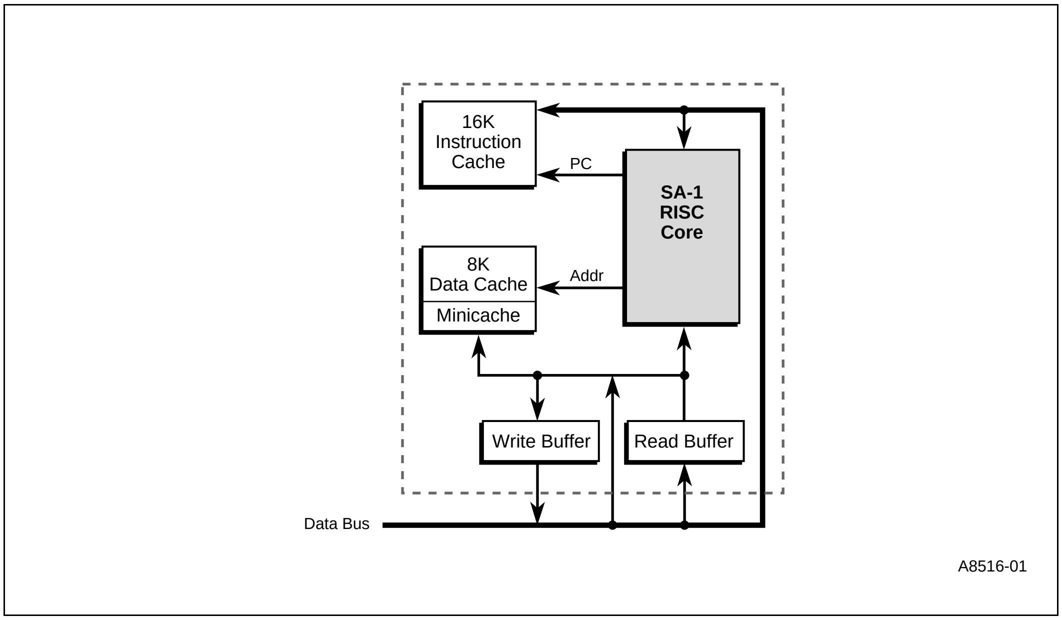
When standard microprocessors (such as the StrongARM, MIPS, Pentium, etc.) execute a memory or I/O request, they have to wait for the data to be returned before they can continue to execute instructions. By comparison, the IXP1200 Microengines do not require the data to be returned before they can continue to execute instructions. The Microengines issue what are called Reference Commands. These Reference Commands are targeted at a specific Functional Unit, and request the Unit to carry out some action (such as fetching data from memory) on behalf of the Microengine. In fact, each Microengine has four Threads, or Contexts, each of them distinct. It is not simply a Microengine that issues a Reference Command, but a specific Thread. Since there are six Microengines, up to 24 Threads can be running. While the targeted Functional Unit carries out the action on behalf of the Microengine Thread, the Microengine can continue to execute instructions. The functional unit sends a signal back to the Microengine Thread to indicate that the task is completed. The Microengine Thread may wait for this signal to be returned (retaining control of the Microengine execution unit), or it may swap out, and allow one of the other three Thread to run instead. This is controlled at the Microengine instruction level, and is described in more detail below. On an instruction by instruction basis, the programmer can decide on whether the Reference Command will cause a swap, or not.

As mentioned above, the Microengines each have 256 32-bit registers. The reason for such a large register set, is that it allows the four Threads to run without interfering with each other. This is because the Microengines implement *Context Relative Addressing* of the registers. When using context relative addressing, the register set is split into 4 equal parts, with each of the Threads able to address only its portion of the register set. In this way, when a Thread swaps out and another Thread runs, there is no need to save state. This allows for *Zero Overhead Context Swaps*, where it is possible for there to be no bubbles in the execution pipeline of the Microengine (the Microengines implement a 5-stage execution pipeline, as described in Section 2.7.2). Thus, frequent swapping of Threads in the Microengines incurs little or no performance penalty. In fact, the architecture of the IXP1200 encourages context swaps as frequently as necessary. In this way, maximum utilization is made of the Microengines while a particular Thread is waiting for something to complete. While other multiprocessor network processor architectures may implement more Microengines, the IXP1200 makes better use of silicon by getting optimal utilization of the Microengines. In the IXP1200 24 separate Threads can be executing completely separate code while sharing common resources.

A Reference Command can cause up to 64 bytes of data to flow between the Microengine transfer registers and a functional unit. For example, if a thread had identified which Virtual Circuit (VC) a packet belongs to, and it needs to examine the QoS parameters for that particular VC, it can fetch them from SDRAM. These parameters are often not just a word or two in length, but may be, say 40 bytes in length. A single Microengine Reference Command can get the SDRAM Unit to transfer all 40 bytes into its transfer registers and signal the thread when it is done. While the Microengine is executing instructions at the core clock frequency (say 232 MHz), the SDRAM clock runs at half the core clock frequency (116 MHz), and takes a number of cycles to complete the request. The thread can swap out and allow another thread to process a different packet, while this transfer is in process.

2.7.1 StrongARM* Core

Figure 2-7. StrongARM Block Diagram



The StrongARM is a full 32-bit RISC microprocessor core running at 232 MHz. It is a custom implementation of the ARM architecture, but with a number of notable features. The ARM architecture has been around since the early nineties, and standard tools for the architecture are available from ARM Limited, the Free Software Foundation (GNU tools), etc. The StrongARM is a 5-stage pipelined processor with 16K bytes of instruction cache (organized as 512 lines of 32 bytes), and 8K bytes of data cache (organized as 256 lines of 32 bytes). In addition, the core contains a 512 byte minicache which is intended for data that is read once, operated on, and then discarded. The intent is for the minicache to reduce flushing of the main data cache. The StrongARM also has a very high MIPS/Watt ratio, making it ideally suited for high performance, low power embedded applications. The ARM architecture is Endian-agnostic in that it can run in either Big-Endian or Little-Endian mode.

Future members of the IXP family of Network Processors may use either the StrongARM* core or the XScale core (the next generation of the StrongARM).

A cacheline fill reads a full 32 bytes, and uses a round robin replacement algorithm. The Data Cache is a write-back cache, and allocates on read, but not on write.

The StrongARM may be used in different ways, depending on the application. If the system already contains a high-level (host) processor, the StrongARM can leave system maintenance to the host processor and act as the exception processor and higher-layer processor for the Microengines. In this case, the StrongARM could run a mini-kernel, and execute routing protocols, while the Microengines would do the fast-path packet processing. The host processor and the IXP1200 communicate over the PCI interface.

In designs where there is no host processor, the StrongARM would be the main processor. It would run a Real Time Operating System, perform maintenance functions, as well as the other functions mentioned above.

The StrongARM Core has access to the resources shown in Table 2-1.

Table 2-1. Summary of StrongARM* Core Internal Connections

Unit	Resource
PCI Unit	Full access to the PCI Bus, including all PCI bus transactions. Full access to PCI Unit registers. Separate, shared 32-bit bus between the StrongARM core (ATU) data bus and the PCI Unit.
SDRAM Unit	Full access to SDRAM.
SRAM Unit	Full access to SRAM, including Flash and other devices hooked up to the SRAM Bus.
IX Bus Unit	Access to Control and Status registers within the IX Bus Unit. Access to Scratchpad RAM within the IX Bus Unit. No access to the Receive or Transmit FIFOs or the IX data bus.
Microengines	Access to Microengines' Program Control Store to program the Microengines. Access to Control and Status Registers, including PC (Program Counter). No access to Microengine Transfer Registers.

The StrongARM has access to the SDRAM, the SRAM, the PCI interface. It does not have access to the IX Bus, as this is the main high-speed data path for the IXP1200, and is better handled by the Microengines. The StrongARM boots from Flash memory over the SRAM interface. It can then load the Microengines with their own programs (the Microengines have their own internal instruction store). The StrongARM communicates with the Microengines via shared registers and a shared on-chip Scratchpad RAM. The StrongARM has the ability to enable or disable any of the Microengines or any of the Microengine threads. The StrongARM can change a Microengine's instruction store at any time by disabling the Microengine, writing to the instruction store, and then re-enabling the Microengine. Though this is a form of downloading the Microengine instruction store "on-the-fly", it is should not be considered "real-time", as downloading the instruction store takes hundreds of clock cycles. The StrongARM can also examine and modify some of the Microengine registers, including the four thread program counters.

The StrongARM may fetch its instructions either from Flash, SDRAM, or SRAM. Typically, the StrongARM would boot from Flash, and then execute from SDRAM. Though the StrongARM and the Microengines share these memory interfaces, the caches on the StrongARM mitigate its affect on Microengine-SDRAM bandwidth.

2.7.2 Microengines

Figure 2-8. Microengine internal structure

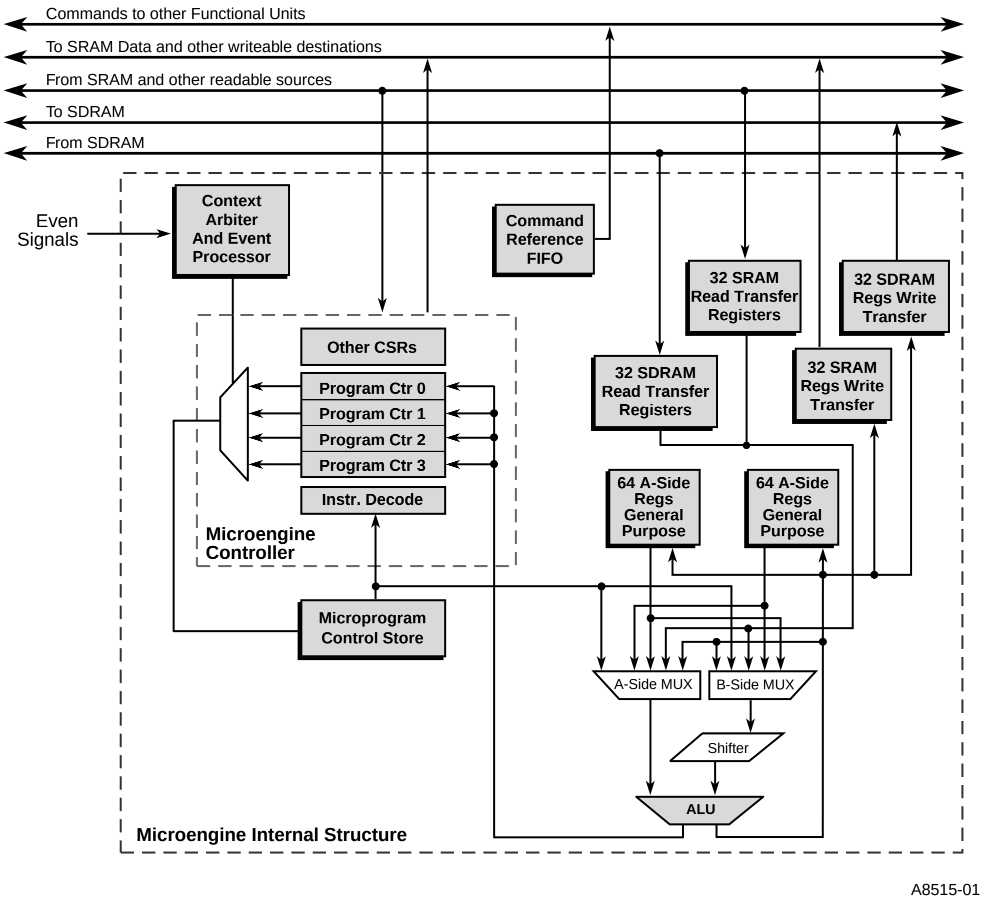

Commands to other Functional Units
To SRAM Data and other writeable destinations
From SRAM and other readable sources
To SDRAM
From SDRAM
Even Signals
Context Arbiter And Event Processor
Command Reference FIFO
32 SRAM Read Transfer Registers
32 SDRAM Regs Write Transfer
32 SDRAM Read Transfer Registers
32 SRAM Regs Write Transfer
Other CSRs
Program Ctr 0
Program Ctr 1
Program Ctr 2
Program Ctr 3
Instr. Decode
Microengine Controller
64 A-Side Regs General Purpose
64 A-Side Regs General Purpose
Microprogram Control Store
A-Side MUX
B-Side MUX
Shifter
ALU
Microengine Internal Structure
A8515-01

Figure 2-8 is a block diagram of a single Microengine. This block is replicated six times in the IXP1200, so each Microengine contains its own instruction store, its own set of 256 registers and other Control/Status Registers (CSRs). All the Microengines are identical, so that functions may be interchanged between them. For example, any Microengine can act as a Scheduler, while a number of other Microengines can do the actual packet processing. There is no fixed function for any of the Microengines, - all are fully programmable.

The Microengines operate at the core clock frequency of the IXP1200, just like the StrongARM (up to 232 MHz). All instructions execute in one clock cycle.

Each Microengine's instruction store is 1K x 32 bits (2K on later steppings of the IXP1200). Since all Microengine instructions are 32-bits in length, a Microengine can contain 1K instructions (2K on later steppings). The Microengines are implemented as 5-stage pipelined processors. In the first stage, the instruction is fetched from the instruction store; in the second stage, the instruction is decoded; in the third stage the operands are fetched from the registers; in the fourth stage the operands proceed through the ALU; and in the fifth stage, the result is written out to the destination register.

The Microengine’s 128 GP (General Purpose) Registers are divided into 2 banks, an A bank and a B bank. This is so that two operands can be fetched simultaneously (in the same clock cycle). The ALU instructions are of the form:

A_register <opcode> **B_register** => **destination_register**

The 128 Transfer Registers are divided into separate SRAM and SDRAM Transfer Registers, and are further divided into Read and Write portions. That is, there are 32 SDRAM Read Transfer Registers, 32 SDRAM Write Transfer Registers, 32 SRAM Read Transfer Registers, and 32 SRAM Write Transfer Registers. Each of these four sets of transfer registers has a separate data path to its respective functional units. Therefore, data can be transferred on all four busses simultaneously. Due to this overlapping of data transfers, as well as Context Switching capability, the IXP1200 is capable of a very high degree of concurrency.

The Microengines can transfer data to and from the other functional units only through these Transfer Registers. As mentioned above, a block of data can be moved to or from the Transfer Registers with one instruction. The read transfer registers are only for data coming from a functional unit, into the Microengine, while the write transfer registers are only for data being written from a Microengine out to one of the functional units. A Microengine cannot read from a write transfer register, and cannot write to a read transfer register.

Therefore, it should be noted that these Transfer Registers are not general purpose registers, rather they act as ports to their respective functional units.

The SDRAM Transfer Registers are used solely for transfer of data to and from the SDRAM unit. However, the SRAM transfer registers are used to transfer data to or from either the SRAM Unit or the IX Bus Unit (i.e. to transfer data to the IX Bus Receive and Transmit FIFOs, the Hash Unit, or the Scratchpad RAM), or the accessible registers within the PCI Unit (DMA Controller registers).

While a Microengine controls the data being transferred to and from its transfer registers, the transfer is actually accomplished by the respective functional unit.

A Microengine has four Program Counters, and therefore up to four threads (also called contexts). Each thread is logically separate from the others, and this is implemented at the hardware level. The four threads can be executing the same code out of the Microengine instruction store, or they can be executing different code. It all depends on where each thread's respective program counter is pointing. For example, it is quite typical to have all four threads of one Microengine executing the same packet processing code (such as IP Verify, Longest Prefix Match, etc.), except that each of the four threads will be operating on a different packet, and will each will have its Program Counter executing at a different point in the program code. By comparison, it is also quite common to have two logically different functions running on the same Microengine. One thread can be controlling the Transmit portion of the IX Bus, while another thread on the same Microengine can be doing the Transmit Scheduling (queuing, etc.). It is completely up to the programmer how the threads are used.

It can be assured that a Microengine thread does not overwrite another thread's registers, as long as the threads use ***context relative addressing***. By using context relative addressing, each of the threads has its own set of General Purpose and Transfer Registers. When a context switch takes place no registers have to be saved, so there is little or no overhead involved in context switching.

Since the Microengines are 5-stage pipelined processors, when a branch instruction is executed, it is possible for instructions following the branch instruction to be aborted in the pipeline. This is fairly typical of pipelined processors. These instructions that follow the branch instruction may have already been decoded, and their operands already fetched. If the branch is taken, processing of these instructions will have to be aborted. To reduce, and at times eliminate, these bubbles in the execution pipeline, the IXP1200 implements deferred execution of instructions. To take advantage of this feature, instructions (up to three) that would normally be placed before the branch instruction, can now be placed after the branch instruction, and a bit-field within the branch instruction indicates that the following instruction(s) should continue to proceed through the execution pipeline. In this way, when the branch is taken and the Microengine has to change its program counter, while it fetches and decodes a new series of instruction, the previous ones (following the branch instruction) continue to be executed. At times, this can reduce the branch penalty to zero. To take advantage of this without increasing the level of complexity for the programmer, the Microengine assembler performs this type of optimization automatically.

In addition, the Microengine instruction set, includes guess branch instructions to further improve instruction pipeline performance. So, if the programmer knows that most times the branch will be taken, the guess-branch-taken instruction can be used (as opposed to the guess-branch-not-taken instruction). When using the guess-branch-taken instruction, when the execution unit decodes this instruction (in stage 3 of the pipeline) it will automatically start fetching the next instruction from the branch destination rather than the next linear instruction.

As far as instruction pipelining is concerned, Context Switching is very similar to executing a branch instruction. Both cause the normal linear fetching of instructions, and execution through the pipeline to be disrupted. So, the Microengines make use of deferred instructions on Context Switches as well. Once again, this reduces Context Switching overhead, making maximum use of Microengine compute power.

Threads also have the ability to use what is called *Absolute Addressing*. Absolute addressing is used when: (a) one or more of the threads requires more than its share of registers, and/or (b) the threads in a Microengine need to communicate with each other and need to use a shared register. By using a shared register, all four threads within a Microengine can communicate with each other without having to go “outside” the Microengine, - very efficient. But by using absolute addressing, one does not give up the benefits of context relative addressing. This is because absolute and context relative addressing are controlled on an instruction by instruction basis. On a given instruction the programmer may indicate whether to use context relative addressing, or absolute addressing. Most often, context relative addressing will be used, thereby keeping things “clean” and logically distinct. It is usually on an exception basis that absolute addressing is used, - when needed.

Context Switching is implemented in a combination of hardware and software. A separate unit within the Microengine, the *Context Arbiter*, continually keeps track of which threads are ready to run. When a context swap occurs, the Context Arbiter is ready to allow the next ready thread to run, - on the next cycle. No cycles are lost in arbitration. The Context Arbiter uses a round robin algorithm to decide which thread to run next, however, any thread that is waiting or disabled, is skipped over. Though the actual context switching is done in hardware, it is controlled by software. A thread will continue to run, until it ***explicitly*** causes itself to be swapped out. An executing thread can cause a context swap to occur by either (a) executing a ctx_swap (context swap) instruction, or by issuing a Reference Command and indicating that it should be swapped out until the Reference Command has completed. A running thread that does not explicitly cause a context swap, will run indefinitely. It is the programmer's responsibility to write his/her code so that this does not happen. The IXP1200 architecture is such, that frequent context swaps are recommended, so that anytime a thread is waiting for a transfer of data, it should swap out.

There is a 64 bit barrel shifter in front of the ALU, allowing an arbitrary shift-and-accumulate to occur within one pass through the execution unit. These hardware features as well as a strong set of bit and byte manipulation instructions allow for a very powerful, and efficient instruction set, particularly suited to networking applications.

The Microengines have access to the resources shown in Table 2-2.

Table 2-2. Summary of Microengine Internal Connections

Unit	Resource
StrongARM Core	The Microengines may interrupt the StrongARM core. The StrongARM core can read a register to determine which Microengine generated the interrupt. No other access to StrongARM Core.
PCI Unit	No access to the PCI Bus. Access only to the Control and Status Registers for the two DMA Controllers in the PCI Unit. By programming these registers, the Microengines may initiate DMA transfers between a block of SDRAM memory and a PCI device.
SDRAM Unit	Full access to SDRAM.
SRAM Unit	Full access to SRAM, including Flash and other devices hooked up to the SRAM Bus.
IX Bus Unit	Full access to the IX Bus Unit, including the Scratchpad RAM, Hardware Hashing Unit, Receive and Transmit FIFOs, Ready Bus, and Control and Status Registers.
Microengines	Each Microengine has access to its Program Control Store so it can execute instructions. The Microengines do not have read or write access to the Control Store (they cannot read or write their own Control Store, only the StrongARM core can do that). Inter-thread signalling between Microengine threads is provided. Each Microengine is self-contained, so one Microengine cannot access the Control Store, or Registers of another Microengine.

The Microengines have full access to SRAM and SDRAM via their transfer registers. The Microengines may make use of the PCI interface, but only by way of the two DMA engines within the PCI Unit. The PCI Unit's DMA Controllers transfer data based on DMA descriptors which are in SDRAM. The StrongARM or the Microengines can construct these DMA descriptors, and then instruct one of the DMA controllers to process the DMA Descriptor. The PCI Unit, and its DMA Descriptors are described in more detail in the PCI Unit section below. The Microengines also have full access to all of the functionality within the IX Bus Unit. The IX Bus is completely controlled by the Microengines, while the StrongARM has only limited access to the IX Bus Unit. The IX-Unit and its features are described in much more detail in the IX Bus Unit section below. The Microengines can interrupt the StrongARM, and leave “messages” for it in a shared area (Scratchpad RAM, SRAM, or SDRAM), but that is the extent to which the Microengines have access or control over the StrongARM processor. Inter-thread signaling and Microengine-StrongARM communication is discussed further in the Inter-process/Thread Synchronization section.

To aid in synchronization, resource-sharing, and inter-process communication, the Microengines implement a number of features. The Microengines have atomic instructions (where the read, modify, and write operations are indivisible, and done without interruption) which can act upon data in SRAM, or in Scratchpad RAM.   The Scratchpad RAM also has an autoincrement (atomic) instruction.

The Microengine instruction set is further detailed in the *IXP1200 Programmer's Reference Manual*.

2.7.2.1 Microengine Data Bandwidth to SRAM Unit and IX Bus Unit

The Microengines make use of a pair of shared busses to transfer data between the Microengines' SRAM Transfer Registers and either the SRAM Unit or the IX Bus Unit (see Figure 2-5 "Microengine Transfer Registers and internal data paths"). (Note that the SDRAM Unit has it's own data paths connecting the SDRAM Unit and the SDRAM Transfer Registers). The Read bus and the Write bus are each 32 bits wide, operate independently of each other, and run at the core clock speed (currently 232 MHz). Data can be transferred to/from the SRAM on odd cycles, while data can be transferred to/from the IX Bus Unit on even cycles.

Table 2-3. Calculation of Internal and External Bandwidths

Calculation based on 232 MHz core clock frequency, 104 MHz IX Bus frequency			
Read/Write	SRAM external bandwidth	32 bits x 116 MHz	3.7 Gbps
Read	SRAM internal bandwidth to Microengine Read Transfer Registers	(32 bits x 232 MHz) / 2	3.7 Gbps
Write	SRAM internal bandwidth to Microengine Write Transfer Registers	(32 bits x 232 MHz) / 2	3.7 Gbps
Read/Write	IX Bus external (pin-side) bandwidth	64 bits x 104 MHz	6.6 Gbps
Read	IX Bus internal bandwidth to Microengine Read Transfer Registers	(32 bits x 232 MHz) / 2	3.7 Gbps
Write	IX Bus internal bandwidth to Microengine Write Transfer Registers	(32 bits x 232 MHz) / 2	3.7 Gbps

As can be see in Table 2-3, since the SRAM internal bandwidth in both the read and write directions is equal to the external bandwidth, there is no internal bottleneck to SRAM.

2.7.3 SRAM Unit

The SRAM is intended to store Lookup Tables, Free Buffer Lists, and Data Buffer Queue Lists. In short, any data that needs to be accessed quickly. While the lookup can be done in SRAM, the results of the lookup can point to larger data structures that are better stored in SDRAM. The SRAM interface is 32 bits wide, and supports either Pipelined SRAMs, or Flow-Through SRAMs. The SRAM data bus width is half that of the SDRAM because it is not intended for bulk data storage, but for fast lookups. The SRAM interface is also intended to be used as a Flash Memory interface (for booting the StrongARM), and a General Purpose I/O Interface (Figure 2-9). Each of these three types of interfaces (SRAM, Flash, General Purpose) can be programmed to have different timing, so that they are not encumbered by the slowest devices sitting on the SRAM interface. The General Purpose Interface can be used to interface to external CAM lookup engines, or to Ethernet MAC access ports (for MAC programming, and accessing MAC manageability registers, - reading RMON and SNMP registers). This type of connection is shown in Figure 2-9.

Figure 2-9. SRAM Unit external interfaces

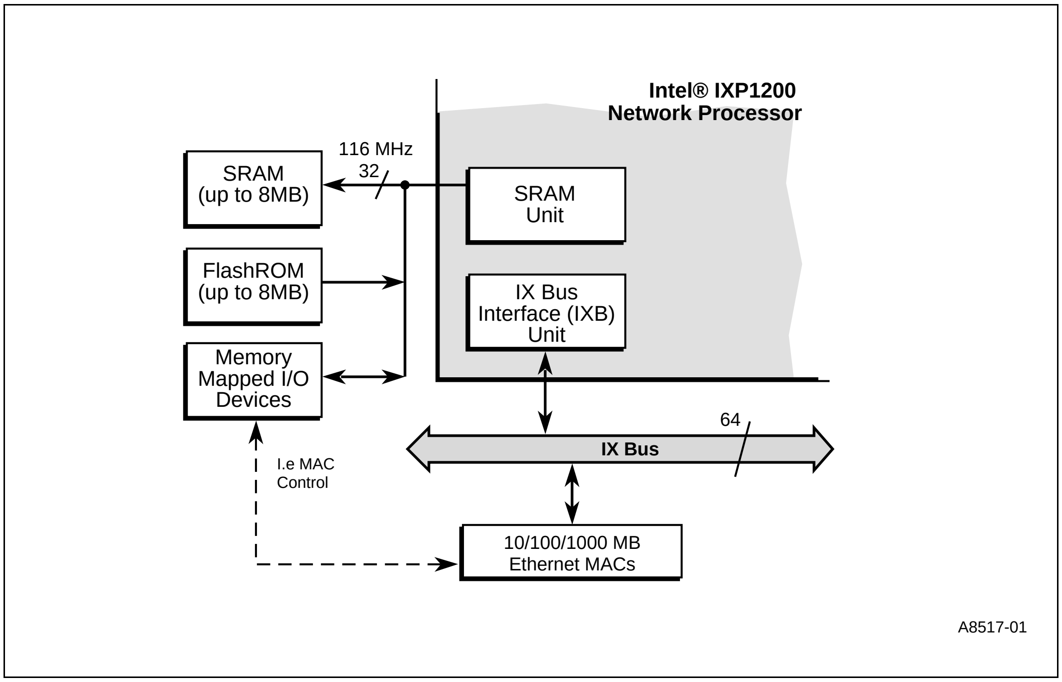


Figure 2-10. SRAM Unit Block Diagram

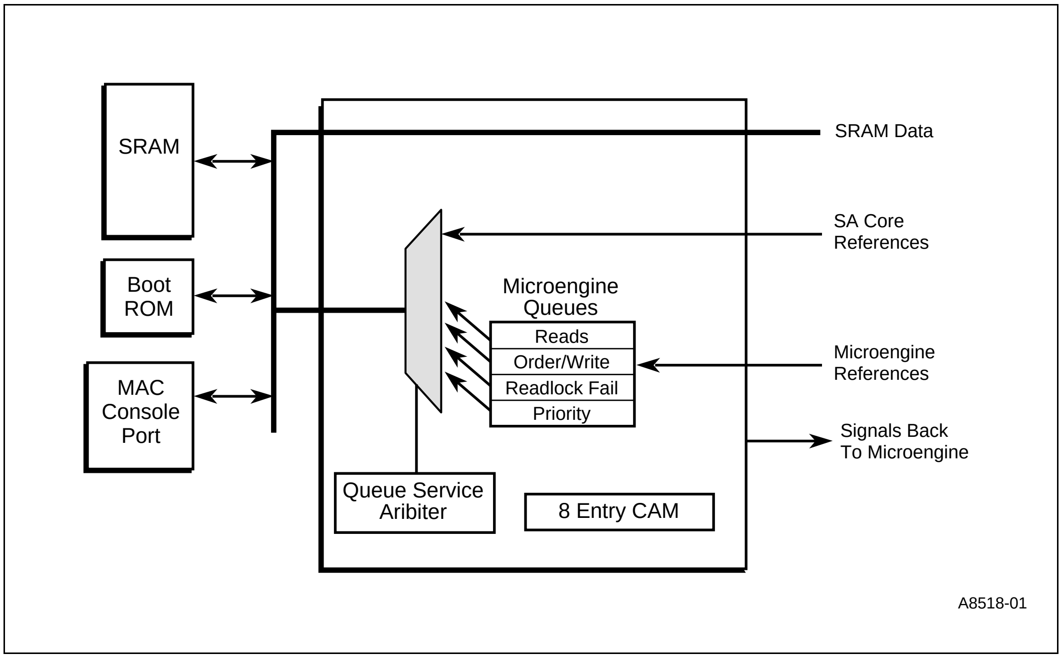


Figure 2-10 is a simplified block diagram of the SRAM Unit. The main function of the SRAM Unit is to take all the SRAM memory references coming from the different Microengine threads as well as the StrongARM, and execute them in as efficient a manner as possible. When a thread issues a Reference Command to the SRAM Unit, it can append an optional token to the end of the instruction. For example,

sram [write,$xfer1,tempa,tempb,4], ***optimize_mem***

This essentially says: “Write the content of the SRAM transfer registers $xfer1 through $xfer4 to the memory locations at the address specified by tempa + tempb. Optimize memory by placing this reference in either the Read or order queue.” The optimize_mem optional token at the end of the instruction instructs the SRAM Unit which queue to put the command reference into. Note that 16 bytes of data are transferred with this instruction (4 transfer registers x 4 bytes each).

The SRAM Queue Arbiter decides on a cycle by cycle basis which SRAM accesses to execute. For example, if the presently executing SRAM cycle is a READ cycle, then the SRAM Unit tries to complete as many READs as possible before going through a bus turnaround cycle and proceeding with any WRITE accesses. By doing this, bus turnaround cycles are avoided and the IXP1200 can get more utilization out of the SRAMs. In practice, this technique manages to squeeze out an additional 18% - 30% of usable SRAM bandwidth.

Since a thread's SRAM references can be “arbitrarily” re-ordered by the SRAM Unit, there must be a way for the programmer to force certain SRAM references to occur in the order in which they are executed. In general, this is not an issue, but under certain circumstances it is necessary for certain accesses to occur in order. For this purpose, an Order/Write queue is provided. If the programmer deems it necessary for certain SRAM references to occur in order, he/she simply appends the *order* token, to those instructions (in place of the *optimize_mem* token used in the earlier example).

A Priority queue is also provided. References in the Priority queue take precedence over any of the references in any of the other queues. To make use of the Priority queue, the *priority* token is appended to the end of the instruction.

The SRAM Unit also includes an 8-entry CAM (Content Addressable Memory) that is used for resource sharing. When using conventional semaphores to restrict access to shared resources, a lot of compute resources are wasted in the continuous polling required to check the semaphore. While the IXP1200 can also use semaphores, the very useful feature of an 8-entry Read Lock CAM has also been implemented. To understand this feature, let us take the simple example of a shared counter (which counts the number of packets processed) that is accessible by all 24 threads. Say one of the threads wants to update the counter. It can execute an SRAM Read Lock instruction indicating a specific SRAM memory location, say location 0x1000. If the Read Lock succeeds, then the thread continues to run, and the instructions following the Read Lock will continue to be executed. Say another thread (which may be within the same Microengine, or within another Microengine) also wants to update the same counter. It will try to take out a Read Lock on the same memory location (0x1000) by executing an SRAM Read Lock instruction with that address. In this case, the CAM recognizes that specific address (0x1000) is within the CAM, and is already locked, and so causes the Read Lock to *fail*. This Read Lock request is now placed in the Read Lock Fail queue, and the thread requesting the Read Lock is now stalled. The next time a Read UNlock Reference Command is executed, the rest of the entries in the Read Lock Fail queue are processed. And when the lock on the address 0x1000 is released, the second thread can continue to run. This feature saves a lot of Microengine cycles since one of the threads is not continuously polling a semaphore, instead it just continues to wait until the read lock is released and the code continues to run. It also saves SRAM cycles, as continuous polling would eat up SRAM bandwidth as well.

It should be noted that the concept of a Read Lock is only logical, in that there is no hardware protection of the SRAM location that is locked (0x1000 in the example above). Instead, as long as each of the threads sharing the resource tries to do a Read Lock before accessing the resource, there is no contention.

Since the StrongARM processor's pipeline stalls when it is waiting for an external access to complete, it is given priority over the Microengines in accessing the SRAM. While this "steals" SRAM cycles from the Microengines, this is completely under the control of the system designer. It is the responsibility of the system designer to ensure that the StrongARM code and the function it is tasked with, do not steal too many SRAM cycles from the Microengines.

The SRAM Unit is connected to the units shown in Table 2-4.

Table 2-4. Summary of SRAM Unit Internal Connections

Unit	Access
StrongARM Core	Full access.
PCI Unit	No connection between the SRAM Unit and the PCI Unit.
SDRAM Unit	No connection between the SRAM Unit and the SDRAM Unit.
IX Bus Unit	No connection between the SRAM Unit and the IX Bus Unit.
Microengines	Full access.

2.7.4 SDRAM Unit

Figure 2-11. SDRAM Unit Block Diagram

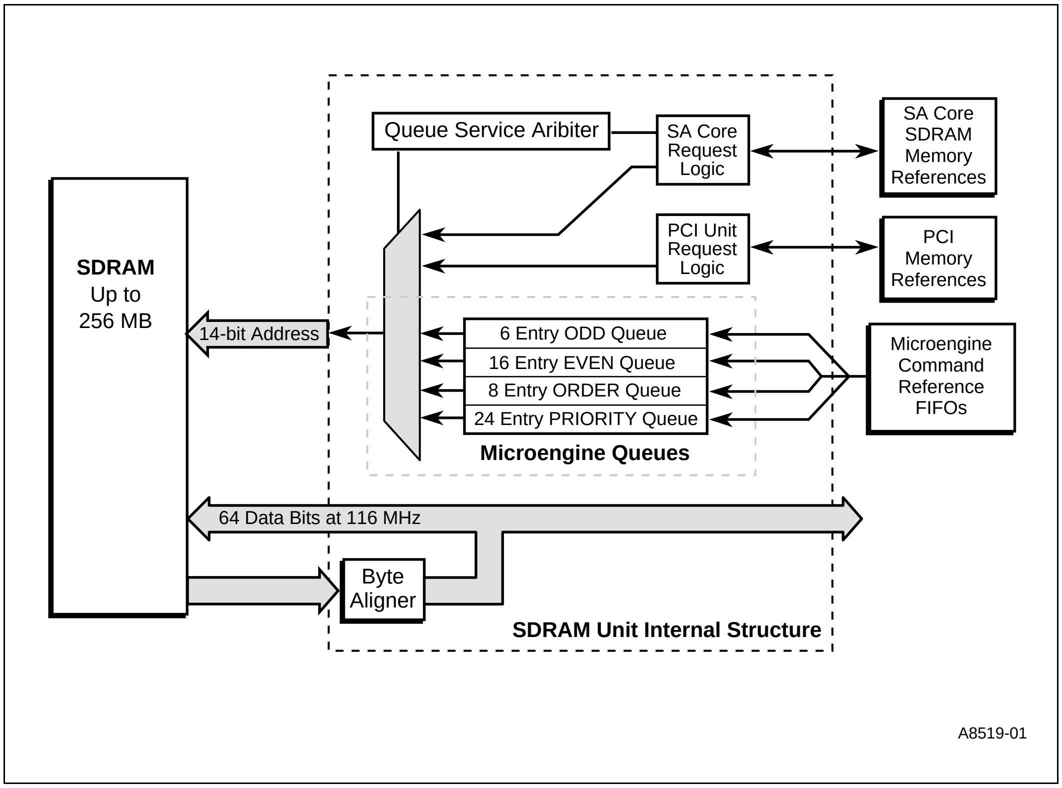


Figure 2-11 is a simplified block diagram of the SDRAM Unit. Like the SRAM Unit, the main function of the SDRAM Unit is to take the various Reference Commands from the Microengines and the StrongARM, and fetch the data on the “pin-side” of the SDRAM in an optimal fashion. In this case, however, it is not READ/WRITE bus turnaround cycles that are avoided, but RAS-Precharge overhead that is reduced. The SDRAM Unit tries to alternate access to ODD banks of SDRAM with accesses to EVEN banks of SDRAM, thereby overlapping the RAS-Precharge time of one bank with an access to the other bank. Once again, Order and Priority queues are provided so that certain accesses can occur in order, and certain access can take priority, respectively. Again, StrongARM accesses take precedence over Microengine accesses for the same reason as with the SRAM Unit.

The PCI Unit also has access to the SDRAM. Any device residing on the PCI bus can access SDRAM memory. As well, the two DMA Controllers in the PCI Unit have the ability to move data between the SDRAM and a PCI device.

The SDRAM Unit also has a Byte Aligner which is in the data path between the SDRAM and the IX Bus Unit. This feature is especially useful in networking applications where a packet header needs to be replaced with new packet header of a different size. This can cause all subsequent words within a packet to be mis-aligned with respect to the SDRAM memory width (a quadword). Without a byte aligner, a Microengine would have to read a word from memory, extract the necessary portion, read a second word from memory, append the new portion, and create a new quadword-aligned result. This would unnecessarily consume a lot of Microengine resources. The Byte Aligner is smart enough to accumulate results from previous reads, so that it can continue to byte align subsequent words from SDRAM.

It should also be noted that the SDRAM Unit has a direct path to the IX Bus Transmit and Receive FIFOs, so the Microengines don't get in the way. So, a single instruction from a Microengine can cause up to 64 bytes of (hardware byte aligned) data to flow between SDRAM and the IX Bus Unit. Meanwhile, the Microengine thread can be performing other useful work.

The SDRAM Unit is connected to the units shown in Table 2-5.

Table 2-5. Summary of SDRAM Unit Internal Connections

Unit	Resource
StrongARM Core	Full access.
PCI Unit	There is a separate, and unshared 32-bit bus connecting the SDRAM Unit and the PCI Bus Unit. This allows devices on the PCI Bus better access to data buffers within the SDRAM Unit. The two DMA Controllers in the PCI Unit have access to SDRAM data.
SRAM Unit	No connection between the SDRAM Unit and the SRAM Unit.
IX Bus Unit	There is a 32-bit bus connecting the SDRAM Unit and the IX-Bus Unit's Receive and Transmit FIFOs.
Microengines	Full access.

2.7.4.1 Internal SDRAM Bandwidth and Internal Data Busses

There are two direct data paths between the SDRAM Unit and the IX Bus Unit. One is an SDRAM *read* data path to transfer data from the IX Bus Unit's Receive FIFOs into SDRAM, and the other is an SDRAM *write* data path to transfer data from SDRAM to the IX Bus Unit's Transmit FIFOs. These two busses operate independently of each other. Whereas both of these internal busses are 32 bits wide, they operate at twice the clock frequency of the SDRAM external (pin-side) data bus. So, while the external SDRAM data bus operates at (64 bits x 116 MHz) 7.4 Gbps, each of the two internal (read & write) data busses also operate at (32 bits x 232 MHz) 7.4 Gbps. Therefore, the internal SDRAM bus bandwidth equals or exceeds the external bus bandwidth, ergo no internal bottleneck in the data path.

2.7.4.2 Chained References

To take advantage of SDRAM architecture and sequential accesses to the same SDRAM page (SDRAM row address stays the same), the IXP1200 implements a feature called *Chained References*. In this mode the SDRAM Unit executes the Chained SDRAM Command References contiguously (and at a high priority so no interruptions can occur) so that sequential chained references can take advantage of SDRAM page hits. To make use of this, the SDRAM reference command must append the optional *chained_reference* token to the end of the instruction.

It should be noted that abnormally long chained references can interfere with SDRAM refresh, and that care should be taken so that these chained references do not unduly interfere with other Microengine thread SDRAM command references.

2.7.5 PCI Unit

Figure 2-12. PCI Unit Block Diagram

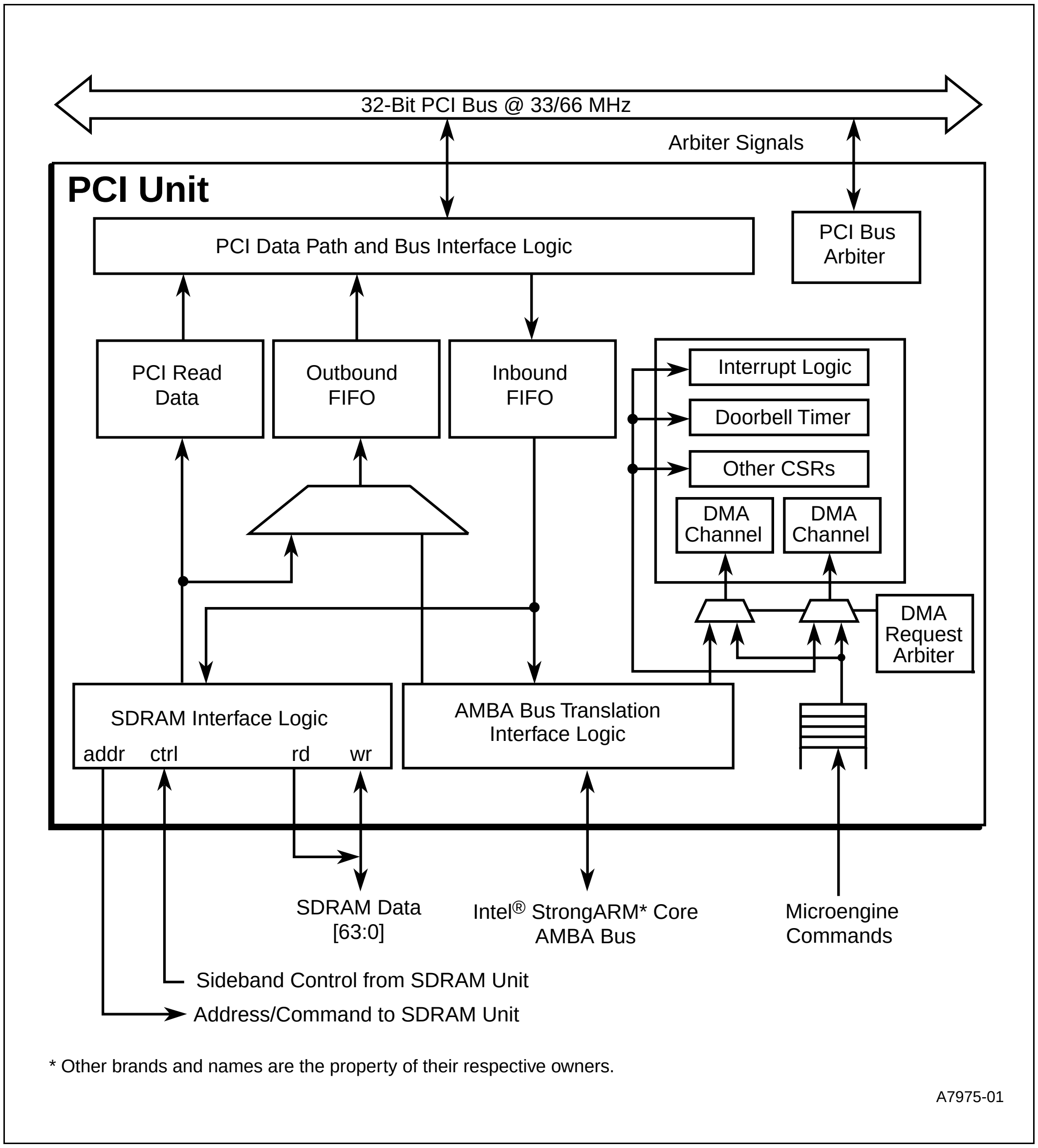

32-Bit PCI Bus @ 33/66 MHz
Arbiter Signals
PCI Unit
PCI Data Path and Bus Interface Logic
PCI Bus Arbiter
PCI Read Data
Outbound FIFO
Inbound FIFO
Interrupt Logic
Doorbell Timer
Other CSRs
DMA Channel
DMA Channel
DMA Request Arbiter
SDRAM Interface Logic
addr
ctrl
rd
wr
AMBA Bus Translation Interface Logic
SDRAM Data [63:0]
Intel® StrongARM* Core AMBA Bus
Microengine Commands
Sideband Control from SDRAM Unit
Address/Command to SDRAM Unit
* Other brands and names are the property of their respective owners.
A7975-01

Figure 2-12 is a simplified block diagram of the PCI Unit. The PCI Unit is a standard 32 bit PCI 2.1 compliant interface with some additional features. The PCI bus can be run at 33 MHz with the standard number of loads, while at 66 MHz, a point to point configuration is supported. For speeds between 33 and 66 MHz, additional loads can be supported, but with a de-rating of the bus speed.

The PCI Unit is connected to the units shown in Table 2-6.

Table 2-6. Summary of PCI Unit internal connections

Unit	Resource
StrongARM Core	Full access. The StrongARM core can access any devices on the PCI Bus. The StrongARM core can program any of the PCI Unit's Control and Status Registers.
SDRAM Unit	Unshared, direct 32-bit bus connecting SDRAM Unit and the PCI Unit. Devices on the PCI Bus have access to SDRAM. DMA Controllers within the PCI Unit can transfer data between SDRAM and devices on the PCI Bus.
SRAM Unit	No connection between the SRAM Unit and the PCI Unit. Devices on the PCI Bus cannot access SRAM. DMA Controllers within the PCI Unit cannot access SRAM.
IX Bus Unit	No connection between the IX Bus Unit and the PCI Unit. Data that needs to be transferred between the IX Bus (Transmit and Receive FIFOs) and the PCI Bus must be transferred by the Microengines (by way of their Transfer Registers).
Microengines	The Microengines can only access the Control and Status Registers for the DMA Controllers within the IX Bus Unit. No other access. The Microengines cannot access devices on the PCI Bus (and vice versa).

The PCI Unit is connected to the SDRAM Unit, and the StrongARM processor. There is a separate, unshared data bus connecting the SDRAM Unit to the PCI Unit. Devices on the PCI bus have full access to SDRAM. The StrongARM processor also has full access to the PCI interface and can be a bus master. The PCI Unit also contains a PCI Bus Arbiter. The Arbiter supports up to 3 bus masters, one of which is the IXP1200 whose arbiter is enabled. If an external PCI Bus Arbiter is required, the internal Arbiter can be disabled (the state of the pci_cfn[1] pin activates or disables the PCI bus arbiter at bootup time). The PCI Unit also supports Doorbells and Messaging.

Two DMA Controllers are integrated into the PCI Unit. Each can be used by either the StrongARM, or by any of the Microengines. The DMA Controllers are programmed by means of DMA Descriptors that reside in SDRAM. DMA Descriptors contain the start address in memory, the number of words that need to be transferred, the destination address on the PCI side, and whether the Descriptor is chained. Chained Descriptors are a series of Descriptors, where each Descriptor points to the next Chained Descriptor, till the last Descriptor is reached. In this way, non-contiguous blocks of data from SDRAM can be moved to PCI as if they were one contiguous block. This is useful for scatter/gather of data.

The PCI Unit contains a number of FIFOs which help to burst data across the PCI bus, making transfer more efficient.

The Microengines have access to only the DMA Controllers within the PCI Unit. There is no connection between the PCI Unit and SRAM, and also no connection between the PCI Unit and the IX Bus Unit. It is not expected that data will need to be transferred between SRAM and the PCI interface. When transferring data between the PCI Unit and the IX Bus, the Microengines have to intervene. Typically, the data would be transferred from the IX Bus Unit's FIFOs directly to SDRAM (note that one instruction from a Microengine can transfer up to 64 bytes between the IX Bus Unit and SDRAM; note also, that there is direct connection between the IX Bus Unit and the SDRAM Unit). Then, the Microengine thread would utilize one of the PCI DMA controllers to transfer data over the PCI interface.

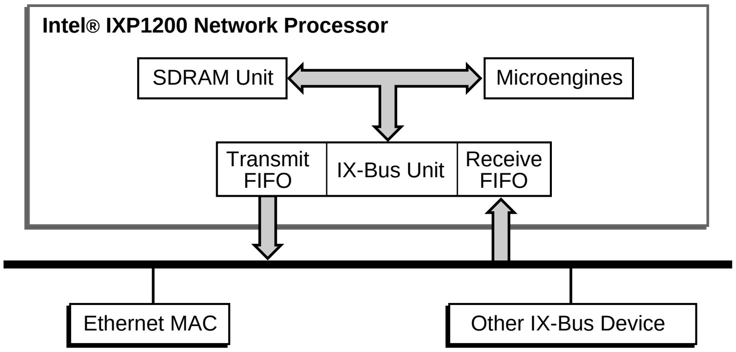
2.7.6 IX Bus Unit

The IX Bus Unit consists of a number of parts: the 64 bit, high-speed data path and related control signals; the Receive and Transmit FIFOs; a separate Ready Bus which is described in more detail later in this section; and lastly it contains some additional functions such as the Scratchpad RAM, a cycle-count register, and the hardware Hashing Unit.

Figure 2-13 is a simplified block diagram showing how data from the IX Bus interface flows through the IXP1200. The thick, block arrows show how data from the IX Bus Interface (the pins), must first go into the Receive FIFO. Once data is in the Receive FIFO, it can then be transferred to either the Microengines (Transfer Registers), or SDRAM. Data to be transmitted to devices on the IX Bus can only come from the Transmit FIFO. So data is first transferred from SDRAM (or the Microengines) into the Transmit FIFO, and then transferred from the Transmit FIFO to the devices on the IX Bus (the pin side interface).

The main task of the IX Bus Unit, is to receive requests from the different Microengine threads, and to execute them. The IX Bus Unit transfers data between the pin-side of the IX Bus interface, and FIFOs within the IX Bus Unit itself. It also transfers data between the IX Bus FIFOs and other functional units within the IXP1200. Thus, the IX Bus Unit is an intelligent “buffering unit”, that buffers data on the IX Bus interface, before it travels to various other units within the IXP1200. These transfers are purely under the control of the Microengines.

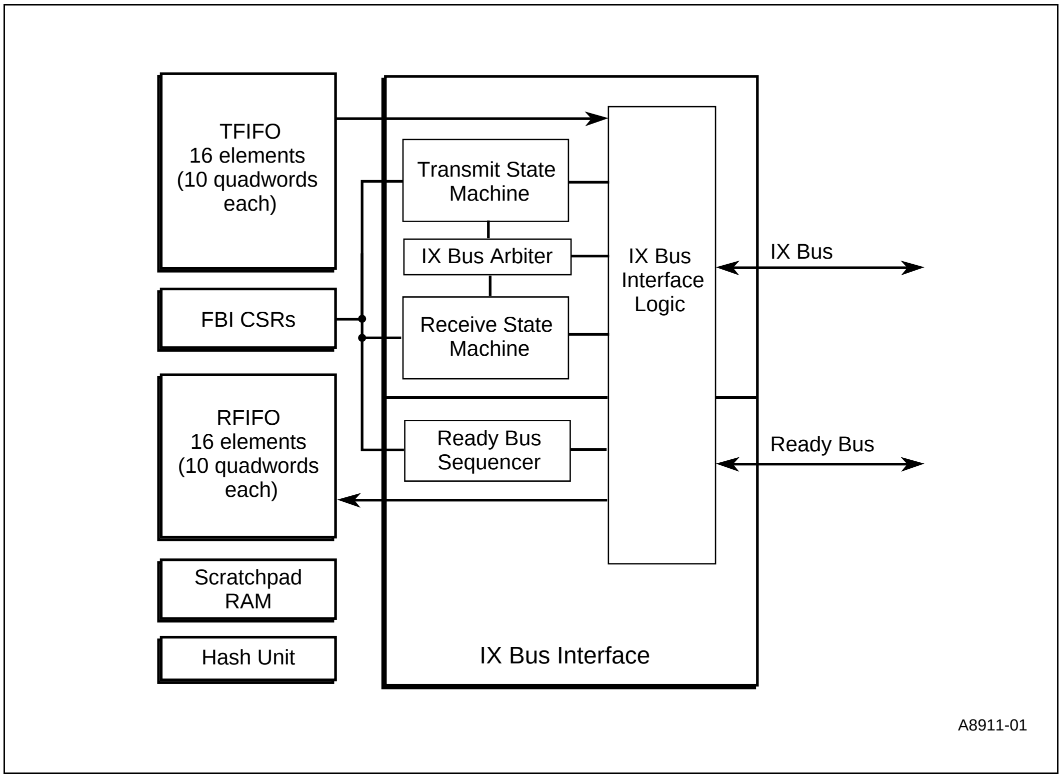
Figure 2-13. IX Bus data flow



A8520-01

The IX Bus Unit contains a Receive FIFO and a Transmit FIFO which buffer the data that flows over the IX Bus. The Receive FIFO in fact really consists of 16 FIFO Elements each 64 bytes deep (though in one of the modes it can be as deep as 72 bytes). These 16 FIFO Elements are shared by all of the Microengine threads. Software has to ensure that only one thread has “ownership” of a FIFO element at a time. When a Microengine thread instructs the IX Bus Unit to fetch data from one of the devices on the IX Bus, it also indicates where to put the data (which FIFO element(s)). Once the data has been fetched from the device on the IX Bus, the IX Bus Unit signals the requesting thread. The thread can then move the data from the Receive FIFO element to wherever it chooses (either SDRAM, or into Microengine Transfer Registers)

Figure 2-14. Simplified IX Bus Unit Block Diagram



2.7.6.1 Ready Bus

The Ready Bus is a separate bus from the IX Data Bus. The Ready Bus has it's own 8-bit data bus and associated control signals. The Ready Bus serves three functions:

- It is used to periodically capture the "ready flags" (MAC FIFO status indicators) from each of the devices on the IX Bus. For example, the Ready Bus would periodically capture the ready flags for each of the MAC devices on the IX Bus. This is especially important for multi-port MAC devices (such as the IXF440) where there are multiple receive and transmit FIFOs, each with it's own ready flag. These flags serve to indicate when the FIFOs reach a pre-programmed threshold level. For example, when one of the MAC's ports Receive FIFO has hit a critical threshold level, the data must be emptied from it soon, or the data will overflow. The Ready Flags are captured by the Ready Bus Sequencer, and made available for inspection by the Microengines. Once a Microengine notices that one of the ports requires attention, it can make sure that the port is serviced.
- The Ready Bus also serves to transfer data between IXP1200s in applications where several IXP1200s share the IX Bus. The 8-bit data bus is used transfer data between IXP1200s (in a daisy chained fashion).
- The Ready Bus also serves to assert flow control signals to the MACs (or other network data ports hooked up to the IX Bus).

The Ready Bus Sequencer is implemented as a programmable state machine. The Microengines or the StrongARM core may program the Controller by writing up to twelve instructions into it. Typically, the last instruction would cause a jump back to the first instruction, causing the program to continually loop. In this way, the Ready Bus Sequencer, continually samples the MACs' Ready Flags and makes them available to the Microengines. The Microengines examine the Ready Flags to determine which ports require servicing.

There are two ways that the Microengines can access these ready flags. One way is by accessing the registers within the IX Bus Unit that contain the ready flags. The second way is by using the *Autopush* mechanism, where the Ready Bus Sequencer can be programmed to "push" the ready flags into a specified Microengine Transfer Register. In the second approach, the Microengine need not issue Reference Commands to access these registers which are external to the Microengine, but only need to check its own (local) Transfer Registers.

2.7.6.2 IX Data Bus Modes

The IX Data Bus can be configured to operate in several different modes depending on the devices that are connected to it. The 64 bit data bus can be made to operate in either a *64 bit Bidirectional Mode*, or a *32 bit Unidirectional Mode*. In the 64 bit bidirectional mode, the data path is wider, however data can travel in only one direction at a time and read/write bus turnaround cycles must take place (dead cycles). In the 32 bit unidirectional mode, the 64 bit data bus is split into two 32 bit busses, one for incoming data, and the other for outgoing data. The 32 bit unidirectional busses operate independently of each other because the receive bus is controlled by the Receive State Machine, while the transmit bus is controlled by the Transmit State Machine. From a programming point of view, the Microengines issue transmit and receive requests to the IX Bus Unit in the same way, regardless of which of the two modes is programmed.

In the 64 bit Bidirectional Mode, it is possible to have several IXP1200 processors sharing the IX Bus. This is called *Shared IX Bus Mode*. In this mode, multiple IXP1200s are connected to the IX Bus, and a token is passed between them (in a daisy chained manner) to determine ownership of the bus.

In addition, the IX Bus control pins may be configured to operate in two different MAC addressing modes: *1-2 MAC Mode*, and *3+ MAC Mode*.

2.7.6.3 Scratchpad RAM

The Scratchpad RAM (1K x 32 bits), which is accessible by the StrongARM core as well as the Microengines, is not a standalone unit but is located within the IX Bus Unit. The Scratchpad RAM supports three different types of operations: read/write, bit operations, and auto-increment operations. While the StrongARM can only perform read/write operations, all three types of operations are available to the Microengines. The bit operations modify the specified bits, and consist of the following operations: set bits, clear bits, test and set, and test and clear. These operations make the Scratchpad RAM area very useful for inter process communication, semaphores, and mailboxes.

The Scratchpad also forms the third memory resource available to the StrongARM and the Microengines. By distributing memory accesses across these three (SRAM, SDRAM, Scratchpad) memories an application can take advantage of these accesses occurring in parallel.

2.7.6.4 Hashing Unit

The Hardware Hashing Unit is also located within the IX Bus Unit. It is capable of doing either 48 bit or 64 bit hashes. The hashing function takes the input 48 (or 64) bits, and produces the hashed 48 (or 64) bit result. The Hashing Unit is only accessible to the Microengines. Once a Microengine thread requests a hash, the IX Bus Unit pulls the data out of the specified Transfer Registers, performs the hash, and returns the result to the Transfer Registers. With a single instruction, a Microengine can request either one, two, or three hashes to be performed. These are queued up by the IX Bus Unit, and then performed sequentially.

The Hashing Unit performs the hashing function in 8 bit blocks, passing the 48 or 64 bits of the data into the unit 8 bits at a time. Whereas the hashing algorithm is fixed, the polynomials are programmable.

2.7.6.5 IXB3208 Bus Scaling Fabric

The IXB3208 is an IX Bus scaling fabric component that offers glueless connection of multiple IXP1200s. Up to eight IXP1200s can be connected using four IXB3208s. Different configurations are possible and more detail may be found in the *IXB3208 Product Brief* and *IXB3208 Datasheet*. The IB3208 implements a full crossbar switching fabric.

The IX Bus Unit is connected to the Units shown in Table 2-7.

Table 2-7. Summary of IX Bus Unit Internal Connections

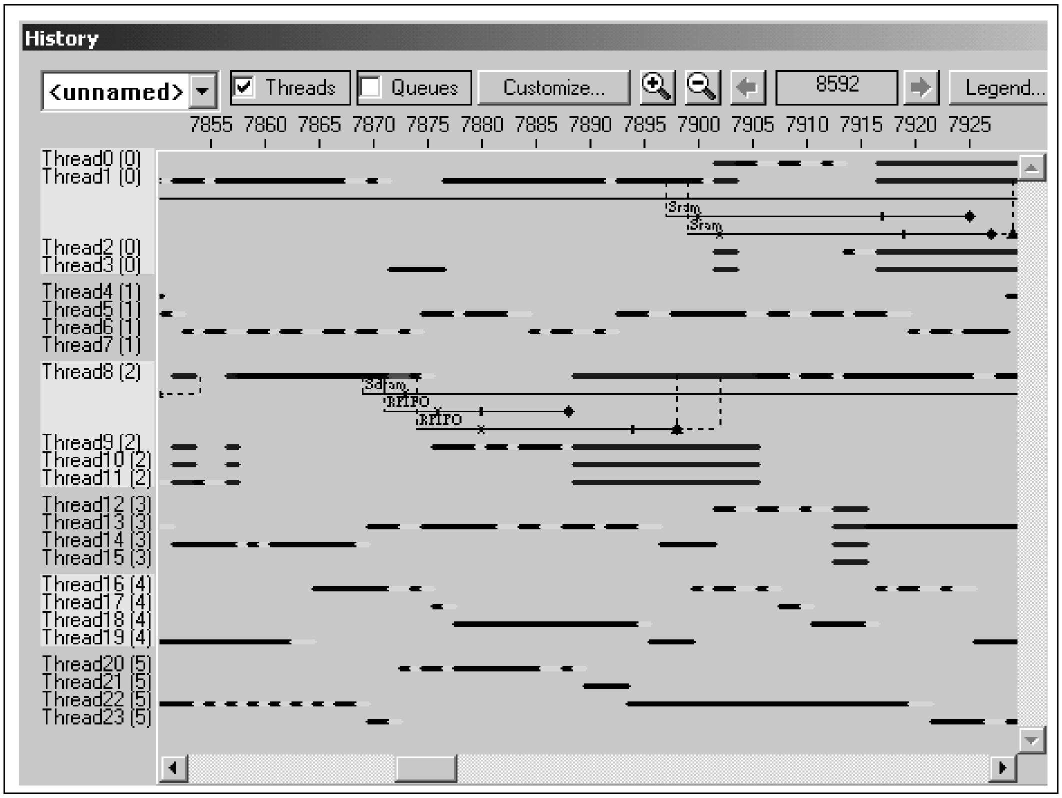
Unit	Resource
StrongARM Core	Limited Access. The StrongARM core can access the Scratchpad RAM within the IX Bus Unit, as well as a number of Control and Status Registers within the IX Bus Unit. See the Appendix for a table of IX Bus registers that appear in the StrongARM memory map. The StrongARM Core cannot access the Receive and Transmit FIFOs within the IX Bus Unit. The StrongARM Core can program the Ready Bus Controller within the IX Bus Unit.
SDRAM Unit	Two direct, unshared, independent busses connect the SDRAM Unit with the IX Bus Unit. The SDRAM read bus is connected to the Receive FIFO, while the write bus is connected to the Transmit FIFO.
SRAM Unit	No connection between the SRAM Unit and the IX Bus Unit's FIFOs. Data that needs to be transferred between the IX Bus (Transmit and Receive FIFOs) and SRAM must be transferred by the Microengines (by way of their Transfer Registers).
PCI Unit	No connection between the IX Bus Unit and the PCI Unit. Data that needs to be transferred between the IX Bus (Transmit and Receive FIFOs) and the PCI Bus must be transferred by the Microengines (by way of their Transfer Registers).
Microengines	Full Access, including access to the Scratchpad RAM, the Hashing Unit, programming the Ready Bus Controller, the Receive and Transmit FIFOs, and all Control and Status Registers.

2.8 Software Development Tools

The IXP1200 software development tools consist of the following elements:

- An Assembler for the Microengine instruction set.
- The StrongARM tools (C Compiler and Assembler) can be obtained from ARM Limited*, or a number of other software tool vendors.
- A Linker and Loader is provided to link and load the Microengine code with the StrongARM code.
- The *Transactor* is a software model of the IXP1200. It eases the debugging of IXP1200 code by providing a data accurate model of the IXP1200 on a cycle by cycle basis. The *Developer's Workbench* graphically displays a history of each of the twenty four threads (Figure 2-15) so the user may easily discern how the software in running and may more easily debug the application.
- The Developer's Workbench is a graphical user interface that ties these tools together to work seemlessly, and ease the development and debugging of IXP1200 code. The Developer's Workbench also has a number of other very useful features, including execution profiling, statistics gathering, a C-like scripting facility, data-watch windows, breakpoints, queue statistics, etc.
- The Transactor also has a software simulation interface called the Foreign Model. By using the Foreign Model, designers can link models of external devices (say MACs, SARs, or any custom devices) into the Transactor model. In this way, a more complete environment for debugging the entire application is provided.

Figure 2-15. Developer's Workbench - Thread History Display



StrongARM* Core 3

3.1 Overview

The Intel® IXP1200 Network Processor Family devices contain an Intel StrongARM* processor based on the Intel® SA-1 core. The IXP1200 documentation refers to this processor as the “StrongARM* core”. The Intel SA-1 core implements the ARM* V4 architecture as defined in the *ARM Architecture Reference Manual*. This chapter supplements the *ARM Architecture Reference Manual* by describing the differences between the StrongARM* core and the fundamental ARM* architecture, the implementation options supported by the IXP1200 (such as MMU, caches, etc.), and how the StrongARM* core is integrated as a component of the overall IXP1200 system.

This chapter is organized as follows:

- Section 3.2, “ARM* Architecture” describes how the StrongARM* core varies from the ARM* V4 architecture. This section supplements the *ARM Architecture Reference Manual*.
- Section 3.3, “Memory Map” describes how the other components of the network processor map into the StrongARM* core 4 Gbyte address space.
- Section 3.4, “FIQ and IRQ Interrupts” describes the interrupt hierarchy used in the IXP1200.
- Section 3.5, “Internal Peripheral Units” describes the UART, general purpose I/O pins, real-time clock, and four 24-bit system timers that can be accessed by the StrongARM* core.
- Section 3.6, “Boot Sequence” describes what the StrongARM* core must do to boot the IXP1200.

3.2 ARM* Architecture

This section describes the ARM* implementation options supported by the StrongARM* core. The StrongARM* core implements the ARM* V4 architecture as defined in the *ARM Architecture Reference Manual*. It is highly recommended that the reader be familiar with the *ARM Architecture Reference Manual* prior to reading this section. In this section, as in StrongARM* and ARM documentation, a word refers to 32 bits and a halfword refers to 16 bits. For all other sections, refer to Table 1-1 for data terminology.

3.2.1 Coprocessors

The StrongARM* core supports Coprocessor 15 (see the *IXP1200 Network Processor Programmer’s Reference Manual* for a description of the Coprocessor 15 registers). All other coprocessor instructions cause an undefined instruction exception. No support for external coprocessors is provided.

Coprocessor 15 contains registers local to the StrongARM* core that control and configure the cache, write buffer, MMU, read buffer, breakpoints, some clocking functions, and other configuration options. These registers are accessed using MRC and MCR instructions to coprocessor 15 with the processor in any privileged mode. Only some of registers 0-15 are valid; the result of an access to an invalid register is unpredictable.

3.2.2 Memory Management Unit (MMU)

The MMU has two primary functions:

- It translates virtual addresses into physical addresses.
- It controls memory access permissions.

The StrongARM* core implements the standard ARM* memory management functions using two 32-entry (4 bytes per entry) fully associative translation buffers (TBs). One is used for instruction accesses and the other for data accesses. Each TB entry can map a segment, a large page, or a small page. On a TB miss, the translation table hardware is invoked to retrieve the translation and access permission information. Once retrieved, if the entry maps to a valid page or section, then the information is placed into the TB. The replacement algorithm in the TB is round-robin. For an invalid page or section, an abort is generated and the entry is not placed in the TB. The data TBs support both the flush-all and flush-single-entry operations, while the instruction TBs support only the flush-all operation. The Memory Management Unit is configured via the Coprocessor 15 registers.

3.2.2.1 MMU Faults and CPU Aborts

The MMU generates the following faults:

Alignment: Generated by a word load or store if the two low-order address bits are nonzero or by a half word load or store if the low-order address bit is a one.

Translation: Generated by access to pages marked invalid by the memory-management page tables.

Domain/Permission: Generated by accesses to memory that are protected by the current mode, domain, and page protection.

3.2.2.2 Data Aborts

The StrongARM* core takes a data abort exception upon MMU-generated exceptions or accesses to reserved memory space.

A linefetch during a cache read can be safely aborted on any word in the transfer. If an abort occurs during the linefetch, the cache is purged so it does not contain invalid data. If the abort happens before the word that was requested by the access is returned, the load is aborted. If the abort happens after the word that was requested by the access is returned, the load completes and the fill is aborted (but no exception is generated).

3.2.2.3 Interaction of the MMU, Icache, Dcache, and Write Buffer

The MMU, Icache, Dcache, and write buffer can be enabled or disabled independently. The Icache can be enabled with the MMU either enabled or disabled. However, the Dcache and write buffer can only be enabled when the MMU is enabled. Because the write buffer is used to hold dirty copyback cached lines from the Dcache, it must be enabled along with the Dcache. Therefore, only four of the eight combinations of the MMU, Dcache, and write buffer enables are valid. There are no hardware interlocks on these restrictions, so invalid combinations cause undefined results.

Table 3-1. Valid MMU, Dcache, and Write Buffer Combinations

MMU	Dcache	Write Buffer
Off	Off	Off
On	Off	Off
On	Off	On
On	On	On

3.2.2.4 MMU Enable/Disable

The following procedures must be observed when enabling and disabling the MMU.

To enable the MMU:

1. Program the Coprocessor 15 TRANSLATION_TABLE_BASE and DOMAIN_ACCESS_CONTROL registers.
2. Program level 1 and level 2 page tables as required.
3. Enable the MMU by setting bit 0 of the Coprocessor 15 CONTROL_CP15 register.

Note: Care must be taken if the translated address differs from the untranslated address. This is because the three instructions following the enabling of the MMU are fetched using “flat translation”, and enabling the MMU may be considered a branch with delayed execution. A similar situation occurs when the MMU is disabled. Consider the following example:

Example 3-1. MMU Enable/Disable Example

```
MOV R1, #0x1
MCR 15,0,R1,0,0 ; Enable MMU
Fetch non-translated
Fetch non-translated
Fetch non-translated
Fetch Translated
```

After enabling the MMU, if the virtual address for the code page being addressed maps to a different physical address, several instructions are fetched before address translation takes effect.

To disable the MMU:

1. Disable the write buffer by clearing bit 3 in the Coprocessor 15 CONTROL_CP15 register.
2. Disable the Dcache by clearing bit 2 in the Coprocessor 15 CONTROL_CP15 register.
3. Disable the Icache by clearing bit 12 in the Coprocessor 15 CONTROL_CP15 register.
4. Disable the MMU by clearing bit 0 in the Coprocessor 15 CONTROL_CP15 register.

Note: If the MMU is disabled and subsequently reenabled, the contents of the TB is preserved. If the contents are now invalid, the TB should be flushed by writing the Coprocessor 15 TLB_OPERATIONS register before reenabling the MMU.

3.2.3 Instruction Cache (Icache)

The StrongARM* core supports a 16 Kbyte instruction cache (Icache) to reduce effective memory access time. Its operation is transparent to program execution. The Icache has 512 lines of 32 bytes (8 words), arranged as a 32-way set associative cache, and uses the virtual addresses generated by the processor core. The Icache is always reloaded a line at a time (8 words). It may be enabled or disabled via the Coprocessor 15 CONTROL_CP15 register, and is automatically disabled on the assertion of the RESET pin or through a software reset sequence.

The Icache supports the flush all function. Replacement is round-robin within a set. The Icache can be enabled while memory management is disabled. When memory management is enabled, the C bit (cacheable) in each translation descriptor entry can disable caching for an area of virtual memory. If memory management is disabled, all addresses are marked as cacheable (C=1).

3.2.3.1 Icache Operation

The instruction cache is searched regardless of the state of the C bit; only reads that miss the cache are affected. If, on an Icache miss, the C bit equals 1 or the Memory Management Unit (MMU) is disabled, a linefetch of 8 words is performed and it is placed in a cache bank with a round-robin replacement algorithm. If, on a miss, the MMU is enabled and the C bit equals 0 for the given virtual address, an external memory access for a single word is performed and the cache is not written. The Icache should be enabled as soon as possible after reset for best performance.

3.2.3.2 Icache Validity

The Icache operates with virtual addresses, so care must be taken to ensure that its contents remain consistent with the virtual-to-physical mappings performed by the memory management unit. If the memory mappings are changed, the Icache validity must be ensured. The Icache is not coherent with stores to memory, so programs that write cacheable instruction locations must ensure the Icache validity. Instruction fetches do not check the write buffer, so data must not only be pushed out of the cache but the write buffer must also be drained. The entire Icache can be invalidated by writing to coprocessor 15 CACHE_CONTROL_OPERATIONS (Register 7). The cache is flushed immediately when the register is written, but the following instruction fetches may come from the cache before the register is written.

3.2.3.3 Icache Enable/Disable and Reset

The Icache is automatically disabled and flushed when the StrongARM* core is reset. Once enabled, cacheable read accesses cause lines to be placed in the cache. If the Icache is subsequently disabled, no new lines are placed in the cache, but the cache is still searched and if the data is found, it is used by the processor. If the data in the cache must not be used, then the cache must be flushed.

To enable the Icache, set bit 12 in the control register. The MMU and Icache may be enabled simultaneously with a single control register write. To disable the Icache, clear bit 12 in the Coprocessor 15 CONTROL_CP15 register.

3.2.4 Data Caches (Dcaches)

The StrongARM* core supports two logically separate data caches: the main data cache and the mini data cache (or minicache) that reduce effective memory access time. Its operation is transparent to program execution.

3.2.4.1 Main Data Cache

The main data cache, an 8 Kbyte write-back Dcache, has 256 lines of 32 bytes (8words) in a 32-way set-associative organization. It is intended for use during most data accesses. This cache allocates on loads to an address space marked in the MMUs translation descriptor as Bufferable (B bit =1) and cacheable (C bit =1). Replacements in the main data cache are selected according to a set of round-robin pointers. At reset, the pointer in each block of the Dcache points to way zero of each 32-way block. As lines are allocated, the pointers are incremented to the next way of the set. After way 31 is allocated, the next line fill replaces (and copies back to memory, if dirty) the data in way zero. The cache supports the flush-all, flush-entry, and copyback-entry functions. The copyback-all function is not supported in hardware. This function can be provided by software.

3.2.4.2 Mini Cache

The minicache is a 512-byte write-back cache. It has 16 lines of 32 bytes (8 words) in a two-way set-associative organization and provides an alternate caching structure for dealing with large data structures that could thrash the main data cache. This cache allocates on loads to an address space marked in the MMUs translation descriptor as not Bufferable (B bit =0) and cacheable (C bit =1). Replacements in the minicache use the same round-robin pointer mechanism as in the main data cache. However, since this cache is only two-way set-associative, the replacement algorithm reduces to a simple least-recently-used (LRU) mechanism.

3.2.4.3 Dcaches Enable/Disable and Reset

The Dcaches are automatically disabled and flushed when the StrongARM* core is reset. To enable the Dcaches, the MMU must first be enabled by setting bit 0 in the CONTROL_CP15 register (Register 1), and then the Dcaches should be enabled by setting bit 2 in the CONTROL_CP15 register. The Dcaches can be disabled by clearing bit 2 in the CONTROL_CP15 register. The Dcaches are also controlled by the cacheable or C bit and the bufferable or B bit stored in the MMU translation descriptor table. For this reason, to use the Dcaches, the MMU must be enabled. The two functions may be enabled simultaneously with a single write to the control register. Once enabled, cacheable read accesses cause lines to be placed in the Dcaches. If subsequently disabled, no new lines are placed in the Dcaches, but they are still searched and if the data is found, it is used by the processor. Write operations continue to update the Dcaches, thus maintaining consistency with the external memory. If the data in the Dcaches must not be used, then the Dcaches must be flushed.

3.2.4.4 Dcache Operation

The Dcaches are accessed in parallel and the design ensures that a particular line entry exists in only one of the two at any time. Both Dcaches use the virtual address generated by the processor and allocate only on loads (write misses never allocate in the cache). Each line entry contains the physical address of the line and two dirty bits. The dirty bits indicate the status of the first and the second halves of the line. When a store hits in the Dcaches, the dirty bit associated with it is set.

When a line is evicted from the Dcaches, the dirty bits are used to decide if all, half, or none of the line is written back to memory using the physical address stored with the line. The Dcaches are always reloaded a line at a time (8 words).

The C bit (cacheable) in each translation descriptor entry determines whether, on load misses, the data being read should be placed in one of the two data caches. The cacheable bit does not affect cache hits. That is, if a data access hits in the cache, the data is assumed to be valid and the load or store is performed. Typically, main memory is marked as cacheable to improve system performance and I/O space as noncachable to stop the data from being stored in the cache. For example, if the processor is polling a hardware flag in I/O space, it is important that the processor is forced to read data from the external peripheral, and not a copy of initial data held in the cache. When the cacheable bit is set (1), a linefetch of 8 words is performed and placed in a cache bank with a round-robin replacement algorithm. When the cacheable bit is cleared (0), an external memory access is performed and the cache is not written.

The B bit (bufferable) does not affect writes that hit the Dcaches. If a store hits in the Dcaches, the store is assumed to be bufferable. Write-backs of dirty lines are treated as bufferable writes. See Section 3.2.5 for more information on the B bit. Table 3-2 summarizes the effects of the B and C bits on the Dcaches.

Table 3-2. Effects of the Cacheable and Bufferable Bits on the Data Caches

		Load		Store	
B	**C**	**Cache Hit**	**Cache Miss**	**Cache Hit**	**Cache Miss**
0	0	Deliver cache data.	Load from memory. – No allocate.	Store to either cache. – Mark line dirty.	Store to memory. – No allocate.
0	1	Deliver cache data.	Allocate to minicache.	Store to either cache. – Mark line dirty.	Store to memory. – No allocate.
1	0	Deliver cache data.	Load from memory. – No allocate.	Store to either cache. – Mark line dirty.	Store to memory. – No allocate.
1	1	Deliver cache data.	Allocate to main data cache.	Store to either cache. – Mark line dirty.	Store to memory. – No allocate.

The Dcaches should be flushed prior to changing the B and C bit in the translation descriptor table mapping. The Dcaches operate with virtual addresses, so care must be taken to ensure that their contents remain consistent with the virtual-to-physical mappings performed by the memory management unit. If the memory mappings are changed, the validity of the Dcaches must be ensured.

3.2.4.5 Software Dcache Flush

The StrongARM* core supports the flush and clean operations on single entries of the Dcaches by writes to the cache operations registers. Flush whole cache is also supported. Since this is a write-back cache, to prevent the loss of data, a flush of the whole cache must be preceded by a sequence of loads to cause the cache to write back any dirty entries. The StrongARM* core memory controller provides an internally decoded memory space to perform coherent Dcache flushing. This space resides in the upper 512 megabytes of the memory map (starting at virtual address E000 0000h) and, when accessed, is detected by the memory controller, which then returns zeros without incurring an external memory latency. The following code causes the main data cache to flush all dirty entries:

```
;+
;Call:
; R0 points to the start of a 8192 byte region of readable data used
; only for this cache flushing routine.
; bl writeBackDC
;Return:
; R0, R1, R2
; Data cache is clean
;-writeBackDC
mov r0, 0hE000000
add r1, r0, #8192
l1
ldr r2, <r0>, #32
teq r1, r0
bne l1
mcr p15, 0, r0, c7, c6, 0
mov pc, r14
```

A similar routine may be written to flush the minicache. To perform this flush, the MMU B and C bit settings must be as described in Section 3.2.4.4. The invalidate all operation also invalidates the minicache. Since the Dcaches work with virtual addresses, it is assumed that every virtual address maps to a different physical address. If the same physical location is accessed by more than one virtual address, the cache cannot maintain consistency, since each virtual address has a separate entry in the cache, and only one entry is updated on a processor write operation. To avoid any cache inconsistencies, doubly mapped virtual addresses should be marked as noncachable.

3.2.5 Write Buffer

The write buffer improves system performance by buffering up to 8 blocks of data of 1 to 16 bytes at independent addresses. It can be enabled or disabled via the W bit (bit 3) in the Coprocessor 15 CONTROL_CP15 register (Register 1). The buffer is disabled and all entries are marked empty following reset. Operation of the write buffer is further controlled by the cacheable or C bit and the bufferable or B bit, which are stored in the MMU translation descriptor table. For this reason, to use the write buffer, the MMU must be enabled. The two functions can be enabled simultaneously with a single write to the control register. For a write to use the write buffer, both the W bit in the CONTROL_CP15 register and the B bit in the corresponding translation table entry must be set. It is not possible to abort buffered writes externally. With the exception of store multiples, stores do not merge with other data at the same line address in the write buffer. A drain write buffer operation is supported.

3.2.5.1 Write Buffer Operation

When the StrongARM* core performs a store, the Dcaches are first checked. If one of the Dcaches hits on the store and the protection for the location and mode of the store allows the write, then the write completes in the Dcaches and the write buffer is not used. If the location misses in the Dcaches, then the translation entry for that address is inspected and the state of the B and C bits determines which of the three following actions are performed (if the write buffer is disabled via the CONTROL_CP15 register, writes are treated as if the B bit is a zero):

- If the write buffer is enabled and the processor performs a write to a bufferable and cacheable location (B=1,C=1), and the data is in one of the caches, then the data is written to that cache, and the cache line is marked dirty. If a write to a bufferable area misses in both data caches, the data is placed in the write buffer and the CPU continues execution. The write buffer performs the external write sometime later. If a write is performed and the write buffer is full, then the processor is stalled until there is sufficient space in the buffer. No write buffer merging is allowed in the StrongARM* core except during store multiples.
- If the write buffer is enabled and the processor performs a write to a bufferable but noncachable location and misses in the Dcaches (B=1,C=0), the data is placed in the write buffer and the CPU continues execution. As with the cacheable case, merging is allowed only on store multiples. The write buffer performs the external write sometime later.
- If the write buffer is disabled or the CPU performs a write to an unbufferable area (B=0), the processor is stalled until the write buffer empties and the write completes externally. This requires several external clock cycles.

3.2.5.2 Enabling and Disabling the Write Buffer

To enable the write buffer, first ensure that the MMU is enabled by setting bit 0 in the CONTROL_CP15 register, then enable the write buffer by setting bit 3 in the CONTROL_CP15 register. The MMU and write buffer can be enabled simultaneously with a single write to the control register.

To disable the write buffer, clear bit 3 in the CONTROL_CP15 register. Any writes already in the write buffer complete normally, but a drain write buffer needs to be done to force all writes out to memory.

Note: The write buffer is used for copybacks from the Dcache even when it is disabled.

3.2.6 Read Buffer

The IXP1200 contains a software-programmable read buffer that can increase the performance of critical loop code by prefetching data. The read buffer enables the preallocation of read-only data into one of four 32-byte buffers without stalling the pipe. For subsequent loads that hit in the read buffer, data is sourced from the buffer instead of the Dcaches at a rate of 1 word per core clock. Also, because the programmer specifies which entry of the read buffer is used, critical data can be “locked” in to eliminate bus latency.

The read buffer is controlled using coprocessor 15, READ_BUFFER_OPERATIONS (Register 9), and provides the capability to allocate 1 word, a half line (4 words), or a full line (8 words) into one of four entries of the read buffer. Half line loads are automatically aligned onto half block boundaries (the lower four address bits are ignored). Full-line loads are automatically aligned onto line boundaries (the lower five address bits are ignored). For partial cache line read buffer loads, only the words actually fetched are marked valid and can be sourced from the buffer. A small queue is used to ensure that subsequent read buffer load instructions go out in order.

Software can flush either a single entry or the entire buffer (four entries).

3.2.6.1 Read Buffer Operation

Read buffer operations are performed by writing instructions to the Coprocessor 15 READ_BUFFER_OPERATIONS register (Register 9). When a read buffer allocate instruction is executed, the virtual address is looked up in the Translation Buffer (TB) to check for a translation hit and possible access violations. If the access misses in the TB, the pipe is stalled until the page is fetched through the normal hardware tablewalk mechanism. If an access violation occurs, the read buffer load is NOPed. For example, a read buffer allocate instruction can generate a data abort. Once the read buffer allocate has received a TB hit and no access violations, a bus access is requested that fills the appropriate buffer without stalling the core pipeline. Subsequent load instructions to this virtual address result in a read buffer hit and data is sourced from the appropriate entry to the core. Any two data words with the same virtual address may not be contained in the read buffer at the same time. If a read buffer allocate references a data word that is already contained in another read buffer entry, then the old read buffer entry is invalidated and the new allocation is performed. It is possible for a portion of a cache block at a given virtual address to be contained in one read buffer entry while another portion of the same block is contained in another read buffer entry. However, a given word can not be in more than one entry at a time.

If a load instruction misses in the read buffer, a normal cache fill is performed (provided the cache is enabled and the page is marked cacheable). It then presents the possibility of having a partial line resident in the read buffer as well as having the line present in one of the Dcaches. This presents coherency issues that must be managed by software. If this situation does occur and the addressed data is in both the Dcache and the read buffer, then the data is sourced from the read buffer. If a read buffer entry contains a partial cache block (1 or 4 words), those words are sourced from the read buffer while the remaining words are sourced from the data cache or memory.

Read buffer allocate instructions are not affected by the cache enable bit (bit 2 in the control register) or by the C bit in the MMU. Any read buffer allocate to a valid read buffer entry causes that read buffer entry to be invalidated, followed by a new allocation for the desired data. This occurs regardless of the address of the data currently in the buffer. For example, back-to-back read buffer allocate instructions to the same entry at the same address invalidates the entry caused by the first instruction prior to performing the second fill.

A read buffer allocate or a load instruction that is issued to a read buffer entry currently being filled stalls until the fill completes. If a data abort is signaled on a read buffer allocate, the fill completes. After that, if a load to that entry is attempted, a data abort exception is issued. The coprocessor 15 register provides the ability to invalidate individual entries in the read buffer or to invalidate the entire buffer in one operation. Read buffer coherency must be managed in software. Writes to addresses present in the read buffer are not written into the buffer. Specific read buffer entries must be invalidated before writing to the addresses or changing the page tables of the entries. Coherency is not checked between the read buffer and the write buffer. The write buffer should be drained prior to performing a read buffer load.

3.2.7 ARM* Instruction Set and Timing

The StrongARM* core implements the instruction set defined in the *ARM Architecture Reference Manual*. Table 3-3 lists the instruction timing for the StrongARM* core. The result delay is the number of cycles that the next sequential instruction would stall if it used the result as input. The issue cycles are the number of cycles that this instruction takes to issue. For most instructions, the result delay is zero and the issue cycles is one. For load and stores, the timing is for cache hits.

Table 3-3. StongARM Core instruction Timing

Instruction Group	Result Delay	Issue Cycles
Data processing	0	1
Mul or Mul/Add giving 32-bit result	1..3	1
Mul or Mul/Add giving 64-bit result	1..3	2
Load single – write-back of base	0	1
Load single – load data zero extended	1	1
Load single – load data sign extended	2	1
Store single – write-back of base	0	1
Load multiple (delay for last register)	1	MAX (2, number of registers loaded)
Store multiple – write-back of base	0	MAX (2, number of registers loaded)
Branch or branch and link	0	1
MCR	2	1
MRC	1	1
MSR to control	0	3
MRS	0	1
Swap	2	2

3.2.8 Exceptions

Exceptions arise when the normal flow of program execution needs to be broken; for example, so that the processor can be diverted to handle an interrupt from a peripheral. The processor state just prior to handling the exception must be preserved so that the original program resumes when the exception routine has completed. Many exceptions may arise at the same time. The StrongARM* core handles exceptions by making use of banked registers to save state. The contents of PC and CPSR are copied into the appropriate R14 and SPSR, and the PC and mode bits in the CPSR bits are forced to a value that depends on the exception. Interrupt disable flags are set where required to prevent otherwise unmanageable nestings of exceptions. In the case of a reentrant interrupt handler, R14 and the SPSR should be saved onto a stack in main memory before re-enabling the interrupt. When transferring the SPSR register to and from a stack, it is important to transfer the whole 32-bit value, and not just the flag or control fields. When multiple exceptions arise simultaneously, a fixed priority determines the order in which they are handled. The priorities are listed in the following section. Most exceptions are fully defined in the *ARM Architectural Reference*. This section specifies the exceptions where the StrongARM* core implementation differs from the *ARM Architectural Reference*.

3.2.8.1 Exception Priorities

When multiple exceptions arise at the same time, a fixed priority system determines the order in which they are handled:

1. Reset (highest priority).
2. Data abort.
3. FIQ.
4. IRQ.
5. Prefetch abort.
6. Undefined instruction, software interrupt (lowest priority).

Not all exceptions can occur at once. Undefined instructions and software interrupts are mutually exclusive because they correspond to particular (nonoverlapping) decodings of the current instruction. If a data abort occurs at the same time as a FIQ, and FIQs are enabled (that is, the F flag in the CPSR is clear), the IXP1200 enters the data abort handler and then immediately proceeds to the FIQ vector. A normal return from FIQ causes the data abort handler to resume execution. Placing data abort at a higher priority than FIQ is necessary to ensure that the transfer error does not escape detection; the time for this exception entry should be added to worst-case FIQ latency calculations.

3.2.8.2 Exception Vector Table

When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These memory addresses are listed in Table 3-4. At boot time, these vectors are mapped to the physical address in the BootROM space. The MMU can remap the virtual address to these vectors into the physical address of faster SDRAM space. The X bit in the Coprocessor 15 CONTROL_CP15 register allows the base virtual address of the interrupt vectors (0000 0000h) to be re-mapped to a base address of FFFF 0000h.

Table 3-4. Exception Vectors

Exception Type	Vector
Reset	0000 0000h
Undefined Instructions	0000 0004h
Software Interrupt	0000 0008h
Prefetch Abort (instruction Prefetch memory abort)	0000 000Ch
Data Abort (data access memory abort)	0000 0010h
IRQ (interrupt)	0000 0018h
FIQ (fast interrupt)	0000 001Ch

3.2.8.3 Hard Reset

Any of the following conditions cause a StrongARM* core reset:

- The RESET_IN# pin is asserted asynchronously.
- The PCI_RST# pin is asserted and the PCI_CFN[0] pin is not asserted.
- The watchdog timer = 0 and the WE (watchdog enable) bit of the SA_CONTROL register = 1.

- The SA1200_RESET register is written from the StrongARM* core. Subfunctions may be reset individually via this register.
- A software reset from the PCI bus occurs.

During reset, the StrongARM* core stops executing instructions, asserts the RESET_OUT# pin, and performs idle cycles on the bus. When reset deasserts, the StrongARM* core does the following:

1. Overwrites R14_svc and SPSR_svc by copying the current values of the PC and CPSR into them. The values of the saved PC and CPSR are not defined.
2. Forces M[4:0]=10011 (32-bit supervisor mode) and sets the I and F bits in the CPSR.
3. Forces the PC to fetch the next instruction from address 0x0000 0000.

At the end of the reset sequence, the MMU, Icache, Dcache, and write buffer are disabled. Alignment faults are also disabled, and little endian mode is enabled. During power up, RESET_IN# must be asserted for 150 milliseconds after VDD and VDDx are stable to allow the internal 3.686 MHz oscillator to stabilize. After the negation of reset, the PLL begins its internally timed locking sequence. The assertion of reset is destructive; the state of the real-time clock and the contents of SDRAM are lost.

3.2.8.4 Abort

An abort can be signaled by the MMU through a data breakpoint or through a reference to reserved memory. An abort indicates that the current memory access cannot be completed or that a prespecified breakpoint address and (optionally) a data pattern has been reached. The StrongARM* core checks for an abort during memory access cycles. When aborted, the StrongARM* core responds in one of two ways:

1. If the abort occurred during an instruction prefetch (a prefetch abort), the prefetched instruction is marked as invalid but the abort exception does not occur immediately. If the instruction is not executed, for example, as a result of a branch being taken while it is in the pipeline, no abort occurs. An abort takes place if the instruction reaches the head of the pipeline and is about to be executed.
2. If the abort occurred during a data access (a data abort), the action depends on the instruction type.
    a. Single data transfer instructions (LDR, STR) abort with no registers modified.
    b. The swap instruction (SWP) is aborted as though it had not executed, though externally the read access may take place.
    c. Block data transfer instructions (LDM, STM) abort on the first access that cannot complete. If write-back is set, the base is NOT updated. If the instruction would normally have overwritten the base with data (for example, an LDM instruction with the base in the transfer list), the original value in the base register is restored.

When either a prefetch or data abort occurs, the StrongARM* core does the following:

1. Saves the address of the aborted instruction plus 4 (for prefetch aborts) or 8 (for data aborts) in R14_abt; saves CPSR in SPSR_abt.
2. Forces M[4:0]=10111 (abort mode) and sets the I bit in the CPSR.
3. Forces the PC to fetch the next instruction from either address 0x0C (prefetch abort) or address 0x10 (data abort).

To return after fixing the reason for the abort, use SUBS PC,R14_abt,#4 (for a prefetch abort) or SUBS PC, R14_abt,#8 (for a data abort). This restores both the PC and the CPSR, and retry the aborted instruction.

The abort mechanism allows a demand paged virtual memory system to be implemented when suitable memory management software is available. The processor is allowed to generate arbitrary addresses, and when the data at an address is unavailable, the MMU signals an abort. The processor traps into system software, which must work out the cause of the abort, make the requested data available, and retry the aborted instruction. The application program needs no knowledge of the amount of memory available to it, nor is its state in any way affected by the abort.

3.2.8.5 Undefined Instruction

If the StrongARM* core attempts to access a coprocessor other than coprocessor 15, it takes the undefined instruction trap. The coprocessor load (LDC), store (STC), and data (CDP) instructions also take the undefined instruction trap. Permissions are set so that access to coprocessor 15 is privileged except where protection is programmable with respect to the read buffer operations.

3.2.9 StrongARM* Core Debug Support

The StrongARM* core supports breakpoints that provide the user with the ability to stop execution after seeing specific data in either the instruction or data streams. Execution then proceeds to an exception routine during which the user may examine the internal state of the machine. This section describes the instruction and data breakpoint facilities. The breakpoints are enabled through additions to coprocessor 15.

3.2.9.1 Instruction Breakpoint

The instruction breakpoint allows the user to stop the processor execution after the execution of an instruction at a selected address. This address is programmed into the instruction breakpoint address and control register (IBCR) via the BREAKPOINT_DEBUG (Register 14). Access to this register is privileged. The IBCR register is 32 bits wide and contains the address value for the breakpoint, and a bit to enable the breakpoint.

When the breakpoint is enabled, the StrongARM* core executes until the instruction at this address is fetched and the fetch address equals the program counter (ignoring bits 0 and 1 of the address). At this point, the processor takes a prefetch abort exception. The interrupt routine must examine R14 (the saved program counter) to determine if the exception was caused by the breakpoint.

3.2.9.2 Data Breakpoint

The data breakpoint allows the user to stop the processor execution after a load or store operation to a particular address. Data Breakpoints are configured through four registers (DBAR, DBVR, DBMR, and DBCR) that are accessed via the BREAKPOINT_DEBUG (Register 14). Access to BREAKPOINT_DEBUG register is privileged. The data breakpoint address is programmed into the data breakpoint address register (DBAR) and is a full 32-bit value (to permit breakpoints on byte accesses).

For stores, the breakpoint condition may also be programmed to include a particular data pattern as well as the reference address. The data value is programmed by way of the data breakpoint value register (DBVR) and the data breakpoint mask register (DBMR). The DBVR is a 32-bit register containing the value against which the store data is compared. The data value can be further qualified through the data breakpoint mask register (DBMR). The DBMR is a 32-bit register containing mask information indicating which bits in the store data should be compared against the DBMR. A 1 in a particular bit position in the DBMR indicates the bit in the DBVR that should be compared against the store data to qualify the breakpoint. To cause a breakpoint on a store data value, the address breakpoint must also be enabled, otherwise no breakpoint occurs.

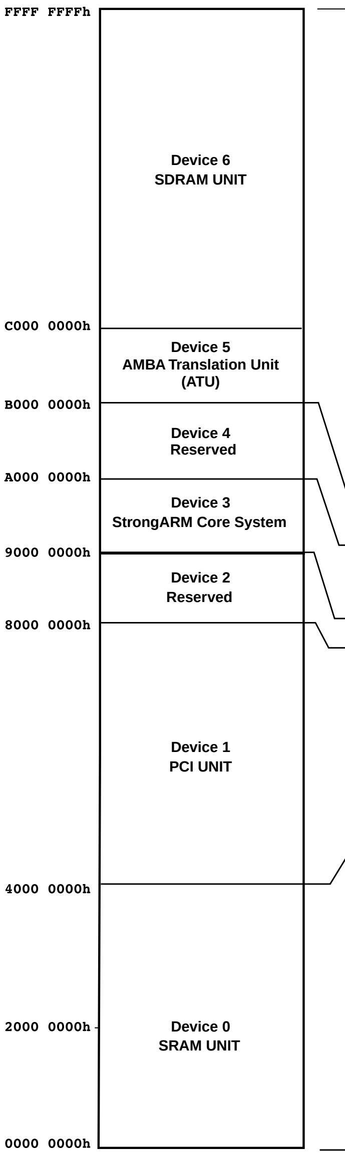
Breakpoints on loads are permitted only through an address match. Breakpoints on load address, store address, and store data are enabled and disabled through the data breakpoint control register (DBCR). A single bit is defined for each action. When a breakpoint is taken, the processor takes a data abort exception and sets bit 9 in the fault status register (FSR).

3.3 Memory Map

The 4 Gbyte StrongARM* address space is decoded into seven internal device select signals. Five of the device selects are used by the IXP1200. Figure 3-1 shows the StrongARM* core memory map.

- Device 0 (0000 0000h through 3FFF FFFFh) is dedicated to static memory devices (BootROM, SRAM, Aynchronous I/O, and SRAM CSRs). The StrongARM* core always boots at address 0000 0000h. The width of the physical devices on the SRAM Unit is always 32 bits.
- Device 1 (4000 0000h to 7FFF FFFFh) is dedicated to the PCI unit. The PCI configuration register and the PCI CSRs portions of this memory space are also mapped into the PCI address space.
- Device 3 (9000 0000h to 9FFF FFFFh) is dedicated to the StrongARM* system registers. These registers control the system clock, first level interrupt registers, UART, general-purpose I/O pins, and real-time clock.
- Device 5 (B000 0000h to BFFF FFFFh) is dedicated to the AMBA translation unit (ATU). The ATU provides an interface between the StrongARM* core and the FBI, scratchpad memory, microengine SRAM read transfer register and microengine local CSRs.
- Device 6 (C000 0000h to FFFF FFFFh) is dedicated to the SDRAM unit. The width of the physical devices on the SDRAM unit is always 64 bits. This space also includes the SDRAM CSRs.

Figure 3-1. StrongARM* Core Memory Map



SDRAM

Address Range	Description
FF00 0000 – FF00 0014	SDRAM Control Registers
D000 0000 – DFFF FFFF	SDRAM Prefetch Memory (256 Mbytes)
C000 0000 – CFFF FFFF	SDRAM non-Prefetch Memory (256 Mbytes)
A000 0000 – A000 4000	Cache Flush Area (16 Kbytes)

Scratch Pad Memory

Address Range: B004 4000 – B004 4FFF

FBI CSRs

Base Address: B004 0000

Microengine SRAM Transfer Register

Base Address	Microengine
B000 6800	Microengine 5
B000 6000	Microengine 4
B000 5800	Microengine 3
B000 5000	Microengine 2
B000 4800	Microengine 1
B000 4000	Microengine 0

Microengine CSRs

Base Address	Microengine
B000 2800	Microengine 5
B000 2000	Microengine 4
B000 1800	Microengine 3
B000 1000	Microengine 2
B000 0800	Microengine 1
B000 0000	Microengine 0

Intel® StrongARM* System Registers

Base Address: 9000 0000

PCI Unit

Address Range	Description
6000 0000 – 7FFF FFFF	PCI Memory Cycle Access
5401 0000 – 5FFF FFFF	Reserved
5400 0000 – 5400 FFFF	PCI I/O Cycle Access
53C0 0000 – 53FF FFFF	Reserved
5300 0000 – 53BF FFFF	PCI Type 0 Configuration Cycle Access
5200 0000 – 52FF FFFF	PCI Type 1 Configuration Cycle Access
4200 0400 – 51FF FFFF	Reserved
4200 0000 – 4200 03FF	Local PCI Configuration Space and PCI Unit CSRs
4000 0000 – 41FF FFFF	Reserved

SRAM Unit

Address Range**	Description
3840 0000 – 385F FFFF	SlowPort
3800 0080 – 3800 00FF	Command FIFO Test
3800 0000 – 3800 0028	SRAM CSRs
2400 0000 (See table ->)	Pop Command
2000 0000 (See table ->)	Push Command
1980 0000 – 19FF FFFF	Bit Test & Set
1900 0000 – 197F FFFF	Bit Test & Clear
1880 0000 – 18FF FFFF	Bit Write Set
1800 0000 – 187F FFFF	Bit Write Clear
1600 0000 – 167F FFFF	CAM Unlock
1400 0000 – 147F FFFF	Write Unlock
1200 0000 – 127F FFFF	Read Lock
1000 0000 – 107F FFFF	Read/Write
0000 0000 – 007F FFFF	BootROM

Descriptor List	Push Operations* (base addr)	Pop Operations
0	2000 0000	2400 0000
1	2080 0000	2480 0000
2	2100 0000	2500 0000
3	2180 0000	2580 0000
4	2200 0000	2600 0000
5	2280 0000	2680 0000
6	2300 0000	2700 0000
7	2380 0000	2780 0000

* To push descriptor, write to base addr + pointer. (Data written has no effect.)

**All unspecified SRAM addresses are reserved.

Notes:

- Devices 0 through 5 address data in longwords.
- Device 6 addresses data in quadwords.
- SRAM Transfer Registers: Thirty-two (32) Microengine specific SRAM Transfer registers are addressed by adding four times their absolute register address (see Table 2-4 of the Programmer's Reference Guide) to the base address of the Microengine being referenced. The core reads the Microengine SRAM Write Transfer Register and writes to the Microengine SRAM Read Transfer Register.

* Other names and brands may be claimed as property of others.

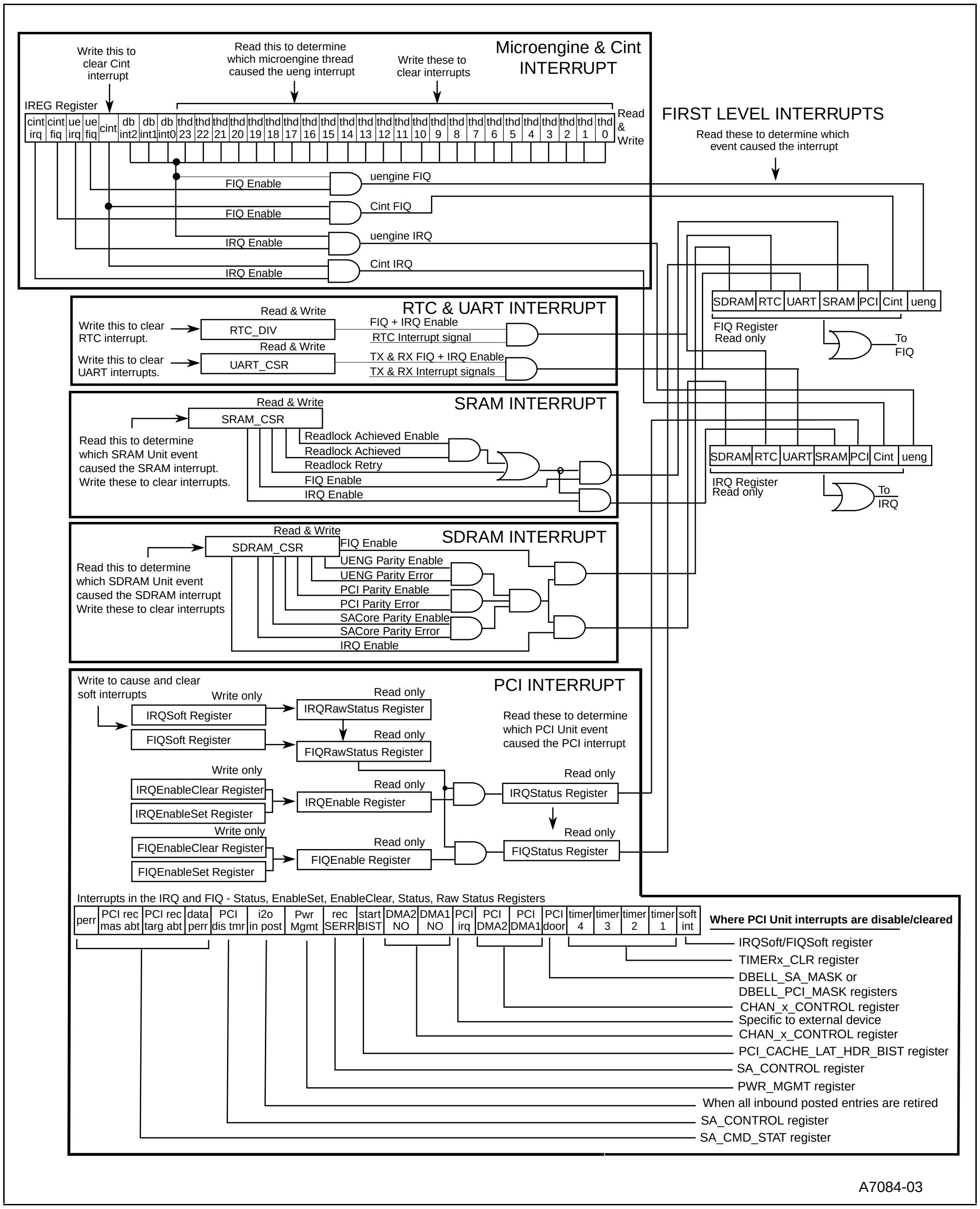
A9461-01

3.4 FIQ and IRQ Interrupts

The StrongARM* core supports two interrupt pins: FIQ and IRQ. FIQ has the higher priority. The StrongARM* core must enable the IRQ and FIQ interrupts by clearing the I and F bits of the Current Operating Status Register (CPSR). To ensure that a pending interrupt is taken, an interrupt-enabling write to CPSR (using the MSR instruction) must be separated from an interrupt-disabling write to the CPSR by at least two instructions.

The IXP1200 allows a number of sources to generate either an IRQ or FIQ. These sources are managed through a series of registers organized in the hierarchy shown in Figure 3-2.

Figure 3-2. FIQ and IRQ Interrupts



3.5 Internal Peripheral Units

This section describes the following peripherals that can be accessed only by the StrongARM* core.

- UART
- Four 24-bit timers
- Four general-purpose I/O pins
- Real-time clock

3.5.1 UART

The StrongARM* core supports a UART that provides a serial communication channel between the StrongARM* core and another device. The UART supports standard baud rates and data formats commonly used by computer RS-232-C ports. The UART supports transmit and receive pins. It does not support controls pins such as RTS, CST, DTR, DSR.

The UART is controlled using three registers:

- UART_CR: UART configuration register.
- UART_SR: UART status register.
- UART_DR: UART data register.

The UART can be configured to support the data formats shown in Table 3-5 via the UART_CR (UART control register).

Table 3-5. UART Supported Data Formats

Settings	Options
Data Size	5, 6, 7, 8 bits.
Stop Bit	1 or 2.
Parity	None, Even, Odd.
Standard Baud Rates	300 bps to 230.4 kbps (system clock = 3.6864 MHz).
StrongARM* Interrupts	Transmit and Receive.
Transmit Break (send all zeroes)	Yes.

The UART can also be enabled to generate an IRQ interrupt when the transmit FIFO is completely empty or when the receive FIFO contains at least one byte of valid data. Software can determine which event generated the interrupt by reading the UART_SR register. After the interrupt is serviced, software must clear the interrupt by writing to the UART_CR register.

As an alternative to using interrupts, software can poll the UART_SR to determine when it should read the receive FIFO or when it can write data to the Transmit FIFO. Three status bits are provided for transmit status, Transmit FIFO Full, Transmit FIFO empty and Transmit FIFO Ready. Transmit Ready indicates that there is room for at least one byte in the transmit FIFO. This is logically equivalent to the opposite of Transmit FIFO Full. Two status bits are provided for receive status, Receive FIFO Full, and Receive FIFO Ready. Receive Ready indicates that there is at least one byte in the Receive FIFO.

The UART transmit and receive FIFOs are mapped to the UART_DR (UART data Register). Both the Transmit and Receive FIFOs provide eight entries. A write to the UART_DR register places the data into the Transmit FIFO which then places data serially onto the TXD pin. Reading the UART_DR register delivers the receive data along with three error status bits affiliated with the data. The three error status bits indicate whether a Receiver Overrun, Error Framing Error, and/or Parity Error occurred during the read.

3.5.1.1 Receive Procedure

1. Configure the UART to the desired data settings by writing the UART_CR register.
2. Enable the receive interrupt in the UART_CR. An IRQ interrupt occurs when the receive FIFO contains at least one byte of data.
3. When a UART interrupt occurs, read the UART_SR to determine if the source of the interrupt was the transmitter (Transmit FIFO empty) or receiver (Receive FIFO Full).
4. If the interrupt source was the receiver, inspect the UART_SR data to determine whether there are any errors associated with the received data. Pay particular attention to the Receiver Overrun Error, as this indicates a break in the data sequence within the receive FIFO. The framing error and parity error apply only to the next byte to be read from the receive FIFO.
5. Read the data in the UART_DR.
6. Determine if the data contains a parity or framing error by looking at bits 8 and 9 of the data. If an overrun error occurred, bit 10 indicates the last valid data that was received before the overrun occurred.

3.5.1.2 Transmit Procedure

1. Configure the UART to the desired settings and enable the UART.
2. Write up to 8 bytes to the UART_DR.
3. Enable the transmit interrupt and select either an FIQ or IRQ interrupt in the UART_CR. An interrupt occurs when the transmit FIFO is empty.
4. When an interrupt occurs, read the data in the UART_SR to determine whether the source of the interrupt was the transmitter (Transmit FIFO empty) or receiver (Receive FIFO Full).
5. If the transmitter generated the interrupt, write eight more bytes into the UART.

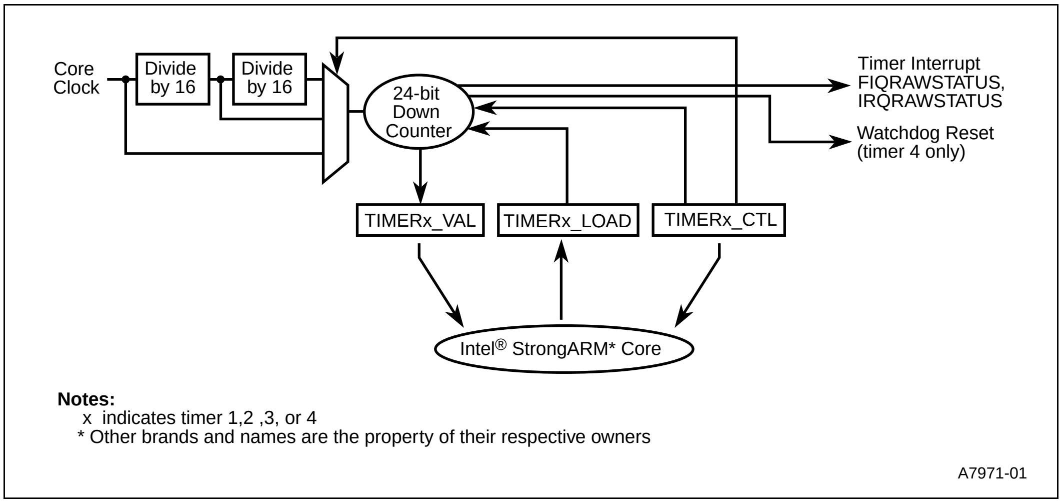
3.5.2 Timers

The IXP1200 supports four 24-bit timers that can be accessed only by the StrongARM* core. Each timer can be preloaded and either free run, or decremented to zero and then reloaded. Each timer is clocked in one of three ways:

- Core Clock
- Core Clock divided by 16
- Core Clock divided by 256

When a timer reaches zero, it generates an interrupt. The interrupt can be enabled or disabled in the IRQ_ENABLE/FIQ_ENABLE registers. These timers are physically located in the PCI unit, so they generate a PCI Interrupt to the StrongARM* core. The interrupt remains asserted until cleared by a write of any data to the associated TMER_CLR register. Figure 3-3 is a block diagram of the timer functions.

Figure 3-3. Timer Block Diagram


Core Clock
Divide by 16
Divide by 16
24-bit Down Counter
Timer Interrupt FIQRAWSTATUS, IRQRAWSTATUS
Watchdog Reset (timer 4 only)
TIMERx_VAL
TIMERx_LOAD
TIMERx_CTL
Intel® StrongARM* Core
Notes:
x indicates timer 1,2 ,3, or 4
* Other brands and names are the property of their respective owners
A7971-01

Timer 4 can be used as a watchdog timer. If the watchdog enable (WE) bit in the SA_CONTROL register is set, a reset sequence is initiated when timer 4 counts to zero.

System software can use the watchdog as follows. Set timer 4 for periodic interrupts and disable the interrupt on IRQ/FIQ. A periodic process (based on one of the other timers) would write to TIMER_4_LOAD. If that process ever fails to write to TIMER_4_LOAD within the countdown time, then the IXP1200 is reset. Once the watchdog enable bit is set, it can only be cleared by a chip reset.

3.5.3 Real-Time Clock (RTC)

The IXP1200 contains a real-time clock (RTC) that provides a general-purpose real-time reference for use by the StrongARM* core. Typically the RTC is programmed to increment the counter on each rising edge of an internally generated 1 Hz clock to provide a time base with a one-second resolution. The 1 Hz clock is generated by dividing the system clock (Typically 3.6864 MHz) by a fixed value of 128 and then by a programmable divisor that is set in the RTC_DIV register.

In addition to the counter, the RTC incorporates a 32-bit alarm count register. On each rising edge of the 1 Hz clock, the counter is compared to the alarm count register. If the values match and if the RTC interrupt is enabled, an interrupt is issued.

The RTC must be initialized by software upon power up. Thereafter, the counter remains valid until power is removed from the IXP1200. The value2 of the RTC registers is unaffected by a reset except for bits 16 to 18 in the RTC_DIV register. These bits are set to zero to clear any pending interrupts and to disable the RTC interrupt to the StrongARM* core. The StrongARM* core is responsible for enabling the interrupts after a reset.

The divider logic is programmable to allow the user to “trim” the counter to adjust for inherent inaccuracies in the system clock frequency. At power-up, the registers that determine the trim amount contain zeros, effectively disabling the trim logic.

Table 3-6 lists the registers used by RTC.

Table 3-6. RTC Registers

RTC Register	Description
RTC_CNTR	RTC Count value: Holds the current one second counter value.
RTC_DIV	RTC Divisor: Holds the programmable divisor used to produce 1Hz clock.
RTC_TINT	Trim Interval Count: This count decrements once every second. When the count equals zero, the next RTC count value used equals RTC_DIV minus RTC_TVAL.
RTC_TVAL	Trim Value: Specifies the number of clocks to trim.
RTC_ALM	Alarm Count: When this register equals RTC_CNTR an FIQ interrupt can be generated.

3.5.3.1 RTC Setup Procedures

The following procedure describes the setup of the RTC.

1. Write the following to the RTC_DIV register:
   a. An appropriate divisor. If the system clock frequency is 3.6864 Mhz, this value is 7080h).
   b. Clear the write enable bit in this register to indicate that this write is to the divisor field.
2. Write the start value for the RTC_CNTR register into the RTC_ALM register.
3. If you choose to implement trimming, write the proper trim interval to the RTC_TINT register. The trim procedure is in Section 3.5.3.3.
4. Write the following to the RTC_TVAL register:
   a. If you choose to implement trimming, the proper value should be provided. If you do not choose to implement trimming, this should be written with zero.
   b. If the prescaler (divide by 128) is to be used (as is the usual case), set the prescaler bit. Otherwise, clear it.
   c. Set the load bit to write the values in the RTC_DIV, RTC_ALM, RTC_TINT, and RTC_TVAL registers to the RTC working registers. The value in the RTC_ALM is loaded into the RTC_CNTR working register.
5. Clear the load bit in the RTC_TVAL register while maintaining the proper trim value in the register.

3.5.3.2 Using the RTC Alarm

The RTC alarm can issue either an IRQ or FIQ interrupt to the StrongARM* core whenever the value in the RTC_CNTR equals the value in the RTC_ALM register. The following procedure outlines the setup of the RTC Alarm:

1. Write the desired alarm value to the RTC_ALM registers.
2. Enable the RTC interrupt by writing to the RTC_DIV register. The write enable bit in the RTC_DIV register must be set when writing to the interrupt fields of this register.

3. When an interrupt occurs, the interrupt service routine must clear the interrupt by writing to the interrupt clear bit in the RTC_DIV register. To clear the interrupt, the write enable bit in the RTC_DIV register must be set.

3.5.3.3 Determining the Trim Values

The inherent inaccuracies of oscillators, aggravated by varying capacitance of the board connections, cause inaccuracies in the 1Hz time base. The RTC supports an automatic periodic trim adjustment in the 1 Hz clock period that improves the accuracy of the RTC. Over time these small inaccuracies of 1 Hz time base add up. When trimming is enabled, every RTC_TINT seconds the second count value (which is usually equal to RTC_DIV) is set to equal RTC_DIV minus RTC_TVAL.

The oscillator frequency is measured to be freq(meas) = 3,688,243 Hz. We arbitrarily pick a TINT value of 15 seconds.

Example 3-2. Automatic Periodic Trim Adjustment

```
RTC_DIV     = (freq(meas)/128)
            = 28814 (708Eh)
RTC_TINT    = Fh (15 seconds)
RTC_TVAL    = (freq(meas) - (RDIV x 128)) x RTINT
            = (3,688,243 -(28814 x 128)) x 15
            = 765 (2FD)
```

3.5.4 General Purpose I/O (GPIO)

The IXP1200 supports four general-purpose I/O pins that are controlled by the StrongARM* core. Each pin can be configured as either an input or an output via the GPIO_EN register. Regardless of whether the pin is configured as an input or output, the GPIO_DATA register contains the data that is present on the GPIO pins. The StrongARM* core writes to those register bits that are configured as outputs and reads the register bits that are configured as inputs.

At reset, the state of GPIO[3] is used to configure the BootROM Bus width. If GPIO[3] is pulled low, the BootROM Bus width is configured to 32 bits. If GPIO[3] is pulled high, the BootROM Bus width is 16 bits. The FBI unit uses some or all of the GPIO pins (as determined by the GPIO_EN register) to control the IX Bus in certain modes. Table 3-7 shows how the GPIO pins are used in the different IX Bus modes.

Table 3-7. GPIO Pins and IX Bus Modes

Mode	GPIO[0]	GPIO[3:1]
Bidirectional mode (2 MAC mode)	Owned by the FBI unit and used as flow control for MAC 1.	Owned by StrongARM* core as GPIOs.
Bidirectional mode (3-7 MAC mode)	Owned by the StrongARM* core as a GPIO.	Owned by StrongARM* core as GPIOs.
Unidirectional mode (2 MAC mode)	Owned by the FBI unit and used as flow control for MAC 1.	Owned by FBI unit and used as transmit FPS[2:0]
Unidirectional mode (3-4 MAC mode)	Owned by the FBI unit and used as an active low enable for an external decoder for the PORTCTL[3:2] signals.	Owned by FBI unit and used as transmit FPS[2:0]

3.6 Boot Sequence

Boot Sequence tasks must be performed by the IXP1200 after reset for proper processor functioning. The boot sequence tasks configure IXP1200 resources to a determined state by writing predetermined values to certain registers. Some register settings are determined by the components selected, such as SDRAM, SRAM, and BootROM. Other register settings are determined by the desired processor performance and system configuration.

The resources that must be configured after reset are the Phase-Locked Loop (PLL), the SRAM controller, the SDRAM controller, and the Memory Management Unit (MMU). There are other resources that if used during the boot sequence must be configured at this time. They are the UART and the PCI Interface.

For a more detailed description of the registers and their settings, please refer to the appropriate sections in the *IXP1200 Network Processor Hardware Reference Manual* and the *IXP1200 Network Processor Programmer's Reference*.

The configuration tasks must be performed in the following sequence.

1. **Configure PLL.** Configure PLL to the desired core clock frequency. Changing this value will affect other configuration settings that are specified in units of the core clock frequency. The register that configures the PLL is:

   PLL_CFG

2. **Configure Clock Switching.** Configure test, clock, and idle control operations.

   StrongARM Coprocessor 15 - TEST_CLOCK_AND_IDLE_CONTROL

3. **Configure SRAM.** Configure the SRAM controller. It is important that these registers be programmed in the order given below. The registers that configure the SRAM controller are:

   SRAM_CSR

   SRAM_SLOW_CONFIG

   SRAM_BOOT_CONFIG

   SRAM_SLOWPORT_CONFIG

4. **Release from Reset.** After reset, the SDRAM controller is not automatically brought out of reset. For this unit to function it must be brought out of reset. This is done by configuring the following register:

   IXP1200_RESET

5. **Configure SDRAM.** Configure the SDRAM controller. This is done through a sequence of register writes. Some of these register settings are defined in terms of the core clock frequency. In this sequence, it is important that the SDRAM_CSR be written last. In doing so, the new parameters will updated in the SDRAM controller. It is recommended that these registers not be reprogrammed after the initial boot sequence. The registers that configure the SDRAM controller are:

   SDRAM_MEMCTL_0

   SDRAM_MEMCTL_1

   SDRAM_INIT

   SDRAM_CSR

6. **Configure and Enable MMU.** Configure the Memory Mapped Unit, Cache, and Buffer. This is done by configuring the following register:

   StrongARM Coprocessor 15 - CONTROL_CP15

7. **Configure PCI.** If loading an image over the PCI during the Boot Sequence is required, the following registers must be configured in the order specified for PCI operations to function:

   PCI_ADDR_EXT

   DRAM_BASE_ADDR_MASK

   If the IXP1200 is the PCI Central Function device, configure the following four registers:

      PCI_MEM_BAR

      PCI_IO_BAR

      PCI_DRAM_BAR

      PCI_CMD_STAT

   SA_CONTROL

8. **Configure Serial Port.** If loading an image over the serial port during the Boot Sequence is required, the following register must be configured.

   UART_CR

Microengines 4

4.1 Overview

Each IXP1200 Network Processor contains six programmable 32-bit RISC processors. These RISC processors are referred to as Microengines and are the distinguishing feature that separates the IXP1200 Network Processor Family from other microprocessors. The Microengines support a 32-bit RISC instruction set tailored to networking and communications applications. The Microengines operate at the IXP1200 core frequency and all instructions execute in a single cycle.

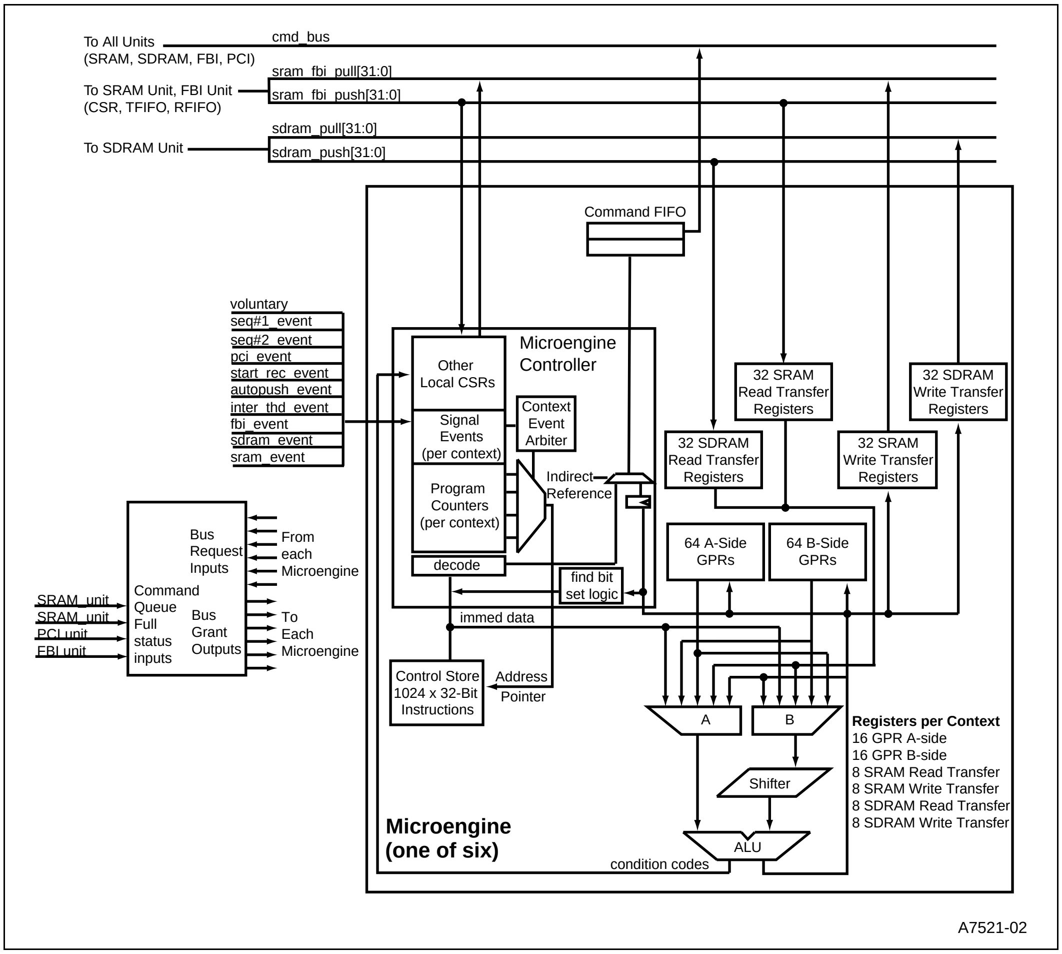
The six Microengines each provide the following features:

- Hardware multithread support for four contexts
- Programmable 1K instruction Control Store (program memory)
- 128 32-bit general purpose registers
- 128 32-bit Transfer Registers (for transferring data into and out of the Microengines)
- Powerful ALU and Shifter capable of performing an ALU and shift operation in a single cycle

4.2 Microengine Block Diagram

This section provides an overview of the Microengine block diagram shown in Figure 4-1.

Figure 4-1. Microengine Block Diagram



4.2.1 Multithread Support

Hardware multithread support allows four separate programs to share execution time on a Microengine. When a program is not executing, each program context is preserved in hardware through separate program counters, signal event states, and relatively addressed register set (General Purpose Registers (GPRs) and Transfer Registers) for each program. When a program is put to sleep, a context switch occurs and another program begins executing. The overhead associated with switching contexts is a maximum of one cycle, however a deferred instruction can be used to eliminate this overhead.

Context Arbiter logic determines which Microengine thread will be allowed to run when a thread puts itself to sleep. The Context Event Arbiter makes the decision based on signal events from the other functional units.

4.2.2 Control Store

Each Microengine contains a programmable Control Store that holds the microcode program. The four program threads associated with the Microengine share the Control Store. The Control Stores support 1024 32-bit instructions and must be programmed by the StrongARM* core upon system initialization.

4.2.3 128 General-Purpose Registers (GPRs)

Each Microengine supports 128 32-bit GPRs. The GPRs can be addressed using relative addressing or absolute addressing. Relative addressing divides the GPRs among the Microengine threads so that each thread has exclusive access to a subset of GPRs (32 maximum). Absolute addressing allows a register to be shared among all the threads within a Microengine.

4.2.4 128 Transfer Registers

Data is moved into and out of the Microengines via the Transfer Registers. Each Microengine supports 128 32-bit Transfer Registers. This register set is divided into 32 SRAM Read, 32 SRAM Write, 32 SDRAM Read, and 32 SDRAM Write Transfer Registers. Each register subset connects to the other functional units via four separate 32-bit data buses. The SDRAM registers are used to move data between the SDRAM Unit and the Microengine. The SRAM Transfer Registers are used to move data between the SRAM Unit or FBI Unit and the Microengine.

The Transfer Registers can be addressed using relative addressing or absolute addressing. Relative addressing divides the Transfer Registers amongst the Microengine threads so that each thread has exclusive access to a subset of Transfer Registers (8 SRAM read, 8 SDRAM read, 8 SRAM write, 8 SDRAM write). Absolute addressing allows a register to be shared among all the threads within a Microengine.

4.2.5 ALU and Shifter

The Microengines contain a powerful 32-bit ALU and Shifter capable of performing an ALU and shift operation in a single cycle. The two inputs of the ALU (A and B) can operate on data supplied by the SRAM/FBI read Transfer Registers, SDRAM read Transfer Registers, GPRs, and immediate data within the instruction. The ALU can perform addition, subtraction, and logical operations as well as generate sign, zero, and carry out condition codes based on these operations.

4.2.6 Command Bus Arbiter

The six Microengines issue references to the other functional units within the IXP1200 (SRAM, SDRAM, FBI, and PCI) via a Command Bus. All six Microengines share the Command Bus. When a Microengine thread executes a reference instruction, a command is placed into a two-entry Command FIFO within the Microengine. The Command Bus Arbiter arbitrates between the Command FIFO within each Microengine to determine which one will be allowed access to the shared Command Bus.

4.2.7 Local CSRs

Each Microengine contains a set of control and status registers. The StrongARM* core accesses the register to program the Control Store and provide control and status information when debugging. A Microengine can access its own set of local CSRs (except USTORE_ADDRESS, USTORE_DATA, or ALU_OUTPUT) using the **local_csr_rd** and **local_csr_wr** Microengine instructions. These registers are described in Table 4-18.

4.3 Microengine Instruction Set

Table 4-1 lists the RISC instructions supported by the Microengines. Each instruction executes in a single cycle. The instructions are described in more detail in the *IXP1200 Network Processor Programmers Reference Manual.*

Table 4-1. Summary of Microengine Instructions (Sheet 1 of 2)

Instruction	Description
Arithmetic, Rotate, and Shift Instructions	
ALU ALU_SHF DBL_SHIFT	Perform an ALU operation. Perform an ALU and shift operation. Concatenate two longwords, shift the result, and save a longword.
Branch and Jump Instructions	
BR, BR=0, BR!=0, BR>0, BR>=0, BR<0, BR<=0, BR=cout, BR!=cout	Branch on condition code.
BR_BSET, BR_BCLR	Branch on bit set or bit clear.
BR=BYTE, BR!=BYTE	Branch on byte equal or not equal.
BR=CTX, BR!=CTX	Branch on current context.
BR_INP_STATE	Branch on event state (e.g., SRAM done).
BR_!SIGNAL	Branch if signal deasserted.
JUMP	Jump to label.
RTN	Return from a branch or a jump.
Reference Instructions	
CSR	CSR reference.
FAST_WR	Write immediate data to the thd_ done CSRs.
LOCAL_CSR_RD, LOCAL_CSR_WR	Read and write CSRs .
R_FIFO_RD	Read the receive FIFO.
PCI_DMA	Issue a request to the PCI Unit.
SCRATCH	Scratchpad reference.
SDRAM	SDRAM reference.
SRAM	SRAM reference.
T_FIFO_WR	Write to the transmit FIFO.

Table 4-1. Summary of Microengine Instructions (Sheet 2 of 2)

Instruction	Description
Local Register Instructions	
FIND_BSET, FIND_BSET_WITH_MASK	Determine position number of first bit set in an arbitrary 16-bit field of a register.
IMMED	Load immediate word and sign extend or zero fill with shift.
IMMED_BO, IMMED_B1, IMMED_B2, IMMED_B3	Load immediate byte to a field.
IMMED_WO, IMMED_W1	Load immediate word to a field.
LD_FIELD, LD_FIELD_W_CLR	Load byte(s) into specified field(s).
LOAD_ADDR	Load instruction address.
LOAD_BSET_RESULT1, LOAD_BSET_RESULT2	Load the result of a find_bset or find_bset_with_mask instruction.
Miscellaneous Instructions	
CTX_ARB	Perform context swap and wake on event.
NOP	Perform no operation.
HASH1_48, HASH2_48, HASH3_48	Perform 1, 2, or 3 48-bit hash operations.
HASH1_64, HASH2_64, HASH3_64	Perform 1, 2, or 3 64-bit hash operations.

4.4 Execution Pipeline

Each Microengine instruction is pipelined through a five-stage execution unit. Table 4-2 shows descriptions of the work performed in each pipeline stage.

Table 4-2. Execution Pipeline

Pipeline Stage	Description
P0	Lookup of instruction
P1	Initial instruction decode and formation of the source register address
P2	Read of operands from source registers
P3	Perform ALU, shift, or compare operations and generate the condition codes
P4	Write result to destination register.

Once the execution pipeline is filled with instructions, an instruction is executed every cycle. Instructions such as branch, jump/return, and context swapping result in a branch decision that may introduce aborted instruction to the pipeline that reduce the efficiency of the Microengines.

A datapath is provided so that if an instruction in P2 specifies an operand that is modified by an instruction in P3, the value of the operand that is used is the value modified by the instruction in P3 rather than the stale data that was fetched in P2.

4.5 Branch Decisions

Any instruction that makes a branch decision may cause one or more instructions in the execution pipeline to be aborted. This section described the branch decision and the effect it has on the execution pipeline. The IXP1200 supports three mechanisms to reduce or eliminate the aborted cycles introduced as a result of the branch decision: deferred branches, setting condition codes earlier, and branch guessing. These topics are discussed after this section.

An instruction that makes a branch decision does so based on the result of an operation that occurs in either the P1, P2, or P3 instruction pipeline stage. The specific pipeline stage where the decision is made depends on the instruction. For this discussion, these instructions are classified into three classes. Table 4-3 lists the instructions by class.

Table 4-3. Instructions Categorized by Class

Class 3	Class 2	Class 1	
br_bclr and br_bset	br=0	br	sdram
br=byte and br=!byte	br!=0	br=ctx	sram
jump	br>0	br!=ctx	hash1_48
rtn	br>=0	ctx_arb	hash2_48
br_!signal	br<0	csr	hash3_48
br_inp_state	br<=0	r_fifo_rd	hash1_64
	br=cout	t_fifo_wr	hash2_64
	br!=cout	scratch	hash3_64

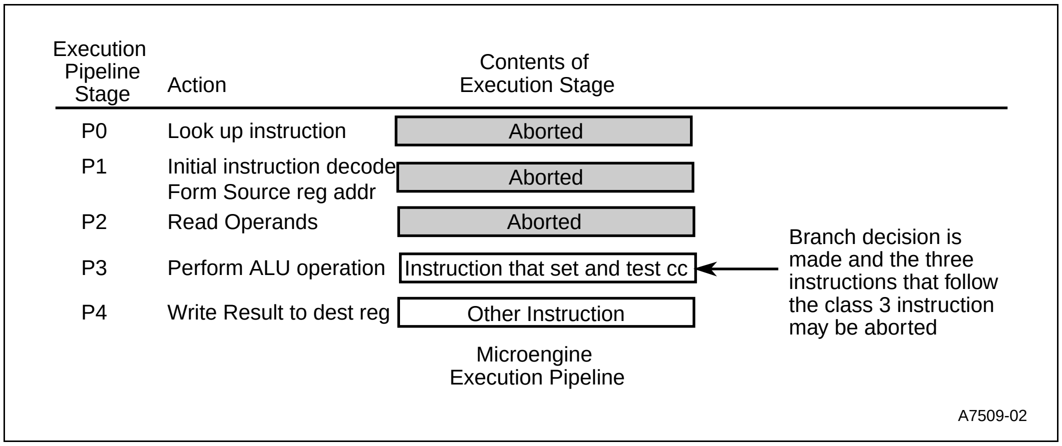
4.5.1 Class 3 Instructions

Class 3 instructions always make the branch decision in the P3 execution pipeline stage. Figure 4-2 illustrates a class 3 instruction causing three instructions to be aborted in the Execution Pipeline. Class 3 instructions are listed in Table 4-4.

Table 4-4. Class 3 Instructions

Instruction	Why Branch Decision Is Made In P3
br_bclr and **br_bset**	ALU operation must be performed to determine if the bit is clear or set.
br=byte and **br=!byte**	ALU operation must be performed to determine if the bytes match.
jump	ALU operation must be performed to determine jump address
rtn	ALU operation must be performed to determine return address
br_!signal	Whenever a signal event is tested, the signal event bit is automatically cleared if it is set. The branch decision is not made until P3 to ensure that the signal event bit is not cleared until we are sure that an another instruction that proceeds it in the execution pipeline will not abort the instruction. If an instruction makes it to P3, the instruction will not be aborted.
br_inp_state	Branch decision is made to ensure that the br_inp_state instruction is not aborted by an instruction that precedes it. If an instruction makes it to P3, the instruction will not be aborted.

Figure 4-2. Class 3 Branch Decision


Execution Pipeline Stage
Action
Contents of Execution Stage
P0
Look up instruction
Aborted
P1
Initial instruction decode
Form Source reg addr
Aborted
P2
Read Operands
Aborted
P3
Perform ALU operation
Instruction that set and test cc
P4
Write Result to dest reg
Other Instruction
Branch decision is made and the three instructions that follow the class 3 instruction may be aborted
Microengine Execution Pipeline
A7509-02

4.5.2 Class 2 Instructions

Class 2 instructions make the branch decision based on condition codes that are generated by another instruction. These instructions listed in Table 4-5.

Table 4-5. Class 2 Instructions

Class 2 instructions			
br=0	**br>0**	**br<0**	**br=cout**
br!=0	**br>=0**	**br<=0**	**br!=cout**

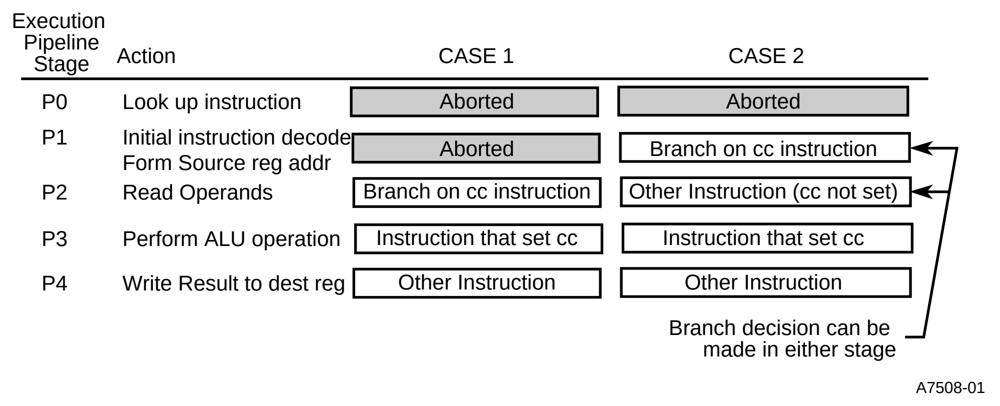
Since condition codes are not generated until the P3 stage, the branch decision can not be made until the instruction that sets the condition codes is in P3 and the branch instruction is in either the P1 or P2 stages. Consider the following cases:

Case 1: If the branch instruction is in the P2 stage when the instruction that generates the condition codes is in the P3 stage, the branch decision causes two instructions to be aborted.

Case 2: If the branch instruction is in the P1 stage when the instruction that generates the condition codes is in the P3 stage, the branch decision causes one instruction to be aborted.

The branch decision can not be made in P0 since the initial instruction decode performed in P1 must occur first.

Figure 4-3. Class 2 Branch Decision



Execution Pipeline Stage	Action	CASE 1	CASE 2
P0	Look up instruction	Aborted	Aborted
P1	Initial instruction decode Form Source reg addr	Aborted	Branch on cc instruction
P2	Read Operands	Branch on cc instruction	Other Instruction (cc not set)
P3	Perform ALU operation	Instruction that set cc	Instruction that set cc
P4	Write Result to dest reg	Other Instruction	Other Instruction

4.5.3 Class 1 Instructions

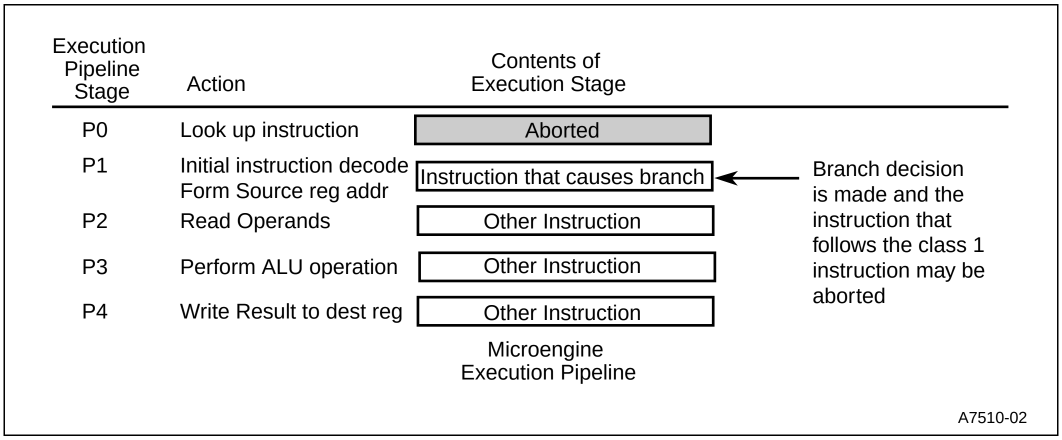
Class 1 instructions make the branch decision in the P1 stage, after the initial decoding of the instruction. Once the instruction is known, (and in the case of **br=ctx**, and **br!=ctx** the context is checked) all the information is available to make the branch decision.

Class 1 instructions can be classified into two groups: branch instructions and context switch instructions. The context switch instructions change the execution contexts as well as branch to the next instruction that is to be executed in the context to which it is switching. Table 4-6 lists the class 1 instructions by category.

Table 4-6. Class 1 Instructions

Branch Instruction	Context Switch Instructions	
br	**ctx_arb**	**hash1_48**
br=ctx	**csr**	**hash2_48**
br!=ctx	**r_fifo_rd**	**hash3_48**
	t_fifo_wr	**hash1_64**
	scratch	**hash2_64**
	sdram	**hash3_64**
	sram	

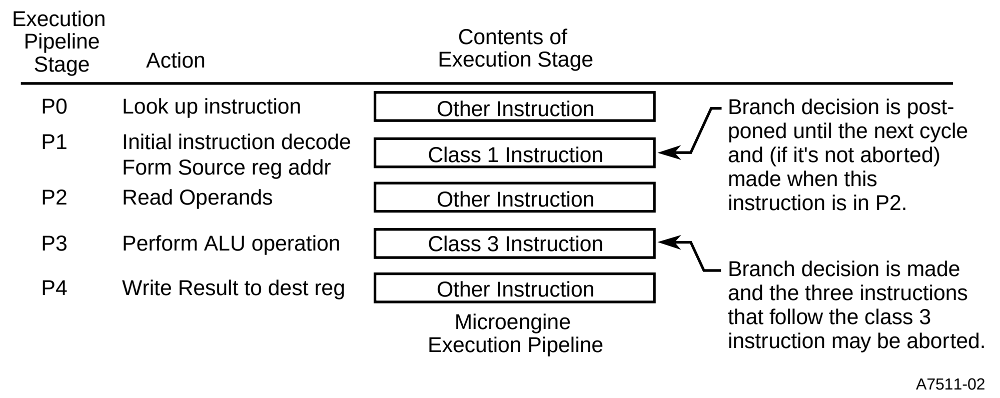
Figure 4-4. Class 1 Branch Decision


Execution Pipeline Stage
Action
Contents of Execution Stage
P0
Look up instruction
Aborted
P1
Initial instruction decode
Form Source reg addr
Instruction that causes branch
P2
Read Operands
Other Instruction
P3
Perform ALU operation
Other Instruction
P4
Write Result to dest reg
Other Instruction
Branch decision is made and the instruction that follows the class 1 instruction may be aborted
Microengine Execution Pipeline
A7510-02

4.5.4 Postponed Branch Decision

The branch decision logic within each Microengine makes one branch decision per cycle. There may be cases where two branch decisions can be made in the same cycle. For example, consider the case where a class 3 instruction is in P3 at the same time as a class 1 instruction is in P0 (Figure 4-5). The branch decision logic makes a decision on the instruction in P3 and defers the decision for the instruction in P1 until the next cycle (when that instruction propagates to P2).

Figure 4-5. Postponed Branch Decision



Execution Pipeline Stage	Action	Contents of Execution Stage
P0	Look up instruction	Other Instruction
P1	Initial instruction decode Form Source reg addr	Class 1 Instruction
P2	Read Operands	Other Instruction
P3	Perform ALU operation	Class 3 Instruction
P4	Write Result to dest reg	Other Instruction

Microengine Execution Pipeline
Branch decision is postponed until the next cycle and (if it's not aborted) made when this instruction is in P2.
Branch decision is made and the three instructions that follow the class 3 instruction may be aborted.
A7511-02

4.5.5 Deferred Branch

Before beginning the discussion of deferred instructions, it should be noted that the IXP1200 Assembler supports an optimization option that automatically performs deferred branch optimization. It does so by recognizing when instructions can and can’t be deferred and rearranges the instruction accordingly. This section is provided for those programmers who understand the result of the assembly optimization and choose to manually insert deferred tokens.

The purpose of a deferred branch is to reduce or eliminate aborted instructions in the execution pipeline. In a deferred branch, an instruction that follows a branch decision is allowed to execute before the branch takes effect (i.e., the effect of the branch is “deferred” in time). If useful work can be found to fill the wasted cycles after the branch instruction, the branch latency can be hidden.

Deferred branches are supported using the “defer” optional token within an instruction. The IXP1200 Assembler can perform the deferred instruction option manually or automatically. The example below demonstrates how a deferred operation eliminates aborted instructions to improve compute efficiency. The first table shows a program that does not use the deferred branch. The second table shows the improved compute efficiency of the same program when deferred branches are used.

Status: E = executed A = aborted N = not executed due to branch

Status	Microcode example with NO deferred instructions
E	immed[$xfer0, 0xff, <<8]
E	immed[$xfer0, 0xff, <<8]
E	alu_shf[gpr1, gpr2, +, $xfer1, <<1]
E	br=0[label1#]
A	alu_shf [gpr3, 0, B, gpr2, <<16] ; Aborted due to branch
A	ld_field_w_clr [gpr1, 0001, $xfer2, >>8] ; Aborted due to branch
--	label1#:
E	alu_shf [gpr3, 0, B, gpr4, <<16]

Status	Microcode example with deferred instructions
E	alu_shf_[gpr1, gpr2, +, $xfer1, <<1]
E	br=0_[label1#], defer[2] ; BRANCH LATENCY FILL OPTIMIZATION: ; the microword below was "pushed" down 2 positions
E	immed[$xfer0, 0xff, <<8] ; BRANCH LATENCY FILL OPTIMIZATION: ; the microword below was "pushed" down 2 positions
E	immed[$xfer0, 0xff, <<8]
N	alu_shf_li[gpr3, 0, b, gpr4, <<16]
N	ld_field_w_clr_b[gpr1, 0001, $xfer2, >>8]
--	label1#:
E	alu_shf_li[gpr3, 0, b, gpr2, <<16]

4.5.6 Setting Condition Codes Early

Class 2 instructions can be arranged so that the condition codes upon which the branch decision is made are set two or more instructions before the branch (refer to the class 2, case 2 example in Section 4.5.2). This allows one aborted instruction to be eliminated because the branch decision can be made one cycle earlier.

4.5.7 Guess Branch

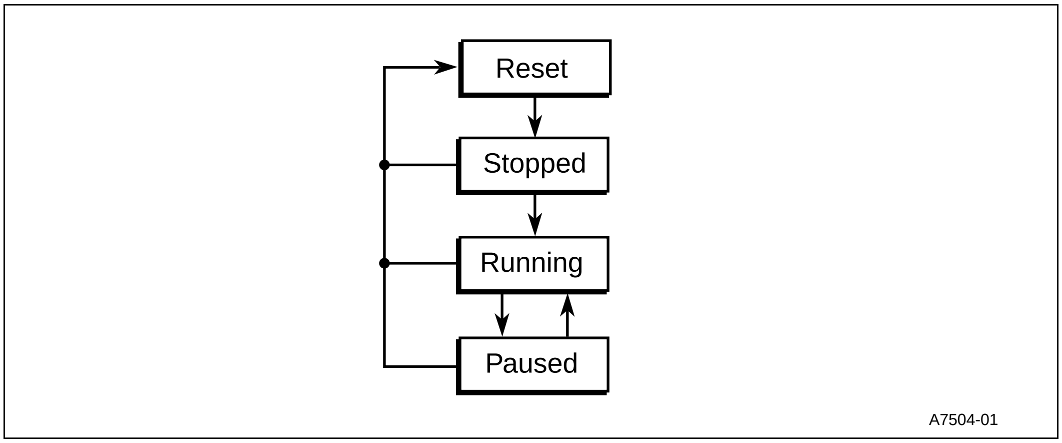
The guess_branch optional token allows instructions to be prefetched from the “branch-taken” path before it makes the actual branch decision. If it is omitted, the operation assumes a branch-not-taken guess. This qualifier is valid on **br_bset**, **br_bclr**, **br_inp_state**, **br_!signal** and on conditional branches affected by ALU condition codes that have been set during the previous instruction. It is not valid on: **br=ctx**, **br!=ctx**, **br=byte**, **br!=byte**, **jump**, or **rtn**. Note, however, that **br=byte** unconditionally guesses the branch taken path, and **br!=byte** unconditionally guesses the branch not taken path.

Supports guess_branch			Does Not Support guess_branch	
br_bset	br=cout	br<0	br	br!=byte
br_bclr	br>0	br=0	br=ctx	jump
br_inp_state	br!=cout	br<=0	br!=ctx	rtn
br_!signal	br>=0	br!=0	br=byte	

4.6 Execution States

A Microengine can be placed in one of four execution states. Figure 4-6 defines the Microengine states and shows the possible transition between states. These states are described in the following sections.

Figure 4-6. Execution States



4.6.1 Reset State

All the Microengines are automatically placed into the Reset state when a system reset is performed. The StrongARM* core or a PCI device may place individual Microengines into the Reset state via the IXP1200_RESET register in the PCI Unit. When a Microengine is placed into reset, the following events occur:

- The Microengine does not execute instructions.
- The Command FIFO is cleared.
- Some bits within the local CSRs are set to their reset state while others are indeterminate. The reset values are defined in the *IXP1200 Network Processor Programmer's Reference Manual*.
- The Control Store, GPRs, and Transfer Registers are in an indeterminate state

4.6.2 Stopped State

When a Microengine is released from the Reset state, it is placed into the Stopped state. The Microengine retains the state caused by a reset, however the StrongARM* core may program the local CSRs and Control Store as well as read and write to the GPRs and Transfer Registers. Since the reset condition disables all four contexts (by writing zeroes to the CTX_ENABLES registers), the Microengines will not execute instructions when it is in the stopped state.

4.6.3 Running State

A Microengine is placed in the Running state when it is not in reset and one or more contexts are enabled via the CTX_ENABLES registers. The StrongARM* core can place the Microengines into the Running state either from the Stopped or the Paused states.

Before the StrongARM* core transitions a Microengine from the Stopped to the Running state, it should first initialize the local CSRs and program the Control Store. The following list describes the basic tasks required to transition a Microengine from the Reset to the Running state.

1. Program the Control Store (refer to Section 4.7).
2. Set the active context number in the ACTIVE_CTX_STS register to the desired context that should start running first (typically context 0).
3. Set the next context number in the CTX_ARB_CNTL register to the desired context that should start running after the first context (typically context 1).
4. Set the CTX_ENABLES register so that the desired contexts are enabled to run.

When the StrongARM* core transitions a Microengine from the Paused to the Running state, it is expected the Microengines local CSRs, Control Store, GPRs and Transfer Registers will all be in a known state and that the transition only requires that the contexts be enabled via the CTX_ENABLES register.

4.6.4 Paused State

A Microengine is placed in the Paused state by disabling the context enables when a Microengine is running (via the CTX_ENABLES register). A Microengine continues to run until it executes an instruction that causes a change of context. Since all the contexts have been disabled, the Microengine gracefully stops and retains the state of its local CSRs, Control Store, GPRs, and Transfer Registers. After the StrongARM* core disables all the contexts, it can poll the AB bit in the ACTIVE_CTX_STS register to determine when the Microengine changes contexts and enters the Paused state.

The StrongARM* core can place a Microengine into the Paused state or a Microengine can place itself into the Paused state. However, a Microengine can not place another Microengine into the Paused state. A Microengine places itself into the Paused state by writing to the CTX_ENABLES register using the **local_csr_wr** instruction. The StrongARM* core can remove a Microengine from the Paused state by writing to the CTX_ENABLES register.

When a Microengine is placed into the Paused state, the previous, current, and next context are reported in the local CSRs. The active context (reported in the ACTIVE_ARB_CNTL register) contains the context that was paused. The previous context (reported in the ACTIVE_CTX_STS register) contains the context that was running before the active context. The next context (reported in the ACTIVE_CTX_STS register) contains the context that is enabled to run after the active context.

4.7 Programming the Microengines

The StrongARM* core should be programmed to load each Control Store upon system initialization. Intel provides a Program Loader that can be used to program the Control Stores. The Program Loader is a library of C functions that loads program images from an object file (created by the IXP1200 assembler) into a Microengine Control Store, updates images with application shared symbol pointers, and initializes Microengine registers. The Program Loader is described in the *IXP1200 Network Processor Software Reference Manual*.

The steps taken by the StrongARM* core to read and write to the Control Store are listed below for programmers that choose to perform these tasks without using the Program Loader libraries.

1. Place the Microengines into the Reset or Paused state.
2. Write the USTORE_ADDRESS register with a Control Store address, making sure the Control Store Enable (CSE) bit in the register is set.
3. Writing an instruction to the USTORE_DATA local CSR loads the instruction into the Control Store.
4. Reading the USTORE_DATA local CSR returns the current instruction at the Control Store address.
5. Repeat steps 2 and 3 for each Control Store address to program the entire Control Store.

4.7.1 Starting Point of Program Execution

Each Microengine thread begins executing at the address specified in the CTX_PC field of the CTX_n_STS local CSR. The StrongARM* core must set this field before a Microengine is removed from reset since its value is indeterminate.

When a programmer designs a Microengine program so that two or more threads execute different programs located in the Control Store, the **br=ctx** or **br!=ctx** instructions can be used to begin program execution. For example, the following three instructions located at the beginning of a Control Store will cause each thread to begin executing at different locations within the Control Store.

```
br=ctx [0, thread_0_start#];instruction format: br=ctx[ctx,label#]
br=ctx [1, thread_1_start#];where ctx is the context label and is 0, 1, 2, or 3
br!=ctx[3, thread_2_start#];and label# is the location to where the branch will occur
```

An alternative approach is to have the StrongARM* core program the CTX_PC field of the CTX_n_STS local CSR to the specific Control Store address where the thread should begin executing.

4.8 Microengine Registers

The sections that follow describe the Microengine general-purpose registers and Transfer Registers.

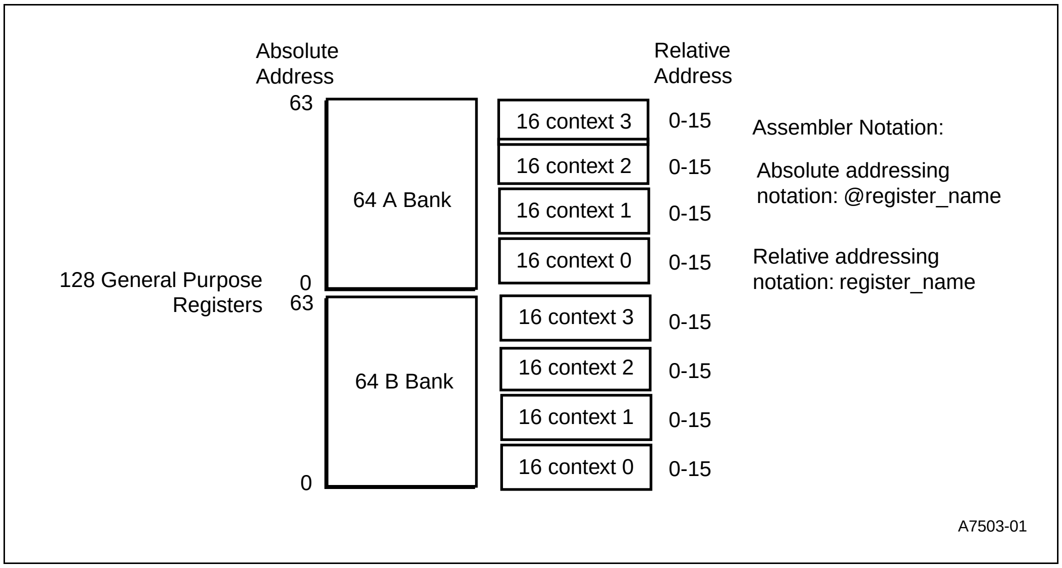
4.8.1 General-Purpose Registers

Each Microengine supports 128 32-bit general-purpose registers (GPRs). The GPRs are divided into an A-bank and a B-bank. Each bank supports one read port with a data path to either the A or B input of the ALU/Shifter and a write port from the output of the ALU/Shifter. The dual ports allow the Microengines to perform simultaneous read and write operations allowing instructions to execute in a single cycle.

The assembler supports symbolic register naming which is a feature provided in higher level programming languages. It allows the programmer to write programs using descriptive symbolic names, and the assembler manages how they are mapped to physical registers. The assembler supports context-relative and absolute address modes.

Context relative addressing logically subdivides the GPRs into four equal regions of 32 registers, so that each thread has exclusive access to one of the regions. Context-relative addressing is a powerful feature that enables four different threads to maintain separate data areas. It also eliminates overhead normally associated with preserving the register set when switching between contexts. Absolute addressing enables any GPR register to be shared among the four threads in a Microengine.

Figure 4-7. GPR Addressing

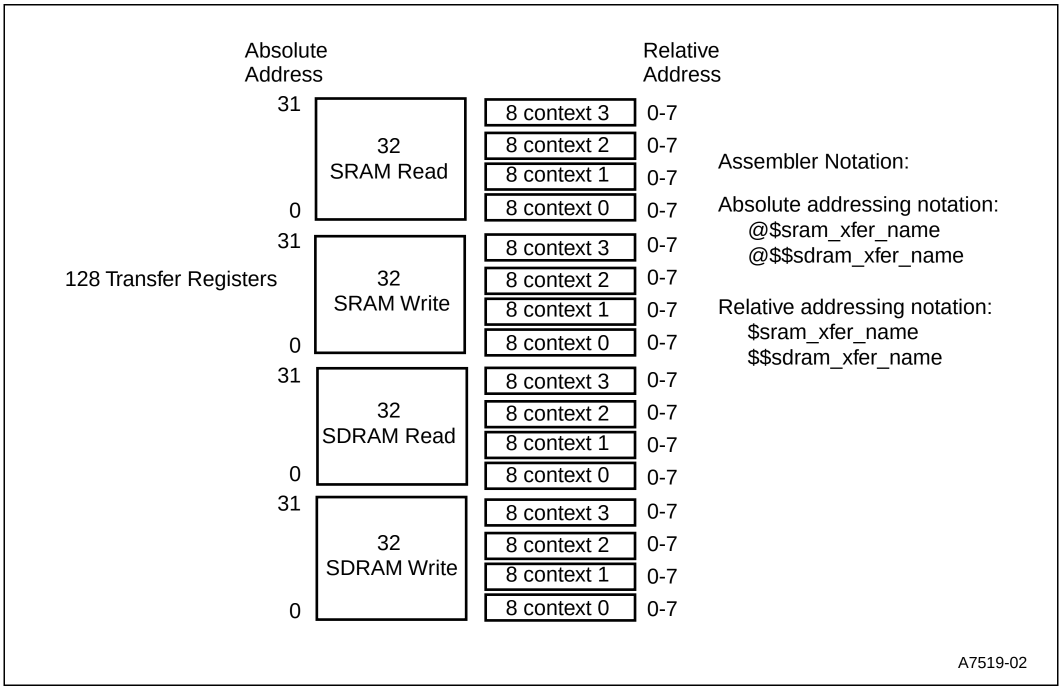


4.8.2 Transfer Registers

Each Microengine supports 128 32-bit Transfer Registers that are used to move data to and from the Microengines. Each Transfer Register supports one read port and one write port. The dual ports allow the Microengines to perform a simultaneous read and write, allowing instructions to execute in a single cycle.

As shown in Figure 4-8, these registers are divided into 64 SRAM and 64 SDRAM Transfer Registers that are connected to the SRAM and SDRAM buses. Of the 64 SRAM/SDRAM Transfer Registers, 32 are connected to the pull buses and are used to move data from the Microengine Transfer Registers to memory and 32 are connected to the push buses and are used to move data to the Microengine Transfer Registers.

Figure 4-8. Transfer Register Addressing



Assembler Notation:

Absolute addressing notation:
@$sram_xfer_name
@$$sdram_xfer_name

Relative addressing notation:
$sram_xfer_name
$$sdram_xfer_name

Table 4-7 lists the Microengine and FBI Ready Bus instructions that use Transfer Registers. Note that all instructions other than the **sdram** instruction use SRAM Transfer Registers.

Table 4-7. Transfer Register Usage (Solicited Access) (Sheet 1 of 2)

Microengine Instruction	Request Destination	Operation
csr	FBI Unit	Read and write an FBI CSR
hash1_48, **hash2_48,** **hash3_48**	FBI Unit	Perform 1, 2, or 3 48-bit hash operations
hash1_64, **hash2_64,** **hash3_64**	FBI Unit	Perform 1, 2, or 3 64-bit hash operations

Table 4-7. Transfer Register Usage (Solicited Access) (Sheet 2 of 2)

Microengine Instruction	Request Destination	Operation
scratch	FBI Unit	Read and write Scratchpad Increment data in the Scratchpad Set or clear specific bits in the Scratchpad
pci_dma	PCI Unit	Write a PCI descriptor address to a PCI DMA channel
sram	SRAM Unit	Read and write SRAM Set or clear specific bits in SRAM Lock memory and read the data Write data and unlock memory Unlock memory Push a pointer onto a link list (up to eight list supported) Pop a pointer off a linked list Read and write BootROM memories Read and write the Slow Port memory space
sdram	SDRAM Unit	Read and write SDRAM
r_fifo_rd	FBI	Read data from the RFIFO into an SRAM Transfer Register
t_fifo_wr	FBI	Write data to the TFIFO from an SRAM Transfer Register

Table 4-8. Transfer Register Usage (Unsolicited Access)

Ready Bus Instruction	Request Destination	Operation
RxAutopush	FBI (Ready Bus)	The Ready Bus may be enabled to write 1,2,or 3 FBI registers to the SRAM Transfer Registers
TxAutopush	FBI (Ready Bus)	The Ready Bus may be enabled to write 2 or 3 FBI registers to the SRAM Transfer Registers

Instructions that support burst counts greater than one also require more than one Transfer Register per read or write operation. The Transfer Register specified in the instruction specifies the register that is the beginning of a contiguous set of registers that receive or supply the data on a read or write operation, respectively. Single transfers to and from the SRAM Transfer Registers always occur in 32-bit increments. Single transfers to and from the SDRAM Transfer Registers always occur in 64-bit increments and require two SDRAM Transfer Registers per read or write operation.

4.8.2.1 Managing Solicited Accesses

The individual Microengine threads are responsible for managing when the Transfer Register may be reused and guaranteeing that the data in the registers is correct. For example, if a write Transfer Register is being used by a thread to provide data to SDRAM, the thread must not overwrite this register until it is provided with an SDRAM signal event indicating that the data has been promoted to SDRAM and the register may now be reused.

A thread can have only one instance of a specific signal event pending at any time. For example, if a thread requests an SRAM signal, it must wait until that signal event occurs before requesting another SRAM signal event.

As a performance optimization, every reference to a particular functional unit does not require a signal event. Instead, a signal event can be overloaded with multiple references as long as the references are issued to the same command queue in the functional unit. Issuing the references to the same queue ensures that the commands will be completed in the order in which they were issued. If a signal event is requested on the last reference, it is implicitly guaranteed that all references have been completed.

If the thread uses multiple command queues in a given functional unit (e.g., SRAM order queue and read queue), the order in which the references are completed can not be guaranteed and therefore a signal event for a particular functional unit cannot be overloaded. In this case, a thread must issue the references in the execution order, request a signal event for each reference, and it must not issue the next reference until the previous reference has completed.

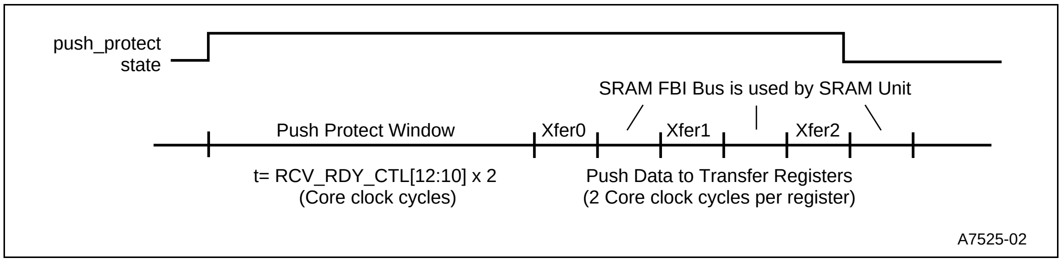
4.8.2.2 Managing Unsolicited Autopush Accesses

The Ready Bus **Autopush** instruction will cause the FBI Push Engine to perform an unsolicited write to a Microengine SRAM read Transfer Register. It is possible for a Microengine thread to read the Transfer Register while the FBI push engine is writing to the same register. If this occurs, the integrity of the data can not be guaranteed. The Autopush Protection Mechanism can be used to ensure that a Microengine read of a Transfer Register and an Autopush write operation will not occur at the same register address at the same time.

The Autopush Protection Mechanism provides an input state to all the Microengines to indicate when an autopush operation is in progress. A Microengine thread should test the push_protect input state using the **br_inp_state** instruction before reading the Transfer Registers that are affected by the Autopush operation. If the push_protect state is not asserted, the Microengine should read the Transfer Registers contiguously beginning at the very next cycle (instruction) after the **br_inp_state** instruction to ensure that the push operation does not collide with the Microengine. If the push_protect state is asserted, the Microengine should wait until it is deasserted before accessing the Transfer Registers.

The timing of the push_protect signal is shown in Figure 4-9. The push protect window is provided to ensure that an Autopush operation does not begin while a Microengine is reading the Transfer Registers. The push protect window is programmable via the RCV_RDY_CTL register and should be set to a value equal to twice the maximum number of registers that are being pushed per Autopush operation. For example, if RCV_RDY_CTL[7:6] is set so that one register is used during an Autopush and XMIT_RDY_CTL[7:6] is set so that three registers are used, then the push protect window should be set to a value of six (3 x 2). The number of registers that can be Autopushed during a TxAutopush or RxAutopush operation can vary from 1 to 3 and is determined by setting RCV_RDY_CTL[7:6] and XMIT_RDY_CTL[7:6].

Figure 4-9. Push Protect Timing


push_protect state
SRAM FBI Bus is used by SRAM Unit
Push Protect Window
Xfer0
Xfer1
Xfer2
t= RCV_RDY_CTL[12:10] x 2
(Core clock cycles)
Push Data to Transfer Registers
(2 Core clock cycles per register)
A7525-02

If the Ready Bus sequencer program contains two **Autopush** instructions back-to-back (e.g., TxAutopush followed by RxAutopush), then a single push_protect signal is asserted with only one push protect window. Otherwise, a push protect signal containing a push protect window is asserted for each Autopush operation.

4.9 ALU and Shifter

Each Microengine contains a powerful 32-bit ALU and shifter capable of performing an ALU and shift operation in a single cycle. The two inputs of the ALU (A and B) can be data supplied by the SRAM read Transfer Registers, SDRAM read Transfer Registers, GPRs, or immediate data within the instruction. The ALU can perform addition, subtraction, and logical operations as well as generate sign, zero, and carry out condition codes based on these operations.

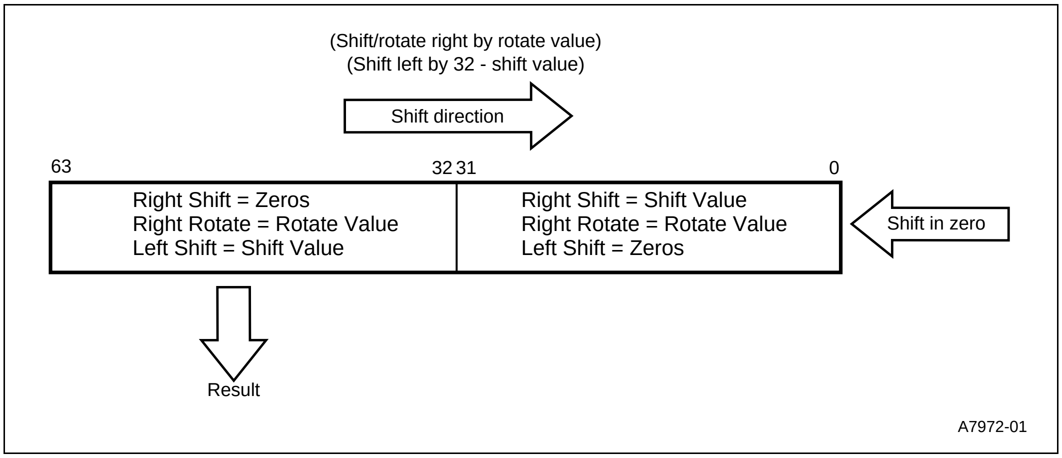
Table 4-9. ALU Operations

ALU Operation	Description
B	B operand (A operand is ignored).
~B	Inverted B operand (A operand is ignored).
+	A operand + B operand.
+carry	A operand + B operand + previous carry-in (carry-in equals previous carry-out).
+4	A operand + B operand truncate to the least significant 4 bits (upper 7 nibbles zeroed).
+8	A operand + B operand truncate to the least significant 8 bits (upper 3 bytes zeroed).
+16	A operand + B operand truncate to the least significant 16 bits (upper 2 bytes zeroed).
-	A operand - B operand.
B-A	B operand - A operand.
AND	A operand AND B operand (Bit-wise AND).
~AND	Inverted A operand AND B operand (Bit-wise AND).
AND~	Inverted B operand AND A operand (Bit-wise AND).
OR	A operand OR B operand (Bit-wise OR).
XOR	A operand XOR B operand (Bit-wise exclusive OR).
+IFsign	If the Sign condition code is set, A operand + B operand. If the Sign condition code is not set, result is B operand. When using +IFsign in an ALU or ALU_SHF instuction, note that the condition code is set by the second instruction immediately preceding the current one. This is in contrast to a branch instruction, where the condition code is set by the instruction immediately preceding the current one.

The B input supports a 64-bit barrel shifter. The shifter shifts a 32-bit input to the right and produces a 32-bit output to the ALU. The right shift operation is performed by placing the shift operand into the lower 32 bits and zeros into the upper 32 bits of the 64-bit shifter and then shifting right by the specified amount. The result is taken from the lower 32 bits.

Right rotate operations are almost identical to right shift operations except that the shift operand is placed into both the upper and lower 32 bits. Left shift operations are emulated by placing the zeros into the lower 32 bits and the shift operand into the upper 32 bits, shifting right by 32 minus the specified the left shift amount, and taking the lower 32 bits as the result. For example, to shift left by 12, the indirect value would be 20 (32-12 = 20). See Figure 4-10.

Figure 4-10. Microengine Shift and Rotate Procedure


(Shift/rotate right by rotate value)
(Shift left by 32 - shift value)
Shift direction
63
32
31
0
Right Shift = Zeros
Right Rotate = Rotate Value
Left Shift = Shift Value
Right Shift = Shift Value
Right Rotate = Rotate Value
Left Shift = Zeros
Shift in zero
Result
A7972-01

The **alu_shf** and **dbl_shf** instructions allow the user to specify an indirect shift value using the optional tokens **<<indirect** or **>>indirect**. When the **>>indirect** token is used, the shift value is specified by the lower 5 bits of the A operand from the previous instruction executed. This 5-bit value specifies the right shift value when the **>>indirect** token is used. When the **<<indirect** token is used, the actual shift value is equal to 32 minus the 5-bit A operand value.

4.9.1 Condition Codes

The ALU supports the following condition codes:

Zero: Set when an ALU operation is performed and the result is zero.

Sign: Set when an ALU operation is performed and bit 31 is set to one.

Carryout: Set when an ALU operation is performed and the carryout bit is set to one.

Only the ALU can generate condition codes (the Shifter does not generate a carryout condition code during a shift operation). The following instructions set the condition codes. All of these instructions set the zero and sign condition codes, but only the ALU and ALU_SHF instruction set the carryout.

ALU
ALU_SHF
BR_BCLR
BR_BSET
BR=BYTE
BR!=BYTE
DBL_SHF
LD_FIELD
LD_FIELD_W_CLR
LOAD_BSET_RESULT1
LOAD_BSET_RESULT2

One set of condition codes is provided for each Microengine. In other words, condition codes are not maintained per context. If a context sets the condition codes, the next context inherits these condition codes and changes them when it executes an instruction that changes the condition codes.

4.9.2 Multiply Support

The Microengines do not support a multiply instruction. However, the ALU instruction supports the **+ifsign** operator which can be used to accelerate multiply operations. The idea is to shift the multiplier bit into the sign bit position to set the sign bit condition code. One cycle after this operation, the **+ifsign** ALU operation can be applied which will perform an A+B operation if the sign bit was set, otherwise the B operand is moved into the destination register. A one cycle delay is present due to internal piping considerations. However, this delay can be hidden in most cases by performing 2 interleaved operations at a time as illustrated in the sample code below. The sample code performs a multiply of a 23-bit number (multiplicand) with an 8-bit number (multiplier).

```
begin#:
; Initialize input registers.
; multiply 8394587 x237= 1989517119
; 0x80175B x 0xED = 0x76959f3f

immed_w0[ multiplicand, 0x175B ]
immed_w1[ multiplicand, 0x80 ]
immed[multiplier, 237]


; Move the LSB into the sign bit. The Shift and accumulate are interleaved in pairs
; to hide cycle delays associated with the +ifsign operation.

start#:
    alu_shf [multiplicand, --, b, multiplicand, <<7]
    immed [accum, 0]

    alu_shf[--, --, b, multiplier, <<31]
    alu_shf[--, --, b, multiplier, <<30]


    alu [accum, multiplicand, +ifsign, accum]
    alu [accum, multiplicand, +ifsign, accum, >>1]

    alu_shf[--, --, b, multiplier, <<29]
    alu_shf[--, --, b, multiplier, <<28]

    alu [accum, multiplicand, +ifsign, accum, >>1]
    alu [accum, multiplicand, +ifsign, accum, >>1]

    alu_shf[--, --, b, multiplier, <<27]
    alu_shf[--, --, b, multiplier, <<26]

    alu [accum, multiplicand, +ifsign, accum, >>1]
    alu [accum, multiplicand, +ifsign, accum, >>1]

    alu_shf[--, --, b, multiplier, <<25]
    alu_shf[--, --, b, multiplier, <<24]

    alu [accum, multiplicand, +ifsign, accum, >>1]
    alu [accum, multiplicand, +ifsign, accum, >>1]

done#:
```

4.10 Changing Contexts

Hardware multithread support allows four separate programs (or threads) to share execution time on a Microengine. When a program is not executing, each program context is preserved in hardware through separate program counters, signal event states, and a relatively addressed register set (General-Purpose Registers (GPRs) and Transfer Registers) for each program. When a program is put to sleep, a context switch occurs and another program begins executing. The overhead associated with switching contexts is a maximum of one cycle, however a deferred instruction can be used to eliminate this overhead. Refer to Section 4.5.5 for more information on deferred instructions.

As with most high performance microprocessor architectures, a Microengine can execute numerous instructions in the time required for a reference to a device external to the Microengine (such as memory or the FBI Unit) to be completed.

The architecture of the IXP1200 Microengines does not require that a Microengine thread wait for references to complete. Instead, the Microengine threads can to put themselves to sleep (switch contexts) or continue executing other instructions after issuing a reference. This increases the processing throughput of the Microengine by allowing it to perform useful work independent of the reference.

To achieve the best performance from the Microengines, context changes should occur at appropriate points within a program execution. For example, if a program thread cannot execute any further instructions without the data supplied by a reference, it should allow another context to run by switching contexts. To ensure context changes occur when it makes the most sense, the programmer is given control of context switching through the program instructions. The multi-threaded architecture of the Microengines can be described as hardware supported, non-preemptive multithreading.

There are two forms of context change instructions. The first is an explicit context arbitrate instruction (**cxt_arb**):

```
ctx_arb[signal_event], optional_token;optional_token = defer[1]
```

This instruction indicates the context should be changed and should be awakened when the specified signal event occurs. All the signal events described in Section 4.10.1 are supported in addition to two other parameters: voluntary and kill.

The voluntary parameter indicates that a thread is voluntarily allowing the other three threads to execute if they are ready (not waiting for a signal event to wake them). If the other threads are not ready to execute, the thread continues to execute the next instruction in its program flow. The penalty for executing a voluntary context change when no other thread is ready to execute is the cycle required to execute the **ctx_arb[voluntary]** instruction.

The **kill** parameter indicates that a context change should occur and the thread should not be woken and allowed to execute. This essentially clears a context enable bit in the CTX_ENABLES register. The StrongARM* core is required to remove a Microengine thread from the kill state by enabling the context in the CTX_ENABLES register.

The programmer may also use the **ctx_swap** optional token in the Microengine instructions that reference the other functional units within the IXP1200. The example below shows how the **ctx_swap** optional token is used within an instruction.

```
sdram[write, $$xfer_reg, addr1, addr2, ref_count], ctx_swap
```

The instructions that support the **ctx_swap** instruction are listed in Table 4-10. The use of the **ctx_swap** optional token does not affect the time required to execute the instruction. The Microengines execute all instructions in a single cycle.

Table 4-10. ctx_swap Instructions

hash1_48	**hash1_64**	**sram**	**pci_dma**
hash2_48	**hash2_64**	**sdram**	**rfifo_rd**
hash3_48	**hash3_64**	**scratch**	**tfifo_wr**
	csr		

4.10.1 Signal Events

Signal events are used to notify a thread when an event occurs elsewhere in the IXP1200. A thread can test these signals using the **br=!signal** instruction or it can switch contexts and request that it be allowed to execute when a signal event occurs. Signal events can be classified into two types: explicitly requested and involuntarily provided.

Explicitly requested events are requested by the programmer via program instructions. The two methods of requesting a signal event are summarized in Table 4-11.

Table 4-11. Signal Event Request Methods

Method	Description
ctx_swap optional token	Switch context and allow it to execute when the signal event associated with this instruction occurs and it is this thread's turn to run. Example: Read SRAM, switch contexts, and begin executing it when the reference has completed. sram [read, $xfer1, temp, 0x15, 4] ctx_swap
sig_done optional token and ctx_arb instruction	The sig_done optional token indicates that the thread should be signaled when the reference has completed, but does not switch contexts. Additional instructions that follow the reference instruction execute immediately following the reference instruction. When the ctx_arb instruction is executed, it indicates that a context switch should occur and it identifies the signal event that will wake the thread. Example: Read SRAM, execute more instructions, then switch contexts and begin executing when the SRAM signal event occurs. sram [read, $xfer2, temp, 0x4, 4], sig_done (more instructions executed here) ctx_arb[sram]

These explicitly requested events are listed in Table 4-12.

Table 4-12. Explicitly Requested Signal Events

Signal Event	Condition when signal are issued
fbi	When a thread requests the signal in a FBI reference (Scratchpad, hash, CSR, RFIFO or TFIFO) and the reference completes.
pci	When a thread requests the signal in a PCI DMA reference.
sram	When a thread requests the signal in an SDRAM reference.
sdram	When a thread requests the signal in an SRAM reference.

Events that are involuntarily provided without explicitly requesting the signal event are listed in Table 4-13.

Table 4-13. Nonexplicit Signal Events

Signal Event	Condition when signal are issued
seq_num1	When Fast Port mode is enabled and a thread increments the enqueue sequence number 1, all threads are signaled.
seq_num2	When Fast Port mode is enabled and a thread increments the enqueue sequence number 2, all threads are signaled.
inter_thread	When StrongARM* core or another thread writes a thread ID to the INTER_THD_SIG register, the thread with the specified ID will be signaled.
start_receive	When a Receive Request is completed by the Receive State Machine, the thread specified in a Receive Request is signaled.
auto_push	When Autopush is enabled and the Ready Bus completes an Autopush operation, the Receive Scheduler thread is signaled.

The seq_num1 and seq_num2 signal events are involuntarily provided to all twenty-four Microengine threads, even though most applications do not require that all threads get the signal event. Threads that do not require these signals may ignore them and are not required to clear the signal events.

The inter_thread, start_receive, and auto_push signal events are provided to specific threads. The threads that are programmed to receive these signal events must periodically test these signal events to determine if the event occurred. The signals can be tested using the **br!=signal** or **ctx_arb** instructions.

The signal events are maintained on a per context basis. Each Microengine contains four CTX_n_SIG_EVENTS (where n=0-3) local CSRs that indicate which signal events have occurred. These signals can be cleared in three ways:

1. Writing to the CTX_n_SIG_EVENTS CSR. Both the StrongARM* core and the Microengines can write to this local CSR. This method is typically used during initialization or debugging.
2. Executing a **br_!signal** instruction. The **br_!signal** instruction allows the programmer to specify the specific signal to test. If the signal event is present, the branch is taken and the signal is automatically cleared.
3. Waking on a signal event. When a thread is woken by a signal event, the signal is automatically cleared.

A thread may request multiple signal events (e.g., SRAM, SDRAM, and FBI). However, only one instantiation of a specific signal event can be pending at one time. For example, a programmer can not have two SRAM signal events pending at a time. Also, a program can only be awakened by a single event. For example, a program can not be awakened by an FBI or an SRAM signal event.

4.10.2 Context Event Arbiter (Waking a Thread)

A Microengine thread can be in one of the following three execution states:

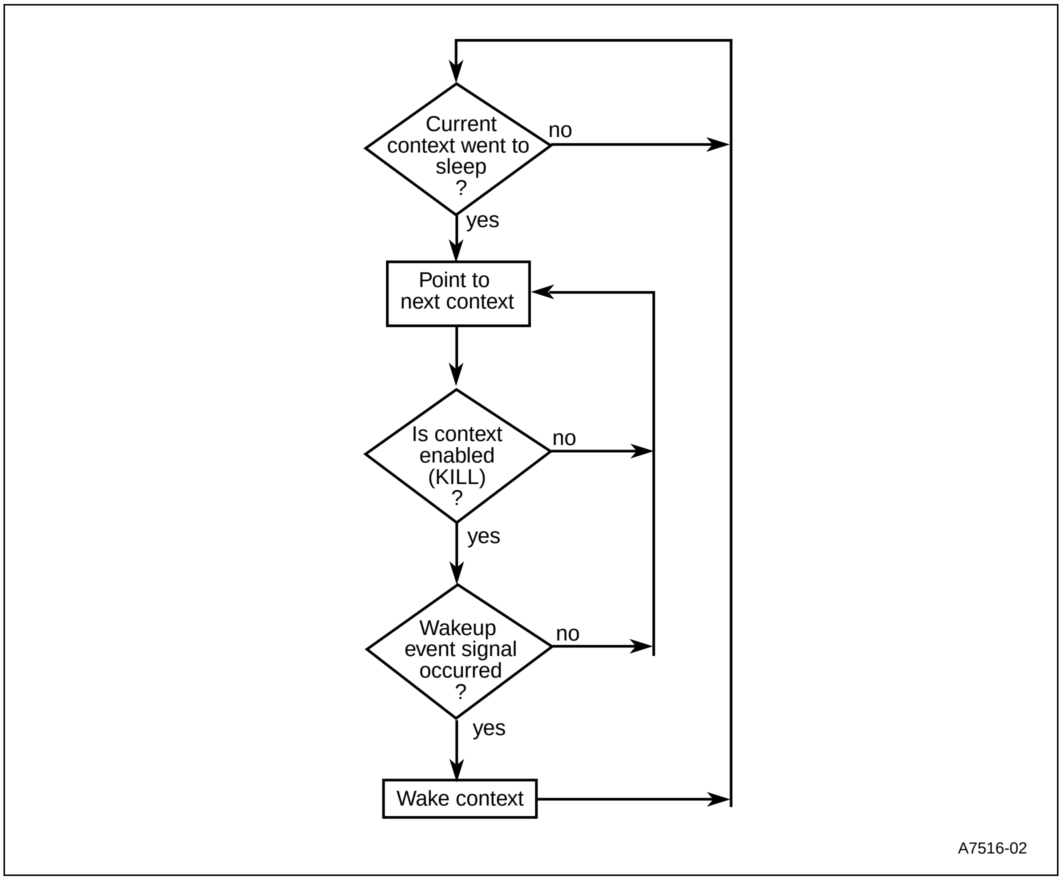
- Executing
- Sleeping (not executing, waiting for a signal event to occur)
- Ready (not executing, has received the signal event that it was waiting to occur, and is waiting for the context arbiter to grant it permission to execute)

The non-preemptive multithreading architecture allows a program thread to execute until it executes an instruction that performs a context switch. When a program thread performs a context switch, the Context Arbiter decides which thread will be allowed to execute next. The Context Event Arbiter makes this decision based on the following:

- A round-robin arbitration scheme provides each thread an equal chance at executing
- Whether the thread is enabled to run (via the CTX_ENABLES register)
- Whether the thread is in the Ready state

The Context Arbiter uses the signal event and wakeup event status provided for each thread and maintained in the CTX_n_WAKEUP_EVENTS (where n = 0, 1, 2, or 3) and CTX_n_SIG_EVENTS local CSRs. The Context Event Arbiter uses a round-robin algorithm to determine which should be woken next. If a context wakeup event is enabled and the event has occurred, the context will be enabled to run. Note that only one event at a time can be enabled in the CTX_n_WAKEUP_EVENTS CSR. This is enforced in the instruction set by allowing only one wakeup event to be specified in any instruction that will cause a context switch.

Figure 4-11. Context Arbitration Policy


Current context went to sleep ?
no
yes
Point to next context
Is context enabled (KILL) ?
no
yes
Wakeup event signal occurred ?
no
yes
Wake context
A7516-02

4.11 Interfacing to Other Functional Units

This section describes how data is transferred between the Microengines and the other functional units within the IXP1200. The Microengine threads reference the other functional units (FBI, PCI, SDRAM, and SRAM Units) by issuing a command and then continuing to execute instructions while the functional unit processes the command independently. This allows the Microengines to perform useful work while references are pending, increasing the overall performance of the Microengines.

The Microengine threads can reference the other functional units in two ways: using immediate data or using Transfer Registers. The **fast_wr** (fast write) instruction writes immediate data to specific registers within the FBI Unit. All other accesses use Transfer Registers. Table 4-14 lists the instructions that use Transfer Registers.

Table 4-14. Instructions Using Transfer Registers

Instructions Using Transfer Registers	
csr	hash1_48
scratch	hash2_48
pci_dma	hash3_48
sram	hash1_64
sdram	hash2_64
r_fifo_rd	hash3_64
t_fifo_wr	

4.11.1 References Using Transfer Registers

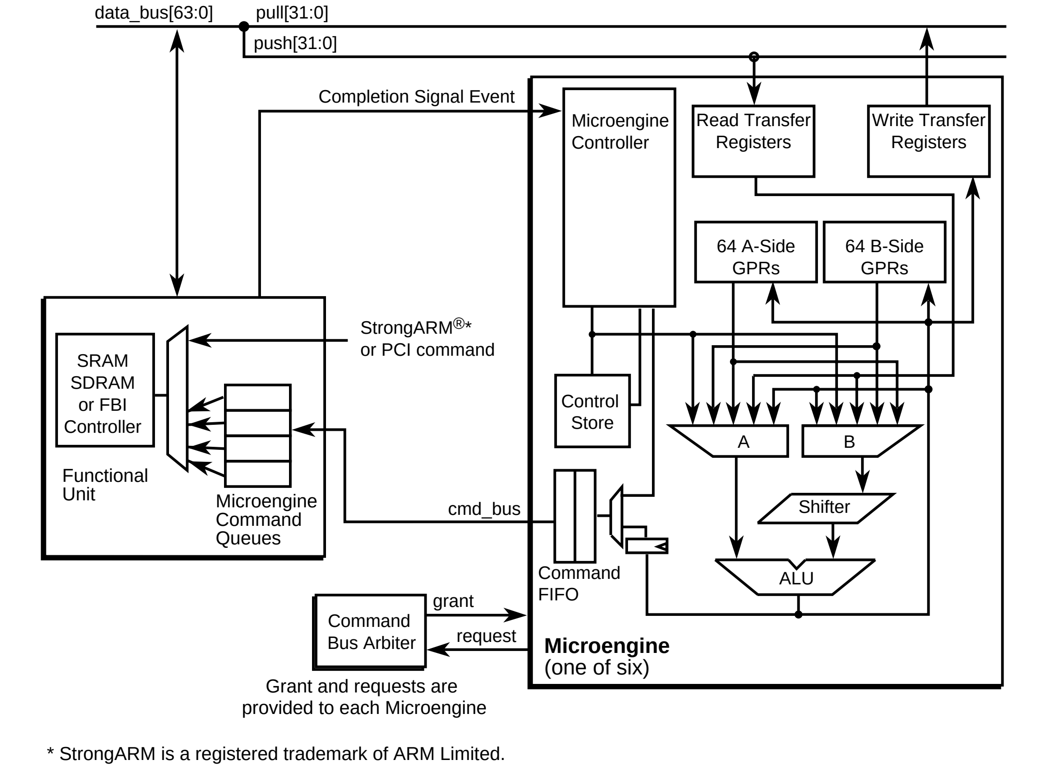
Figure 4-12 shows a simplified Microengine along with another functional unit. The simplification is that only one set of push/pull buses is shown and only one shared functional unit is illustrated. This figure will be used to describe how a Microengine issues a reference using Transfer Registers.

The following steps are taken when a Microengine issues a reference:

- Set up the Transfer Registers,
- Issue command,
- Queue command at the functional unit,
- Move data between Microengine and functional unit, and
- Signal completion

These steps are described in the sections that follow.

Figure 4-12. Microengine References Using Transfer Registers



* StrongARM is a registered trademark of ARM Limited.

A7505-02

4.11.1.1 Setting Up the Transfer Registers

Before performing a write operation, the write data should be placed into one or more contiguous write Transfer Registers. For read operations, the read data is written into one or more contiguous read Transfer Registers by the functional unit that is referenced.

The reference instruction specifies the register name at the beginning of a set of contiguous registers and the burst size of the transfer. Since the microcode assembler automatically assigns registers based on symbolic names, the assembler supports the .xfer_order assembler directive. This directive informs the assembler that a list of register names must be allocated contiguously. It also allows the programmer to define the names of the Transfer Registers in which the data is read from or written to.

4.11.1.2 Issuing a Command

When reference instructions are executed, a Microengine issues a command to the appropriate functional unit. The command describes the operation that is requested and indicates which Microengine and thread issued the command. When a command is issued, it is placed into a two-entry Command FIFO located in the Microengine. All six Microengines contain their own Command FIFOs, and each FIFO is shared among the four program threads within the

Microengine. The six Command FIFOs from the six Microengines are tied to a shared time-multiplexed Command Bus that operates at the Core clock frequency. When a command is placed into the Command FIFO, a bus request is submitted to the Command Bus Arbiter.

The Command Bus Arbiter arbitrates between the six Microengines to determine which Command FIFO will be serviced next. The Command Bus Arbiter uses the following information to determine which Command FIFO is to be serviced.

- A priority scheme between the types of commands.
- A rotating priority scheme between the Microengines.
- A back-pressure signal from the functional units.

The arbitration scheme is shown in Figure 4-13.

If the Command Bus arbiter serviced a Command FIFO that contains a chained SDRAM command, the Command Bus Arbiter will only allow SDRAM commands from the Command FIFO where the chained SDRAM was issued to be granted access to the Command Bus until the chain is completed.

Note: Note, however, that commands to other units are granted access to the Command Bus.

The priority after chained SDRAM commands is as follows:

- SRAM
- Non-chained SDRAM
- FBI/PCI

Back-to-back accesses to the FBI or PCI are not allowed. They are not considered separate units by the arbitration. If a command is issued to the FBI on one cycle, the next cycle cannot issue a command to either the FBI or the PCI. Similarly, if a command is issue to the PCI on one cycle, the next cycle cannot issue a command to either the FBI or the PCI.

Note: Note that consecutive grants are not allowed for SDRAM (chained or otherwise), FBI, or, PCI. Consecutive grants are allowed to SRAM, but not allowed to SDRAM or FBI/PCI. FBI and PCI appear as a single unit to the arbiter.

For grants within a priority, one Microengine is selected as the lowest priority Microengine. The highest priority Microengine is the next higher indexed Microengine (this wraps around, so if Microengine 5 is the lowest priority, then highest priority is assigned to Microengine 0). The Microengines have lower priority as the index increases from highest priority Microengine (again wrapping from 5 to 0). Microengine commands to a functional unit will not be serviced, until there are no more commands for higher priority functional unit or to the same functional unit from any other higher priority Microengine.

If a chained SDRAM, SRAM or normal SDRAM operation is not pending (any one of the three) and an FBI/PCI command could be issued but for pending commands to higher priority units on a cycle, the following cycle will not be used to issue a command to the FBI/PCI. FBI/PCI commands are eligible to be issued on the cycle subsequent to the following cycle, subject to other pending commands and the priorities of target units and Microengines. When the grant to the FBI/PCI is suppressed as described above, the following cycle will not be suppressed by those same conditions. If a different command gets a grant that following cycle, then the FBI/PCI command may be suppressed on the next cycle.

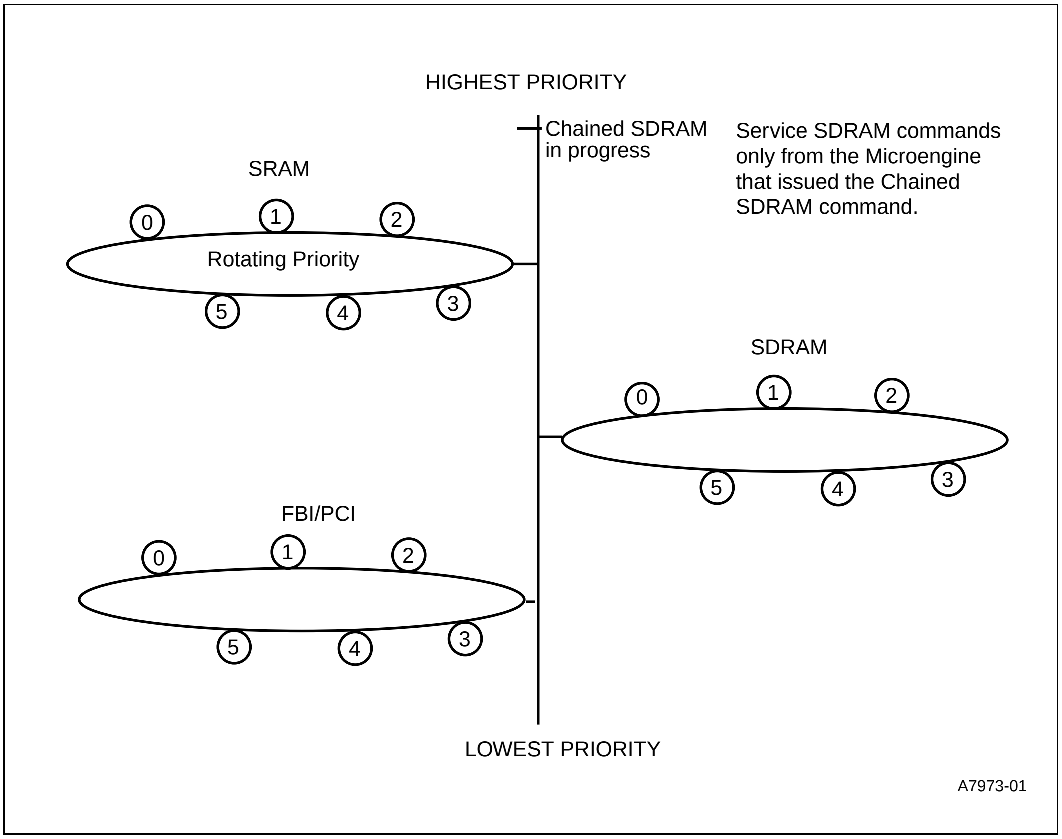
The index of the lowest priority Microengine increments to the next higher value when it has no outstanding commands in its Command FIFO. Thus, the Microengine with lowest priority will remain as the lowest priority until all its pending requests are serviced, at which point it will become the second lowest priority. A Microengine could wait indefinitely before it can issue a command to a functional unit, if the higher priority Microengines are constantly issuing commands to that functional unit.

If any of the command queues within a functional unit fills to a threshold level typically equal to six less than the number of commands that queue can hold, the functional unit will apply a back-pressure mechanism to ensure that the queue does not overflow. This threshold level ensures that there is room in the queues for any commands currently being sent to the functional unit from the six Microengines. If the Command Bus Arbiter receives a back-pressure signal, it will not service any Microengine whose Command FIFO contains a command that is targeted for the functional unit that applied the back-pressure mechanism.

The Command FIFOs are drained at the Core frequency, and, therefore, are typically drained at a rate faster than the commands are placed into them. However, if a program executes multiple consecutive references, the Command FIFO may fill. If the Command FIFO on a Microengine fills and another reference is issued, that Microengine stops executing instructions (referred to as stalling) until the command can be placed into the Command FIFO.

Also, the rate at which commands are issued to the functional units is greater than the rate at which the functional units are capable of processing a command. For example, the SRAM Unit reads data from the SRAM devices at ½ the Core frequency, however, commands are submitted to the SRAM Unit every Core clock cycle. This ensures that functional units maybe supplied with commands, which in turn ensures that the unit has useful work to perform.

Figure 4-13. Command Bus Arbitration Policy


HIGHEST PRIORITY
Chained SDRAM in progress
Service SDRAM commands only from the Microengine that issued the Chained SDRAM command.
SRAM
0
1
2
Rotating Priority
5
4
3
SDRAM
0
1
2
5
4
3
FBI/PCI
0
1
2
5
4
3
LOWEST PRIORITY
A7973-01

4.11.1.3 Command Serviced in Queue

After the command is placed onto the Command Bus, it is deposited into a command queue within the targeted functional unit. These command queues accept commands from, and hence are shared between, all of the Microengines. These command queues are shared among the other Microengines. The functional unit removes the commands from the command queues and performs the operation in the order that appears in the queue. Several different command queues are provided at each functional unit, and, these queues are described in more detail in the FBI, SDRAM, SRAM, and PCI sections.

4.11.1.4 Moving Data to and from Transfer Registers

When the functional unit processes the command, it reads or writes the data to or from the Microengine Transfer Registers. The data is provided on one of two 64-bit buses: the SRAM Bus or the SDRAM Bus. These 64-bit buses are divided into a 32-bit push and a 32-bit pull bus. The bus utilization is described in Table 4-15. Note that the peak bandwidth of each bus is equal to the peak bandwidth of the external buses.

Table 4-15. Bus Utilization

Operation	Internal 32-Bit Data Bus	Transfers Occur on Core Clock Cycle	Internal Peak Bandwidth	External Peak Bandwidth
SDRAM read	SDRAM Pull bus	odd and even	Core freq x 32-bits	½ Core freq x 64-bits
SDRAM write	SDRAM Push bus	odd and even		
SRAM read	SRAM Pull bus	odd	½ Core freq x 32-bits	½ Core freq x 32-bits
SRAM write	SRAM Push bus	odd		
FBI read	SRAM Pull bus	even	½ Core freq x 32-bits	FCLK x 64-bits
FBI write	SRAM Push bus	even		

4.11.1.5 Signaling Completion

Reference instructions provide two optional tokens that indicate that a Microengine thread should be signaled when the transfer completes. The first optional token is sig_done. This token indicates that the thread should be signaled when the operation is completed. The **br_!signal** instruction tests the signal events and the **ctx_arb** instruction indicates that the thread should be put to sleep and woken when a signal event occurs.

The second optional token is ctx_swap. It allows a reference instruction to specify that the thread should be signaled when the operation is completed and the thread should be put to sleep (swapped out) and woken when the signal event occurs. The ctx_swap optional token is equivalent to using the sig_done optional token with the reference instruction and a separate **ctx_arb** instruction.

4.11.2 PCI DMA

The Microengines can initiate DMA transfers across the PCI using either one or both DMA channels. The StrongARM* core allocates the DMA channels under software control. The StrongARM* core can dynamically change the allocation of the channels by modifying the DMA_INTER_MODE register. It is expected that, using interprocessor communications, the StrongARM* core and the Microengines signal each other when a channel allocation change is required. After that point, neither should issue any more DMA requests and the StrongARM* core waits until all pending requests have completed. The StrongARM* core can detect when the DMA channels are idle by enabling the DMA “Not Busy” interrupt in the IRQ_ENABLE register.

The PCI Unit supports a four-entry DMA request FIFO for requests made by the Microengines. This provides for five outstanding Microengine DMA requests, one in the allocated DMA Channel, and four in the DMA FIFO. Since there is no hardware back-pressure mechanism to indicate when the DMA FIFO is full, Microengine software must maintain status as to the number of pending Microengine DMA requests to ensure it does not exceed a total of five.

If the Microengines own both DMA channels, the DMA requests can be issued to either the next available DMA channel or directly to DMA channel 2. When a DMA request is targeted at DMA channel 2, it ensures that the DMA requests will be completed in the order in which they were issued.

The procedure for issuing PCI DMA transfers is described in Section 5.2.2.3, “Microengine Initiated DMA Channel Operation”.

4.11.3 FAST_WR Instruction

The Microengines support a **fast_wr** instruction to improve performance when writing to a subset of FBI registers. The **fast_wr** instruction supplies 10-bit immediate data in the reference command. This eliminates the need for the FBI Unit to read the data from a Microengine SRAM Transfer Register when it processes the command.

The meaning of the 10-bit immediate data is shown in Table 4-16. Some registers do not require the entire 10-bits. The programmer should ensure that data larger than what is required for the register is not written since it will extend past the Microengine assigned bit position field.

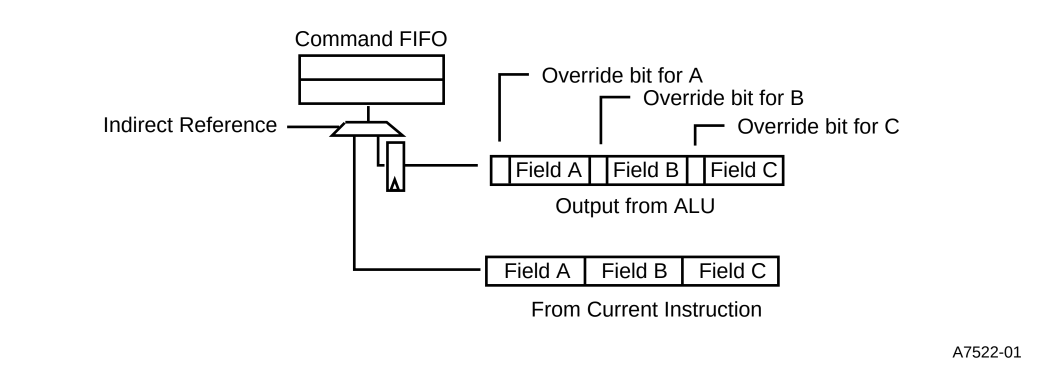
Table 4-16. Fast_wr 10-Bit Immediate Data

Register	10-Bit Immediate Data
Inter_thd_sig	Thread number of the thread that is to be signaled (0 to 23).
Thread_done	A 2-bit message that is shifted into a position relative to the thread that is writing the message. The shift operation is performed by the FBI Unit. The meaning of the message is determined through software.
Thread_done_incr1 Thread_done_incr2	Same as thread_done except that either the enqueue_seq1 or enqueue_seq2 is incremented.
xmit_validate	The Transmit FIFO element number (0 to 15) that is marked to indicate that the data is valid in this element.
incr_enq_num1 incr_enq_num2	Write a one to increment the Enqueue Sequence Number by one (the Sequence Number is always incremented by one).
self_destruct	Specifies the bit position (0-31) that will be set.
ireg	The 10-bit immediate data supplied with the instruction is shifted by the FBI Unit in two segments. Bits 6 through 0 are shifted left by an amount equal to the thread ID writing the data. Bits 9 through 7 are always shifted into the BP2 through BP0 positions regardless of the Microengine writing the data.

4.11.4 Indirect References

Indirect references allow the programmer to redefine a reference (a command that is placed onto the Command Bus) at run time. This is referred to as an indirect reference. As shown in Figure 4-14, if an instruction specifies an indirect reference, the reference is redefined by the output generated by the previous ALU instruction.

Figure 4-14. Indirect References



The output of the ALU is latched after each instruction is executed. If an instruction specifies the indirect reference using the indirect_ref optional token, the command placed on the Command Bus is redefined using data provided at the output of the previous ALU instruction. The format of the data is defined in the *IXP1200 Network Processor Programmer's Reference Manual* and depends on the instruction being modified.

The indirect reference data may modify more than one parameter within in an instruction. An override bit is provided on a per data field basis to indicate which parameter should be modified. When the indirect_ref optional token is specified, it enables the data multiplexer that redefines the command and the override bits enable the specific bit fields.

Example 4-1 shows how two instructions are used to create an indirect reference.

Example 4-1. Indirect Reference

```
alu_shf[--, --, b, 0x13, <<16]
t_fifo_wr[$xfer0, tfifo_addr, 0, 1], indirect_ref
```

The first instruction, **alu_shf**, determines how the second instruction is defined. In this example, bit field 20 is the override bit and bit fields 16 to 19 indicates the burst count size which is changed to a burst count of four quadwords (the value specified in bits 19:16 plus one). The indirect_ref optional token in the second instruction specifies that the ALU output from the first instruction should be used to redefine the reference.

Table 4-17 shows the instructions that support an indirect reference and the parameters that may be modified using an indirect reference.

Table 4-17. Indirect Reference Instructions (Sheet 1 of 2)

Instruction	Command	Indirect Reference Options
fast_wr		• Specify the Microengine and thread ID the FBI Unit uses to specify the thread that is signaled when the reference is complete, and how the data is interpreted when the specified FBI register is THREAD_DONE, THREAD_DONE_INCR1, THREAD_DONE_INCR2, and IREG. • Specify the immediate data.
scratch	read/write	• Specify the Microengine and thread ID the FBI Unit uses to specify the thread that is signaled when the reference is complete. • Specify the number of longwords that should be written to scratchpad memory (16 max.). • Specify the Microengine SRAM Transfer Register (0-31).
	bit_wr	• Specify the Microengine and thread ID the FBI Unit uses to specify the thread that is signaled when the reference is complete. • Specify whether the pre-modified data should be returned. • Specify whether the operation is set or clear. • Specify the Microengine SRAM Transfer Register (0-31).

Table 4-17. Indirect Reference Instructions (Sheet 2 of 2)

Instruction	Command	Indirect Reference Options
sdram	read/write	• Specify the Microengine and thread ID the SDRAM Unit uses to determine the thread that is signaled when the reference is complete and the Microengine where the Transfer Registers reside. • Specify the Transfer Register (0-31) in the Microengine. • Specify the number of quadwords that should be written to SDRAM (16 max.). • Specify a byte mask for read-modify-write operations.
	tfifo_wr	• Specify the Microengine and thread ID the SDRAM Unit uses to determine the thread that is signaled when the reference is complete. • Specify the number of quadwords read from SDRAM and written to the TFIFO (16 max.). • Specify the byte alignment. • Specify the TFIFO quadword address.
	rfifo_wr	• Specify the Microengine and thread ID the SDRAM Unit uses to determine the thread that is signaled when the reference is complete. • Specify the number of quadwords that should be read from the RFIFO and written to SDRAM (16 max.). • Specify the RFIFO quadword address.
sram	read/write	• Specify the Microengine and thread ID the SRAM Unit uses to determine the thread that is signaled when the reference is complete and the Microengine where the Transfer Registers reside. • Specify the number of longwords moved between the SRAM and the Microengine (16 max.). • Specify the SRAM Transfer Register (0-31) in the Microengine.
	push/pop	• Specify the Microengine and thread ID the SRAM Unit uses to determine the thread that is signaled when the reference is complete and the Microengine where the Transfer Registers reside. • Specify the push/pop register. • Specify the SRAM Transfer Register (0-31) in the Microengine.
	bit_wr	• Specify the Microengine and thread ID the SRAM Unit uses to determine the thread that is signaled when the reference is complete and the Microengine where the Transfer Registers reside. • Specify whether the pre-modified data should be returned • Specify whether the operation is set or clear • Specify the SRAM Transfer Register (0-31) in the Microengine.
r_fifo_rd		• Specify the Microengine and thread ID the FBI Unit uses to determine the thread that is signaled when the reference is complete and the Microengine where the Transfer Registers reside. • Specify the number of quadwords that should be read from the RFIFO (16 max.). • Specify the SRAM Transfer Register (0-31) in the Microengine.
t_fifo_wr		• Specify the Microengine and thread ID the FBI Unit uses to determine the thread that is signaled when the reference is complete and the Microengine where the Transfer Registers reside. • Specify the number of quadwords that should be written to the TFIFO (16 max.). • Specify the SRAM Transfer Register (0-31) in the Microengine.

4.12 Local CSRs

The local CSRs described in Section 4.2.7 are memory mapped into the StrongARM* core 4-Gigabyte address space. These registers are used by the StrongARM* core to program the Control Stores and provide control and status information when debugging. These registers are described in Table 4-18.

Table 4-18. Local CSRs

Local CSR	Description
USTORE_ADDRESS	Microstore Address: Used to load the Control Store[1].
USTORE_DATA	Microstore Data: Used to load the Control Store[2].
ALU_OUTPUT	ALU Output: Used for debugging by the StrongARM* core for reading the GPRs and Read Transfer Registers.
ACTIVE_CTX_STS	Active Context Status: Active context number, whether it is running, and the current PC.
ENABLE_SRAM_JOURNALING	Enable SRAM Journaling: Used for debugging to create journals in SRAM.
CTX_ARB_CTL	Context Arbiter Control: Context number of the last context to run and the number of the next context ready to run.
CTX_ENABLES	Context Enables: Used during debugging to enable a context to run when arbiter grants permission.
CC_ENABLE	Condition Code Enables: The previous and current condition codes.
CTX_n_STS (n = 0 to 3)	Context Status: Status indicating that the context is ready to run, and the program counter where it will start.
CTX_n_SIG_EVENTS (n = 0 to 3)	Context Signal Events: Status of event signals that have occurred (one register per context).
CTX_n_WAKEUP_EVENTS (n = 0 to 3)	Context Wakeup Events: Determines which wakeup event is enabled to wake the context (one register per context).

[1] Specifies the Control Store address being addressed by the StrongARM* core.
[2] Specifies the data at the Control Store Address specified in the USTORE_ADDRESS register.

The local CSRs are mapped into the StrongARM* core 4-Gbyte address space. A Microengine thread can access the set of local CSRs within its Microengine, but it can not access the local CSRs of another Microengine. The Microengines threads have access to all the local CSRs except USTORE_ADDRESS, USTORE_DATA, and ALU_OUTPUT. The **local_csr_rd** and **local_csr_wr** Microengine instructions are used to access the registers.

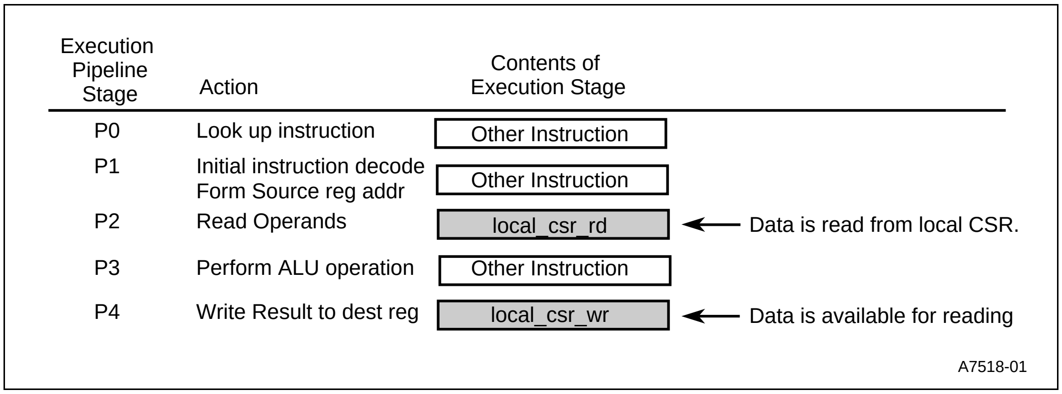
Reading a local CSRs requires two instructions, **local_csr_rd** followed by any instruction that uses immediate data as a source operand. The **local_csr_rd** instruction reads the contents of the CSR and provides the data as immediate data to the very next instruction. The next instruction places the immediate data into a GPR or write Transfer Register. If the next instruction does not contain an immediate data source operand field, then the opportunity to read the CSR data from the previous instruction is lost. Example 4-2 shows how a local CSR is written and then immediately read.

Example 4-2. Reading and Writing a Local CSR

```
alu_shf[reg,temp,b,0x1,<<8]      ;write a 1 to bit 8
local_csr_wr[ctx_enables,reg]    ;set the bit in the CSR
nop                              ;required because of pipeline
local_csr_rd[ctx_enables]        ;setup the data as immediate
alu_shf[csr_data,--,b,0x1,>>8]   ;move data in to LSB of GPR
```

As shown in the example, a **local_csr_rd** must not immediately follow a **local_csr_wr**. This is due to issues concerning the Microengine instruction execution pipeline. Since the CSR data is written in P4, and the CSR is read in P2, a 1-cycle latency must be enforced to ensure the correct data is read.

Figure 4-15. Reading and Writing Local CSRs



Execution Pipeline Stage	Action	Contents of Execution Stage	
P0	Look up instruction	Other Instruction	
P1	Initial instruction decode Form Source reg addr	Other Instruction	
P2	Read Operands	local_csr_rd	← Data is read from local CSR.
P3	Perform ALU operation	Other Instruction	
P4	Write Result to dest reg	local_csr_wr	← Data is available for reading

4.13 Find Bit Set Instructions

The Microengines support six instructions that are used to determine the first bit set within a GPR or Read Transfer Register. The format of the instructions is as follows:

```
find_bset [test_operand], optional_token
find_bset [test_operand, shf_cntl], optional_token
find_bset_with_mask [mask_operand, test_operand], optional_token
find_bset_with_mask [mask_operand, test_operand, shf_cntl], optional_token

load_bset_result1 [dest_reg], optional_token
load_bset_result2 [dest_reg], optional_token
```

The first four instructions return the bit position number of the first set bit in a 16-bit field of a Microengine register. The search begins at the LSB. A shift control token is provided in the instruction so that any arbitrary 16-bit field in the register can be evaluated and a mask value can also be specified so that certain bits are not evaluated.

The result of the operation is deposited into one of two result registers that reside in the Microengine Controller and can not be directly read by the Microengines like the GPRs or Transfer Registers. Instead, the Microengines must explicitly move the contents on the Result registers into one of the Microengine GPRs or Transfer Registers using the **load_bset_result1** and **load_bset_result2** instructions.

The Result registers contains three bit fields described in Table 4-19.

Table 4-19. Result Register Bits

Bit	Name	Description
31:9	RES	Reserved. Read as 0
8	LK	Indicates that the register is locked and contains a valid result.
7:4	INST#	Indicates the **find_bset** instruction that found a set bit. n = The n + 1 **find_bset** instruction locked the register. Each time a **find_bset** instruction is executed, a 4-bit counter is incremented. When an instruction finds a bit set, the counter value is placed in this field.
3:0	BIT POS	Indicates the bit position within the 16-bit field in which the first set bit was found (0 = lsb; 15 = msb).

When a **find_bset** instruction is executed, the test data and optionally the mask data is moved from a GPR or Transfer Register through the ALU/Shifter and into the Find Bit Set logic. The Microengine Controller may also supply the mask data as immediate data (in this case, the data still passes through the ALU).

After the data is delivered to the Find Bit Set logic, the logic increments a counter to track how many **find_bset** instructions have been executed. When a **find_bset** instruction finds a bit set, it writes the bit number and the value of the instruction counter, sets the lock bit, and locks the register so that subsequent **find_best** instructions do not overwrite the result register. If the first result register is locked, the second result register will be loaded and locked when the next **find_bset** instruction is executed and detects a bit set in its test value. If both result registers are locked, the result is not reported. When an instruction specifies the clr_result optional token, both result registers unlocked and cleared and the instruction counter is also reset. This allows a programmer to use multiple **find_bset** or **find_bset_with_mask** instructions to find the first set bit within up to 16 consecutive 16-bit fields (i.e., up to 8 registers of data). For example, The following program will find the first bit set in a 32-bit register and the following case examples describe the result.

Example 4-3. Find First Bit Set Program Example

```
find_bset [test_operand], clr_result; clear the result registers and test the lower 16 bits
find_bset [test_operand, >>16]; test the upper 16 bits
nop          ; required for latency (explained later)
nop          ; required for latency
nop          ; required for latency
load_bset_result1 [dest_reg]; read the first result
```

Case 1: If one or more bits were set in the lower 16 bits of the register, the result1 register would contain data that:

- Set the lock bit indicating that a bit was set in the data provided
- Set the instruction number to 0, indicating that the first bit set instruction discovered a first bit set (for this example, assume a bit was set in the lower 16 bits)
- Set the bit position number to the value of the bit position number that was set (0 to 15).

Case 2: If one or more bits were also set in the upper 16 bits of the register, the result1 register would be as in case 1 and the result2 register would contain data that:

- Set the lock bit indicating that a bit was set in the data provided
- Set the instruction number to 1, indicating that the second bit set instruction discovered a bit set (for this example, a bit was also discovered in the upper 16 bits)

- Set the bit position number to the value of the bit position number that was set (0 to 15).

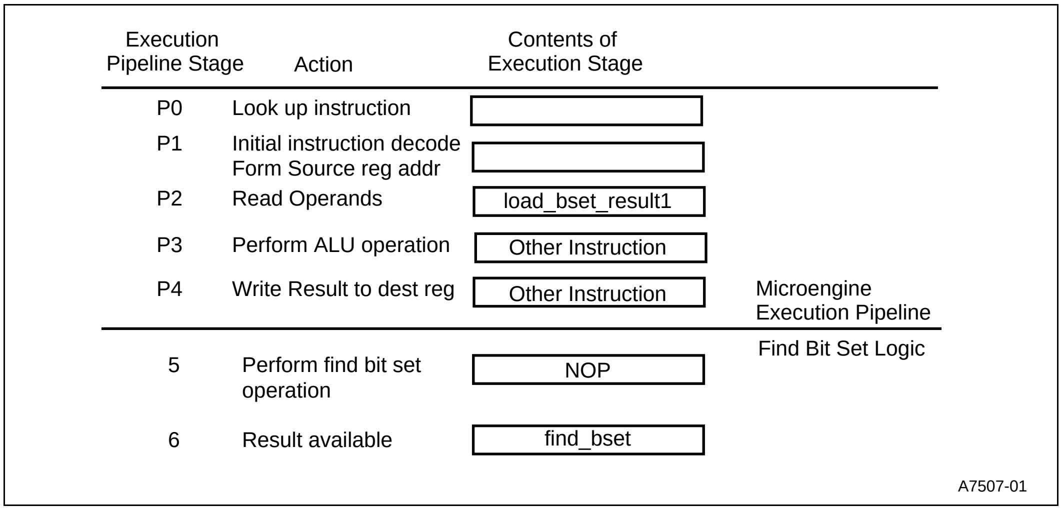
Case 3: If a bit is not set in either the upper or lower 16 bits, both result registers are cleared.

Case 4: If a bit is not set in the lower 16 bits but one or more bits are set in the upper 16 bits, the result1 register contains valid data (as with instruction number set to 1 in case 1) and the result2 register contains zero.

The programmer must explicitly move the data from the Result register to a GPR or Transfer Register using the **load_bset_result1** and **load_bset_result2** instructions. These instructions cause the Microengine Controller to write the Result register data as immediate data that passes through the ALU and into a GPR or Transfer Register.

The programmer must ensure that there are three instructions between the time that a **find_bset** and a **load_bset_result** instruction are executed. This is due to the combined execution pipelines of the Microengine and the Find Bit Set logic. As shown in Figure 4-16, with a three instruction latency, by the time the **find_bset** instruction gets to the Result register, the **load_bset_result1** instruction will have propagated to the read operands stage and the bit set data will be available to read. The find bit set operations should be completed before a context change to ensure that another thread does not clear the result register before the result could be read.

Figure 4-16. Load Bit Set and Find Bit Set Execution



4.14 Input States

Each Microengine is provided with two status signals (referred to as “input states”) from the FBI Unit. A brief description of the input states is provided below. A more detailed description of the FBI Unit is contained in Chapter 6.

state_name	Description
rec_req_avail	This state is set to 1 if there is room for another receive request in the two-entry REC_REQ FIFO. Refer to Section 6.6.4.1, “Issuing a Receive Request” for a description of how this state is used.
push_protect	This state is set to 1 if a Ready Bus Autopush instruction is being executed and the FBI Push engine is currently writing to an SRAM Read Transfer Register. Refer to Section 6.6.3.7, “Autopush Operation” and Section 4.8.2.2, “Managing Unsolicited Autopush Accesses” for a description of how this state is used.

The input states represent the current state of the associated logic in the FBI Unit. The Microengines have access to the input states through the **br_inp_state** instruction. This instruction allows a Microengine thread to test the state, and branch if the state is set.

4.15 Inter-Processor Communications

This section describes the three methods of inter-processor communication provided by the IXP1200.

- **Thread-to-StrongARM* Core Communications:** Allows any program thread executing on any Microengine to communicate with the StrongARM* core. This is supported through interrupts and signal events.
- **Thread-to-Thread Communication:** Allows any program thread executing on any Microengine to communicate with another thread executing on any other Microengine in the same IXP1200. This is supported through signal events.
- **Multiple IXP1200 Communications (Thread-to-Thread):** Allows any program thread executing on any Microengine to communicate with another thread executing on any other Microengine in the different IXP1200s. This is supported through the Ready Bus Get and Send instructions. Refer to Section 6.6.3.10, “Ready Bus Communications” for information on Ready Bus communications.

4.15.1 Generating StrongARM* Core Interrupts

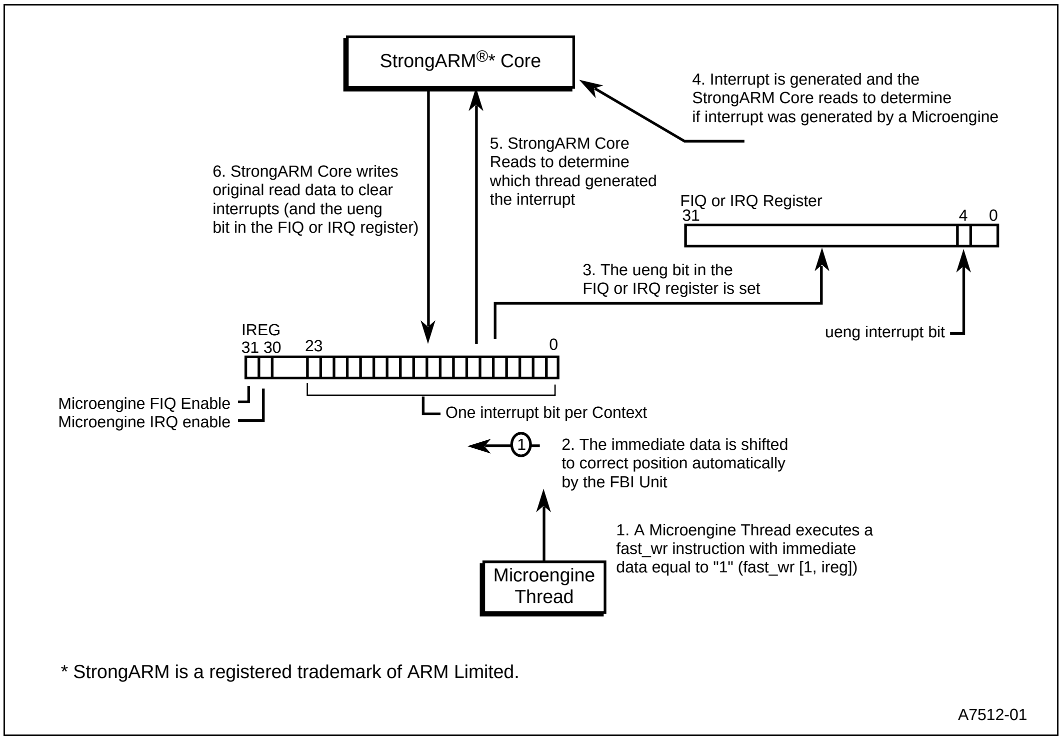
Communication between the Microengine threads and the StrongARM* core is supported in hardware using a signaling mechanism. The individual Microengine threads may interrupt the StrongARM* core by generating either an FIQ or IRQ interrupt. The interrupt type is selected in the IREG register. The StrongARM* core can signal the individual Microengine threads by generating an **inter_thd_sig** signal event. Refer to Figure 4-17.

The Microengine threads generate an interrupt to the StrongARM* core by setting an interrupt bit in the IREG register. The IREG register provides one interrupt bit for each Microengine thread (24 total). All of the Microengine threads set the interrupt bit using the **fast_wr** instruction and to write a value of 1. The IREG register (which resides in the FBI Unit) will shift the 1 to the appropriate bit position based on the Microengine thread that issued the **fast_wr** instruction.

Note: The Microengines should not write values other than 1 to the IREG register since the immediate data may set other bits within the IREG register.

When the StrongARM* core gets an interrupt, it reads either the FIQ or IRQ registers to determine if the interrupt was generated by a Microengine and then reads the IREG register to determine which thread generated the interrupt. From the StrongARM* core perspective, the individual bits in the IREG register are cleared by writing 1s. This allows the StrongARM* core to clear interrupt bits that are set by writing back the original value that was read.

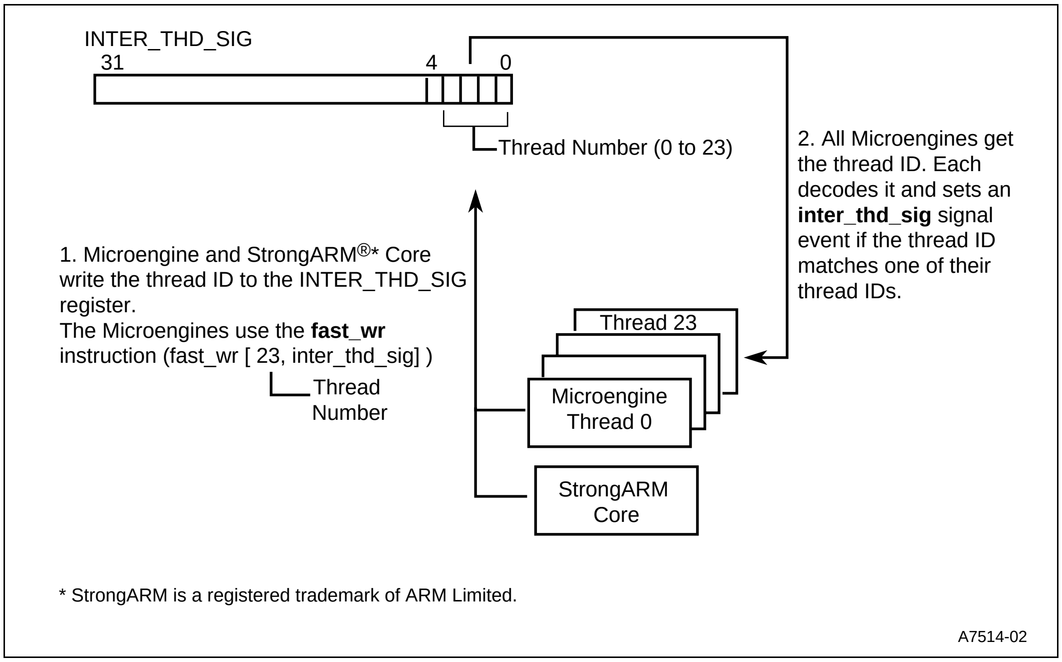
Figure 4-17. StrongARM* Interrupts


StrongARM®* Core
4. Interrupt is generated and the StrongARM Core reads to determine if interrupt was generated by a Microengine
5. StrongARM Core Reads to determine which thread generated the interrupt
6. StrongARM Core writes original read data to clear interrupts (and the ueng bit in the FIQ or IRQ register)
FIQ or IRQ Register
31
4
0
3. The ueng bit in the FIQ or IRQ register is set
ueng interrupt bit
IREG
31 30
23
0
Microengine FIQ Enable
Microengine IRQ enable
One interrupt bit per Context
1
2. The immediate data is shifted to correct position automatically by the FBI Unit
1. A Microengine Thread executes a fast_wr instruction with immediate data equal to "1" (fast_wr [1, ireg])
Microengine Thread
* StrongARM is a registered trademark of ARM Limited.
A7512-01

4.15.2 Generating Inter-thread Signal Events

The StrongARM* core or any Microengine thread can generate an **inter_thd_sig** signal event to any Microengine thread by writing a thread number (0 to 23) to the INTER_THD_SIG register (Figure 4-18). The thread number in this register is broadcast to all Microengines. Each Microengine decodes and the thread address and sets the **inter_thd** bit in the CTX_n_SIG_EVENTS local register if the target is a context in the particular Microengine. A Microengine thread can branch (using the **br_!signal** instruction) or it can wake up on this signal event. Writes to the INTER_THD_SIG are not queued, therefore two back-to-back writes to the same thread ID may result in only one signal.

Figure 4-18. Inter-thread Signal Events


INTER_THD_SIG
31
4
0
Thread Number (0 to 23)
2. All Microengines get the thread ID. Each decodes it and sets an **inter_thd_sig** signal event if the thread ID matches one of their thread IDs.
1. Microengine and StrongARM®* Core write the thread ID to the INTER_THD_SIG register.
The Microengines use the **fast_wr** instruction (fast_wr [23, inter_thd_sig])
Thread Number
Thread 23
Microengine Thread 0
StrongARM Core

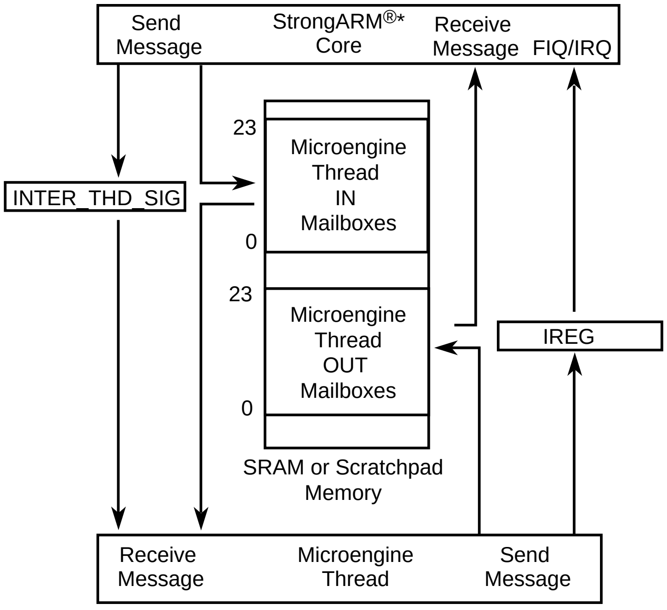
* StrongARM is a registered trademark of ARM Limited.

A7514-02

4.15.3 Communication Example

A software semantic employing mailboxes can be used in conjunction with the signal mechanisms described previously to provide communication between Microengine threads and between the Microengine threads and the StrongARM* core. In the example shown in Figure 4-19, the StrongARM* core writes a message to one of twenty-four Microengine thread "In Mailboxes" and signals the Microengine thread using inter-thread signaling. The Microengine periodically branches on the inter-thread signal and if it detects the signal, it reads its In Mailbox and processes the message. After it completes its task, it can write a response to its "Out Mailbox" and signal the StrongARM* core with an interrupt. The StrongARM* core interrupt service routine determines which thread generated the interrupt and reads that thread's Out Mailbox.

Figure 4-19. StrongARM* Core and Microengine Communication



* StrongARM is a registered trademark of ARM Limited.

A7513-02

4.16 Chained SDRAM References

The Microengine can chain multiple SDRAM references together to ensure each reference will be processed by the SDRAM Unit immediately after the other. The Command Bus Arbiter gives a chained reference a higher service priority over all other requests from the other Microengines to ensure that the commands are delivered successively to the SDRAM Unit. The SDRAM Unit also gives chained references a higher service priority over other Microengines, StrongARM* core, and PCI Unit SDRAM references. Chained referenced are initiated using the chain_ref optional token with the **sdram** instruction. Chained references are supported between SDRAM Transfer Registers and the SDRAM Unit, RFIFO and the SDRAM Unit, and the TFIFO and the SDRAM Unit.

Example 4-4. ;Filling Two TFIFO Elements Using Chained References

```
alu[temp, op1, B, 15, <<16];setup indirect ref - ref_cnt = 16
alu[--, temp, +, elem_0, <<4];setup indirect ref - qword addr
sdram[t_fifo_wr,--,0,sdram_addr,1], indirect_ref, chain_ref

alu[temp, op1, B, 15, <<16];setup indirect ref - ref_cnt = 16
alu[--, temp, +, elem_1, <<4];setup indirect ref - qword addr
sdram[t_fifo_wr, --, 16, addr, 4], indirect_ref ;(no "chain_ref" indicates end of
                                                ; chain)
```

4.17 Debugging Support

Each Microengine provides hardware support for debugging software running on the IXP1200 Network Processor. Intel provides software that takes advantage of these debugging capabilities. This software includes the IXP Developer Workbench and debug libraries that execute on the StrongARM* core. The Debug libraries are written in the C language, allowing it to be ported to different operating systems.

The sections that follow describe the hardware support provided by the IXP1200 and how software can take advantage of this support to provide the following basic debugging capabilities:

- Determining if a Microengine is executing.
- Stopping, starting, and hopping the Microengines.
- Setting breakpoints.
- Reading the local register set.
- Creating a journal.

4.17.1 Determining If a Microengine is Executing

The ACTIVE_CTX_STS (Active Context Status) local CSR indicates whether or not a context is currently executing on the Microengine. If so, it provides the context number and the PC of the instruction it is currently executing. The StrongARM* core can poll this register to determine if the context is running or whether it is idle.

If the Microengine context is idle, the StrongARM* core can read the CTX_n_SIG_EVENTS and CTX_n_WAKEUP_EVENTS registers for each context to determine the signal event for which the current context is waiting and which signal events have occurred.

If none of the Microengine contexts have received a signal event, the StrongARM* core can determine the next instruction that should be executed by reading the current PC for each of the contexts from the CTX_n_STS local CSR.

4.17.2 Stopping, Starting, and Hopping the Microengines

The StrongARM* core can place the Microengines into the Running and Paused states, described in Section 4.6. This allows the Microengine to execute for a period of time and then be placed in a paused state. Once a Microengine is in a paused state, information about the present state of the system (GPRs, Transfer Registers, memory, and CSRs) can be read by the StrongARM* core and transmitted to an I/O debug port (e.g., the serial port, Ethernet interface, etc.).

The user can manually place the Microengines into a paused state or a Microengine can place itself into the paused state automatically each time an instruction that causes a context switch is executed.

Since a Microengine can place itself into the paused state automatically, it is possible for the user to "hop" the execution of the Microengines from one context change point to the next. This allows the user to monitor the progress of a Microengine program as it executes in hardware. The term "hop" is used instead of the common term "step" because the execution progress is monitored after a context switch point rather than each time an instruction is executed.

4.17.3 Breakpoints

Breakpoints are supported by replacing instructions in the program with branch instructions to a breakpoint routine. The StrongARM* core can dynamically insert and remove breakpoints into the Control Store during runtime when a Microengine is in the Paused or Stopped state.

Once a Microengine thread enters a breakpoint routine, a Microengine can place itself into the non-destructive Paused state and interrupt the StrongARM* core. The StrongARM* core can then perform a breakpoint function such as stopping all other Microengines and then reading the Microengine's register set. The StrongARM* core is used to restart the Microengine from the point at which the breakpoint occurs.

Note that breakpoints in the different Microengines occur asynchronously, so a breakpoint may pause one Microengine while the others may continue to execute until the StrongARM* core manually places them into a paused state. Or, they may pause themselves by executing their own breakpoint routine.

Breakpoints are inserted into a program by replacing the instruction where the breakpoint should occur with a branch instruction to a breakpoint routine.

The following procedure describes how the StrongARM* core can insert breakpoints into program while a Microengine is in the paused or stopped state.

1. The StrongARM* core saves the address and the instruction where the breakpoint will occur.
2. The StrongARM* core replaces the instruction where the breakpoint will occur with a branch instruction to a breakpoint routine placed in unused code space.
3. If the user chooses to break only if a certain context hits the breakpoint, the first instruction of the breakpoint routine should be a **br!=ctx[n]** to step 7. This causes the Microengine to execute the replaced instruction and return to the main program. If the user chooses not to break only if a certain context hits the breakpoint, this step can be eliminated.

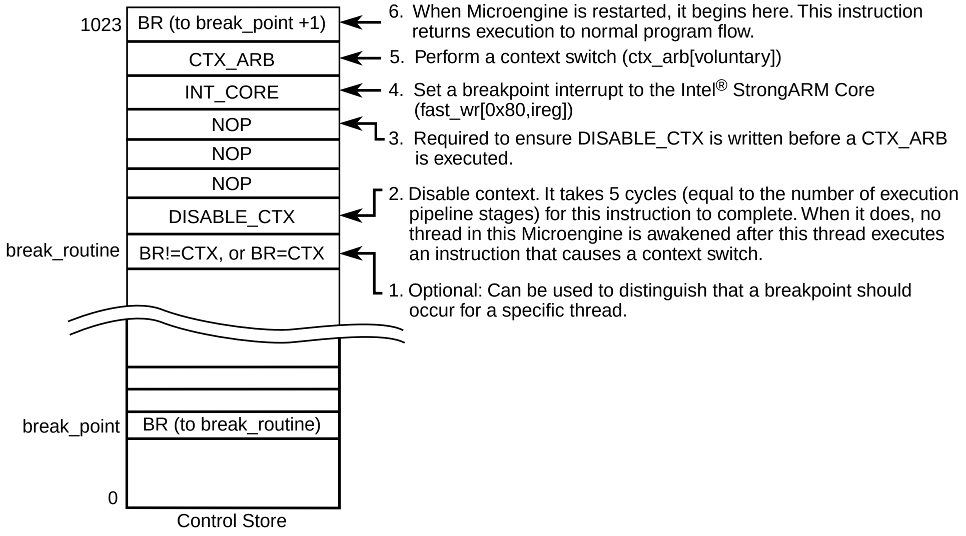
4. The Microengine disables all its other contexts by writing 0x0 to the CTX_ENABLES register using the **local_csr_wr** instruction.
5. The Microengine executes a **fast_wr** instruction to the IREG register, using the data 0x80 to generate a breakpoint interrupt.

Note: The StrongARM* interrupt service routine should be programmed to service the interrupt as desired. For example, the StrongARM* core may stop all the other Microengines and read the Microengine register set and journal area.

6. The Microengine thread then goes to sleep by executing the **ctx_arb[voluntary]** instruction. This allows the Microengine to gracefully stop executing and since all other contexts are disabled, no other thread will begin executing. Due to pipelining considerations, four instructions should be executed after the **local_csr_wr** and before the **ctx_arb** instruction.
7. The StrongARM* core enables the Microengine to begin executing by writing to the CTX_ENABLES register to enable the appropriate threads. The Context Arbiter detects the enabled contexts and allows the program thread specified in the NCTX field of the CTX_ARB_CNTL local register to execute. When a Microengine thread is woken, it will start executing at the instruction following step 7. This is where the original instruction removed from the program in step 1 should be placed.
8. A branch back to the address of the original instruction + 1 enables the Microengine to continue on the normal execution path.

Example 4-5 shows a pseudo code example of a breakpoint routine.

Example 4-5. Breakpoint Routine Example



6. When Microengine is restarted, it begins here. This instruction returns execution to normal program flow.
5. Perform a context switch (ctx_arb[voluntary])
4. Set a breakpoint interrupt to the Intel® StrongARM Core (fast_wr[0x80,ireg])
3. Required to ensure DISABLE_CTX is written before a CTX_ARB is executed.
2. Disable context. It takes 5 cycles (equal to the number of execution pipeline stages) for this instruction to complete. When it does, no thread in this Microengine is awakened after this thread executes an instruction that causes a context switch.
1. Optional: Can be used to distinguish that a breakpoint should occur for a specific thread.

* Other brands and names are the property of their respective owners.

A7974-01

A breakpoint only applies to the Microengine in which the breakpoint is inserted.

4.17.4 Reading Microengine GPR and Read Transfer Registers

When a Microengine is placed into the Stopped or Paused state, and the ECS bit in the USTORE_ADDRESS local CSR is set, the Microengine loops on the instruction specified in the UADDR field in the USTORE_ADDRESS local CSR. The ALU_OUTPUT CSR can be read by the StrongARM* core to view the ALU output from the instruction currently executing at this address. This allows the StrongARM* core to read the GPRs and Read Transfer Registers using the following procedure:

1. The StrongARM* core places the Microengine into either the Stopped or Paused state by disabling all its other contexts. This is accomplished by writing 0x0 to the CTX_ENABLE register.
2. The StrongARM* core loads an **alu** instruction at some unused address in the Control Store. The format of the **alu** instruction is as follows:
   ```
   alu[--,0, B, read_register_address]
   ```
   where:
   "-" = no destination register
   read_register_name = address of the register to read, valid registers are GPRs, SRAM Read registers and SDRAM Read registers.
3. The StrongARM* core reads the ALU_OUTPUT register.
4. The StrongARM* core Repeats steps 2 and 3, changing the **alu** instruction to output different GPRs or Read Transfer Registers

4.17.5 Creating a Journal

Journaling allows a Microengine to write debugging data (or other data) to a location in SRAM. The SRAM Unit assists the Microengines in creating a journal by maintaining the pointers to the journal area in SRAM. There are three pointers that specify the start (base), end, and current position of the journal area. These are initialized via the SRAM_AUTO_BASE, SRAM_AUTO_END, and SRAM_AUTO_PTR SRAM registers.

Once these SRAM registers are initialized, a write to the SRAM_AUTO_PTR register causes the following data to be written to SRAM at the address in the pointer register:

[31:29] Microengine ID/StrongARM* core ID
[28:27] Thread ID
[26:0] User defined data

The SRAM Unit supplies the Microengine ID and thread ID if the source is a Microengine. Otherwise, the StrongARM* core should write the entire 32 bits using a Microengine ID of 7 and a thread ID of 0 (bits [31:27] = 11100). After each write, the current pointer is incremented. When the current pointer equals the end pointer, the next value of the current pointer will wrap around so that the current value equals the start pointer. Journaling is enabled on a per Microengine basis via the ENABLE_SRAM_JOURNALING Microengine CSR. The user can view the journal data by programming the StrongARM* core to read the journal area in SRAM and dump the data to an IXP1200 I/O port (UART, PCI, etc).

The Journal registers should be initialized in the order shown in Table 4-20 to ensure proper operation.

Table 4-20. Journal Register Initialization Order

Initiation Order	SRAM CSR	Reason
1	SRAM_AUTO_BASE	Writing this register sets the base address in the SRAM_AUTO_BASE register.
2	SRAM_AUTO_END	Writing this register resets the SRAM_AUTO_PTR register to the contents of the SRAM_AUTO_BASE register.

PCI Unit 5

5.1 Overview

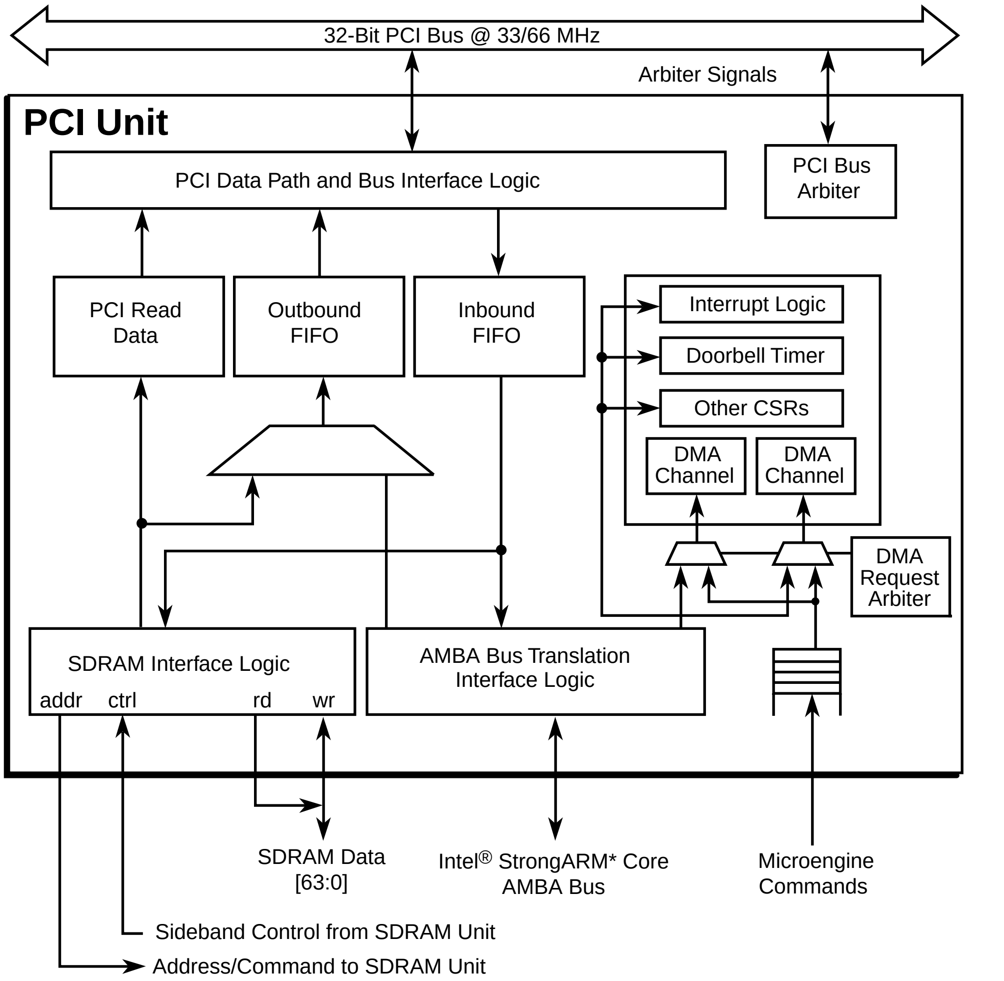
This chapter contains information on the IXP1200 Network Processor Family PCI unit. It is organized as follows:

- Section 5.2, "Hardware Description describes the functional blocks of the PCI unit.
- Section 5.3, "PCI Transactions describes which PCI transactions are supported and how they are implemented by the PCI unit.

As shown in Figure 5-1, the main functional blocks of the PCI unit are as follows:

- Data Path and Bus Interface Logic
- FIFOs
- SDRAM Interface Logic
- AMBA Bus Translation Interface Logic
- PCI bus arbiter
- DMA channels
- I_2O* message unit
- Timers

Figure 5-1. PCI Unit Block Diagram



* Other brands and names are the property of their respective owners.

A7975-01

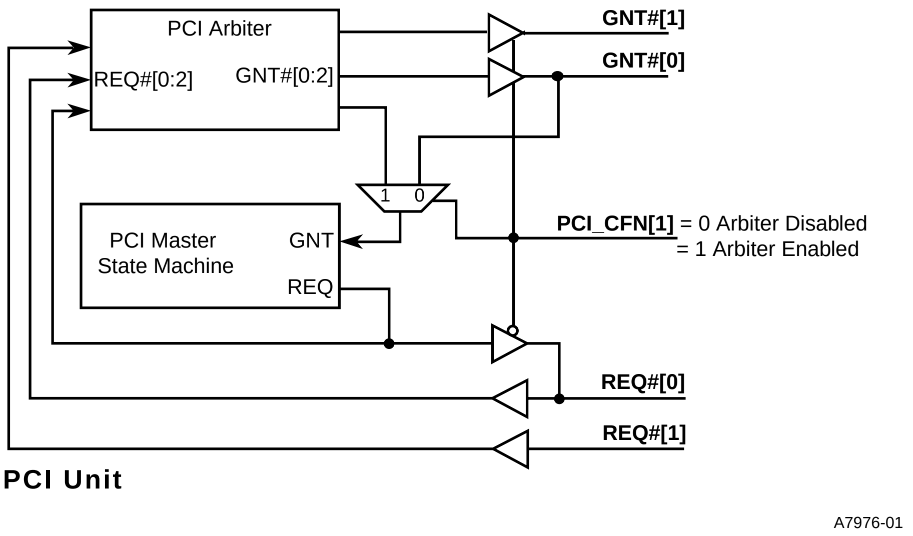
5.2 Hardware Description

5.2.1 PCI Bus Arbiter

The PCI unit contains a PCI bus arbiter that supports two external masters in addition to the IXP1200. To enable the arbiter, the **PCI_CFN[1]** pin must be hardwired to 1. Figure 5-2 illustrates how the arbiter is configured via the **PCI_CFN[1]** pin.

Figure 5-2. PCI Bus Arbiter

Pin	PCI_CFN[1] = 0	PCI_CFN[1] = 1
GNT#[0]	PCI Bus Grant Input to IXP1200	PCI Bus Grant Output to Master 1
GNT#[1]	Not Used, Tied High	PCI Bus Grant Output to Master 2
REQ#[0]	PCI Bus Request Output from IXP1200	PCI Bus Request Input from Master 1
REQ#[1]	Not Used, Float	PCI Bus Request Input from Master 2



The arbiter uses a simple round-robin priority algorithm. The arbiter asserts the grant signal corresponding to the next request in the round-robin during the currently executing transaction on the PCI bus (this is also called hidden arbitration). If the arbiter detects that an initiator has failed to assert **frame_l** after 16 cycles of both grant assertion and PCI bus idle condition, the arbiter deasserts the grant. That master does not receive any more grants until it deasserts its request for at least one PCI clock cycle.

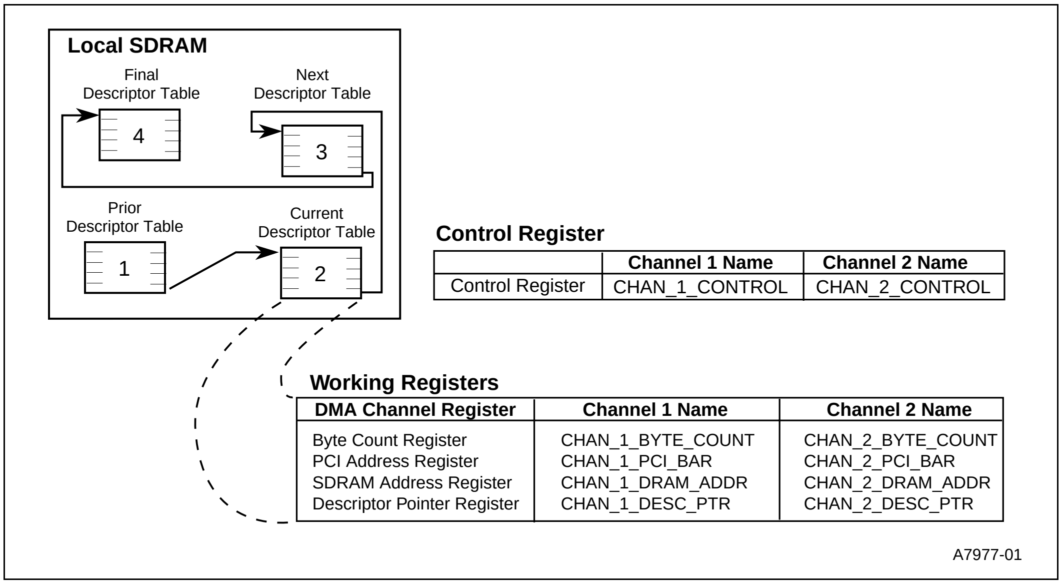
To prevent bus contention, if the PCI bus is idle, the arbiter never asserts one grant signal in the same PCI cycle in which it deasserts another. It deasserts one grant, and then asserts the next grant after at least one full PCI clock cycle has elapsed to provide for bus driver turnaround.

5.2.2 DMA Channels

There are two DMA channels, each of which can move blocks of data from SDRAM to the PCI or from the PCI to SDRAM. The DMA channels read parameters from a list of descriptors in memory, perform the data movement, and stop when the list is exhausted. The descriptors are loaded from predefined SDRAM entries or may be set directly by a configuration write.

Figure 5-3 shows DMA descriptors in local SDRAM. Each descriptor occupies four Dwords and is aligned on a quadword boundary. The DMA channels read the descriptors from local SDRAM into the four DMA working registers once the control register has been set to initiate the transaction. This control register is set automatically for microengine transactions but must be set explicitly for StrongARM* core transactions. This starts the DMA transfer. The register names for the two channels are listed in Figure 5-3. After a descriptor is processed, the next descriptor is loaded in the working registers. This process repeats until the chain of descriptors is terminated (i.e., the End Of Chain bit is set).

Figure 5-3. DMA Descriptor Reads



	Channel 1 Name	Channel 2 Name
Control Register	CHAN_1_CONTROL	CHAN_2_CONTROL

DMA Channel Register	Channel 1 Name	Channel 2 Name
Byte Count Register	CHAN_1_BYTE_COUNT	CHAN_2_BYTE_COUNT
PCI Address Register	CHAN_1_PCI_BAR	CHAN_2_PCI_BAR
SDRAM Address Register	CHAN_1_DRAM_ADDR	CHAN_2_DRAM_ADDR
Descriptor Pointer Register	CHAN_1_DESC_PTR	CHAN_2_DESC_PTR

There is no restriction on byte alignment of the source address or the destination address. DMA reads are always unmasked reads (all byte enables asserted) either from SDRAM or the PCI. For PCI-to-SDRAM transfers, the PCI command is Memory Read, Memory Read Line, or Memory Read Multiple according to the PCI read type field bits [6:5] in the CHAN_1_CONTROL or CHAN_2_CONTROL register. After each read, the byte count is decremented by the number of bytes read, and the source address is incremented by one Dword.

When moving a block of data, the IXP1200 internal hardware adjusts the byte enables so that the write data is aligned properly on Dword boundaries and that only the correct bytes are written if the initial and final data writes require masking.

5.2.2.1 Allocation of the DMA Channels

Both the StrongARM* core and the six microengines can access the two DMA channels. The channels can function in one of the following modes, as determined by the DMA_INF_MODE register:

- The StrongARM* core owns both DMA channels.
- The microengines own both DMA channels.
- The StrongARM* core owns DMA channel 1 and the microengines own DMA channel 2 (default).

The DMA mode (as specified by the DMA_INF_MODE register) can be changed by the StrongARM* core under software control. The StrongARM* core software should signal the microengines to suspend DMA transactions and wait until both DMA channels are free before changing the mode. The StrongARM* core software should determine when both DMA channels are free either by polling IRQ_RAW_STATUS/FIQ_RAW_STATUS register bits [17:16] or by reading the equivalent IRQ_STATUS/FIQ_STATUS register after having set the appropriate mask bits in the IRQ_ENABLE/FIQ_ENABLE register.

5.2.2.2 StrongARM* Core Initiated DMA Channel Operation

The StrongARM* core can either set up the descriptors in SDRAM or it can write the first descriptor table directly to the DMA channel registers.

When descriptors and the descriptor list are in SDRAM, the procedure is as follows:

1. The StrongARM* core writes the address of the first descriptor into the DMA Channel Descriptor Pointer register (CHAN_1_DESC_PTR or CHAN_2_DESC_PTR).
2. The StrongARM* core writes the DMA Channel Control register (CHAN_1_CONTROL or CHAN_2_CONTROL) with miscellaneous control information and also sets the channel enable bit (bit 0). The channel initial descriptor bit (bit 4) in the CHAN_1_CONTROL or CHAN_2_CONTROL register must also be cleared to indicate that the first descriptor is in SDRAM.
3. Depending on the DMA channel number, the DMA channel reads the descriptor block into the CHAN_1_BYTE_COUNT, CHAN_1_PCI_BAR, CHAN_1_DRAM_ADDR, and CHAN_1_DESC_PTR registers or the CHAN_2_BYTE_CNT, CHAN_2_PCI_BAR, CHAN_2_DRAM_ADDR, and CHAN_2_DESC_PTR registers.
4. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done (bit 2) in the CHAN_1_CONTROL or CHAN_2_CONTROL register.
5. If the end of chain bit (bit 31) in the CHAN_1_BYTE_COUNT or CHAN_2_BYTE_COUNT register is clear, the channel reads the next descriptor and transfers the data (steps 3 and 4 above). If bit 31 is set, the channel sets the channel chain done bit (bit 7) in the CHAN_1_CONTROL or CHAN_2_CONTROL register and then stops.
6. If enabled via the IRQ_ENABLE or FIQ_ENABLE registers, the DMA channel interrupts the StrongARM* core when the descriptor list is exhausted.

When single descriptors are written directly into the DMA channel registers, the procedure is as follows:

1. The StrongARM* core writes the descriptor values directly into the DMA channel registers. The end of chain bit (bit 31) in the CHAN_1_BYTE_COUNT or CHAN_2_BYTE_COUNT register must be set, and the value in the CHAN_1_DESC_PTR or CHAN_2_DESC_PTR register is not used.
2. The StrongARM* core writes the base address of the DMA transfer into the CHAN_1_PCI_BAR or CHAN_2_PCI_BAR register.

3. When the first descriptor is in the CHAN_1_BYTE_COUNT or CHAN_2_BYTE_COUNT register, the CHAN_1_DRAM_ADDR or CHAN_2_DRAM_ADDR register must be written with the address of the data to be moved.
4. The StrongARM* core writes the CHAN_1_CONTROL or CHAN_2_CONTROL register with miscellaneous control information, along with setting the channel enable bit (bit 0). The channel initial descriptor in register bit (bit 4) in the CHAN_1_CONTROL or CHAN_2_CONTROL register must also be set to indicate that the first descriptor is already in the channel descriptor registers.
5. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done bit (bit 2) in the CHAN_1_CONTROL or CHAN_2_CONTROL register.
6. Since the end of chain bit (bit 31) in the CHAN_1_BYTE_COUNT or CHAN_2_BYTE_COUNT register is set, the channel sets the channel chain done bit (bit 7) in the CHAN_1_CONTROL or CHAN_2_CONTROL register and then stops.
7. If enabled via the IRQ_ENABLE or FIQ_ENABLE registers, the DMA channel interrupts the StrongARM* core when the descriptor list is exhausted.

5.2.2.3 Microengine Initiated DMA Channel Operation

Microengine initiated DMA channel operations occur as follows:

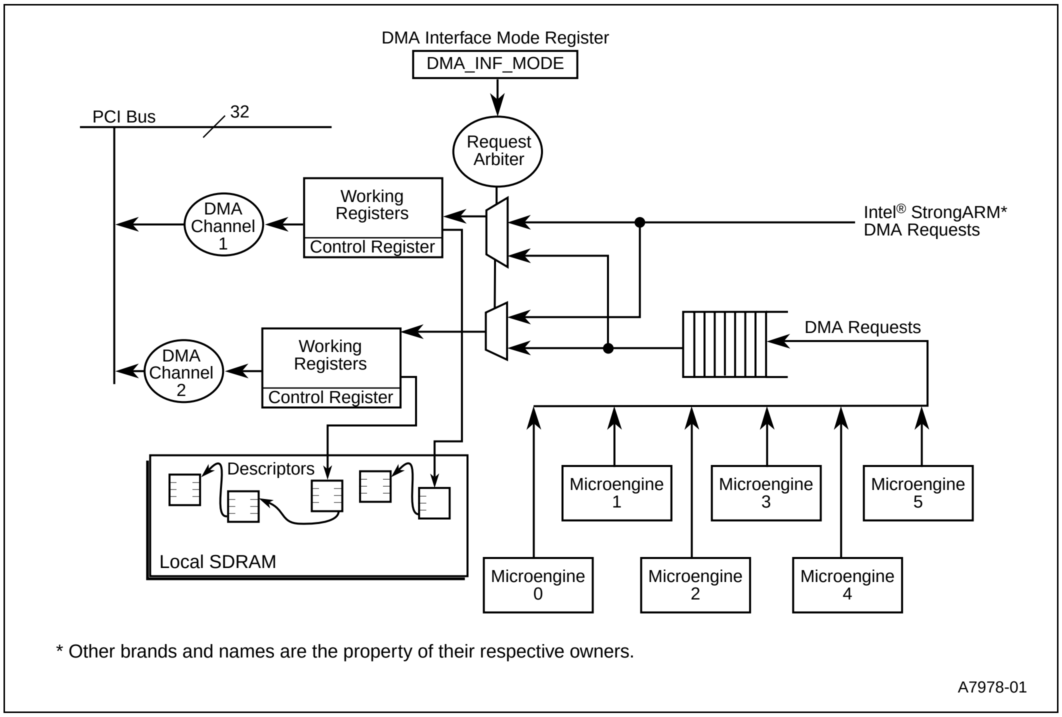
1. A microengine thread sets up the descriptors in SDRAM. Microengines can not write directly to the DMA descriptor register in the PCI unit.
2. The microengine thread initiates a DMA operation by writing the pointer to the first descriptor in SDRAM to the DMA channel using the PCI_DMA instruction.
3. The DMA channel reads the descriptor block into the Channel Control, Channel PCI Address, Channel SDRAM Address, and Channel Descriptor Pointer registers.
4. The DMA channel transfers the data until the byte count is exhausted, and then sets the channel transfer done bit in the DMA Channel Control Register.
5. If the end of chain bit in the DMA Channel Byte Count register is clear, the channel reads the next descriptor and transfers the data. If it is set, the channel sets the chain done bit in the DMA Channel Control Register and then stops.
6. If enabled (via the PCI_DMA instruction), the DMA channel can signal the microengine thread when the descriptor list is exhausted.

There is an eight entry DMA request FIFO that holds pending microengine DMA descriptor requests. This provides for nine outstanding microengine DMA requests, one in the allocated DMA Channel, and eight in the DMA FIFO. There is no hardware back-pressure mechanism to indicate when the DMA FIFO is full so microengine software must maintain status as to the number of outstanding microengine DMA requests that are pending, to ensure it does not exceed the total of nine.

If the microengine owns both DMA channels, DMA requests can be issued to either the next available DMA channel or directly at DMA channel 2. If the PCI_DMA instruction has the **order_queue** flag set, use DMA channel 2. Otherwise, use the next available DMA channel. Targeting a DMA request at channel 2 ensures that DMA requests complete in the order they were issued.

Figure 5-4 illustrates a microengine initiated DMA channel operation.

Figure 5-4. DMA Channel Operations Initiated by Microengine



* Other brands and names are the property of their respective owners.

A7978-01

5.2.2.4 SDRAM-to-PCI Transfer

For a SDRAM-to-PCI transfer, the DMA channel reads data from SDRAM and places it into the outbound FIFO for transfer to the PCI Bus when the following conditions are met:

- There is enough free space in the FIFO.
- The PCI interburst delay for the DMA channel has elapsed (see bits [9:8] of CHAN_1_CONTROL or CHAN_2_CONTROL register).
- The SDRAM controller has completed any previous requests and grants access to the PCI unit.

The number of Dwords read from SDRAM is specified by the read length field bits (bits [18:16]) of the CHAN_1_CONTROL or CHAN_2_CONTROL register. At the beginning or end of a transfer, fewer Dwords may be read depending on alignment and byte count.

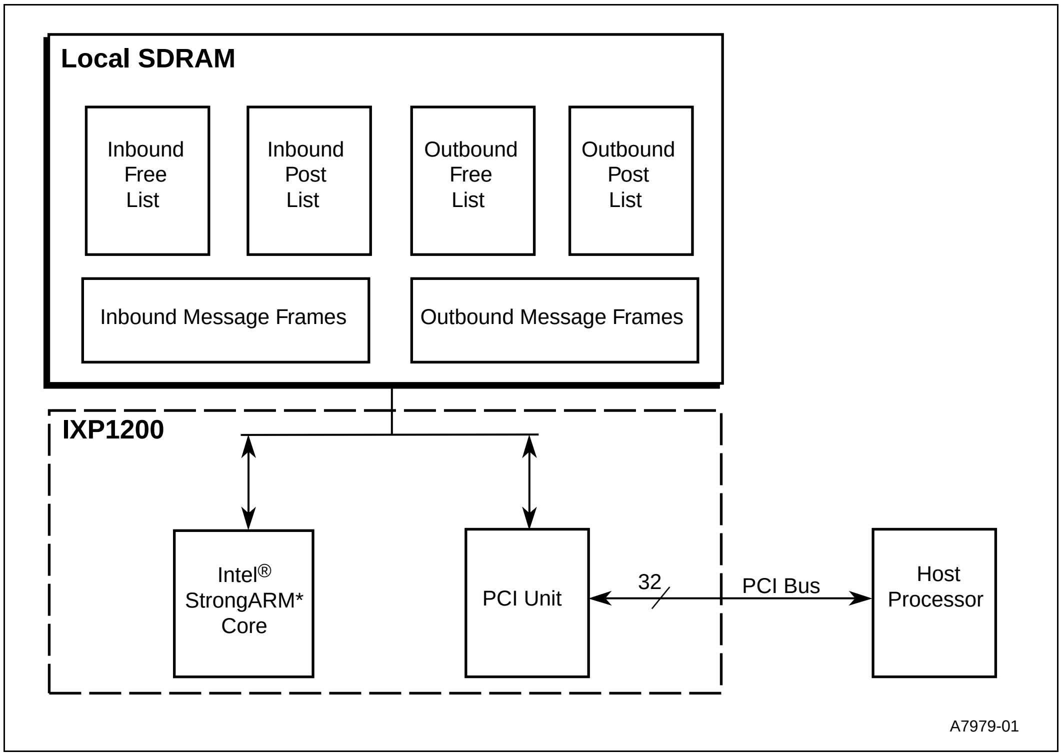
5.2.2.5 PCI-to-SDRAM Transfer

For a PCI-to-SDRAM transfer, the DMA channel issues a sequence of PCI read request commands through the Outbound FIFO. As the data is read, the DMA continues to issue new requests until the specified number of Dwords has been read. If the target disconnects before that number of Dwords has been read, the PCI unit waits for the PCI interburst delay before starting another read. The PCI transaction also terminates if there is insufficient space in the Inbound FIFO to receive the requested data or if the master latency timer expires. In all cases, all Dwords that were read into the Inbound FIFO are written into SDRAM.

5.2.3 I_2O* Message Unit

This section describes the operation of the PCI unit's I_2O message unit. The I_2O message unit provides a standardized message-passing mechanism between a device on the PCI and the StrongARM* core. It provides a means for the host processor on the PCI to read and write lists over the PCI at offsets of 40h and 44h from the first base address.

Figure 5-5. I_20 Overview Diagram



The message unit supports four logical FIFOs located in local SDRAM:

- Inbound free list FIFO
- Inbound post list FIFO
- Outbound free list FIFO
- Outbound post list FIFO

The FIFOs are used to hold message frame addresses (MFAs). The MFAs are offsets (pointers) to the message frames. The PCI unit does not interpret the MFA values other than to recognize the special indicator for an invalid MFA (which is FFFF FFFFh), nor does it access the message frames.

The I_2O Inbound FIFOs are used to manage messages that are I/O requests from the host processor to the StrongARM* core. The I_2O Outbound FIFOs are used to manage messages that are replies from the StrongARM* core to the host processor. The FIFOs are stored in SDRAM. The size of all four FIFOs are of equal size as determined by bits [12:10] in the SA_CONTROL register. Table 5-1 lists the four pointers maintained in the PCI unit and the four pointers that are maintained by the StrongARM* core as variables in software.

Table 5-1. FIFO Pointers

Maintained by the PCI Unit	Maintained by the StrongARM* Core within SDRAM
Inbound free list head (I2O_INB_FLIST_HPTR)	Inbound free list tail (I2O_INB_FLIST_TPTR)
Inbound post list tail (I2O_INB_PLIST_TPTR)	Inbound post list head (I2O_INB_PLIST_HPTR)
Outbound free list tail (I2O_OUTB_FLIST_TPTR)	Outbound free list head (I2O_OUTB_FLIST_HPTR)
Outbound post list head (I2O_OUTB_PLIST_HPTR)	Outbound post list tail (I2O_OUTB_PLIST_TPTR)

5.2.3.1 I_2O Inbound FIFO Operation

The I_2O Inbound FIFO operation is as follows:

- Initialization
  a. The StrongARM* core allocates memory space for both the inbound free list and inbound post list FIFOs, and initializes the inbound pointers (both PCI unit registers and software variables) with the address of the first MFA.

  b. The StrongARM* core initializes the inbound free list FIFO by writing valid MFA values to all entries. For each write to the inbound free list FIFO, the StrongARM* core must also do a write to the inbound free list count (I2O_INB_FLIST_CNT) register to increment the number of entries.
- Host posts an inbound message
  a. When it needs to send a request message, the host processor removes an MFA from the head of the inbound free list (via a read over the PCI bus to the PCI unit register offset 40h).

  b. The host processor writes the request message to the MFA in IXP1200 SDRAM (via a target write over the PCI Bus).

  c. The host processor places the MFA onto the tail of the inbound post list (via a write over the PCI bus to offset 40h). The PCI unit internally increments by 1 the inbound post list count register, which interrupts the StrongARM* core (if not masked by IRQ_ENABLE or FIQ_ENABLE). This interrupt occurs after the write has completed to ensure that the message is in SDRAM before the interrupt.
- StrongARM* core accepts inbound message
  a. The StrongARM* core removes the MFA from the head of the inbound post list. For each read of the inbound post list, the StrongARM* core must also do a write to the inbound post list count register to decrement the number of entries.

  b. The StrongARM* core reads the request message from the MFA and performs the application-specific action based on the message.

  c. The StrongARM* core writes the MFA to the tail of the inbound free list so that the message frame can be reused at a future time. It also writes to the inbound free list count register to increment the number of entries.

  d. It is possible for the host to post more than one message before the StrongARM* core accepts any posted messages. The interrupt to the StrongARM* core remains asserted as long as there is at least one message posted.

5.2.3.2 I_2O Outbound FIFO Operation

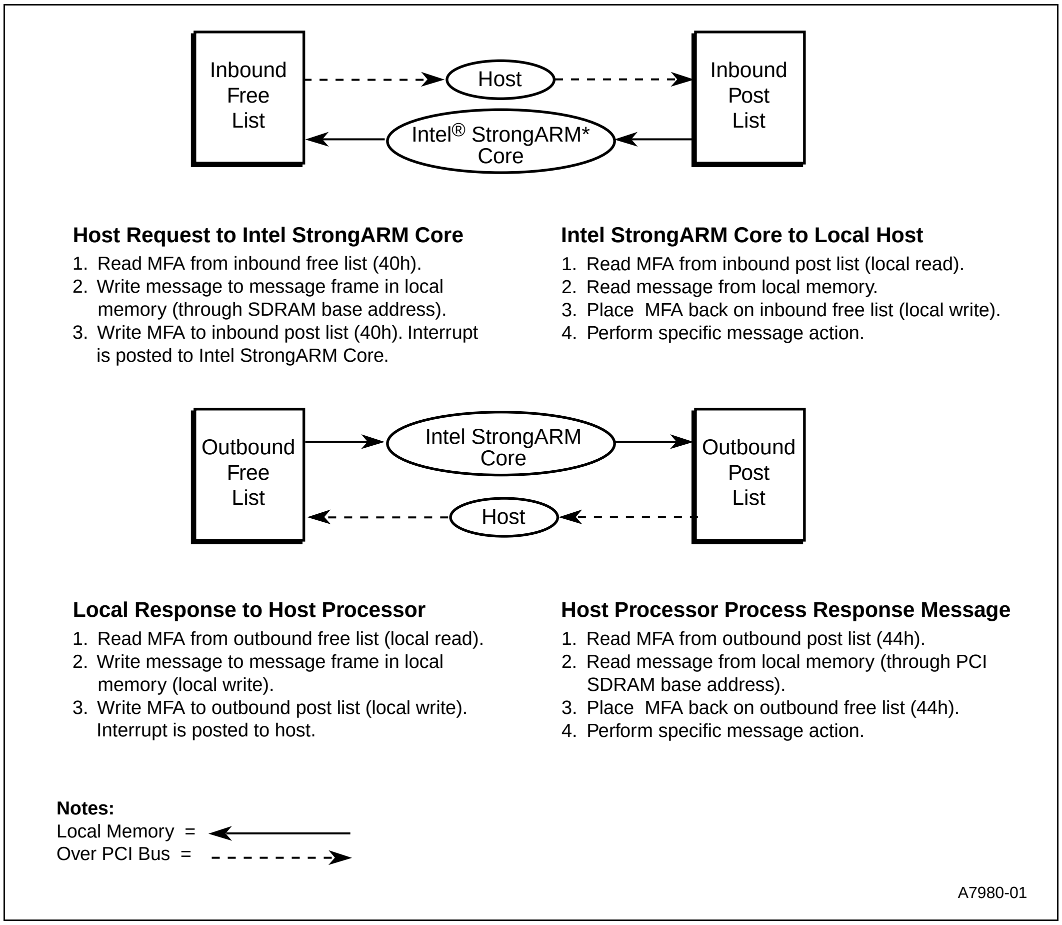
The I_2O Outbound FIFO operation is as follows:

- Initialization
  a. The I_2O outbound FIFOs are initialized by the StrongARM* core by writing the appropriate data into SDRAM.
  b. The StrongARM* core allocates memory space for both the outbound free list and outbound post list FIFOs, and initializes the outbound pointers (both PCI unit registers and software variables) with the address of the first MFA.
  c. The host processor initializes the outbound free list FIFO by writing valid MFAs to all entries.
- StrongARM* core posts an outbound message
  a. When it needs to send a reply message, the StrongARM* core removes an MFA from the head of the outbound free list.
  b. The StrongARM* core writes the reply message to the MFA in local SDRAM.
  c. The StrongARM* core places the MFA onto the tail of the outbound post list. The StrongARM* core must also do a write to the outbound post list count register to increment the number of entries. The PCI unit asserts **pci_irq_l** when the value in the outbound post list count register is not zero (if not masked by outbound interrupt mask register).
- Host accepts outbound message
  a. The host processor removes the MFA from the head of the outbound post list (via a read over the PCI bus to offset 44h). The PCI unit internally decrements the value in the outbound post list count register.
  b. The host processor reads the reply message from the MFA (via reads over the PCI bus) and performs the application-specific action based on the message.
  c. The host processor writes the MFA to the tail of the outbound free list (via a write over the PCI bus to offset 44h) so that the message frame may be reused at a future time.

- Circulation of MFAs

Figure 5-6 shows the circulation of MFAs from free lists to post lists and back. Initially all inbound MFAs are on the inbound free list and all outbound MFAs are on the outbound free list.

Figure 5-6. Circulation of MFAs



Host Request to Intel StrongARM Core

1. Read MFA from inbound free list (40h).
2. Write message to message frame in local memory (through SDRAM base address).
3. Write MFA to inbound post list (40h). Interrupt is posted to Intel StrongARM Core.

Intel StrongARM Core to Local Host

1. Read MFA from inbound post list (local read).
2. Read message from local memory.
3. Place MFA back on inbound free list (local write).
4. Perform specific message action.

Local Response to Host Processor

1. Read MFA from outbound free list (local read).
2. Write message to message frame in local memory (local write).
3. Write MFA to outbound post list (local write). Interrupt is posted to host.

Host Processor Process Response Message

1. Read MFA from outbound post list (44h).
2. Read message from local memory (through PCI SDRAM base address).
3. Place MFA back on outbound free list (44h).
4. Perform specific message action.

5.2.4 Mailbox and Doorbell Registers

Mailbox and Doorbell registers provide hardware support for communication between the StrongARM* core and a host processor on the PCI bus.

Four mailbox registers (MAILBOX_0, MAILBOX_1, MAILBOX_2, and MAILBOX_3) are provided so that messages can be passed between the StrongARM* core and a device on the PCI bus. All four registers are 32 bits and can be read and written with byte resolution from both the StrongARM* core and PCI. How the registers are used is application dependent and the messages are not used internally by the PCI Unit in any way. The mailbox registers are often used with the Doorbell interrupts.

Doorbell interrupts provide an efficient method of generating an interrupt as well as encoding the purpose of the interrupt. The PCI Unit supports a Doorbell register that is used by a PCI device to generate a StrongARM* core FIQ or IRQ interrupt and by the StrongARM* core to generate a PCI interrupt. A source generating the Doorbell interrupt can write a software-defined bitmap to the register to indicate a specific purpose. This bitmap is translated into a single interrupt signal to the destination (either a PCI interrupt or a StrongARM interrupt). When an interrupt is received, the Doorbell registers can be read and the bit mask can be interpreted. If a larger bit mask is required than that provided by the Doorbell register, the Mailbox registers can be used to pass up to four longwords of data.

The doorbell interrupts are controlled through the four registers shown in Table 5-2.

Table 5-2. Doorbell Interrupt Registers

Register Name	Description
DOORBELL	Used to generate the Doorbell interrupts.
DOORBELL_SETUP	Used to initialize the Doorbell register and for diagnostics.
DBELL_PCI_MASK	Used to determine which bits in the Doorbell register generate a PCI interrupt.
DBELL_SA_MASK	Used to determine which bits in the Doorbell register generate a StrongARM interrupt.

The StrongARM* core and PCI devices write to the DOORBELL register to generate up to 32 doorbell interrupts. The DBELL_PCI_MASK register and DBELL_SA_MASK register are used to allocate any of the 32 bits in the DOORBELL register as either generating a StrongARM* core interrupt or generating a PCI interrupt. The DBELL_SA_MASK and DBELL_PCI_MASK bits should be assigned so that the individual Doorbell bits in the DOORBELL register are enabled to interrupt either the PCI device or the StrongARM* core (not both). Normally, the allocation of doorbells is performed during the design of the application by deciding how many unique causes for interrupts are required.

Each bit in the DOORBELL register is implemented as an SR flip-flop. The StrongARM* core writes a 1 to set the flip-flop and the PCI device writes a 1 to clear the flip-flop. Writing a 0 has no effect on the register. The PCI and StrongARM interrupt signals are gated by the DOORBELL Register (output of the SR flip-flop) and the state of the DBELL_SA_MASK and DBELL_PCI_MASK registers.

To summarize, depending on the state of the DBELL_SA_MASK and DBELL_PCI_MASK registers:

To assert an interrupt (i.e., to "push a doorbell"):

- A write of 1 to the corresponding bit of the DOORBELL Register generates an interrupt. This is the case for either PCI device or StrongARM* core, since the write of 1 changes the doorbell bit to the proper asserted state (i.e., 0 for a StrongARM interrupt and 1 for a PCI interrupt).

To dismiss an interrupt:

- A write of 1 to the corresponding bit of the DOORBELL register bit clears an interrupt. This is the case for either PCI device or StrongARM* core, since the write of 1 changes the doorbell bit to the proper deasserted state (i.e., 1 for a StrongARM interrupt and 0 for a PCI interrupt).

Table 5-3 illustrates the register settings and the corresponding interrupt functions.

Table 5-3. Doorbell Interrupt Functions

DBELL_SA_ MASK Bit	DBELL_PCI_ MASK Bit	Action	DOORBELL Bit	Result
0	0	Don’t care.	Don’t care.	Doorbell Bit is not used.
0	1	StrongARM* core writes 1 to DOORBELL.	1	Interrupt to PCI is set.
		PCI device writes 1 to DOORBELL.	0	Interrupt to PCI is cleared.
1	0	PCI device writes 1 to DOORBELL.	0	Interrupt to StrongARM* core is set.
		StrongARM* core writes 1 to DOORBELL.	1	Interrupt to StrongARM* core is cleared.
1	1	Don't care.	1	Illegal.

Figure 5-7 and Figure 5-8 illustrate how a Doorbell interrupt is asserted and cleared by both the StrongARM* core and a PCI device.

Figure 5-7. How the StrongARM* Core Generates Doorbell Interrupts to a PCI Device

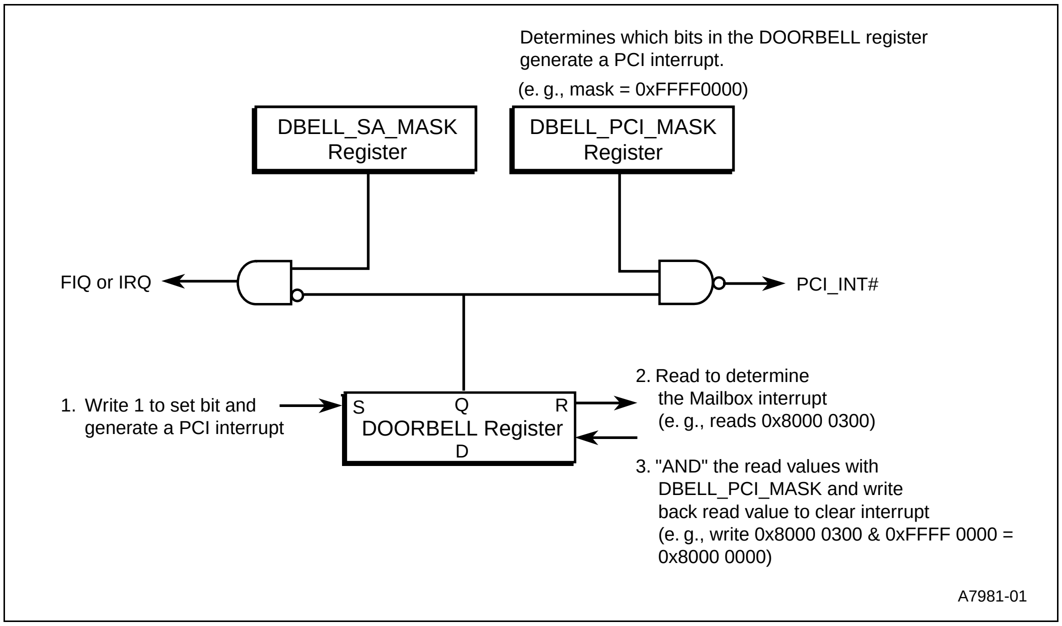
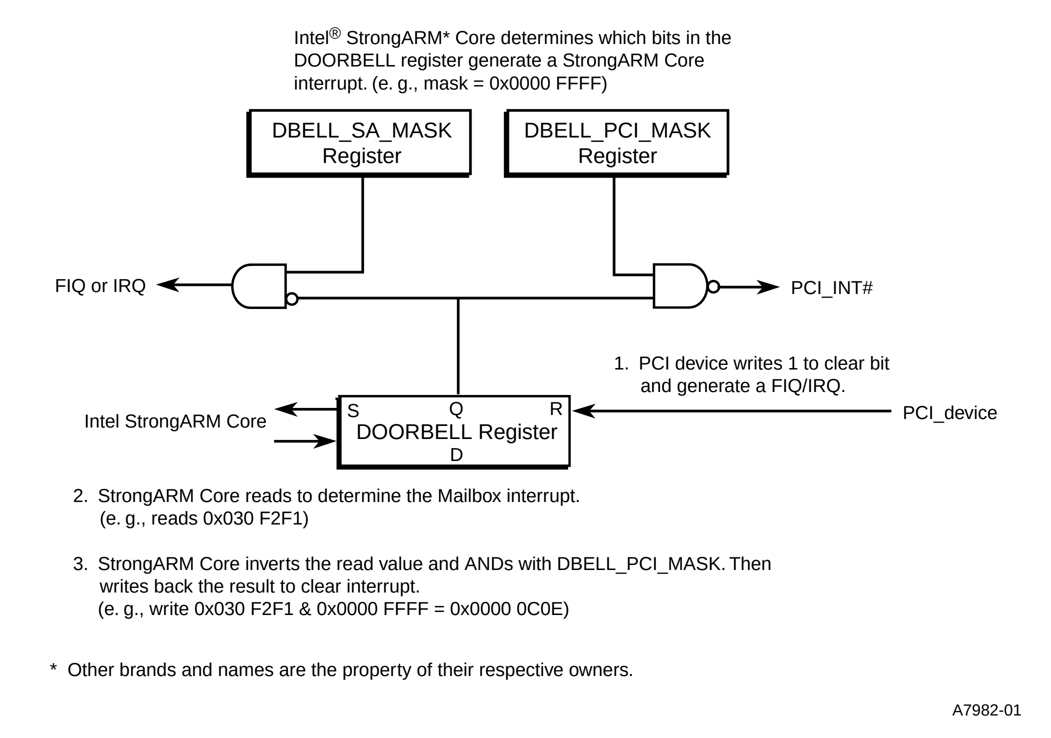

Determines which bits in the DOORBELL register generate a PCI interrupt.
(e. g., mask = 0xFFFF0000)
DBELL_SA_MASK Register
DBELL_PCI_MASK Register
FIQ or IRQ
PCI_INT#
1. Write 1 to set bit and generate a PCI interrupt
S
Q
R
DOORBELL Register
D
2. Read to determine the Mailbox interrupt (e. g., reads 0x8000 0300)
3. "AND" the read values with DBELL_PCI_MASK and write back read value to clear interrupt (e. g., write 0x8000 0300 & 0xFFFF 0000 = 0x8000 0000)
A7981-01

Figure 5-8. How the PCI Device Generates Doorbell Interrupts to the StrongARM* Core



Intel® StrongARM* Core determines which bits in the DOORBELL register generate a StrongARM Core interrupt. (e. g., mask = 0x0000 FFFF)

1. PCI device writes 1 to clear bit and generate a FIQ/IRQ.

2. StrongARM Core reads to determine the Mailbox interrupt. (e. g., reads 0x030 F2F1)

3. StrongARM Core inverts the read value and ANDs with DBELL_PCI_MASK. Then writes back the result to clear interrupt. (e. g., write 0x030 F2F1 & 0x0000 FFFF = 0x0000 0C0E)

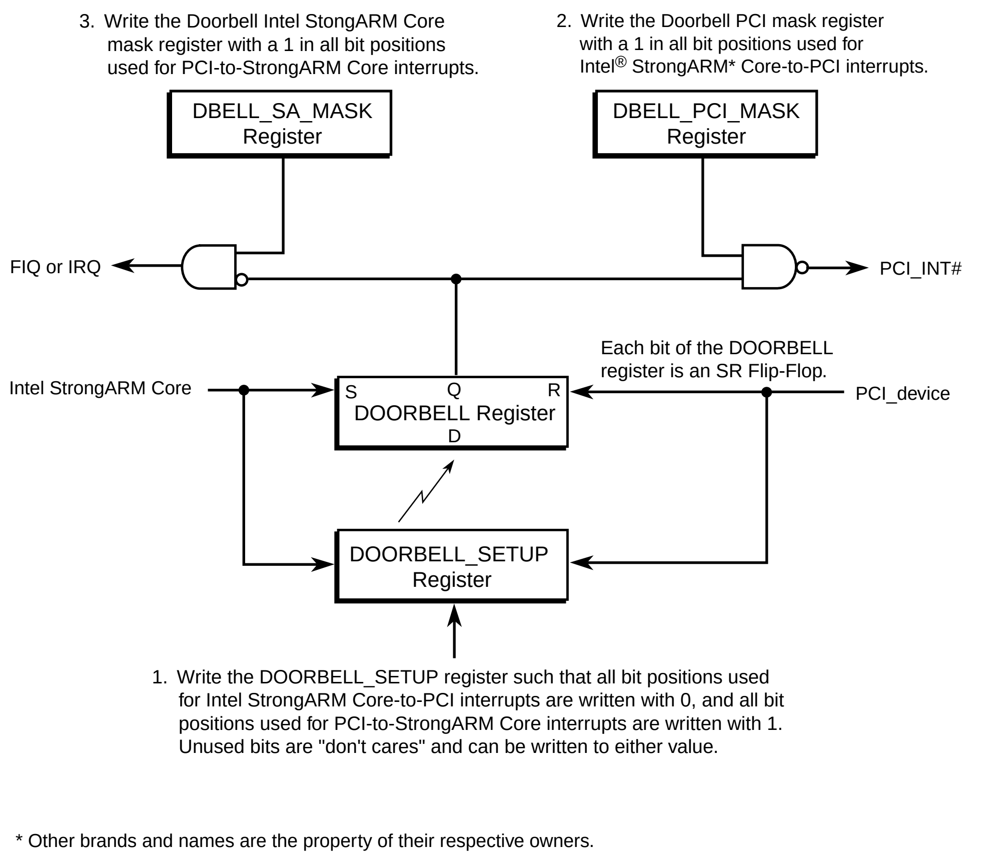
* Other brands and names are the property of their respective owners.

The DOORBELL_SETUP register allows the StrongARM* core and a PCI device to perform two functions that are not possible using the DOORBELL register. This register is used during setup and diagnostics and is not used during normal doorbell operations. First, it allows the StrongARM* core and PCI device to clear an interrupt that it has generated to the other device. For example, during initialization, the StrongARM* core should clear the PCI Doorbell interrupts by writing to the DOORBELL_SETUP register. Second, it allows the StrongARM* core and PCI device to generate a doorbell interrupt to itself. This can be used for diagnostic testing. Each bit in the DOORBELL_SETUP register is mapped to the D input of the SR flip-flop of the DOORBELL register so any write operation to the DOORBELL setup register is reflected in the DOORBELL register.

During system initialization, the doorbell registers must be initialized. This is accomplished by doing the following (see Figure 5-9):

1. Write a 0 to all bits of the DOORBELL register for doorbells that interrupt the PCI and a 1 to all bits of DOORBELL register for doorbells that interrupt the StrongARM* core. This is accomplished by writing to the DOORBELL_SETUP register (which is an alias of the DOORBELL register).
2. Write a 1 to all bits of the DBELL_ PCI_MASK register for doorbells that interrupt from the StrongARM* core to the PCI device.
3. Write a 1 to all bits of the DBELL_SA_MASK register for doorbells that interrupt from a PCI device to the StrongARM* core.

Figure 5-9. How the StrongARM* Core Initializes Doorbell Interrupts


3. Write the Doorbell Intel StongARM Core mask register with a 1 in all bit positions used for PCI-to-StrongARM Core interrupts.
2. Write the Doorbell PCI mask register with a 1 in all bit positions used for Intel® StrongARM* Core-to-PCI interrupts.
DBELL_SA_MASK Register
DBELL_PCI_MASK Register
FIQ or IRQ
PCI_INT#
Each bit of the DOORBELL register is an SR Flip-Flop.
Intel StrongARM Core
S
Q
R
DOORBELL Register
D
PCI_device
DOORBELL_SETUP Register
1. Write the DOORBELL_SETUP register such that all bit positions used for Intel StrongARM Core-to-PCI interrupts are written with 0, and all bit positions used for PCI-to-StrongARM Core interrupts are written with 1. Unused bits are "don't cares" and can be written to either value.
* Other brands and names are the property of their respective owners.
A7983-01

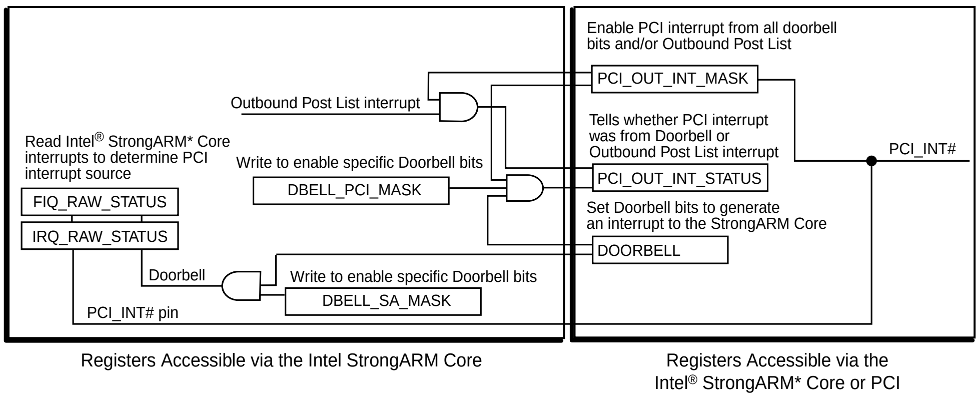
5.2.5 PCI Interrupt Pin

An external PCI interrupt can be generated in two ways:

1. A Message Frame Address (MFA) is added to the outbound post list (i.e., the outbound post list is empty, or
2. The StrongARM* core initiates a Doorbell interrupt.

Figure 5-10 shows how PCI interrupts are managed via the PCI and the StrongARM* core.

Figure 5-10. PCI Interrupts


Enable PCI interrupt from all doorbell bits and/or Outbound Post List
PCI_OUT_INT_MASK
Outbound Post List interrupt
Tells whether PCI interrupt was from Doorbell or Outbound Post List interrupt
PCI_INT#
Read Intel® StrongARM* Core interrupts to determine PCI interrupt source
Write to enable specific Doorbell bits
DBELL_PCI_MASK
PCI_OUT_INT_STATUS
FIQ_RAW_STATUS
Set Doorbell bits to generate an interrupt to the StrongARM Core
IRQ_RAW_STATUS
DOORBELL
Doorbell
Write to enable specific Doorbell bits
DBELL_SA_MASK
PCI_INT# pin
Registers Accessible via the Intel StrongARM Core
Registers Accessible via the Intel® StrongARM* Core or PCI

* Other brands and names are the property of their respective owners.

A7984-01

5.3 PCI Transactions

All PCI transactions occur between the StrongARM* core, microengines (via the DMA channels), PCI unit, and SDRAM unit. A description of the three FIFOs used in performing the transactions and their respective data follows:

- Outbound FIFO
  - StrongARM* core write addresses and data for PCI
  - StrongARM* core read addresses for PCI
  - DMA write addresses and data for PCI
  - DMA read addresses for PCI
- Inbound FIFO
  - PCI write addresses and data for SDRAM
  - PCI read addresses for SDRAM
  - DMA read data from PCI to SDRAM
  - StrongARM* core read data from PCI
- PCI Read FIFO
  - PCI read data from SDRAM

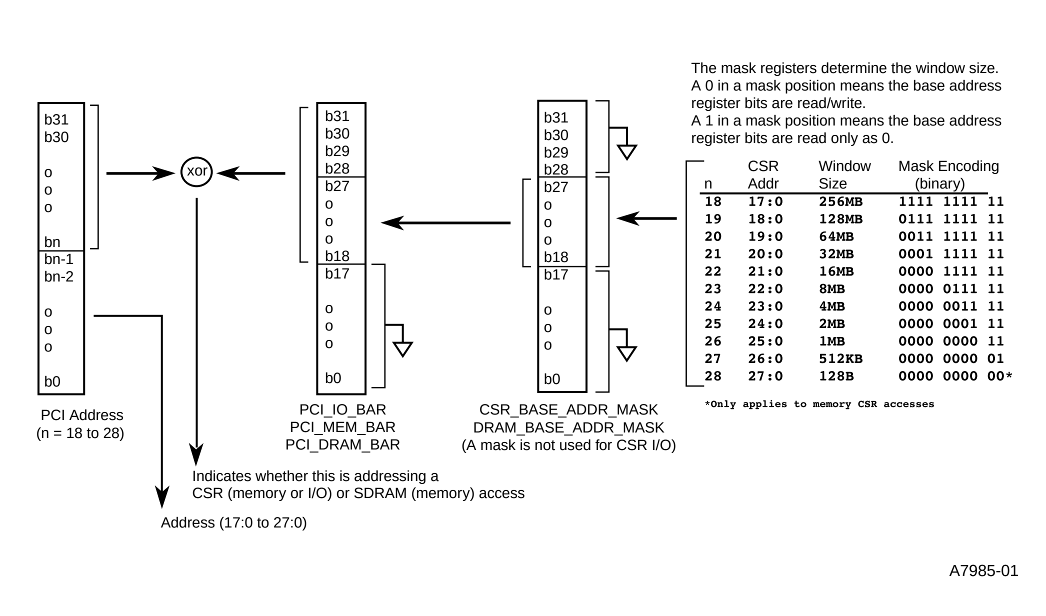
5.3.1 Generating the Address

This section describes how the internal and PCI addresses are generated.

5.3.1.1 Target Transactions - Internal Address Generation

When the IXP1200 is a target, the internal CSR or SDRAM address is generated when the PCI address matches the appropriate base address register (Figure 5-11). Mask registers are provided to set the window size into the SDRAM address space.

Figure 5-11. Target Transactions - Internal Address Generation



The mask registers determine the window size.
A 0 in a mask position means the base address register bits are read/write.
A 1 in a mask position means the base address register bits are read only as 0.

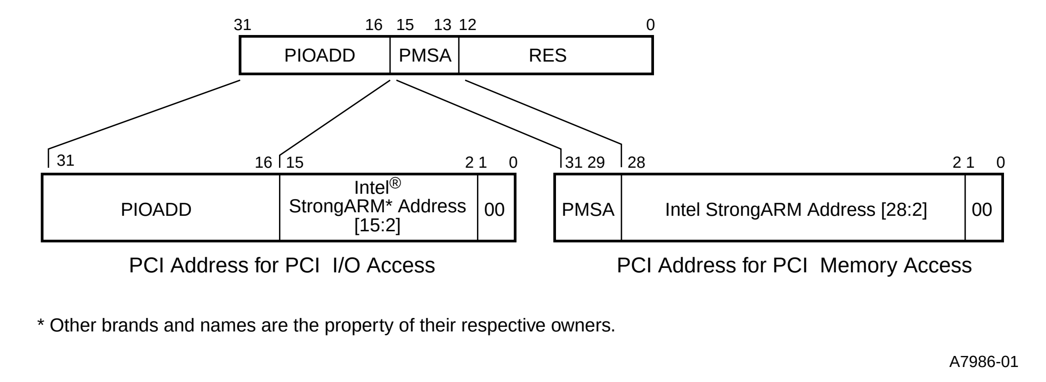
n	CSR Addr	Window Size	Mask Encoding (binary)
18	17:0	256MB	1111 1111 11
19	18:0	128MB	0111 1111 11
20	19:0	64MB	0011 1111 11
21	20:0	32MB	0001 1111 11
22	21:0	16MB	0000 1111 11
23	22:0	8MB	0000 0111 11
24	23:0	4MB	0000 0011 11
25	24:0	2MB	0000 0001 11
26	25:0	1MB	0000 0000 11
27	26:0	512KB	0000 0000 01
28	27:0	128B	0000 0000 00*

*Only applies to memory CSR accesses

5.3.1.2 Master Transactions - PCI Address Generation

When the IXP1200 is a master, the PCI address is generated based on the PCI address extension register (PCI_ADDR_EXT). Refer to Figure 5-12.

Figure 5-12. Master Transactions - Internal Address Generation


31
16
15
13
12
0
PIOADD
PMSA
RES
31
16
15
2 1
0
PIOADD
Intel® StrongARM* Address [15:2]
00
PCI Address for PCI I/O Access
31 29
28
2 1
0
PMSA
Intel StrongARM Address [28:2]
00
PCI Address for PCI Memory Access
A7986-01

* Other brands and names are the property of their respective owners.

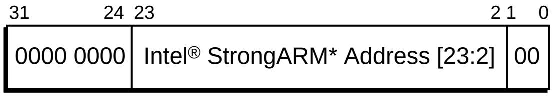
5.3.1.3 Master Configuration Transactions - PCI Address Generation

When the IXP1200 is a master and the StrongARM* core performs a configuration cycle access, the PCI address is generated based on the StrongARM address as shown in Table 5-4 and Figure 5-13.

Table 5-4. Master Configuration Transactions

Cycle	Conditions
Type 1 Configuration Cycle	None.
Type 0 Configuration Cycle	StrongARM* core address bits [23:22] are NOT equal to 11.

Figure 5-13. Master Configuration Transactions



* Other brands and names are the property of their respective owners.

A7987-01

5.3.2 Enabling PCI Bus Transactions

The IXP1200 PCI unit is initialized to an inactive, disabled state. No CSR or other accesses are accepted by the PCI unit until the StrongARM* core has set the Initialize complete bit (bit 0) in the SA_CONTROL register. This bit should be set after the StrongARM* core has initialized the various PCI base address and mask registers..

5.3.3 PCI Target Transactions

This section describes the target response of the PCI unit to various PCI cycles.

5.3.3.1 Unsupported PCI Cycles As Target

The following PCI transactions are not supported by the IXP1200 as a target:

- I/O write to SDRAM
- I/O read to SDRAM
- Type 1 configuration write
- Type 1 configuration read
- Special cycle
- IACK cycle
- Dual-address cycle

The following commands are aliased:

- Memory Write and Invalidate is aliased to a memory write.

- Memory Read Line and Memory Read Multiple to the CSR address space (not SDRAM space) are aliased to a memory read.

5.3.3.2 Memory Write to SDRAM (Target Write)

PCI memory write to SDRAM occurs if the PCI address matches the SDRAM base address register (PCI_DRAM_BAR) or the CSR base address register (PCI_MEM_BAR), and the PCI command is either a memory write or a memory write and invalidate.

The PCI memory write data is collected in the Inbound FIFO and written to SDRAM when the SDRAM is available. The PCI unit requests the SDRAM at the end of each eight Dword boundary of the PCI burst (or at the end of the burst). If PCI address bits [1:0] are not 00 (that is, nonlinear increment mode), and the master attempts to continue the burst past the first Dword, the PCI unit signals a target disconnect.

If fewer than 16 free longwords are available in the Inbound FIFO at the start of the write, the PCI unit signals retry to the PCI master. If the Inbound FIFO fills during the write, the PCI unit signals target disconnect to the PCI master.

5.3.3.3 Memory Read, Read Line, Read Multiple to SDRAM (Target Read)

A PCI memory read from SDRAM occurs if the PCI address matches the SDRAM base address register (PCI_DRAM_BAR) or the CSR base address register (PCI_MEM_BAR), and the command is either a Memory Read, Memory Read Line, or Memory Read Multiple.

The read is completed as a PCI delayed read. That is, on the first occurrence of the read, the PCI unit signals a retry to the PCI master. If there are no prior reads pending, the PCI unit latches the address and command and places it into the Inbound FIFO. When the address reaches the head of the FIFO, the PCI unit reads the SDRAM. If the delayed read latch is full and a new read to a different address is attempted, that read gets a retry response and no change is made in the delayed read latch. In other words, the new read does not displace the in-progress read.

When the read data is returned from SDRAM into the PCI Read FIFO, the PCI unit begins decrementing its discard timer. If the PCI bus master has not repeated the read by the time the discard timer reaches zero, the PCI unit discards the read data, invalidates the delayed read address, and sets Discard Timer Expired (bit 8) in the StrongARM* core Control Register (SA_CONTROL). If enabled, the PCI unit interrupts the StrongARM* core. The discard timer counts 2^{15} (32768) PCI clocks.

When the master repeats the read command, the PCI unit compares the address and checks that the command is a Memory Read, a Memory Read Line, or a Memory Read Multiple (that is, all memory read command types are aliased for a match). If there is a match, the response is as follows:

- If the read data has not yet been read from SDRAM, the response is retry.
- If the read data has been read from SDRAM, assert **trdy_l** and deliver the data. If the master attempts to continue the burst past the amount of data read from SDRAM, the PCI unit signals a target disconnect.

A Memory Read Multiple command initiates a streaming prefetch from SDRAM so that it can supply read data at the maximum PCI bus data rate. Streaming prefetch works as follows:

- The PCI unit reads 32 Dwords when the read is initially queued.

- When the PCI master repeats the read, the PCI unit starts to deliver data. When the PCI master consumes the 17th Dword, the PCI unit reads the next 16 Dwords from SDRAM.
- As long as the PCI master continues to consume the 16th Dword of subsequent blocks, the PCI unit reads the next 16 Dwords.

The PCI unit never deasserts **trdy_l** while supplying SDRAM read data. It deasserts **trdy_l** only after all prefetched data has been transferred.

Streaming prefetch does not start if a PCI write to SDRAM is queued into the Inbound FIFO between the delayed read being started and the read data being delivered. This allows the write data to be written to SDRAM. In this case, only the first 32 Dwords are read. Streaming prefetch stops when either of the following events occur:

- The PCI master ends the cycle (by deasserting **frame_l**). In this case, the PCI unit immediately discards any remaining prefetched data.
- Other bus activity between the StrongARM* core or microengines and SDRAM prevents data from being available in time. This is the target disconnect case, and the PCI unit never negates **trdy_l** for the burst.

5.3.3.4 Type 0 Configuration Write

A PCI configuration write to a configuration register occurs when the following conditions are satisfied:

- **idsel** is asserted,
- The PCI command is a configuration write, and
- The PCI address bits [1:0] are 00.

The PCI write data is written to a configuration register selected by PCI address bits [7:2]. The PCI byte enables determine which bytes are written. If a nonexistent configuration register is selected within the configuration register address range, the data is discarded and no error action is taken. If the PCI master attempts to do a burst longer than one Dword, the PCI unit signals a target disconnect.

5.3.3.5 Type 0 Configuration Read

A PCI configuration read to a configuration register occurs when the following conditions are satisfied:

- **idsel** is asserted,
- The PCI command is a configuration read, and
- The PCI address bits [1:0] are 00.

The data from the configuration register selected by PCI address bits [7:2] is returned on **ad[31:0]**. If a nonexistent configuration register is selected within the configuration register address range, the data returned are zeros and no error action is taken. If the PCI master attempts to do a burst longer than one Dword, the PCI unit signals a target disconnect.

5.3.3.6 Write to CSR

A PCI write to a CSR occurs if either of the following conditions are satisfied:

- The PCI address matches the CSR memory base address register (PCI_MEM_BAR), and the PCI command is either a Memory Write or Memory Write and Invalidate.
- The PCI address matches the CSR I/O base address register (PCI_IO_BAR), and the PCI command is an I/O write.

The data is written to the CSR with offset equal to PCI address bits [7:2]. The I_2O Outbound FIFO and I_2O Inbound FIFO are handled differently (see Section 5.3.3.8). The PCI byte enables determine which bytes are written. If a nonexistent CSR is selected within the CSR address range, the data is discarded and no error action is taken. If the PCI master attempts to do a burst longer than one Dword, the PCI unit signals a target disconnect.

5.3.3.7 Read to CSR

A PCI read to a CSR occurs if either of the following conditions are satisfied:

- The PCI address matches the CSR memory base address register (PCI_MEM_BAR), and the PCI command is either Memory Read, Memory Read Line, or Memory Read Multiple.
- The PCI address matches the CSR I/O base address register (PCI_IO_BAR), and the PCI command is an I/O read.

The data from the CSR with an offset equal to PCI address [7:2] is returned on **ad[31:0]**. The I_2O Outbound FIFO and I_2O Inbound FIFO (memory write offsets 40h and 44h) are handled differently (see Section 5.3.3.9). If a nonexistent CSR is selected within the CSR address range, the data returned is zeros. If the PCI master attempts to do a burst longer than one Dword, the PCI unit signals a target disconnect.

5.3.3.8 Write to I_2O Address

A PCI write to I_2O address space occurs when the following conditions are satisfied:

- The PCI address matches the CSR memory base address register (PCI_MEM_BAR),
- The PCI command is either Memory Write or Memory Write and Invalidate, and
- The register offset is either 40h or 44h.

A write address of 40h causes data to be written to SDRAM using the indirect address defined in I2O_INB_PLIST_TPTR and also increments this register. Similarly, a write address of 44h results in a data write to the address defined in I2O_OUTB_FLIST_TPTR and also increments this register. If the PCI master attempts to do a burst longer than one Dword, the PCI unit signals a target disconnect.

Write data is discarded if the PCI address matches the CSR I/O base address register PCI_IO_BAR), the PCI command is an I/O write, and the register offset is either 40h or 44h.

5.3.3.9 Read to I_2O Address

A PCI read to I_2O address space occurs when all three statements are true:

- The PCI address matches the CSR memory base address register (PCI_MEM_BAR),
- The PCI command is either Memory Read, Memory Read Line, or Memory Read Multiple
- The I_2O Outbound FIFO (I2O_OUTB_FIFO) or I_2O Inbound FIFO (I2O_INB_FIFO) register is addressed (offset 40h or 44h).

This read is completed as a delayed read. On the first occurrence of the read, the PCI unit signals a retry to the PCI master. If the delayed read latch is not full, the PCI unit latches the address and command, and places it into the Inbound FIFO.

When the read address reaches the head of the FIFO, the PCI unit reads the SDRAM at the address in the I_2O inbound free list head pointer (I2O_INB_FLIST_HPTR) register (offset 40h), or the I_2O outbound post list head pointer (I2O_OUTB_PLIST_HPTR) register (offset 44h). If the list being read is empty, the value FFFF FFFFh is substituted for the data read from the SDRAM.

When the master repeats the read command, the PCI unit compares the address and checks that the command is a Memory Read, a Memory Read Line, or a Memory Read Multiple (that is, all memory read command types are aliased for a match). If there is a match, the response is as follows:

- If the read data has not yet been read from SDRAM, the response is retry.
- If the read data has been read from SDRAM, assert **trdy_l** and deliver the data.

If the PCI master attempts to do a burst longer than one data phase, the PCI unit signals a target disconnect.

If the delayed read latch is full and a new read from a different address is attempted, that read gets a retry and no change is made in the delayed read latch. In other words, the new read does not displace the in-progress read.

If the PCI address matches the CSR I/O base address register (PCI_IO_BAR), the PCI command is an I/O read, and the register offset is either 40h or 44h. The PCI unit returns 0 for the read data.

5.3.4 PCI Master Transactions

The following sections describe the PCI master transactions performed by the PCI unit. All PCI master transactions performed by the PCI unit are caused by either StrongARM* core loads and stores that fall into the various PCI address ranges or by DMA channels. PCI write cycles are caused by StrongARM* core writes (stores) and SDRAM-to-PCI DMAs. PCI read cycles are caused by StrongARM* core reads (loads) and PCI-to-SDRAM DMAs. The command register (PCI_COMMAND) bus master bit must be set for the PCI unit to perform any of the transactions in this section.

5.3.4.1 Unsupported PCI Cycles As Master

PCI dual address cycles (DACs) are not supported by the PCI unit as a master.

5.3.4.2 Memory Write, Memory Write and Invalidate

This section describes Memory Write and Memory Write and Invalidate commands from the DMA channels. The PCI unit attempts to use a Memory Write and Invalidate command when the following conditions are met. If any condition is not met, the PCI unit uses the Memory Write command.

- Memory write and invalidate enable bit in the command register (PCI_COMMAND) is a 1.
- Cache line size is set to a value of 4, 8, or 16 in the CACHE_LNSIZE register. Supported during DMA transfers only. For master transactions initiated by the StrongARM* core, StrongARM* core software should limit the burst size to one Dword.

- The address of the first Dword in the burst is aligned to a cache line as defined in the cache line size (CACHE_LNSIZE) register.
- The number of Dwords in the burst is an integer multiple of the cache line size.
- There are no unoccupied bytes in the burst.

The following general rules apply to the memory write command transactions regardless of whether the command was initiated by the StrongARM* core or the DMA channels:

- If the PCI unit receives either a target retry response or a target disconnect response before all of the write data has been delivered, it resumes the transaction at the first opportunity, using the address of the first undelivered Dword.
- If the PCI unit receives a master abort, it discards all of the write data from that transaction and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core.
- If the PCI unit receives a target abort, it discards all of the remaining write data from that transaction, if any, and sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM* core.
- The PCI unit can deassert **frame_l** prior to delivering all data due to the master latency timer. If this occurs, it resumes the memory write at the first opportunity, using the address of the first undelivered Dword.

StrongARM* core CommandsPCI Memory Write or Memory Write and Invalidate commands occur when the StrongARM* core writes to the PCI memory space. The PCI address is derived from the StrongARM address and the PCI address extension register (PCI_ADDR_EXT). The PCI byte enables for each data phase are generated as appropriate based on the StrongARM* core store instruction.

DMA Channels CommandsPCI Memory Write and Memory Write and Invalidate commands are initiated from a DMA channel programmed to perform a SDRAM-to-PCI transfer. The PCI address is based on the DMA channel PCI address register (CHAN_1_PCI_BAR, CHAN_2_PCI_BAR). The PCI byte enables are all asserted with the possible exception of during the first or last Dword of a DMA transfer. The length of the burst is normally 8 or 16 Dwords, with the possible exception of the first or last burst of a DMA transfer.

5.3.4.3 Memory Read, Memory Read Line, Memory Read Multiple

This section describes Memory Read, Memory Read Line, and Memory Read Multiple commands from either the StrongARM* core or DMA channel.

The following general rules apply to the command transactions regardless of whether the command was initiated by the StrongARM* core or the DMA channels:

- If the PCI unit receives a target retry response, it repeats the same PCI command at the first opportunity.
- If the PCI unit receives a master abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core
- If the PCI unit receives a target abort, it sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM.

From StrongARM* core PCI memory reads are done when the StrongARM* core performs reads from the PCI memory space. The PCI address is derived from the StrongARM address and the PCI address extension register (PCI_ADDR_EXT). For Memory Read, the PCI byte enables are generated as appropriate based on the StrongARM* core load instruction. For Memory Read Line and Memory Read Multiple, the PCI byte enables assert for all data phases.

For Memory Read, one Dword is read.

From DMA The PCI command used is specified by the DMA channel PCI read type field in the DMA channel control register (CHAN_1_CONTROL, CHAN_2_CONTROL). The PCI address is based on the DMA channel PCI address register (CHAN_1_PCI_BAR, CHAN_2_PCI_BAR). The PCI byte enables assert for all data phases. The maximum number of Dwords that can be read is specified in the DMA channel control register.

5.3.4.4 I/O Write

An I/O write occurs when the StrongARM* core address is in the PCI I/O space. The PCI address is derived from the StrongARM address and the PCI address extension register (PCI_ADDR_EXT). The PCI byte enables for each data phase are generated as appropriate based on the StrongARM* core store instruction. One Dword is written.

The following general rules apply to I/O write transactions:

- If the PCI unit receives a target retry response, it repeats the I/O write command at the first opportunity.
- If the PCI unit receives a master abort, it discards the write data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core.
- If the PCI unit receives a target abort, it discards the write data and sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM* core.

5.3.4.5 I/O Read

An I/O read occurs when the StrongARM* core address is in the PCI I/O space. The PCI address is derived from the StrongARM address and the PCI address extension register (PCI_ADDR_EXT). The PCI byte enables for each data phase are generated as appropriate based on the StrongARM* core load instruction. One Dword is read.

The following general rules apply to the I/O read transactions:

- If the PCI unit receives a target retry response, it repeats the I/O read command at the first opportunity.
- If the PCI unit receives a master abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core.
- If the PCI unit receives a target abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM* core.

5.3.4.6 Configuration Write

A configuration write occurs when the StrongARM* core address is in the PCI configuration address range. The PCI address is derived from the StrongARM address and depends on whether type 0 or type 1 configuration space is addressed.

It is the responsibility of StrongARM* core software to generate an address that is meaningful for the PCI configuration cycle. For example, typically **ad[23:11]** bits are used for the **idsel** of PCI devices during a type 0 configuration cycle, so only one of those bits is a 1. The PCI byte enables are generated as appropriate based on the StrongARM* core store instruction. One Dword is written.

The following general rules apply to configuration write transactions:

- If the PCI unit receives a target retry response, it repeats the configuration write command at the first opportunity.
- If the PCI unit receives a master abort, it discards the write data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core.
- If the PCI unit receives a target abort, it discards the write data and sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM* core.

5.3.4.7 Configuration Read

A configuration read occurs when the StrongARM* core address is in the PCI configuration space. The PCI address is derived from the StrongARM address and depends on whether the type 0 or type 1 configuration space is addressed. The PCI byte enables are generated as appropriate based on the StrongARM* core load instruction. One Dword is read.

The following general rules apply to configuration read transactions:

- If the PCI unit receives a target retry response, it repeats the configuration read command at the first opportunity.
- If the PCI unit receives a master abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core.
- If the PCI unit receives a target abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM* core.

5.3.4.8 Special Cycle

A special cycle occurs when the StrongARM* core address is in the PCI IACK/Special space. The special cycle is caused by a StrongARM* core write. The PCI address is undefined. The PCI byte enables are generated as appropriate based on the StrongARM* core store instruction. One Dword is written. Special cycles are broadcast to all PCI agents, so **devsel_l** is not asserted and no errors can be received.

5.3.4.9 IACK Read

An IACK read occurs when the StrongARM* core address is in the PCI IACK/Special space. An IACK read is caused by a StrongARM* core read. The PCI address is undefined. The PCI byte enables are generated as appropriate based on the StrongARM* core load instruction. One Dword is read.

The following general rules apply to IACK read transactions:

- If the PCI unit receives a target retry response, it repeats the IACK read at the first opportunity.
- If the PCI unit receives a master abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received master abort bit, which, if enabled, interrupts the StrongARM* core.
- If the PCI unit receives a target abort, it substitutes FFFF FFFFh for the read data and sets the status register (PCI_STATUS) received target abort bit, which, if enabled, interrupts the StrongARM* core.

5.3.4.10 PCI Request Operation

The PCI unit asserts **req_l[0]** to act as bus master on the PCI for StrongARM* core and DMA originated transactions. It deasserts **req_l[0]** for two cycles when it receives a retry or disconnect response from the target.

However, if **gnt_l[0]** is asserted, the PCI unit can start a PCI transaction regardless of the state of **req_l[0]**.

When the PCI unit requests the PCI bus, it performs a PCI transaction when **gnt_l[0]** is received. Once **req_l[0]** is asserted, the PCI unit never deasserts it prior to receiving **gnt_l[0]** (nor deasserts it after receiving **gnt_l[0]** without doing a transaction).

5.3.4.11 Master Latency Timer

When the PCI unit begins a PCI transaction as master, asserting **frame_l**, it begins decrementing its master latency timer. When the timer value reaches zero, the PCI unit checks the value of **gnt_l[0]**. If **gnt_l[0]** is deasserted, the PCI unit deasserts **frame_l** (if it is still asserted) at the earliest opportunity. This is normally the next data phase for all transactions except for the memory write and invalidate command (MWI). For MWI, it is the data phase at the top of a cache line. When the command is MWI and the latency timer expires while the last Dword of a cache line is being delivered, and **frame_l** is still asserted, the master delivers the entire next cache line before stopping the transaction.

5.3.5 Errors As PCI Target

PCI target errors are listed as follows:

- Address parity error
- Write data parity error
- Read data parity error

5.3.5.1 Address Parity Error

An address parity error is detected when the **par** signal driven by the PCI master does not match the expected parity for the address and command. This causes the following actions to occur:

- The status register (PCI_STATUS) detected parity error bit is set.
- If the (potentially corrupted) address or command matches any of the PCI unit base address registers, then the PCI unit claims the cycle and proceeds as though the address was correct.
- If the command register (PCI_COMMAND) parity error response bit is a 1, command register SERR enable bit is a 1, and **pci_cfn[0]** is a 0, then **serr_l** is asserted for one cycle and the status register (PCI_STATUS) signaled system error bit is set.

5.3.5.2 Write Data Parity Error

A write data parity error is detected when the **par** signal received by the PCI unit does not match the expected parity for data and byte enables. This causes the following to occur:

- The status register (PCI_STATUS) detected parity error bit is set.
- If the command register parity error response bit is a 1, then:
  - **perr_l** is asserted.
  - If the data destination is SDRAM, and SDRAM parity is enabled, incorrect parity is written to the SDRAM. The write is completed to the intended destination (that is, CSR, SDRAM, or ROM) despite the detection of a parity error.

5.3.5.3 Read Data Parity Error

A read data parity error is detected when the PCI master asserts **perr_l** in response to read data driven by the PCI unit. No action is taken by the PCI unit.

5.3.6 Errors As PCI Master

PCI master errors are listed as follows:

- Master abort
- Write data parity error
- Target abort on write
- Read data parity error
- Target abort on read

5.3.6.1 Master Abort

A master abort occurs when **devsel_l** is not asserted within five cycles after the PCI unit asserts **frame_l**. This causes the following actions to occur:

- Status register (PCI_STATUS) received master abort bit is set (except if the transaction is a special cycle).
- If the transaction was a write from StrongARM* core or DMA, all write data for the transaction is discarded.
- If the transaction was a StrongARM* core read, FFFF FFFFh is inserted for read data.
- If the transaction was a DMA read or write, the channel error bit in the channel control register (CHAN_1_CONTROL, CHAN_2_CONTROL) is set, stopping the channel.

5.3.6.2 Write Data Parity Error

A write data parity error occurs when the PCI target asserts **perr_l** in response to write data driven by the PCI unit. This causes the following actions to occur:

- If the command register (PCI_COMMAND) parity error response bit is a 1, then:
  - — The status register (PCI_STATUS) detected parity error data bit is set.
  - — If the transaction was a DMA, the channel error bit in the channel control register (CHAN_1_CONTROL, CHAN_2_CONTROL) is set, stopping the channel.

5.3.6.3 Target Abort on Write

When the PCI target signals a target abort, the following actions occur:

- Status register (PCI_STATUS) received target abort bit is set.
- Any remaining write data is discarded.
- If the transaction was a DMA, the channel error bit in the channel control register (CHAN_1_CONTROL, CHAN_2_CONTROL) is set, stopping the channel.

5.3.6.4 Read Data Parity Error

A read data parity error is detected when the **par** signal that is received by the PCI unit does not match the expected parity for data and byte enables. This causes the following actions to occur:

- The status register (PCI_STATUS) detected parity error bit is set. If the command register (PCI_COMMAND) parity error response bit is set, then the status register (PCI_STATUS) data parity error detected bit is set and the PCI unit asserts **perr_l**.
- Dwords with bad parity are marked as such in the Inbound FIFO.
  - — For DMA operations, if SDRAM parity is enabled, incorrect parity is written to SDRAM. The channel error bit in the channel control register (CHAN_1_CONTROL, CHAN_2_CONTROL) is set, stopping the channel.

5.3.6.5 Target Abort on Read

When the PCI target signals a target abort, the following actions occur:

- If the read is a demand read for the StrongARM* core, it substitutes FFFF FFFFh for the read data.
- If the transaction was a DMA, the channel error bit in the channel control register (CHAN_1_CONTROL, CHAN_2_CONTROL) is set, stopping the channel.

FBI Unit 6

This section describes the FBI Unit.

Section 6.1, “FBI Architecture” on page 6-156: Describes the architecture of the FBI Unit.

Section 6.2, “Push/Pull Engine Interface” on page 6-158: Describes how data is moved between the FBI Unit and the other functional units.

Section 6.3, “Scratchpad Memory” on page 6-162: Describes the internal Scratchpad memory.

Section 6.4, “Hash Unit” on page 6-165: Describes the Hash Unit.

Section 6.5, “FBI CSRs” on page 6-168: Describes the FBI CSRs.

Section 6.6, “IX Bus Interface” on page 6-174: Describes the IX Bus and Ready Bus interfaces.

6.1 FBI Architecture

The FBI Unit provides on-chip Scratchpad memory, hash index generation hardware, an interface to the IX Bus, and functions accessible through the FBI CSRs. The StrongARM* core and the Microengines access these resources through the FBI Push/Pull Engine interface. Figure 6-1 is a block diagram of the FBI Unit. A description of FBI resources appears in Table 6-1.

Figure 6-1. FBI Unit Block Diagram

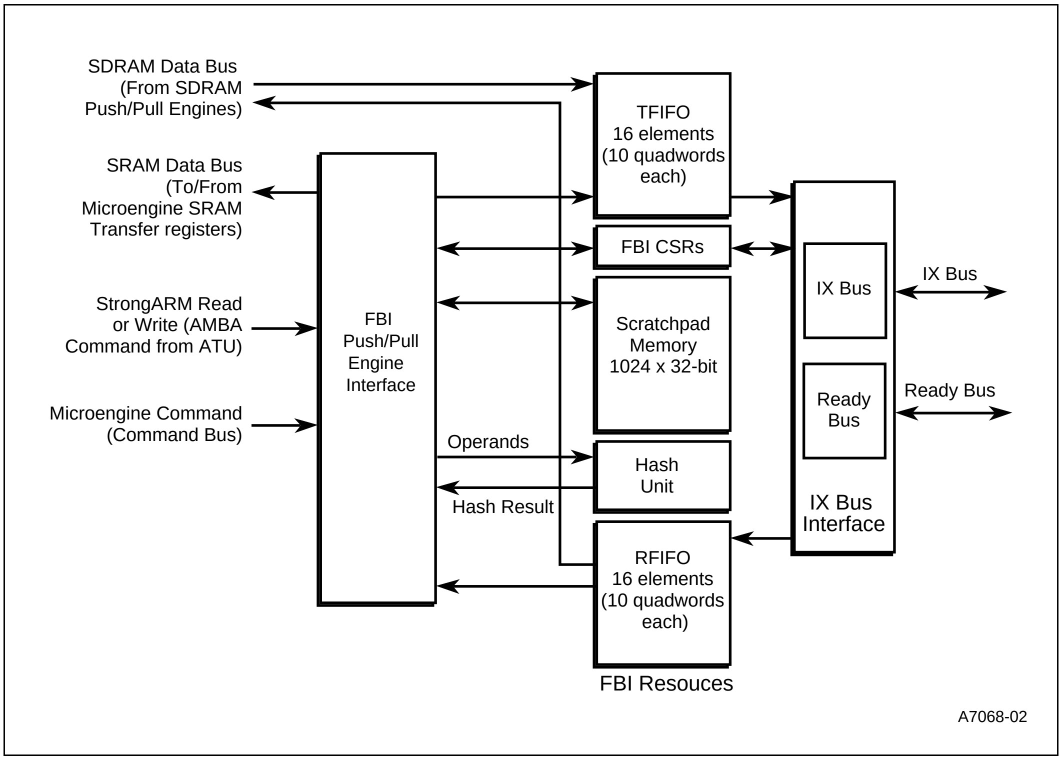


Table 6-1. FBI Resources

FBI Resource	Purpose	Accessed By
Hash Unit	Used to generate hash indexes for 48-bit or 64-bit data.	Microengines
Scratchpad Memory	4 Kbytes of internal memory (1 K x 4 byte). Operations supported: Read and Write operations. Bit test/set, Bit test/clear, and Increment operations.	 StrongARM* core/Microengines Microengines
FBI CSRs	Registers are used for: • FBI Unit configuration. • Accessing IX Bus Interface (IX Bus and Ready Bus). • Inter-thread signaling. • 64-bit counter. • Clear-on-read register (SELF_DESTRUCT).	StrongARM* core[1]/ Microengines
IX Bus	Transmitting and receiving data on the IX Bus. The IX Bus is accessed through the FBI CSRs.	Microengines

Table 6-1. FBI Resources (Continued)

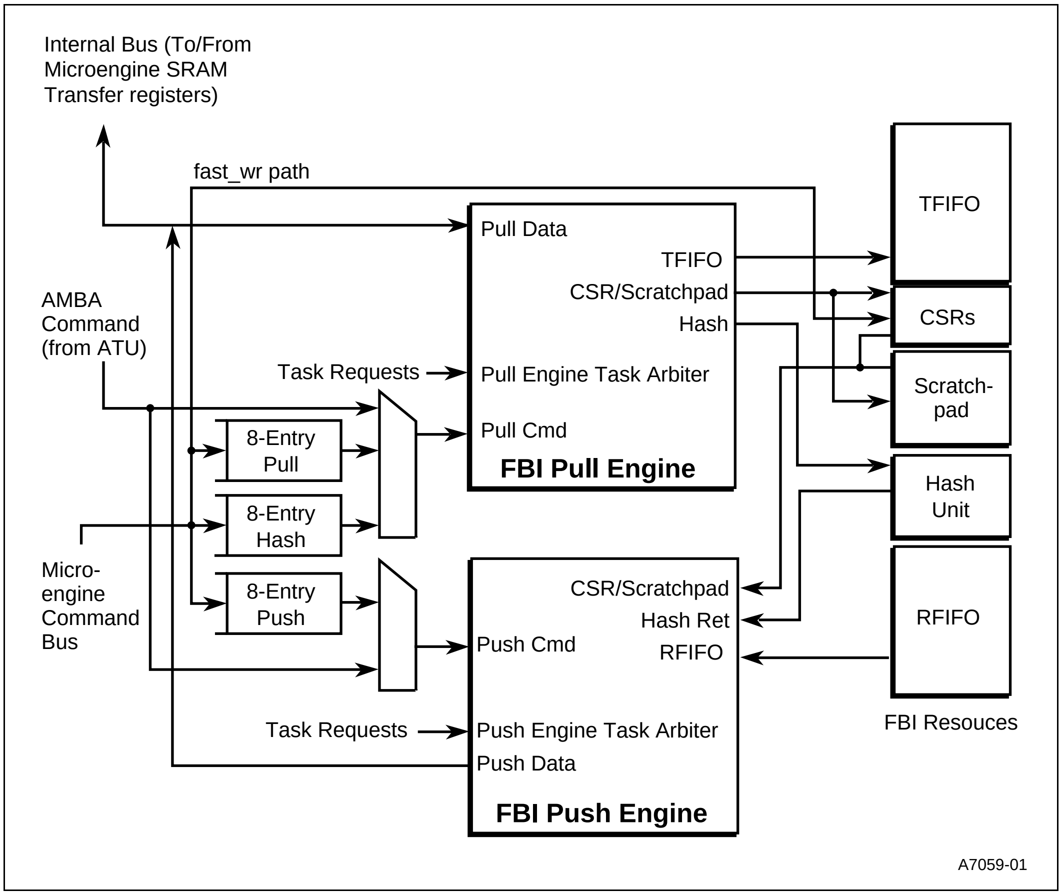
FBI Resource	Purpose	Accessed By
Ready Bus	8-bit data bus that is used: • To read MAC FIFO Ready Flags. • To assert flow control to a MAC device. • As a communication channel to another IXP1200. The Ready Bus is accessed through the FBI CSRs.	Microengines
RFIFO	Data buffers that hold data received from the IX Bus.	Microengines (read)
TFIFO	Data buffers that hold data to be transmitted to the IX Bus.	Microengines (write)

1. The StrongARM* core does not have access to the fast_wr function supported by some FBI registers, the CYCLE_CNT register, or the Hash Unit.

6.2 Push/Pull Engine Interface

The StrongARM* core and the Microengines issue commands to the Push/Pull Engine Interface when accessing an FBI resource. The Push/Pull Engine Interface places the commands into queues, arbitrates which commands to service, and moves data between the FBI resources, the StrongARM* core, and Microengines.

Figure 6-2. Push/Pull Engine Interface Block Diagram



6.2.1 Push and Pull Engines

The Push/Pull Engine Interface contains a Push Engine and a Pull Engine for moving data to and from an FBI Resource and the StrongARM* core and Microengines. The Push and Pull Engines operate independently and in parallel with each other. This provides improved performance for moving data through the FBI Unit.

The terms Push and Pull are used relative to the FBI Unit. The Push Engine pushes data out of the FBI Unit to a Microengine SRAM Transfer Register or the StrongARM* core, and the Pull Engine pulls data out of a Microengine SRAM Transfer Register or the StrongARM* core and places it into an FBI resource.

In addition to servicing requests from the StrongARM* core and Microengines, the FBI Push and Pull Engines also service requests from the Ready Bus for performing an autopush operation to a Microengine SRAM Transfer Register. The FBI Push and Pull each contain a task arbiter that determines which task it performs next. Table 6-2 defines the priorities for the task arbiters.

Table 6-2. FBI Push and Pull Task Priorities

Priority	Pull Engine	Push Engine
1	StrongARM (AMBA)	StrongARM (AMBA)
2	Hash (in progress)	Scratchpad (test&set/clear return)
3	Hash (new operation)	Rx Autopush
4	Pull Queue	Tx Autopush
5		Hash (return data)
6		Push Queue

6.2.2 Microengine Initiated FBI References

When a thread issues a request to an FBI resource, a command is driven onto the internal command bus and placed into command queues within the FBI Unit. Table 6-3 shows which queues are used for each instruction type.

Table 6-3. Instructions Assigned to FBI Command Queues

Hash Queue	Pull Queue	Push Queue
hash1_48	csr (write)	csr (read)
hash2_48	t_fifo_wr	r_fifo_rd
hash3_48	scratch (write)	scratch (read)
hash1_64	scratch (increment)	
hash2_64	scratch (test & set/clear)	
hash3_64		

The **fast_wr** instruction does not use the FBI Push or Pull Engines. It uses logic that services the instruction as soon as the write request is issued to the FBI CSR.

The **sdram** instruction, when used with the t_fifo_rd or r_fifo_wr command, relies on the SDRAM Unit Push/Pull Engine to move data between SDRAM and the RFIFO and TFIFO.

6.2.3 StrongARM* Core Initiated FBI References

The AMBA Translation Unit (ATU) translates StrongARM* core AMBA bus transactions to FBI read and write operations. StrongARM* core reads and writes bypass the Microengine queues and are temporarily latched until either the Push or Pull Engine is free to service the request. The Push and Pull task arbiters give StrongARM* core requests the highest priority to minimize StrongARM* core read stalls.

6.2.4 FBI Signal Events (to Microengines)

The FBI Unit provides a signal event to each Microengine thread to indicate when a reference to an FBI resource is completed. A Microengine thread must explicitly request the sig_done or ctx_swap optional tokens in the following instructions: **t_fifo_wr**, **r_fifo_rd**, **csr**, **hash**, and **scratch**.

6.2.5 Command Ordering

Since the FBI Unit contains multiple command queues, two references destined for different FBI queues may not complete in the order in which they were issued.

A single thread may overload the FBI signal (SIG_DONE) with multiple references if the references are to the same FBI queue. For example, if a Microengine thread executes the following instructions, the execution order is maintained since both commands are placed into the read command queue. In this case, when the FBI signal is asserted, this indicates both references have completed.

```
r_fifo_rd[$xfer1, 0, rfifo_addr, 2]
scratch[read, $xfer0, 0, addr, 1], sig_done ; Or sig_swap
```

If multiple references are not issued to the same FBI queue, a separate FBI signal must be requested for each reference to maintain ordering. For example, if a Microengine thread executes the instructions below, the execution order may not be maintained since one command is placed into the read queue and the other into the write queue. In this case, the first instruction requests the FBI signal and waits until it is received. After the FBI signal is returned, the next instruction may request another FBI signal.

```
r_fifo_rd[$xfer1, 0, rfifo_addr, 2], ctx_swap ;Doing FIFO read, moving data.
scratch[write, $xfer0, 0, addr, 1], sig_done (or ctx_swap);Write scratchpad RAM.
```

6.2.6 FBI Command Bus Arbiter Signaling

The Microengine Command Bus Arbiter determines which commands (from the Microengines and StrongARM* core) are placed onto the internal Command bus. The FBI Unit provides a signal to the Command Bus Arbiter to indicate when the FBI Unit can not accept any more commands because one or more of the three FBI command queues are full. This allows the StrongARM* core and Microengines to issue references to the FBI without managing a back pressure mechanism in software. Even if the FBI Unit is not accepting any more commands, the Command Bus Arbiter can continue to issue commands to other Units.

6.2.7 Scratchpad Test and Set/Clear Instructions

If a **scratch** instruction specifies a test_and_set_bits or a test_and_clear_bits operation, a Microengine submits a command to the FBI Pull Engine queue to request that the bit mask supplied in an SRAM Transfer Register can be read into the FBI Unit. Once the bit mask is read, data is read from Scratchpad and a request is made to the Push Engine to write the original data back to the SRAM Transfer Register. The Scratchpad memory data is modified per the mask data and written back to Scratchpad memory. If specified in the instruction, the Microengine thread is signaled after the data is returned to the Microengine.

Example 6-1. Scratchpad Test and Set Bits

```
scratch[bit_wr, $sram_xfer_reg, op1, op2, test_and_set_bits]
```

Where:

$sram_xfer_reg	The beginning of a contiguous set of registers that supply the scratchpad data on write operation or contain the read data after a read operation.
op1 and op2	These operands are added together to specify the scratchpad address

6.2.8 Scratchpad Increment Instruction

If a **scratch** instruction specifies an increment operation, the Microengine submits a command to the FBI Pull Engine queue even though a pull operation is not performed. When the increment command is serviced, data is read from Scratchpad memory, incremented by one, and written back to Scratchpad memory.

Example 6-2. Scratchpad Increment

```
scratch[incr, --, op1, op2, 1], optional_token
```

Where:

--	Indicates an SRAM Transfer Register is not used for this command.
op1 and op2	These operands are added together to specify the scratchpad address.
1	Specifies a burst size of one.
optional_token	sig_done, ctx_swap, defer [1], indirect_ref.

6.2.9 Hash Instruction

The **hash** instructions require the service of the FBI Pull and Push Engines. The Microengines submit a hash command to the FBI Pull Engine queue to request that the data used to generate the hash index be read from the SRAM Transfer Registers into the Hash Unit. Once the hash operation is complete, a request is made to the Push Engine to write the hash index data back to the SRAM Transfer Register. If specified in the instruction, the Microengine thread is signaled after the hash index data is returned to the Microengine.

One to three hash operations can be performed with a single Microengine **hash** instruction. As described in Section 6.4.1, the Hash Unit has two input buffers which may or may not be full. Therefore, when multiple hashes are initiated by a single **hash** instruction, the FBI Pull Engine feeds the data read from the SRAM Transfer registers into the Hash Unit one at a time. This translates into a separate Pull Engine service request for each piece of data used to generate the hash index. The Pull Engine Task Arbiter gives a higher priority to a hash operation in progress than a new hash operation request.

6.3 Scratchpad Memory

The FBI unit contains 1024 x 32-bit of Scratchpad memory that is accessible by the StrongARM* core and Microengines. The Scratchpad memory performs three basic operations: read and write operations, bit operations and an increment operation. The Microengines support all operations while the StrongARM* core only supports the read and write operations.

Scratchpad memory is provided as a third memory resource (in addition to SRAM and SDRAM) that is shared by the Microengines and the StrongARM* core. The Microengines and the StrongARM* core can distribute memory accesses between these three memory resources to provide a greater number of memory accesses occurring in parallel.

6.3.1 Read and Write Operations

Read and write operations to Scratchpad memory are supported by both the Microengines and the StrongARM* core.

The Microengine scratchpad read and write instructions have the following formats:

```
scratch[read, $sram_xfer_reg, op1, op2, cnt], optional_token
scratch[write, $sram_xfer_reg, op1, op2, cnt], optional_token
```

Where:

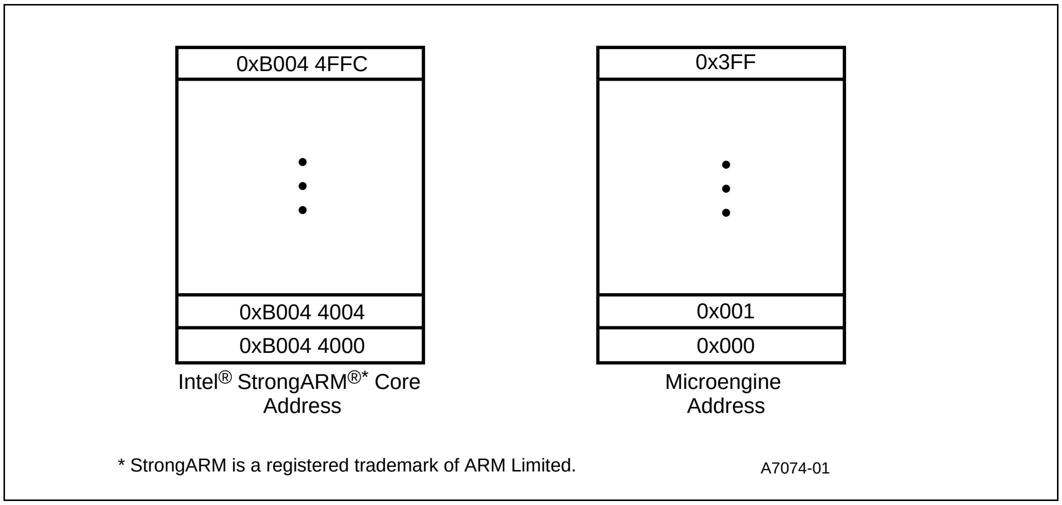
$sram_xfer_reg	The beginning of a contiguous set of registers which supply the scratchpad data on write operation or contain the read data after a read operation.
op1 and op2	These operands are added together to specify the scratchpad address
cnt	A burst count of one to eight.
optional_token	sig_done, ctx_swap, defer [1], indirect_ref

A Microengine initiates a write operation by first writing one to eight 32-bit data elements into an SRAM Transfer Register, and then executing the **scratch[write...]** instruction. The reference is placed into the FBI Pull command queue. When the command is serviced, the FBI Unit reads the data from the SRAM Transfer Registers, performs a write operation to Scratchpad memory, and, if specified, signals the Microengine thread when the operation is complete.

A Microengine initiates a read operation by executing the **scratch[read...]** instruction. The reference is placed into the FBI Push command queue. When the command is serviced, the FBI Unit performs a read operation to Scratchpad memory, pushes the data into the specified SRAM Transfer Registers, and, if specified, signals the Microengine thread when the operation is complete.

The Scratchpad memory is mapped into the StrongARM memory space at addresses 0xB004 4000 to 0xB004 4FFF (4 Kbytes of addresses). The StrongARM* core reads and writes Scratchpad memory on longword address boundaries (byte and halfword are not supported).

Figure 6-3. Scratchpad Memory Mapping



* StrongARM is a registered trademark of ARM Limited.

6.3.2 Bit Write Operations

A bit write operation is one in which the specified bits at a 32-bit Scratchpad location are set or cleared using a single Microengine instruction.

Note: The FBI Unit does not support bit operations for the StrongARM* core.

A Microengine initiates a bit write operation by first writing a 32-bit bit mask into an SRAM Transfer Register, and then executing the **scratch** instruction using the **bit_wr scratch** command. The instruction has the following format:

```
scratch[bit_wr, $sram_xfer_reg, op1, op2, bit_op], optional_token
```

Where:

$sram_xfer_reg	The register that contains the bit mask and original data for test operations
op1 and op2	These operands are added together to specify the Scratchpad address
bit_op	Specifies one of the following bit operations: • set_bits • clear_bits • test_and_set_bits • test_and_clear_bits
optional_token	sig_done, ctx_swap, defer [1], indirect_ref

The reference is placed into the FBI Pull command queue. When the command is serviced, the FBI Unit reads the bit mask from the SRAM Transfer Register, performs a read-modify-write operation on the Scratchpad data, and, if specified, signals the Microengine thread when the operation is complete.

If the instruction specifies a **test_and_set_bits** or a **test_and_clear_bits** operation, the FBI Unit submits a command to the FBI Pull engine to deliver the original data back to the SRAM Transfer Register prior to signaling the Microengine thread. A Microengine thread should always perform a context swap on a **test_and_set_bits** or a **test_and_clear_bits** operation if it is immediately followed by a read operation.

6.3.3 Auto Increment Operations

An auto increment operation is one in which the data at a specified 32-bit Scratchpad location is incremented by one.

A Microengine initiates an auto increment operation by executing the **scratch[incr,...]** instruction:

```
scratch[incr, --, op1, op2, 1], optional_token
```

Where:

--	Indicates an SRAM Transfer Register is not used for this command.
op1 and op2	These operands are added together to specify the scratchpad address.
1	Specifies a burst size of one.
optional_token	sig_done, ctx_swap, defer [1], indirect_ref.

The reference is placed into the FBI Pull command queue. When the command is serviced, the FBI Unit reads the data from Scratchpad memory, performs a read-increment-write operation on the Scratchpad data, and, if specified, signals the Microengine thread when the operation is complete.

6.4 Hash Unit

The FBI Unit contains a Hash Unit that can take 48-bit or 64-bit data and produce a 48-bit or a 64-bit hash index, respectively. The Hash Unit is accessible by the Microengines only.

6.4.1 Hashing Operation

Up to three hash indexes can be created using a single Microengine instruction. The Microengine hash instructions have the following format:

```
hash1_48[$sram_xfer_reg], optional_token
hash2_48[$sram_xfer_reg], optional_token
hash3_48[$sram_xfer_reg], optional_token
hash1_64[$sram_xfer_reg], optional_token
hash2_64[$sram_xfer_reg], optional_token
hash3_64[$sram_xfer_reg], optional_token
```

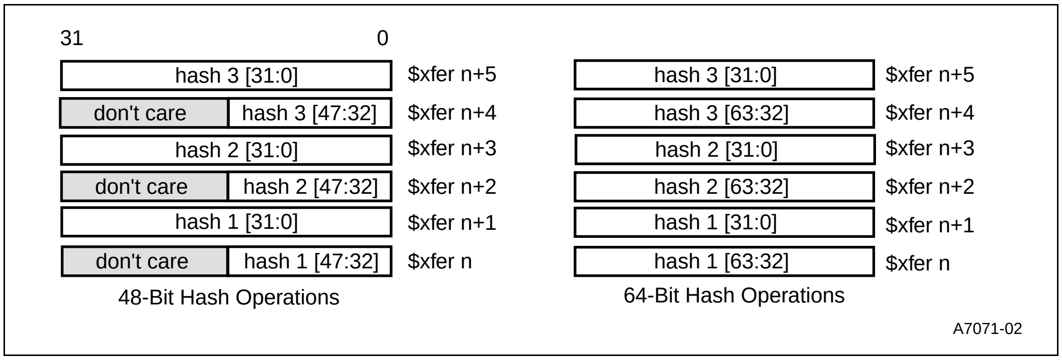
Where:

$sram_xfer_reg — The beginning of a contiguous set of registers that supply the data used to create the hash index and contain the hash index upon completion of the hash operation.

optional_token — sig_done, ctx_swap, defer [1]

A Microengine initiates a hash operation by writing a contiguous set of SRAM Transfer Registers with the data to be used to generate the hash index and then executing the hash instruction. Two SRAM Transfer Registers are required to create each hash index. In the case of the 48-bit hash, the Hash Unit ignores the upper two bytes of the first Transfer Register.

Figure 6-4. How SRAM Transfer Registers are Used



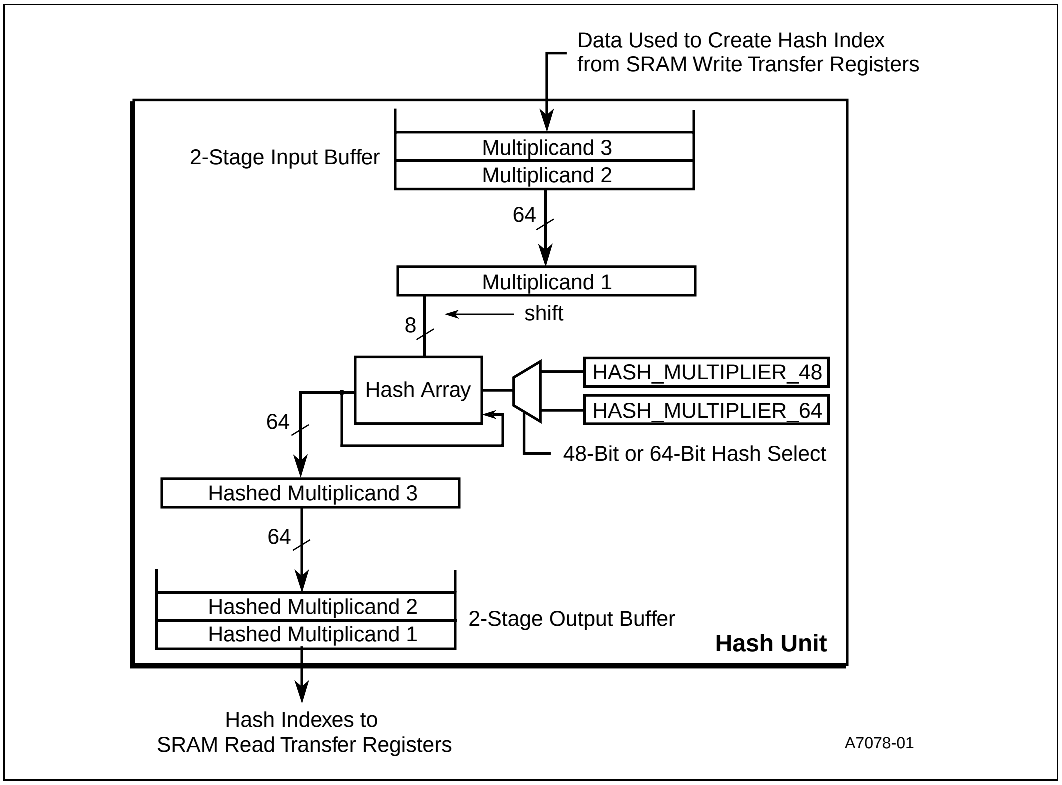
The reference is placed into the FBI Hash command queue. The FBI Pull Engine arbitrates between the AMBA, Hash, and Pull command queues and services the Hash queue according to the priority shown in Section 6.2.1. When the command is serviced, the FBI Pull Engine reads the data from the SRAM Transfer Registers and deposits the data into a 64-bit two-stage buffer. The two-stage buffer allows the data for up to three hash operations to be read into the Hash Unit in a single burst. The first data to be hashed enters the hash array while the next two wait in the two-stage buffer.

The Hash Unit uses a hard-wired polynomial algorithm and a programmable hash multiplier to create hash indexes. Two separate multipliers are supported, one for 48-bit hash operations and one for 64-bit hash operations. The multiplier is programmed through FBI registers (HASH_MULTIPLIER_64_LO, HASH_MULTIPLIER_64_HI, HASH_MULTIPLIER_48_LO, HASH_MULTIPLIER_48_HI).

The multiplicand is shifted into the hash array eight bits at a time. The hash array performs a ones-complement multiply and polynomial divide, calculated using the multiplier and 8 bits of the multiplicand. The result is placed into an output register and also feeds back into the array. This process is repeated 6 times for a 48-bit hash (8 bits x 6 = 48) and 8 times for a 64-bit hash (8 bits x 8 = 64). After an entire multiplicand has been passed through the hash array, the resulting hash index is placed into a two-stage output buffer. After each hash index is completed, the Hash Unit requests service from the FBI Push Engine so that the hash index can be returned to the Microengines SRAM Transfer Registers. If specified by the instruction, the FBI Push engine signals the Microengine after all the hashes specified in the instruction have been completed.

The number of Core clock cycles required to perform a single hash operation equals: two cycles through the input buffers, six or eight cycles through the hash array, and two cycles through the output buffers. Because of the pipeline characteristics of the Hash Unit, performance is improved if multiple hash operations are initiated with a single instruction rather than separate hash instructions for each hash operation.

Figure 6-5. Hash Operation Flow



6.4.2 Hash Algorithm

The hashing algorithm used by the IXP1200 allows flexibility and uniqueness since it can be programmed to provide different results for a given input. The algorithm uses binary polynomial multiplication and division under modulo-2 addition. The input to the algorithm is a 48-bit or 64-bit value.

The data used to generate the hash index is considered to represent the coefficients of an order-47 polynomial in x. The input polynomial (designated as A(x)) has the form:

$A(x) = a_{47} * x^{47} + a_{46} * x^{46} + ... + a_2 * x^2 + a_1 * x + a_0$ (48-bit hash operation)

$A(x) = a_{63} * x^{63} + a_{62} * x^{62} + ... + a_2 * x^2 + a_1 * x + a_0$ (64-bit hash operation)

This polynomial is multiplied by a programmable hash multiplier using a modulo-2 addition. The hash multiplier, M(x) is stored in FBI CSRs and represents the polynomial

$M(x) = m_{47} * x^{47} + m_{46} * x^{46} + ... + m_2 * x^2 + m_1 * x + m_0$ (48-bit hash operation)

$M(x) = m_{63} * x^{63} + m_{62} * x^{62} + ... + m_2 * x^2 + m_1 * x + m_0$ (64-bit hash operation)

Since multiplication is performed using modulo-2 addition, the result is an order-94 polynomial or an order-126 polynomial with coefficients that are also 1 or 0. This product is divided by a fixed generator polynomial given by:

$G(x) = x^{48} + x^{36} + x^{25} + x^{10} + 1$ (48-bit hash operation)

$G(x) = x^{64} + x^{54} + x^{35} + x^{17} + 1$ (64-bit hash operation)

The division results in a quotient Q(x), a polynomial of order-46 or order-62, and a remainder R(x), a polynomial of order-47 or order-63. The operands are related by the equation:

$A(x) * M(x) / G(x) = \mathbf{R(x)} + Q(x)$

The generator polynomial has the property of irreducibility. As a result, for a fixed multiplier M(x), there is a unique remainder R(x) for every input A(x). The quotient Q(x), can then be then discarded, since input A(x) can be derived from its corresponding remainder R(x). A given bounded set of input values A(x) (say 8 K or 16 K table entries), with bit weights of an arbitrary density function can be mapped one-to-one into a set of remainders R(x) such that the bit weights of the resulting Hashed Arguments (a subset of all values of R(x) polynomials) are all about equal. In other words, there is a high likelihood that the low order set of bits from the Hash Arguments are unique, so they can be used to build an index into the table. If the hash algorithm does not provide a uniform hash distribution for a given set of data, the programmable hash multiplier (M(x)) may be modified to provide better results.

6.5 FBI CSRs

The FBI CSR registers are accessible by the StrongARM* core and Microengines and are used to:

- Configure and control the IX Bus Interface.
- Configure the Hash Unit.
- Generate inter-thread signaling.
- Generate Microengine initiated StrongARM* core interrupts.
- Read a 64-bit cycle count register (Microengine only access).

6.5.1 CSR Reads and Writes

The Microengines can read or write any FBI CSR using the **csr** instruction. One FBI CSR is read per **csr** instruction except when GET_CMD and CYCLE_CNT registers are read. The GET_CMD is mapped to an 8-entry x 32-bit FIFO and up to four entries can be read in a single instruction. Reading the CYCLE_CNT register returns a 64-bit count value into two SRAM Transfer Registers.

The StrongARM* core can read or write any of the FBI registers except the 64-bit CYCLE_CNT register. The FBI registers are mapped into the StrongARM address space.

6.5.2 FAST_WR Support

The Microengine support a **fast_wr** instruction that bypasses the Push and Pull queues to improve performance when writing to a subset of FBI registers. The **fast_wr** instruction is not supported for the StrongARM* core. The **fast_wr** instruction supplies 10-bit immediate data in the reference command. This eliminates the need for the FBI Pull engine to read data from a Microengine Transfer Register when it processes the command.

The meaning of the 10-bit immediate data is shown in Table 6-4. Some registers do not require the entire 10 bits. The programmer should ensure that data larger than what is specified for the register is not written, since it extends past the Microengine assigned bit field into another Microengine assigned bit field.

Table 6-4. 10-Bit Immediate Data (Sheet 1 of 2)

Register	10-Bit immediate data
INTER_THD_SIG	Thread number of the thread that is to be signaled.
THREAD_DONE	A 2-bit message that is shifted into a position relative to the thread that is writing the message. The message is determined through software.
THREAD_DONE_INCR1 THREAD_DONE_INCR2	Same as thread_done except that either the enqueue_seq1 or enqueue_seq2 is also incremented.
XMIT_VALIDATE	The Transmit FIFO element number (0 to 15) that is to marked to indicate that the data is valid in this element.

Table 6-4. 10-Bit Immediate Data (Sheet 2 of 2)

Register	10-Bit immediate data
INCR_ENQ_NUM1 INCR_ENQ_NUM2	Write a one to increment the Enqueue Sequence Number by one (The Sequence Number is always incremented by one).
SELF_DESTRUCT	Specifies the bit position (0-31) that is set.
IREG	The 10-bit immediate data supplied with the instruction is shifted in two segments to the appropriate fields. Bits 6 through 0 are shifted left by an amount equal to the thread number writing the data. Bits 9 through 7 are always shifted into the BP2 through BP0 positions regardless of the Microengine writing the data.

6.5.3 FBI CSR Description Summary

The FBI CSRs are listed by function and briefly described in the following sections. Refer to the *IXP1200 Network Processor Programmer's Reference Manual* for more information on the FBI CSRs.

6.5.3.1 IX Bus Receive Registers

Table 6-5. Receive Request Registers

Register	Description
RCV_REQ	Receive Request Register. Written by a Microengine thread to issue a Receive Request to the Receive State Machine. (Implemented as a 2-entry FIFO).
RCV_CNTL	Receive Control Register. Written by the Receive State Machine to provide control information about the data in an RFIFO. (Implemented as a 4-entry FIFO).

The purpose of the Receive Request Registers is described in detail in Section 6.6.4.

Table 6-6. Sequence Numbers Registers

Register	Description
SOP_SEQ1 SOP_SEQ2	SOP sequence number incremented by the Receive State Machine for Fast Ports 1 and 2.
ENQUEUE_SEQ1 ENQUEUE_SEQ2	Enqueue SOP sequence number incremented by the Microengine threads for Fast Ports 1 and 2.

The purpose of the Sequence Numbers Registers is described in detail in Section 6.6.8.

6.5.3.2 IX Bus Transmit Registers

The purpose of these registers is described in detail in Section 6.6.5.

Table 6-7. IX Bus Transmit Registers

Register	Description
XMIT_VALIDATE	Transmit Validate Register. Written by a Microengine thread (using the **fast_wr** instruction) to indicate the data and control information in a TFIFO element is valid.
XMIT_PTR	Transmit Pointer Register. Read by a Microengine thread to determine which TFIFO element is transmitted next by the Transmit State Machine.

6.5.3.3 IX Bus and Ready Bus Configuration Registers

The purpose of these registers is described in detail in Section 6.6.1 and Section 6.6.3.

Table 6-8. IX Bus and Ready Bus Configuration Registers

Register	Description
RDYBUS_TEMPLATE_CTL	Ready Bus/IX Bus Configuration Register. Used to: • Enable the Ready Bus Sequencer. • Select 1-2 MAC or 3+ MAC mode. • Select 32-bit unidirectional or 64-bit bidirectional IX Bus mode. • Specify the Ready Bus as a master or slave. • Select Shared IX Bus mode. • Select IX Bus Status mode. • Select IX Bus little or big endian mode.
RCV_RDY_CTL	Ready Bus/IX Bus Configuration Register. Used to: • Select the Fast Port mode. • Configure the Ready Bus for Rx autopush operation.
XMIT_RDY_CTL	Ready Bus/IX Bus Configuration Register. Used to: • Keep a copy of the current TFIFO Valid flags. • Configure the Ready Bus for Tx autopush operation.
RDYBUS_TEMPLATE_PROG3 RDYBUS_TEMPLATE_PROG2 RDYBUS_TEMPLATE_PROG1	Ready Bus Program Register. Contains the Ready Bus Sequencer program.
RDYBUS_SYNCH_COUNT_DEFAULT	Ready Bus Synchronize Counter Register. Contains the count value used to determine the minimum rate at which the Ready Bus Sequencer repeats the program.
FP_READY_WAIT	Fast Port Mode Configuration Register. Specifies how many IX Bus clock cycles the Receive State Machine waits before returning to the same Fast Port between the time it reads the last data transfer on the IX Bus and the time the Receive State Machine samples the receive ready pins (FAST_RX1 and FAST_RX2) to start the next receive.
REC_FASTPORT_CTL	Fast Port Status Register. Provides the current status of the header and body thread assignments for the two Fast Ports.

6.5.3.4 Ready Bus Control Registers

The purpose of these registers is described in detail in Section 6.6.3. Table 6-9 provides a summary.

Table 6-9. Ready Bus Control Registers

Register	Description
FLOWCTL_MASK	MAC Flow Control Mask Register. Written by the Microengines to specify the MAC and the 8-bit mask that determine which port asserts flow control.
RCV_RDY_CNT	Status register that provides the following: • Receive Ready Count. • Receive Request Count. • Fast Receive Ready flags for ports 1 and 2. • FLOWCTL_MASK register valid data status. • Initial IX Bus owner/Ready Bus Master.
RCV_RDY_HI RCV_RDY_LO	Receive Ready Flags. Status registers that indicate which MAC Ports Rx FIFOs have data available.
XMIT_RDY_LO XMIT_RDY_HI	Transmit Ready Flags. Status registers that indicate which MAC Ports Tx FIFOs have room for data.
GET_CMD SEND_CMD	Get and Send registers. Typically used to send data to another IXP1200. These registers are each mapped to eight-entry FIFOs.

6.5.3.5 Hash Unit Configurations Registers

The purpose of these registers is described in detail in Section 6.4.1. Table 6-10 provides a summary.

Table 6-10. Hash Unit Configurations Registers

Register	Description
HASH_MULTIPLIER_64_HI HASH_MULTIPLIER_64_LO	64-bit Hash Multiplier. These registers contain the programmable hash multiplier for generating 64-bit hash indexes.
HASH_MULTIPLIER_48_HI HASH_MULTIPLIER_48_LO	48-bit Hash Multiplier. These registers contain the programmable hash multiplier for generating 48-bit hash indexes.

6.5.3.6 FBI Interrupt/Signal Registers

Table 6-11. FBI Interrupt/Signal Registers

Register	Description
IREG	StrongARM* core FBI Interrupt Register. Used to enable CINT and Microengine thread interrupts to the StrongARM* core. Microengines threads write this register to generate a StrongARM interrupt.
INTER_THD_SIG	Inter-thread Signaling Register. Any thread or the StrongARM* core can write a Microengine thread number to this register to signal an inter-thread signal event.

6.5.3.7 Thread Status Registers

Table 6-12. Thread Status Registers

Register	Description
THREAD_DONE	Thread Processing Status (write). All 24 Microengine threads can write 2-bit messages to this register using the **fast_wr** instruction to indicate current processing status.
THREAD_DONE_REG0 THREAD_DONE_REG1	Thread Processing Status (read). Each of the 24 Microengine threads can read the 2-bit messages that were written using the **fast_wr** instruction.
THREAD_DONE_INCR1 THREAD_DONE_INCR2	Thread Processing Status (write). These registers are the same as THREAD_DONE except that they also increment Fast Port enqueue sequence numbers (in ENQUEUE_SEQ1 and ENQUEUE_SEQ2) when written.

6.5.3.8 Miscellaneous Registers

The purpose of these registers is described in detail in Section 6.5.4 and Section 6.5.5. Table 6-13 provides a summary.

Table 6-13. Miscellaneous Registers

Register	Description
CYCLE_CNT	Cycle Count Register. A 64-bit counter that can be read by any Microengine thread (can not be read by the StrongARM* core).
SELF_DESTRUCT	Self Destruct Register. This register is written during a **fast_wr** instruction with data in the range of 0 through 31 (decimal). Writing this register sets a bit that corresponds to the data written. When the register is read, all bits are cleared to 0 after the original data is read.

6.5.4 Cycle Count Register

The IX Bus Interface contains a free running 64-bit counter that provides an ~3000-year modulo count that is incremented once each Core clock cycle (F_{core}). This counter is read by accessing the CYCLE_CNT register. This is the only register read by the Microengine as a single 64-bit register into two SRAM Transfer Registers using the **csr** instruction. The StrongARM* core can not read this register.

The use of the Cycle Counter is defined by the programmer. The Microengine threads can read this register to perform time-sensitive tasks such as regulating transmit bandwidth.

6.5.5 Self Destruct Register

The SELF_DESTRUCT register is written using a **fast_wr** instruction. Writing this register sets a bit that corresponds to the data written. For example, writing a value of 3 sets bit 3. Multiple writes can be performed to set additional bits without changing the state of the other bits in the register. The register is read using the **csr** instruction. When the register is read, all bits are cleared to 0 after the original data is read.

When a Microengine executes a **fast_wr** instruction, the data bypasses the FBI command queues to the Push and Pull engines. Since a CSR read of this register places a **csr read** command into the Push queue, the order may not be preserved if a CSR read is followed by a **fast_wr** instruction.

The use of the SELF_DESTRUCT register is defined by the programmer. One possible use is as a fast method of communication between threads.

6.5.6 Thread Status Registers (THREAD_DONE)

The THREAD_DONE registers are provided so that the Microengines can quickly report their current processing status. Other Microengine threads or the StrongARM* core can read the register to get the processing status of the Microengine threads. These registers contain twenty-four 2-bit status fields, one for each Microengine thread.

The Microengines write a 2-bit message to the THREAD_DONE register using a **fast_wr** instruction.

Each Microengine thread can be programmed to write a 2-bit message that represents their current processing status to one of these fields using the **fast_wr** instruction. The **fast_wr** immediate data is shifted into the correct bit position within the register based on the thread ID of the Microengine thread that issued the fast_wr command.The data should be constrained to values 0, 1, 2, or 3, otherwise the 10-bit immediate data overwrites the bit fields for other threads. The thread processing status of the Microengine threads are read using the **csr read** instruction. Since the number of bits required to support 24 2-bit messages exceeds 32 bits, two registers (THREAD_DONE_REG0 and THREAD_DONE_REG1) must be read to read the status for all the Microengine threads.

The definition of the 2-bit status field is determined through software. An example of the 2-bit encoding is shown in the following example:

Example 6-3. 2-bit Encoding Example

Message	Description
00	Thread is still active
01	Thread task is complete but it has not seen an EOP
10	Thread task is complete and the EOP was seen
11	Not used

6.6 IX Bus Interface

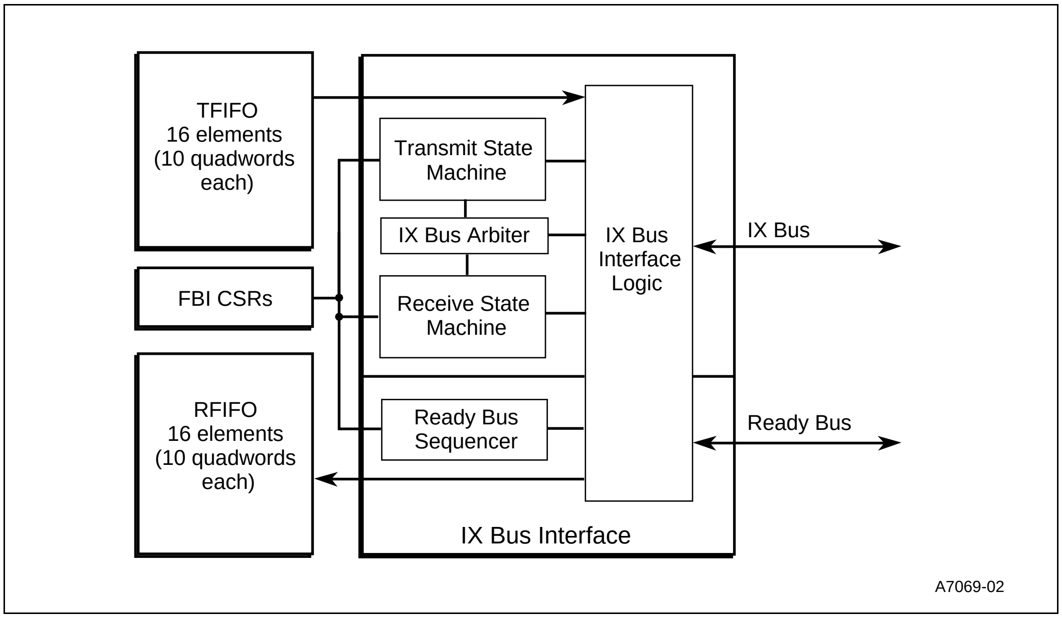
The IX Bus Interface consists of two buses: The Ready Bus and the IX Bus.

The Ready Bus is controlled by a programmable sequencer and is used for:

- Retrieving the MAC Receive and Transmit FIFO Ready flags from MAC devices.
- Asserting Flow Control to MAC devices.
- Inter-IXP1200 communications.

The IX Bus is used to transfer data to and from slave devices (i.e., MACs). It is controlled by a Transmit State Machine and a Receive State Machine. An IX Bus Arbiter is provided for selecting which state machine owns the IX Bus. These components are described in the following sections.

Figure 6-6. IX Bus Interface

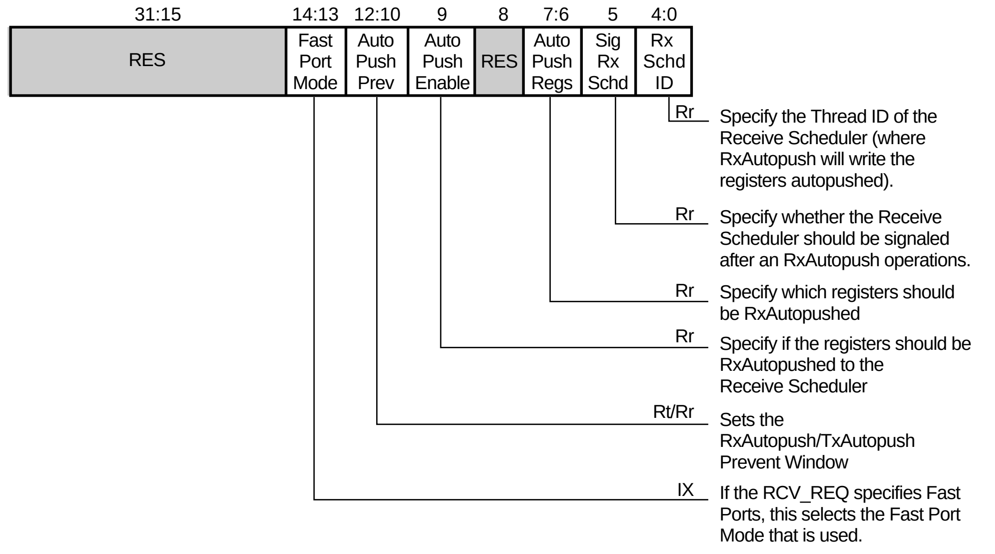


6.6.1 Configuring the IX Bus and Ready Bus

This IX Bus is configured by software through a series of registers and through a hardware configuration pin. The figures that follow provide a summary of the FBI registers that are used to configure the IX Bus and the Ready Bus. These registers are described in more detail in the*IXP1200 Network Processor Programmer's Reference Manual*.

The TK_IN pin (which is used as a token input pin) has a secondary function used at power-up to configure the IX Bus. It determines whether an IXP1200 powers up as a Ready Bus master or slave and whether an IXP1200 initially owns the IX Bus upon power-up. This is important since it defines which IXP1200 should drive the IX Bus and Ready Bus to a known state prior to configuring the IX Bus through software. If TK_IN is pulled high during reset, an IXP1200 will initially own the IX Bus, is configured as the Ready Bus master, and drives both of these buses. If TK_IN in pulled low during reset, an IXP1200 will not drive these buses. In a single IXP1200 system, the TK_IN pin should be pulled high. System designs that use the Shared IX Bus mode should have only one IXP1200 pulling this pin high.

Figure 6-7. RCV_RDY_CTL Register



Specify the Thread ID of the Receive Scheduler (where RxAutopush will write the registers autopushed).

Specify whether the Receive Scheduler should be signaled after an RxAutopush operations.

Specify which registers should be RxAutopushed

Specify if the registers should be RxAutopushed to the Receive Scheduler

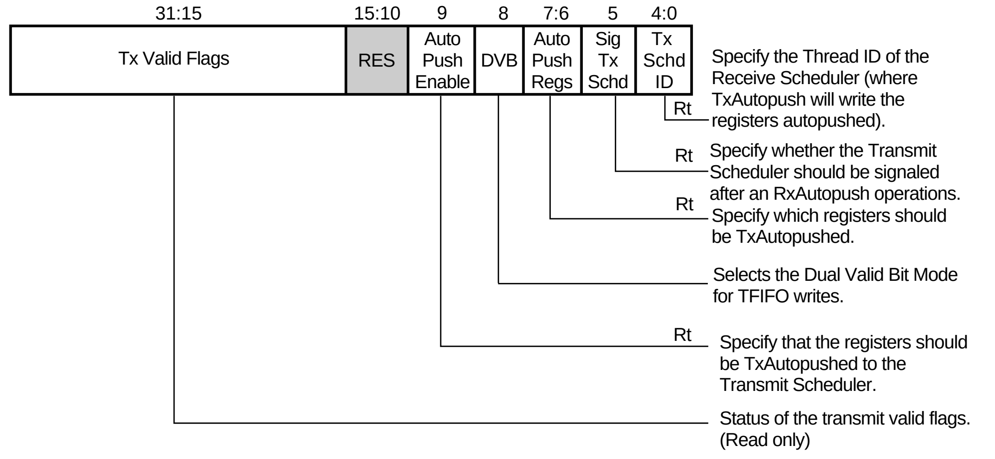
Sets the RxAutopush/TxAutopush Prevent Window

If the RCV_REQ specifies Fast Ports, this selects the Fast Port Mode that is used.

Note:
Rt -Ready Bus configuration (transmit)
Rr -Ready Bus configuration (receive)
IX - IX Bus configuration

A7050-01

Figure 6-8. XMIT_RDY_CTL Register



Specify the Thread ID of the Receive Scheduler (where TxAutopush will write the registers autopushed).

Specify whether the Transmit Scheduler should be signaled after an RxAutopush operations.

Specify which registers should be TxAutopushed.

Selects the Dual Valid Bit Mode for TFIFO writes.

Specify that the registers should be TxAutopushed to the Transmit Scheduler.

Status of the transmit valid flags. (Read only)

Note:
Rt -Ready Bus configuration (transmit)
Rr -Ready Bus configuration (receive)
IX - IX Bus configuration

A7051-01

Figure 6-9. RDYBUS_TEMPLATE_CTL Register

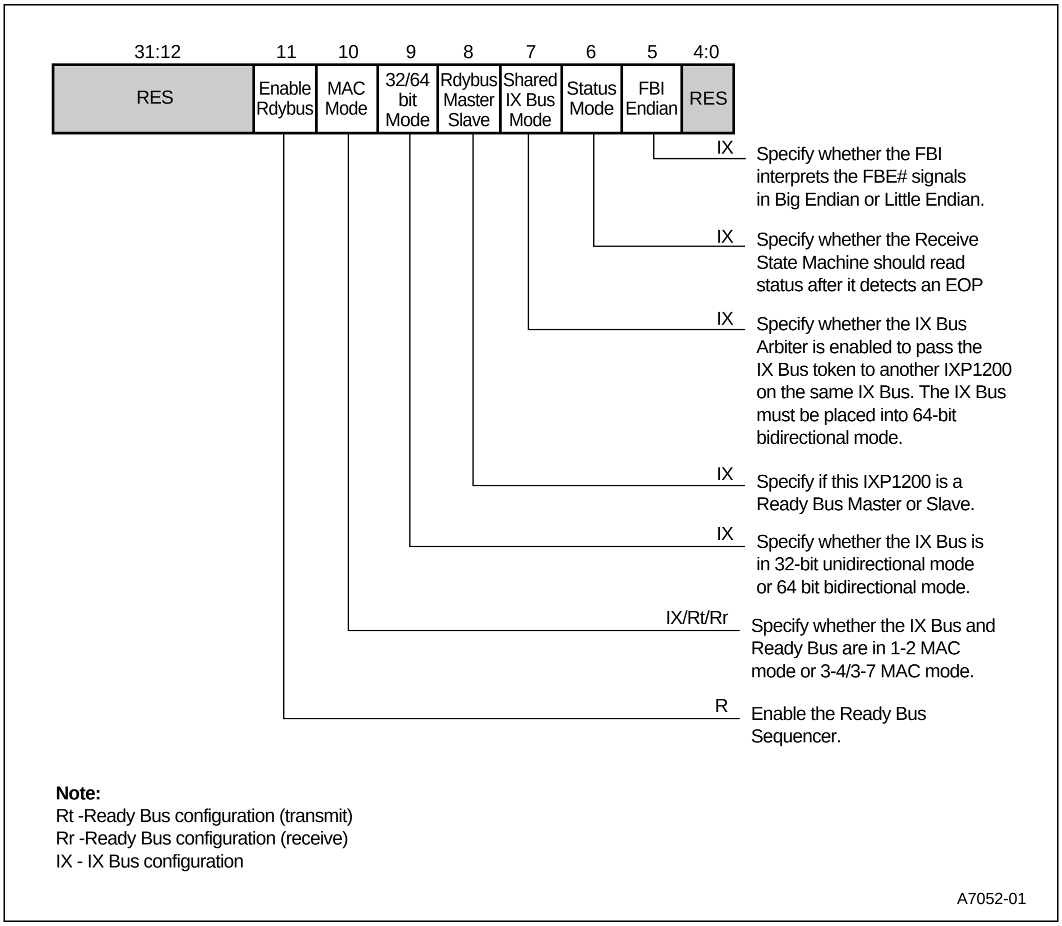

31:12
11
10
9
8
7
6
5
4:0
RES
Enable Rdybus
MAC Mode
32/64 bit Mode
Rdybus Master Slave
Shared IX Bus Mode
Status Mode
FBI Endian
RES
IX
Specify whether the FBI interprets the FBE# signals in Big Endian or Little Endian.
IX
Specify whether the Receive State Machine should read status after it detects an EOP
IX
Specify whether the IX Bus Arbiter is enabled to pass the IX Bus token to another IXP1200 on the same IX Bus. The IX Bus must be placed into 64-bit bidirectional mode.
IX
Specify if this IXP1200 is a Ready Bus Master or Slave.
IX
Specify whether the IX Bus is in 32-bit unidirectional mode or 64 bit bidirectional mode.
IX/Rt/Rr
Specify whether the IX Bus and Ready Bus are in 1-2 MAC mode or 3-4/3-7 MAC mode.
R
Enable the Ready Bus Sequencer.
Note:
Rt -Ready Bus configuration (transmit)
Rr -Ready Bus configuration (receive)
IX - IX Bus configuration
A7052-01

Figure 6-10. RDYBUS_SYNCH_COUNT_DEFAULT Register

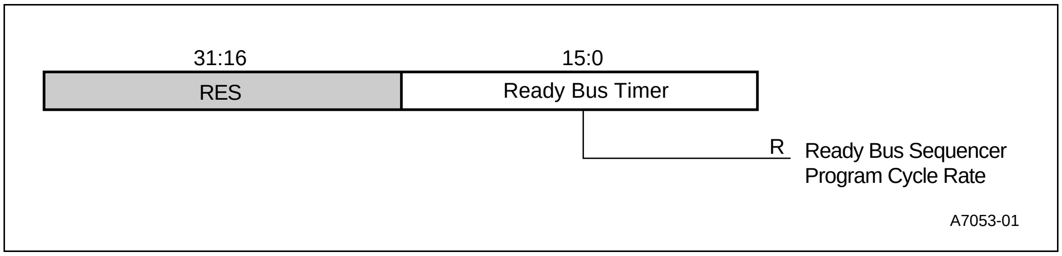
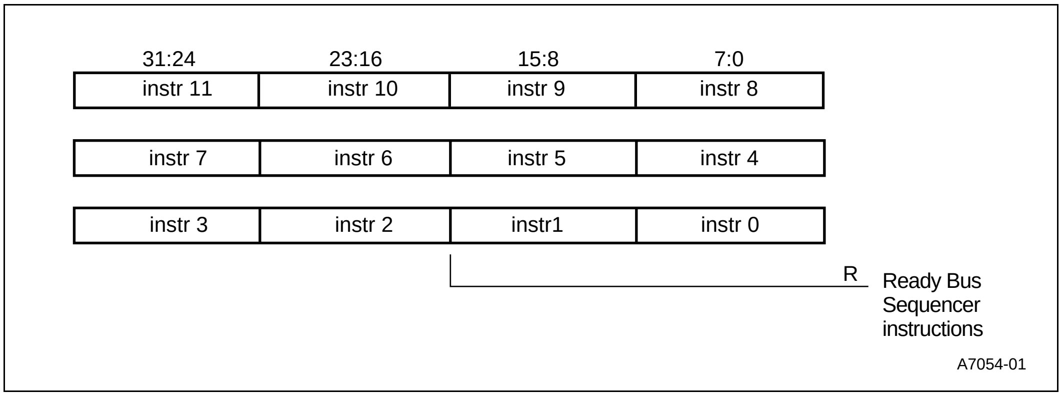


Figure 6-11. RDYBUS_TEMPLATE_PROGx Registers



6.6.2 IX Bus and Ready Bus Modes

This section describes the different modes of operation supported by the IX Bus and Ready Bus.

6.6.2.1 64-bit Bidirectional and 32-bit Unidirectional IX Bus Modes

64-bit bidirectional IX Bus mode or 32-bit unidirectional IX Bus mode is selected via the RDYBUS_TEMPLATE_CTL register. In 64-bit bidirectional IX Bus mode, the IX Bus is a single 64-bit data bus used for transmitting as well as receiving data on the IX Bus. The IX Bus Arbiter switches between granting ownership to the Transmit State Machine, the Receive State Machine, or another IXP1200.

In 32-bit unidirectional IX Bus mode, the IX Bus becomes two 32-bit data buses. One is used for transmitting and the other is used for receiving data on the IX Bus. The IX Bus Arbiter is not used in this mode since the Receive State Machine always owns the Receive IX Bus and the Transmit State Machine always owns the Transmit IX Bus.

From a programming perspective, there is no difference in how microcode issues receive and transmit requests to the State Machines.

6.6.2.2 1-2 MAC Mode and 3+ MAC Mode

These modes are selected via the RDYBUS_TEMPLATE_CTL register. The 1-2 MAC mode eliminates the need for external decode logic for the PORTCTL# and RDYCTL# signals. All the signals required for selecting the IX Bus and Ready Bus functions are provided directly by the IXP1200 for up to two MAC devices.

3+ MAC mode requires external decode logic for the PORTCTL# and RDYCTL# signals. If this mode is selected in 32-bit unidirectional mode, the port control signals support up to 4 MAC devices. If this mode is selected in 64-bit bidirectional mode, the port control signals support up to 7 MAC devices. From a programming perspective, there is no difference in how microcode issues receive and transmit requests to the State Machines. Refer to Section 6.6.3 for more information on these modes.

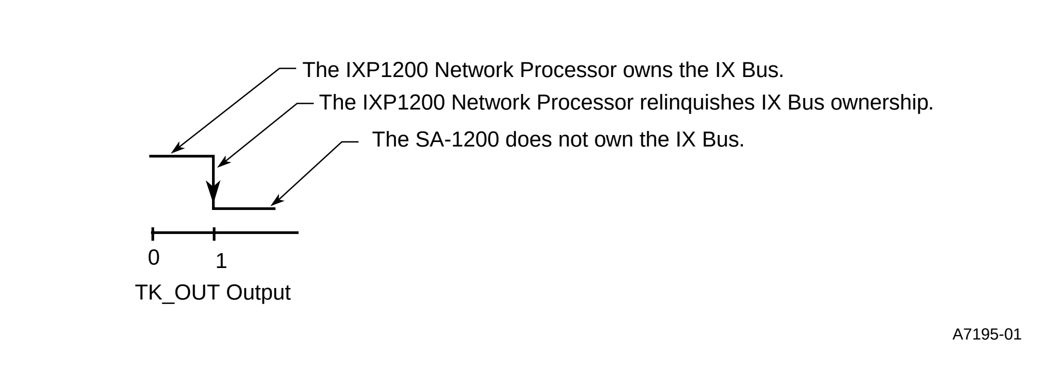
6.6.2.3 Shared IX Bus Mode

The IXP1200 supports multiple IXP1200s on the IX Bus in the 64-bit bidirectional mode. Bus ownership is established by passing a token between the IXP1200s.

The Shared IX Bus mode affects how the IX Bus Arbiter makes arbitration decisions. When the Shared IX Bus mode is not selected, the IX Bus arbitration scheme switches between granting IX Bus ownership to the Receive and Transmit State Machines in a single IXP1200. If the IXP1200 is placed into the Shared IX Bus mode, the IX Bus arbitration scheme switches between granting IX Bus ownership to the Receive State Machine, Transmit State Machine, and another IXP1200. IX Bus ownership is passed between IXP1200s using a token scheme.

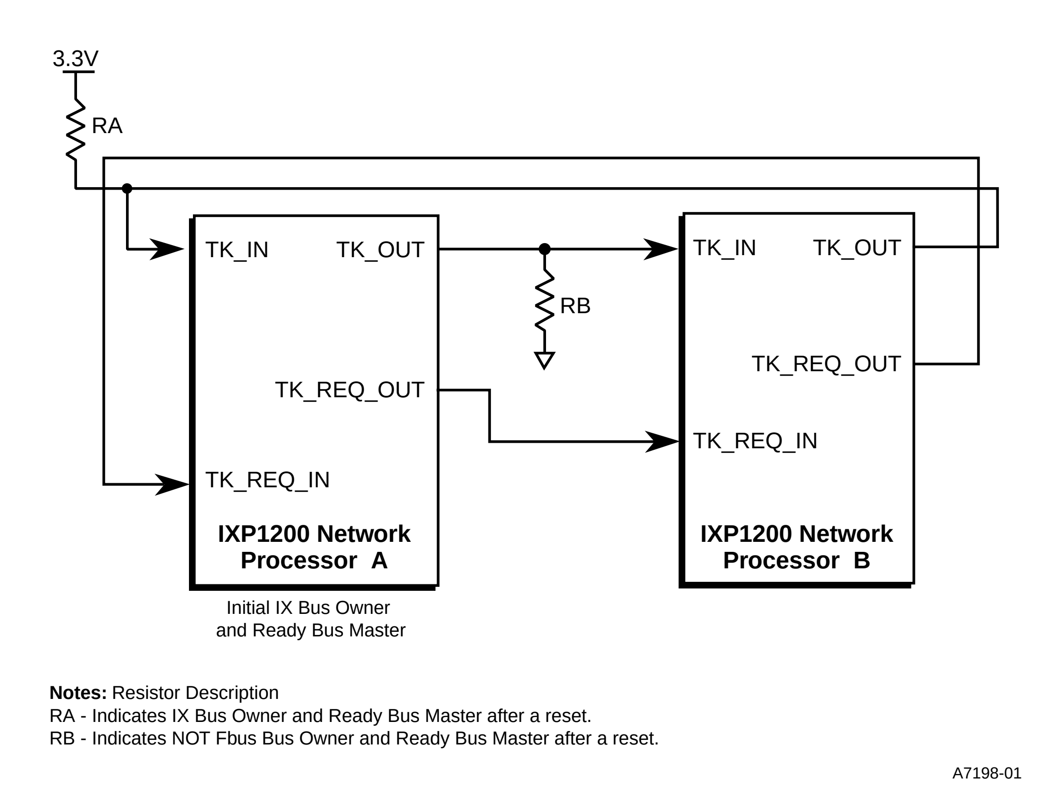
The token passing scheme uses four pins on the IXP1200: two token request pins (TK_REQ_OUT, TK_REQ_IN) and two token pins (TK_OUT, TK_IN). Refer to Figure 6-12. An IXP1200 requests the token by asserting the TK_REQ_OUT pin whenever it has a Receive Request or Transmit Request pending. The TK_REQ_OUT pin should be tied to the TK_REQ_IN pin of another IXP1200. The current IX Bus owner passes ownership of the IX Bus by asserting the TK_OUT pin low. The TK_OUT pin should be tied to the TK_IN pin of another IXP1200. The falling edge at a TK_IN pin indicates that bus ownership has been relinquished and the IXP1200 should take ownership of the bus.

Figure 6-12. TK_OUT Output


The IXP1200 Network Processor owns the IX Bus.
The IXP1200 Network Processor relinquishes IX Bus ownership.
The SA-1200 does not own the IX Bus.
0
1
TK_OUT Output
A7195-01

Cascading the connections of the token pins allows the token to be passed to multiple IXP1200s. Figure 6-13 shows the pin connections for two IXP1200s in a shared IX Bus configuration.

Figure 6-13. Dual IXP1200 System Using Dynamic Token Passing


3.3V
RA
TK_IN
TK_OUT
RB
TK_IN
TK_OUT
TK_REQ_OUT
TK_REQ_OUT
TK_REQ_IN
TK_REQ_IN
IXP1200 Network Processor A
IXP1200 Network Processor B
Initial IX Bus Owner and Ready Bus Master
A7198-01

Notes: Resistor Description
RA - Indicates IX Bus Owner and Ready Bus Master after a reset.
RB - Indicates NOT Fbus Bus Owner and Ready Bus Master after a reset.

The IXP1200s are placed into Shared IX Bus mode via the RDYBUS_TEMPLATE_CTL register. The following settings are required:

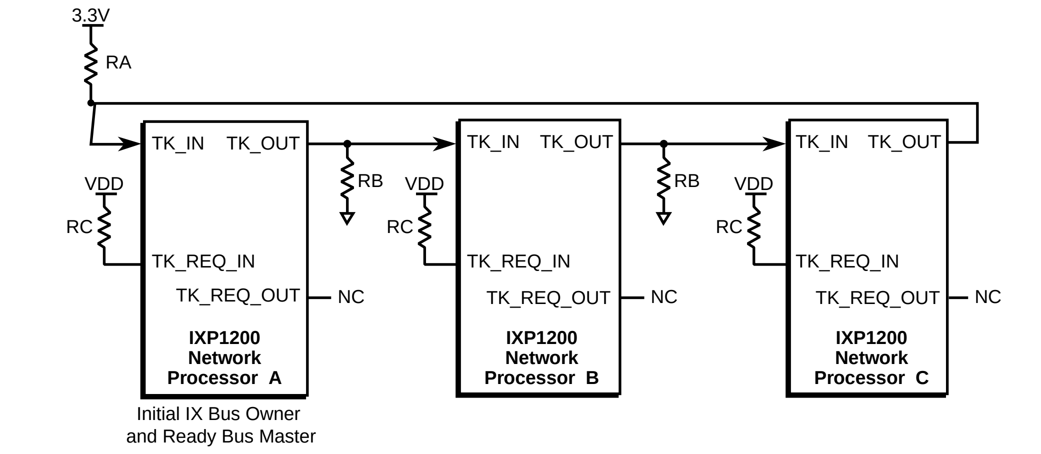
- 64-bit bidirectional mode
- Shared IX Bus mode
- One IXP1200 designated as an initial Ready Bus master and the others as Ready Bus slaves.

The TK_IN pin is also used as a configuration pin after as reset. (This is in addition to the token in function). The TK_IN pin must be either pulled high or low to indicate which IXP1200 is designated as a Ready Bus Master/initial IX Bus owner. An IXP1200 designated as a Ready Bus Master/initial IX Bus owner drives these buses after a reset. Only one IXP1200 should be designated as the Ready Bus Master/initial IX Bus owner since only one IXP1200 should drive the IX Bus and Ready Bus at any one time.

If the TK_IN pin is pulled high, an IXP1200 becomes the initial IX Bus Owner, the Ready Bus master, and it drives both buses. After a reset, the FBI Unit reads the TK_IN pin to determine if it is the Ready Bus master. The Receive State Machine looks for a valid Receive Request, and the Transmit State Machine looks to see if the first TFIFO element contains valid data and then checks to see if another IXP1200 is requesting IX Bus ownership by reading the state of the TK_REQ_IN. If none of these cases are true, the IX Bus is driven to a known state (no bus operations are performed) and the IX Bus arbiter waits for one of the cases to occur.

System designs using more than two IXP1200s can use the token request pins in two configurations. The first configuration (shown in Figure 6-14) uses a fixed token passing scheme where all the TK_REQ_IN pins are pulled high to indicate that the token should always be passed regardless of whether or not another IXP1200 needs access to the bus.

Figure 6-14. Shared IXP1200 System Using Fixed Token Passing

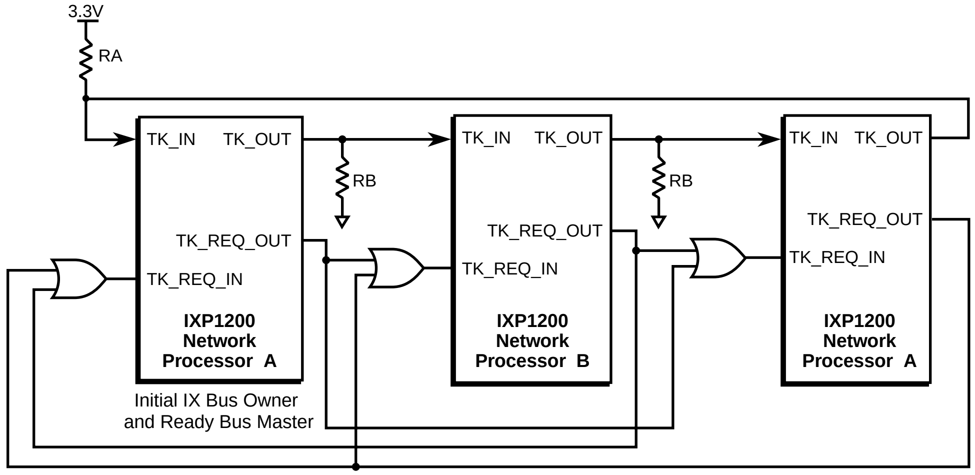

3.3V
RA
TK_IN
TK_OUT
RB
VDD
RC
TK_REQ_IN
TK_REQ_OUT
NC
IXP1200 Network Processor A
IXP1200 Network Processor B
IXP1200 Network Processor C
Initial IX Bus Owner and Ready Bus Master

Notes: Resistor Description
RA - Indicates Initial IX Bus Owner and Ready Bus Master.
RB - Indicates NOT the Initial IX Bus Owner and Ready Bus Master.
RC - Indicates always pass token.

A7197-01

The second configuration (shown in Figure 6-15) uses a dynamic token passing scheme where all the TK_REQ_OUT pins are logically "ORed" to indicate when any one of the IXP1200s need ownership of the IX Bus. If any one of the IXP1200s need ownership of the IX Bus, the token is passed in round-robin order to the next IXP1200.

Figure 6-15. Shared IXP1200 System Using Dynamic Token Passing


3.3V
RA
TK_IN
TK_OUT
RB
TK_REQ_OUT
TK_REQ_IN
IXP1200 Network Processor A
Initial IX Bus Owner and Ready Bus Master
TK_IN
TK_OUT
RB
TK_REQ_OUT
TK_REQ_IN
IXP1200 Network Processor B
TK_IN
TK_OUT
TK_REQ_OUT
TK_REQ_IN
IXP1200 Network Processor A

Notes: Resistor Description
RA - Indicates Initial IX Bus Owner and Ready Bus Master.
RB - Indicates NOT the Initial IX Bus Owner and Ready Bus Master.

A7196-01

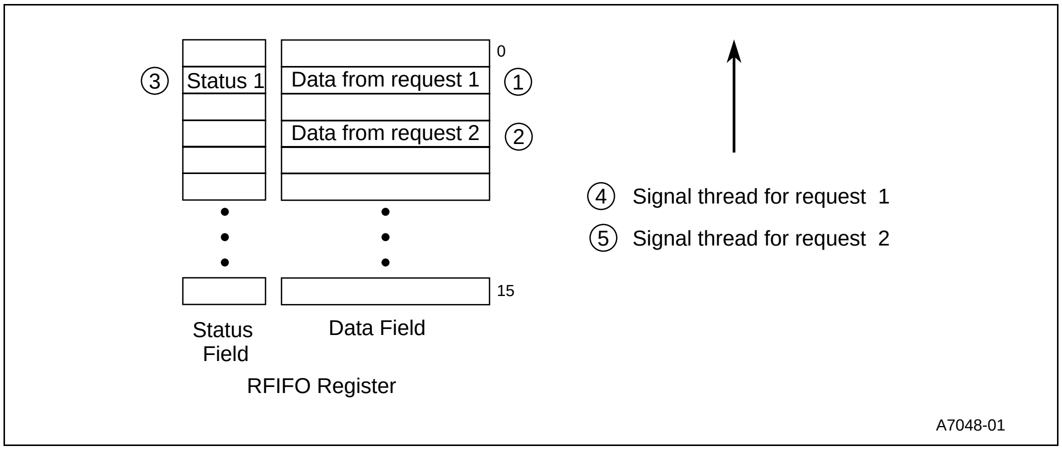
6.6.2.4 Status Mode

Status mode is selected when bit 6 in the RDYBUS_TEMPLATE_CTL register is 0. In Status mode, the Receive State Machine expects status to be provided on the IX Bus during the next data transfer on the same port following EOP. The Receive State Machine automatically reads the status and places the data into the status field of the RFIFO element. The only thing the IXP1200 does with this status field is to copy the inverted state of bit 8 to the Receive Error bit (bit 18) in the RCV_CNTL register.

For 64-bit bidirectional IX Bus mode, one quadword is always read and placed into this RFIFO status field. For 32-bit unidirectional IX Bus mode, REC_REQ[26] specifies whether the status length is one longword or one quadword. The one quadword option in 32-bit unidirectional mode requires that the Receive State Machine access the IX Bus twice. The extended data option (see Section 6.6.4.6) is not supported if the one quadword status option selected.

Based on the Receive Request, the Receive State Machine always attempts to read a maximum number of quadwords based on the request parameters (see Section 6.6.4.3). Due to pipelining issues, the Receive State Machine begins processing the next Receive Request before the current request is complete. Therefore, if an EOP is detected on the last IX Bus data cycle for request 1, the Receive State Machine allows request 2 to finish before going back for the status for request 1.

Figure 6-16. Receive State Machine Behavior when EOP Occurs on the Last IX Bus Data Cycle



6.6.3 Ready Bus

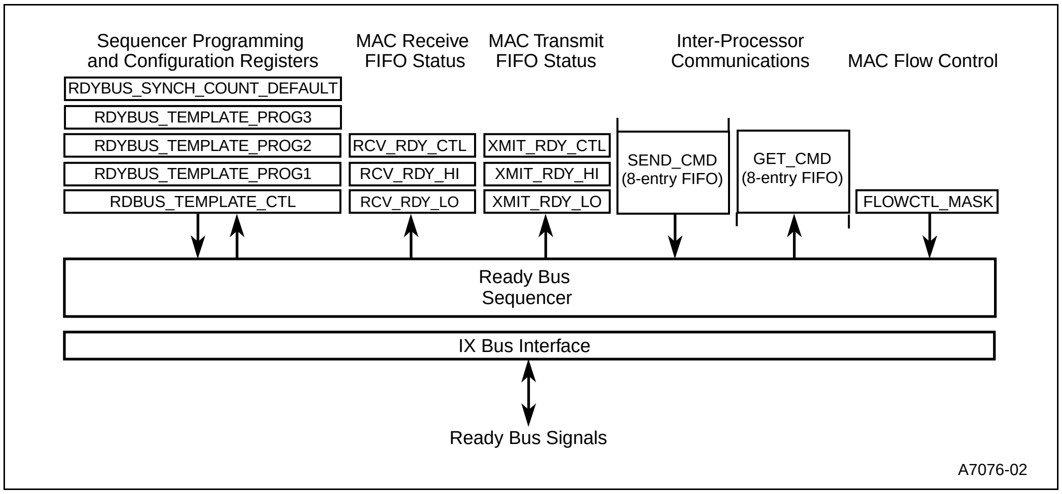
The Ready Bus is a separate bus from the IX Bus, and is used for the following purposes:

- Retrieving the MAC Receive and Transmit FIFO Ready flags from MAC devices.
- Asserting Flow Control to MAC devices.
- Inter-IXP1200 communications.

The Ready Bus is controlled by a programmable sequencer and is comprised of an 8-bit data bus (RDYBUS[7:0])and five control pins (RDYCTL#[4:0]). The control pins select which of the three functions is being performed on the bus. In 1-2 MAC mode, separate control signals are provided by the IXP1200 for each function. In this mode, the GPIO[0] pin is also used by the Ready Bus as a control signal, and Inter-IXP1200 communications are not supported. 3+ MAC mode requires an external decode of the RDYCTL#[4:0] signals to provide the individual select signals for each function. All of the Ready Bus functions are supported in 3+ MAC mode.

Figure 6-17 is a block diagram of the Ready Bus and the relevant FBI CSRs used to program and control the Ready Bus. The purpose of each of these components is described in this section.

Figure 6-17. Ready Bus Block Diagram

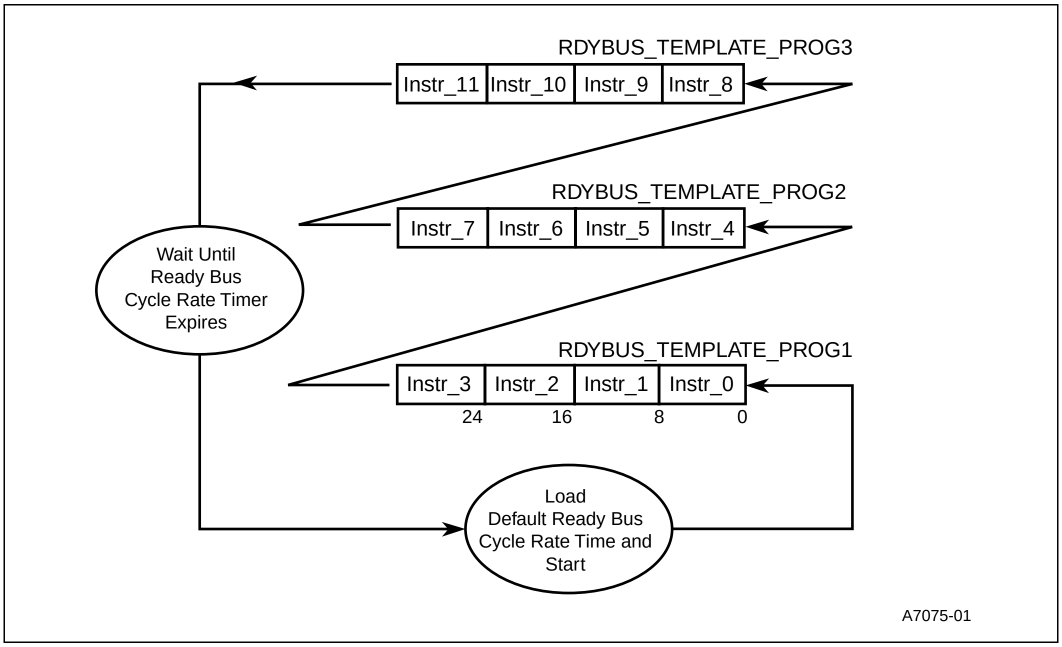


6.6.3.1 Ready Bus Sequencer

The Ready Bus is controlled by a programmable sequencer. The sequencer is configured and programmed using FBI registers. The sequencer is configured and enabled after reset and then runs freely.

The Ready Bus Sequencer instructions are loaded into the three RDYBUS_TEMPLATE_PROGx registers. These registers hold a total of twelve 8-bit instructions. A sequence rate count can be programmed into the RDYBUS_SYNC_COUNT_DEFAULT register to determine the maximum rate at which the program repeats the instructions. The Ready Bus Sequencer is enabled via the RDYBUS_TEMPLATE_CTL register. When it is enabled, the sequence rate counter is started. Instruction execution begins with **instr_0** and completes after **instr_11**. The sequencer then waits until the Ready Bus sequence rate timer expires and repeats execution of instructions 0 through 11. The sequence rate count has no effect if it is less than the time required to execute the instructions.

Figure 6-18. Ready Bus Instruction Sequence



The RDYBUS_TEMPLATE_PROG registers can be programmed by both the Microengines (using the **csr** instruction) and the StrongARM* core. In typical applications, the StrongARM* core programs the sequencer once as part of the power-up initialization sequence.

The steps required to set up the Ready Bus Sequencer are as follows:

1. Load the Sequencer program into the three RDYBUS_TEMPLATE_PROGx registers.
2. Load the cycle rate into the RDYBUS_SYNCH_COUNT_DEFAULT register.
3. Configure the Transmit Autopush settings via the XMIT_RDY_CTL register.
4. Configure the Receive Autopush settings via the RCV_RDY_CTL register.
5. Configure the pin signaling, Inter-IXP1200 modes, and enable the Ready Bus Sequencer via the RDY TEMPLATE_CTL register.

6.6.3.2 Ready Bus Master and Slave Modes

The Ready Bus supports a Master-Slave mode for systems that require multiple IXP1200s to share a common Ready Bus. Only one IXP1200 is configured as a Ready Bus master; the others are configured as Ready Bus slaves. A Ready Bus master executes a Ready Bus program while the other IXP1200s snoop the Ready Bus to determine what action they should perform.

The TK_IN pin is used as a configuration pin to determine which IXP1200 powers up as a Ready Bus master and which powers up as slaves. This pin defines which IXP1200 should drive the Ready Bus to a known state prior to configuring the Ready Bus through software. If TK_IN is pulled high during reset, an IXP1200 becomes the Ready Bus master. All other IXP1200s should be designated as slaves by pulling TK_IN low. In a single IXP1200 system, this pin should be configured to set the IXP1200 as the Ready Bus master.

After power-up, software can determine if it is configured as the Ready Bus master by reading a status bit within the RCV_RDY_CNT register. Software can then program the Ready Bus accordingly. If the IXP1200 is the Ready Bus master, the Ready Bus should be configured as if it were a single IXP1200. If the IXP1200 is a Ready Bus slave, the program and sequence count does not need to be loaded into the FBI registers, but the RDYBUS_TEMPLATE_CTL register must be configured to select 3+ MAC mode and Slave mode as well as enable the Ready Bus Sequencer.

6.6.3.3 Ready Bus Instructions

The Ready Bus supports the following instructions:

- **NOP**
- **rxrdy**
- **RxAutopush**
- **get1**
- **send**
- **flwctl**
- **txrdy**
- **TxAutopush**
- **get2**

Table 6-14 through Table 6-22 provide descriptions of these instructions, the 8-bit opcodes for each variation of the instruction, and the number of IX Bus clock cycles required to perform each instruction.

NOP Perform no operation on the Ready Bus. In Master-Slave mode, the slave also performs no operation.

Table 6-14. NOP Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
NOP1	1F	3
NOP2	3F	6
NOP3	5F	9
NOP4	7F	12
NOP5	9F	15
NOP6	BF	18
NOP7	DF	21
NOP8	FF	24

rxrdy Collect the Receive Ready flags from the specified MAC ports and place them into the RCV_RDY_LO and RCV_RDY_HI registers. In Master-Slave mode, the slave snoops the Ready Bus and reads the Receive Ready flags into its RCV_RDY_LO and RCV_RDY_HI registers.

Table 6-15. rxrdy Instruction Variations

MAC Range	Opcode (Hex)	#IX Bus Clock Cycles		MAC Range	Opcode (Hex)	#IX Bus Clock Cycles
rxrdy MAC 0	FB	3		**rxrdy** MAC 3	98	3
rxrdy MAC 0 to 1	DB	6		**rxrdy** MAC 3 to 4	78	6
rxrdy MAC 0 to 2	BB	9		**rxrdy** MAC 3 to 5	58	9
rxrdy MAC 0 to 3	9B	12		**rxrdy** MAC 3 to 6	38	12
rxrdy MAC 0 to 4	7B	15				
rxrdy MAC 0 to 5	5B	18		**rxrdy** MAC 4	6B	3
rxrdy MAC 0 to 6	3B	21		**rxrdy** MAC 4 to 5	4B	6
				rxrdy MAC 4 to 6	2B	9
rxrdy MAC 1	DA	3				
rxrdy MAC 1 to 2	BA	6		**rxrdy** MAC 5	4A	3
rxrdy MAC 1 to 3	9A	9		**rxrdy** MAC 5 to 6	2A	6
rxrdy MAC 1 to 4	7A	12				
rxrdy MAC 1 to 5	5A	15		**rxrdy** MAC 6	29	3
rxrdy MAC 1 to 6	3A	18				
rxrdy MAC 2	B9	3				
rxrdy MAC 2 to 3	99	6				
rxrdy MAC 2 to 4	79	9				
rxrdy MAC 2 to 5	59	12				
rxrdy MAC 2 to 6	39	15				

txrdy Collect the Transmit Ready flags from the specified MAC ports and place them into the XMIT_RDY_LO and XMIT_RDY_HI registers. In Master-Slave mode, the slave snoops the Ready Bus and reads the Transmit Ready flags into its XMIT_RDY_LO and XMIT_RDY_HI registers.

Table 6-16. txrdy Instruction Variations

MAC Range	Opcode (Hex)	# IX Bus Clock Cycles		MAC Range	Opcode (Hex)	# IX Bus Clock Cycles
txrdy MAC 0	F7	3		**txrdy** MAC 3	94	3
txrdy MAC 0 to 1	D7	6		**txrdy** MAC 3 to 4	74	6
txrdy MAC 0 to 2	B7	9		**txrdy** MAC 3 to 5	54	9
txrdy MAC 0 to 3	97	12		**txrdy** MAC 3 to 6	34	12
txrdy MAC 0 to 4	77	15				
txrdy MAC 0 to 5	57	18		**txrdy** MAC 4	67	3
txrdy MAC 0 to 6	37	21		**txrdy** MAC 4 to 5	47	6
				txrdy MAC 4 to 6	27	9
txrdy MAC 1	D6	3				
txrdy MAC 1 to 2	B6	6		**txrdy** MAC 5	46	3
txrdy MAC 1 to 3	96	9		**txrdy** MAC 5 to 6	26	6
txrdy MAC 1 to 4	76	12				
txrdy MAC 1 to 5	56	15		**txrdy** MAC 6	25	3
txrdy MAC 1 to 6	36	18				
txrdy MAC 2	B5	3				
txrdy MAC 2 to 3	95	6				
txrdy MAC 2 to 4	75	9				
txrdy MAC 2 to 5	55	12				
txrdy MAC 2 to 6	35	15				

RxAutopush Perform RxAutopush operation. The RxAutopush operation does the following:

- Increments the Receive Ready Count in the RCV_RDY_CNT register.
- If enabled via the RCV_RDY_CTL, it automatically writes the RCV_RDY_CNT, RCV_RDY_LO, and RCV_RDY_HI registers to the Receive Scheduler SRAM Transfer Registers.
- If enabled via the RCV_RDY_CTL register, the Receive Scheduler thread is signaled via the AUTO_PUSH_SIG context event signal when the RxAutopush operation is complete.

In Master-Slave mode, Ready Bus signaling is generated by the Ready Bus master. The slave snoops the Ready Bus, and, if it detects autopush signaling, it also performs an RxAutopush.

Table 6-17. RxAutopush Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
RxAutopush	0C	3

TxAutopush Perform TxAutopush operation. The TxAutopush operation does the following:

- If enabled via the XMIT_RDY_CTL, it automatically writes the XMIT_PTR, XMIT_RDY_LO, and XMIT_RDY_HI registers to the Transmit Scheduler SRAM Transfer Registers.
- If enabled via the XMIT_RDY_CTL register, the Transmit Scheduler thread is signaled via the AUTO_PUSH_SIG context event signal when the TxAutopush operation is complete.

In Master-Slave mode, Ready Bus signaling is generated by the Ready Bus master. The slave snoops the Ready Bus, and, if it detects autopush signaling, it also performs a TxAutopush.

Table 6-18. TxAutopush Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
TxAutopush	9C	3

flwctl Writes the 8-bit data from the FLOWCTL_MASK register to the device also specified in the FLOWCTL_MASK register. In Master-Slave mode, only the master executes the **flwctl** instruction. When a slave detects a flow control operation, it does not perform any action. If the FLOWCTL_MASK register contains invalid data, a single cycle **nop** operation is performed.

Table 6-19. flwctl Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
flwctl	00	6 - flowctl_mask contains valid data 1 - flowctl_mask contains invalid data

get1 Used for Ready Bus communications between multiple IXP1200s. When this instruction is executed, data is read from a Ready Bus slave SEND_CMD FIFO into the master GET_CMD FIFO. The **get1** instruction is supported only in 3+ MAC bidirectional mode and 3+ MAC unidirectional mode. One IXP1200 is designated as the Ready Bus master that sends data to other devices on the Ready Bus. The other device can be another IXP1200 in Slave mode. When the slave detects a **get1**, it drives the data onto RDYBUS[7:0]. Up to eight longwords can be read from a slave in a single **get1** instruction.

Table 6-20. Get1 Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
get1_1lw	1E	9
get1_2lw	3E	17
get1_3lw	5E	25
get1_4lw	7E	33
get1_5lw	9E	41
get1_6lw	BE	49
get1_7lw	DE	57
get1_8lw	FE	65

get2 Same as **get1** except it is used to get a message from another Ready Bus master in a chain configuration (i.e., two IXP1200 Ready Buses separated by an external FIFO). The IXP1200 is able to decode **get1** but cannot decode **get2**. **get2** can be decoded by external hardware.

Table 6-21. Get2 Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
get2_1lw	0E	9
get2_2lw	2E	17
get2_3lw	4E	25
get2_4lw	6E	33
get2_5lw	8E	41
get2_6lw	AE	49
get2_7lw	CE	57
get2_8lw	EE	65

send Used for Ready Bus communications between multiple IXP1200s. When this instruction is executed, data is sent from the master SEND_CMD FIFO to the Ready Bus. The **send** instruction is supported only in 3+ MAC bidirectional mode and 3+ MAC unidirectional mode. One IXP1200 is designated as the Ready Bus master that sends data to other devices on the Ready Bus. The other devices can be another IXP1200 in Slave mode or another IXP1200 in a chained configuration. Refer to Section 6.6.3.2 for more information on Master-Slave modes.

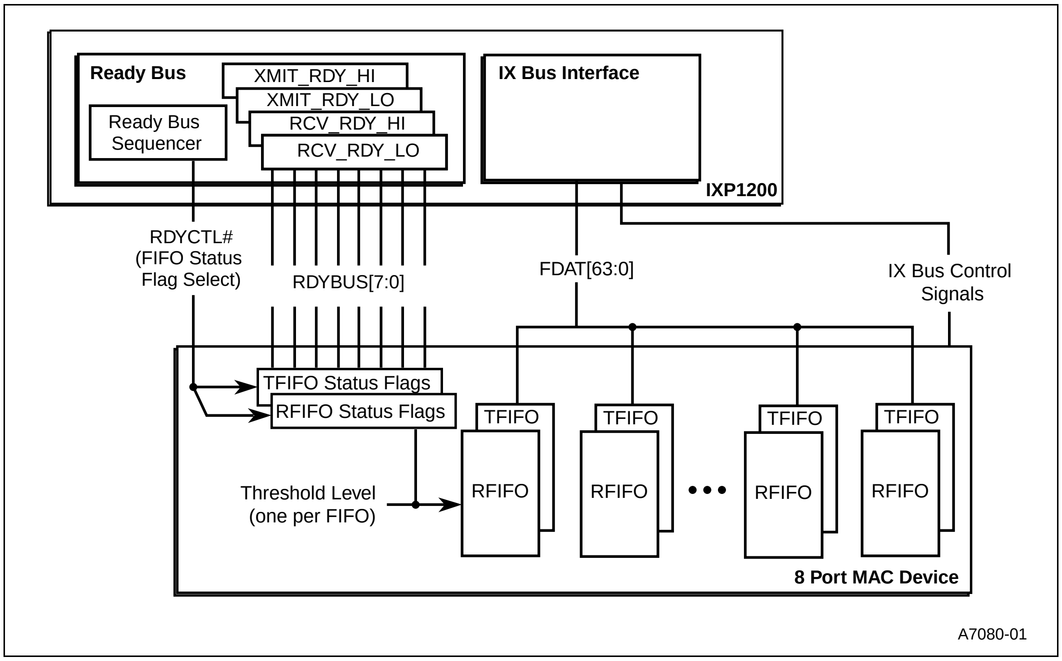
Table 6-22. send Instruction Variations

Description	Opcode (Hex)	# IX Bus Clock Cycles
send_1lw	0D	9
send_2lw	2D	17
send_3lw	4D	25
send_4lw	6D	33
send_5lw	8D	41
send_6lw	AD	49
send_7lw	CD	57
send_8lw	ED	65

6.6.3.4 Reading the MAC FIFO Ready Flags

A typical MAC device (such as the Intel® 21440 Octal 10/100 Mbps Ethernet Controller) provides transmit and receive Ready flags that indicate whether the amount of data in a FIFO has reached a certain threshold level. The Ready Bus Sequencer (Figure 6-19) periodically polls the Receive and Transmit FIFO Ready Flags and places them into FBI registers if the Ready Bus Sequencer is programmed with the **rxrdy** and **txrdy** instructions.

Figure 6-19. MAC Device Sequencer Flow



6.6.3.5 Receive Ready Flags

When the Ready Bus Sequencer is programmed with a **rxrdy** instruction, it reads the Receive Ready flags from the MAC device(s) specified in the instruction and places the flags into the RCV_RDY_HI and RCV_RDY_LO registers. Each bit in these registers corresponds to a MAC port on the IX Bus and up to fifty-six ports total from up to seven MAC devices are supported. A single **rxrdy** instruction allows the Ready flags to be read from multiple MAC devices.

The IXP1200 also supports two Fast Port Receive Ready Flags. These flags are discussed in more detail in Section 6.6.8. The Fast Port Receive Ready flags are placed into the RCV_RDY_CNT register. All three registers can be autopushed to a Receive Scheduler SRAM Transfer Registers if the Ready Bus Sequencer executes the **RxAutopush** instruction, and the Autopush operation is enabled via the RDYBUS_TEMPLATE_CTL register.

If an IXP1200 is configured as a Ready Bus slave, it snoops the Ready Bus control signals. When it detects that a master is performing a read of the Receive Ready flags, it also latches the data into its RCV_RDY_LO and RCV_RDY_HI registers.

6.6.3.6 Transmit Ready Flags

When the Ready Bus Sequencer is programmed with a **txrdy** instruction, it reads the Transmit Ready flags from the MAC device(s) specified in the instruction and places them into the XMIT_RDY_LO and XMIT_RDY_HI registers. Each bit in these registers corresponds to a MAC port on the IX Bus and up to fifty-six ports total from up to seven MAC devices are supported. A single **txrdy** instruction allows the Ready flags to be read from multiple ports. These registers can be autopushed to the Transmit Scheduler SRAM Transfer Registers if the Ready Bus Sequencer executes the **TxAutopush** instruction and the Autopush operation in enabled via the XMIT_RDY_CTL register.

If an IXP1200 is configured as a Ready Bus slave, it snoops the Ready Bus control signals. When it detects that a master is performing a read of the Transmit Ready flags, it also latches the data into its XMIT_RDY_LO and XMIT_RDY_HI registers.

6.6.3.7 Autopush Operation

The Ready Bus Sequencer supports two Autopush instructions: **RxAutopush** and **TxAutopush**.

The **RxAutopush** operation performs the following functions:

- Increments the Receive Ready Count in the RCV_RDY_CNT register
- If enabled via the RCV_RDY_CTL register, it automatically writes the RCV_RDY_CNT, RCV_RDY_LO, and RCV_RDY_HI registers to the Receive Scheduler SRAM Transfer Registers.
- If enabled via the RCV_RDY_CTL register, the Receive Scheduler thread is signaled via the AUTO_PUSH_SIG context event signal when the RxAutopush operation is complete.

The TxAutopush operation performs the following functions:

- If enabled via the XMIT_RDY_CTL register, it automatically writes the XMIT_PTR, XMIT_RDY_LO, and XMIT_RDY_HI registers to the Transmit Scheduler SRAM Transfer Registers.
- If enabled via the XMIT_RDY_CTL register, the Transmit Scheduler thread is signaled via the AUTO_PUSH_SIG context event signal when the TxAutopush operation is complete.

In Master-Slave mode, the Ready Bus master generates the Ready Bus signaling and the slave snoops the Ready Bus. When it detects autopush signaling, it also performs an autopush operation.

Autopush Versus Polling

There are two methods for reading the Ready Bus registers: polling and autopush.

The advantages to autopush are the following:

- The latest data is available to a Microengine thread without having to explicitly read it.
- Lower latency associated with reading the registers compared to polling.
- Since lower latency is achieved, a Microengine thread may spend more time performing other tasks rather than waiting for the read operations to complete.

Polling requires that a Microengine thread periodically issue read references to the FBI Unit. One example of a polling algorithm uses two Microengine threads to poll and process the Ready Bus registers. The main Microengine thread performs the typical Receive or Transmit Scheduler function, and a secondary Microengine thread running on the same Microengines performs the simple task of reading the registers periodically into an absolute addressed SRAM Transfer Registers. When the main scheduler thread is ready for the new data, it reads the SRAM Transfer Register. The advantage of this method is that the secondary scheduler thread would read the registers while the main scheduler thread is waiting for references to complete. This allows the register read operations to be hidden.

Tx Autopush and Rx Autopush Sequence

Although 3+ MAC mode supports unique Rx and Tx autopush signaling, RDYCTL#[4] is always ignored by a Ready Bus slave. A slave always cycles between transmit and receive autopush if it detects the generic autopush signaling on the bus. The first autopush after the sequencer is enabled is always interpreted by the slave as a Tx Autopush. When programming the sequence of instructions in Master-Slave mode, the programmer should ensure that the sequencer pushes the Transmit Ready flags first. The tables that follow show three examples of the sequence of instruction for an autopush operation.

Example 6-4. Autopush Method 1

Instruction Sequence	Action	Comment
txrdy MAC 0-1	Master asserts signaling, slave snoops Tx Ready flags	Flags are pushed in order and there is more time between the two instances when the Autopush prevent state is asserted.
TxAutopush	Master asserts signaling and pushes Tx Ready flags, slave pushes Tx Ready flags	
rxrdy MAC 0-1	Master asserts signaling, slave snoops Rx Ready flags	
RxAutopush	Master asserts signaling and pushes Rx Ready flags, slave pushes Rx Ready flags	

Example 6-5. Autopush Method 2

Instruction Sequence	Action	Comment
txrdy MAC 0-1	Master asserts signaling, slave snoops Tx Ready flags	There may be a minor impact on performance since the Scheduler is not able to access the Transfer Registers for two consecutive Autopush Prevent Window time periods.
rxrdy MAC 0-1	Master asserts signaling, slave snoops Rx Ready flags	
TxAutopush	Master asserts signaling and pushes Tx Ready flags, slave pushes Tx Ready flags	
RxAutopush	Master asserts signaling and pushes Rx Ready flags, slave pushes Rx Ready flags	

Example 6-6. Autopush Method 3

Instruction Sequence	Action	Comment
rxrdy MAC 0-1	Master asserts signaling, slave snoops Rx Ready flags	Incorrect method: Flags are pushed out of order.
RxAutopush	Master asserts signaling and pushes Rx Ready flags, slave pushes Rx Ready flags	
txrdy MAC 0-1	Master asserts signaling, slave snoops Tx Ready flags	
TxAutopush	Master asserts signaling and pushes Tx Ready flags, slave pushes Tx Ready flags	

Autopush Protection

The Microengine SRAM Transfer Registers and the GPRs are implemented as register files so multiple registers can be read at one time. This allows a Microengine to read an operand for one instruction and write the result from another in a single cycle. However, only one source can access a particular register at any one time. Access contention can occur when the SRAM Transfer Registers are accessed by both the Microengines and the autopush write operation at the same time.

To prevent contention, the FBI Unit provides the PUSH_PROTECT state. All Microengine threads can read this state using the **br_inp_state** instruction, although in most applications only the Transmit and Receive Scheduler threads use it. When the FBI Unit is ready to autopush to the Transfer Register, it asserts the PUSH_PROTECT state and waits for a time specified in Autopush Prevent Window field of the RCV_RDY_CTL before writing to the Transfer Registers.

Whenever a Microengine thread reads the SRAM Transfer Registers, it should check the PUSH_PROTECT state (using the **br_inp_state** instruction). If it is set, it should wait until it is cleared employing a tight loop structure. Otherwise, it should immediately read the registers and move them to GPRs. The order in which the Autopush operation occurs is: transfer register 0, transfer register 1, and transfer register 2. The Microengine should also move the data from the SRAM Transfer Registers to GPRs in this order.

6.6.3.8 Interpreting The Receive Ready Flags

The Ready Bus polls the MAC FIFO Status Flags periodically and asynchronously to other events occurring in the IXP1200. Ideally, the rate at which the MAC FIFO Ready flags are polled is greater than the maximum rate at which the data is arriving at the MAC ports. This presents the issue of determining whether the MAC FIFO Ready flags read by the Ready Bus are new or whether they have already been read. The IXP1200 provides two methods that can be used to determine which flags are new and which are old.

Receive Ready Count

Each time the **RxAutopush** instruction is executed, it increments the Receive Ready Count in the RCV_RDY_CNT register. The Receive Ready Count can be used by the Receive Scheduler to determine whether the state of specific flags have to be evaluated or whether they can be ignored because Receive Requests have been issued and the port is currently being serviced. For example, if the FIFO threshold for a Fast Ethernet port is set so that the Receive Ready flags are asserted when 64 bytes of data are in the MAC RFIFO, it can be said that the flags do not change until the next 64 bytes arrive, 5120 ns later. If the Sequencer is programmed to collect the Receive Ready flags four times each 5120 ns period, the next three sets of Ready flags that are collected can be ignored.

This method of tracking the freshness of the Ready flags is relatively simple to implement in microcode. However, there is a chance that the Ready flags are ignored when they are actually reporting new data in the FIFO. For most applications, this inaccuracy does not have any negative repercussion for the receive process since it is reported when the next Ready flags are collected. If greater accuracy is required, the Receive Request Count can be used.

Receive Request Count

Each time a Receive Request is completed and the receive control information is pushed onto the RCV_CNTL FIFO, the FBI Unit increments the Receive Request Count. This counter is recorded in the RCV_RDY_CNT register the first time the Ready Bus Sequencer executes either an **rxrdy** or **txrdy** instruction for each program execution loop. The Receive Scheduler can use this count to track which how many requests the Receive State machine has completed. As the Receive Scheduler thread issues commands, it can maintain a list in chronological order of the Receive Requests that it submits and the ports associated with each request. The Receive Request count (RCV_RDY_CNT[15:8]) can be used to determine which requests it can retire from the list:

a. Receive Scheduler reads the RCV_RDY_CNT register:

   Receive Request Count read is 4 (no Receive Requests outstanding).

   MAC Receive FIFO Ready Flags set for MAC 1/port 3, MAC 1/port 7, and MAC 2/port2.

b. Receive Scheduler issues Receive Requests to MAC 1/port 3, MAC 1/port 7, and MAC 2/port2.

c. Request for MAC 1/port 3 is serviced by the hardware.

d. Receive Scheduler reads the RCV_RDY_CNT register again:

   Receive Request Count read is 5.

   MAC Receive FIFO Ready Flags set for MAC 1/port 3, MAC 1/port 7, and MAC 2/port2.

e. Interpretation:

   Ready flags for MAC 1/port 7 and MAC 2/port 2 are old, since those requests have not been processed.

   Ready flag for MAC 1/port 3 is new, since the Receive State Machine has processed that request and there is data available.

6.6.3.9 Flow Control

The flow control function allows the Ready Bus Sequencer to place a single byte of data onto the Ready Bus to be used as flow control for the MACs. Device select signals are provided for up to seven separate devices allowing the data to be targeted to multiple devices (the number of MAC devices depends on the MAC mode).

A Microengine thread writes the 8-bit data and a MAC number to the FLOWCTL_MASK register. A Valid flag (the FCMA bit in the RCV_RDY_CNT register) is set each time new data is written to the FLOWCTL_MASK register. A Microengine thread can determine if valid data has been written to the FLOWCTL_MASK register by reading the Valid flag. When the **flwctl** instruction is executed and there is valid data in the FLOWCTL_MASK register, the Ready Bus Sequencer places the data onto the Ready Bus. If the Valid flag is not set, the sequencer executes a NOP for a single IX Bus cycle and it proceeds to the next instruction.

In the Master-Slave mode, only the master can assert flow control to the MAC devices. When a slave detects a flow control operation, it does not perform any action.

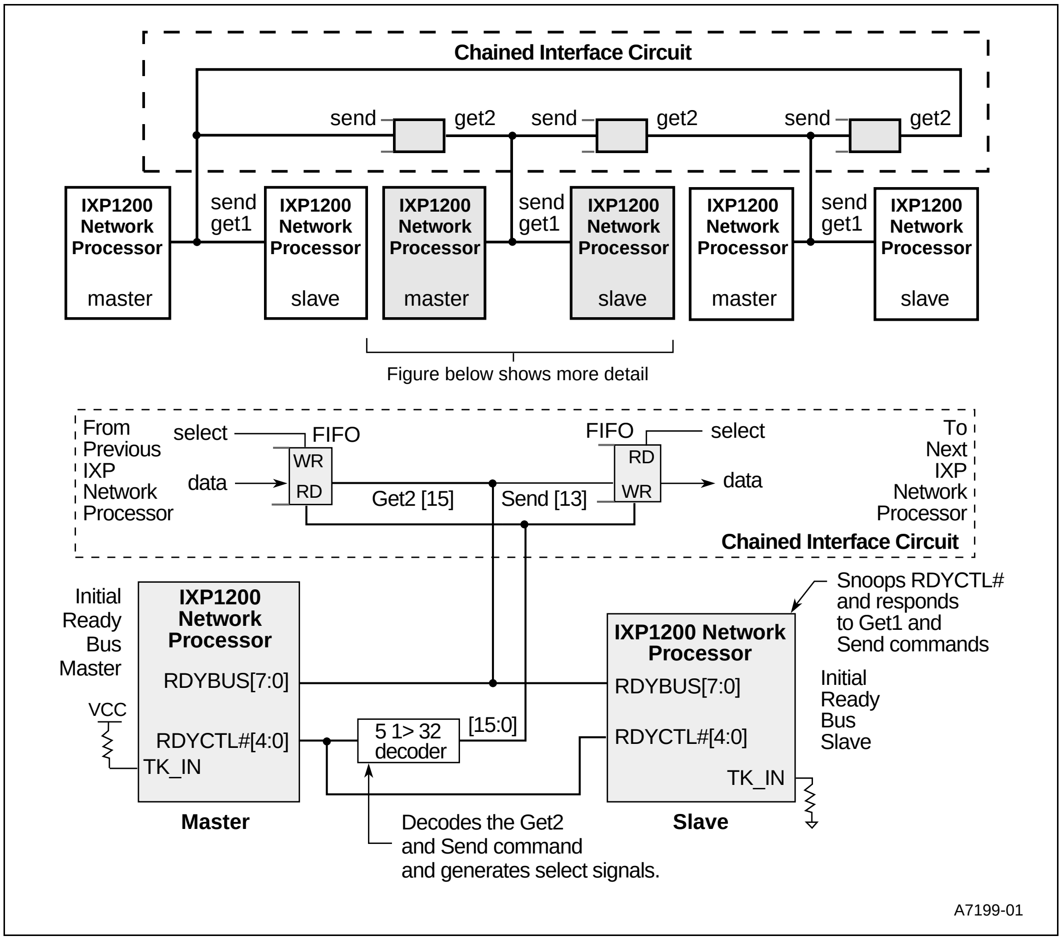
MAC devices (such as the Intel 21440 Octal 10/100 Mbps Ethernet Controller) use the 8-bit data as a flow control signal to indicate that back pressure should be applied to data arriving on the network. A Microengine thread can also send status information using the flow control function during hardware debugging.

6.6.3.10 Ready Bus Communications

The Ready Bus provides a communications channel between the Microengines in different IXP1200s. The StrongARM* core does not have access to Ready Bus Communications. The communications channel has a deterministic latency that is dependent on the Ready Bus sequencer programming. Ready Bus Communications is only supported in 3+ MAC mode.

Ready Bus communications support two modes: Master-to-Slave and Chained Master-to-Master. These modes can be used separately or together. The Master-to-Slave mode provides communication between a single master and single slave and does not require any interface logic between the two IXP1200s. System designs with more than two IXP1200s can use the Master-to-Master mode to pass messages to the next IXP1200 in a chained configuration. The Master-to-Master mode requires external interface logic. The figure below shows multiple IXP1200s in both the Master-to-Slave and Master-to-Master mode configurations.

Figure 6-20. Master-to-Slave and Master-to-Master Chained Ready Bus Configuration

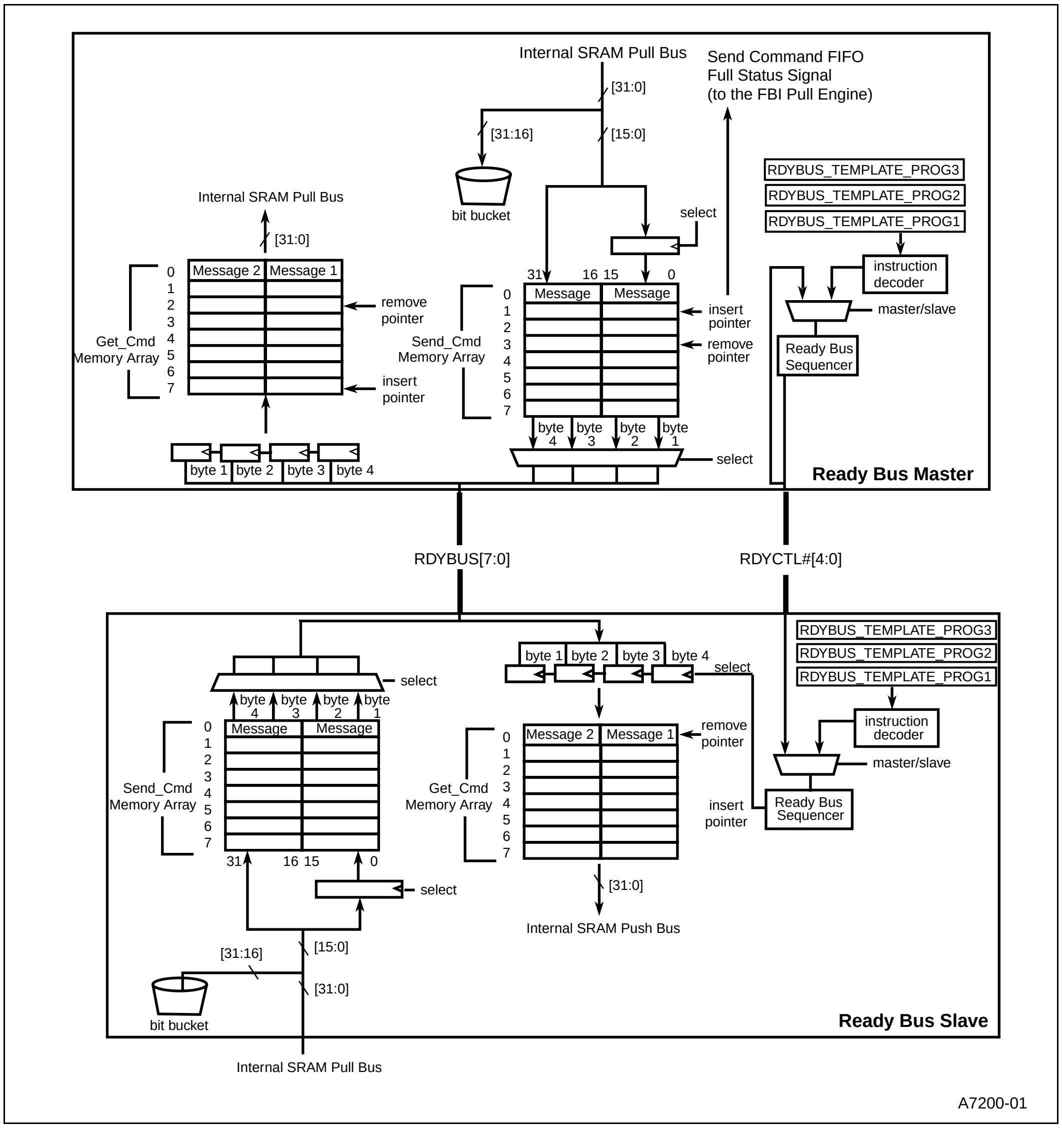

Chained Interface Circuit
send
get2
send
get2
send
get2
IXP1200 Network Processor master
send get1
IXP1200 Network Processor slave
IXP1200 Network Processor master
send get1
IXP1200 Network Processor slave
IXP1200 Network Processor master
send get1
IXP1200 Network Processor slave
Figure below shows more detail
From Previous IXP Network Processor
select
FIFO
WR
data
RD
Get2 [15]
Send [13]
FIFO
select
RD
WR
data
To Next IXP Network Processor
Chained Interface Circuit
Initial Ready Bus Master
IXP1200 Network Processor
RDYBUS[7:0]
VCC
RDYCTL#[4:0]
TK_IN
Master
5 1> 32 decoder
[15:0]
Decodes the Get2 and Send command and generates select signals.
Snoops RDYCTL# and responds to Get1 and Send commands
IXP1200 Network Processor
RDYBUS[7:0]
RDYCTL#[4:0]
TK_IN
Initial Ready Bus Slave
Slave
A7199-01

The Ready Bus Sequencer supports two get instructions (**get1** and **get 2**) and one send instruction (**send**). The **get1** and **send** instructions are used to communicate in the Master-to-Slave mode, and the **get2** and **send** instructions are used in the Master-to-Master mode. Unique signaling is provided on the RDYCTL#[4:0] pins whenever the Ready Bus Sequencer executes these instructions. A Ready Bus slave snoops the RDYCTL#[4:0] signals and responds to the **get1** or **send** signaling. In the Master-to-Master mode, a Ready Bus master communicates with another Ready Bus Master though external FIFOs. The RDYCTL# signals must be externally decoded to control access to the FIFOs. Since the **send** instruction is shared by the Master-to-Slave and Master-to-Master modes, whenever the **send** instruction is executed, the send data is written to both the Ready Bus slave and the chained Send FIFO. A Microengine thread in the receiving IXP1200 must determine if it is the target for the send data.

The Ready Bus Master Sends and Gets messages each time the Get or Send instruction is executed by the Ready Bus Sequencer. The data is transferred across the Ready Bus at a deterministic rate since the time required to loop through a Ready Bus program loop execution is deterministic (and so is the rate at which the **get** and **send** instructions are executed). A message is defined as 16 bits. However the **get** and **send** instructions specify the size of a transfer across the Ready Bus in increments of longwords. For reasons explained later in this section, it is expected that most applications will implement a single Microengine thread to manage Ready Bus communications between IXP1200s.

The paragraphs that follow describe the **get** and **send** Ready Bus instructions. Refer to Figure 6-21 while reading the descriptions for information on how the data flows during instruction execution.

Figure 6-21. Data Flow for Get and Send Instructions



Send

A Ready Bus master sends data to another IXP1200 by writing 16-bit messages to the SEND_CMD register using the **csr** instruction. Two 16-bit messages must be written to the SEND_CMD register before the messages are placed onto the Send_Cmd memory array within the FBI Unit.

If the Send_Cmd memory array contains more than 4 longwords, and a Microengine thread issues another write request to the SEND_CMD register, a back-pressure mechanism will prevent the FBI Pull Engine from processing the request (and any other requests behind it in the FBI Pull queue) until data is removed from the memory array and driven onto the Ready Bus. When there are at least 4 entries available to service the next write request, the Pull engine is allowed to complete the write request. This mechanism allows up to four longwords to be written in a single instruction and ensure proper operation of the back-pressure mechanism.

The **send** instruction supports writes to the SEND_CMD register of up to eight longwords in a single instruction. If the programmer chooses to manage the rate at which the data is written to the SEND_CMD register in software so that the FBI Pull engine does not stall, bursts greater than four can be used. One way to manage the rate at which data is written to the SEND_CMD register is to have a single Microengine thread write to the SEND_CMD register. This Microengine thread writes data to the SEND_CMD register and waits until the data is sent before writing more data. A Microengine thread can determine if messages have been sent by placing an **autopush** instruction in the Ready Bus program. When this instruction is executed, the RCV_RDY_CNT field in the RCV_RDY_CNT register is incremented and, if enabled in the RCV_RDY_CTL register, the **autopush** signal event will occur. A Microengine Thread can either wait for the **autopush** signal event or it can poll the RCV_RDY_CNT register to determine if the RCV_RDY_CNT field has incremented.

When the Ready Bus Sequencer executes the **send** instruction, it always sends the amount of data specified in the **send** instruction regardless of whether there is valid data in the memory array. If there is no valid data, zeroes are sent.

The Ready Bus Slave snoops the RDYCTL# signals, and when it detects the **send** signaling, it reads the data, one byte at a time, on the Ready Bus data pins (RDYBUS[7:0]), latches the data internally, and places the data into the GET_CMD memory array after a longword is assembled.

A Microengine thread in the Ready Bus Slave reads the messages from the GET_CMD register. A dedicated signaling mechanism is not provided for the Microengine thread to determine when data has been written to the GET_CMD memory array. However, the RCV_RDY_CNT field in the RCV_RDY_CNT register or the Autopush signal event can be used as described earlier in this section. For this reason, it is expected that most applications will have a single Microengine thread manage Ready Bus communications so that only one source is reading from the GET_CMD Register.

The Microengine thread must determine if the data it reads from the GET_CMD register is valid. If the Ready Bus master does not send valid data to the slave, the 32-bit data contains all zeros.

For applications where the Ready Bus Master sends messages to both a Ready Bus Slave and to another Ready Bus Master in the chained configuration, both devices respond to the **send** signaling. In this case, the Microengine thread must distinguish if it is the target for the data. A software semantic that can be implemented to distinguish if it is the target for the data is to have the sending Microengine thread set the MSB of a message to a zero for a slave IXP1200 and a one for a chained IXP1200.

Get

The Ready Bus Sequencer in the Ready Bus Master gets data from another IXP1200 each time the **get1** or **get2** instruction is executed. An IXP1200 configured as a Ready Bus Slave recognizes the signaling on the RDYCTL# pins when a **get1** instruction is executed and drives data from its Send_Cmd memory array onto the Ready Bus pins (RDYBUS[7:0]). A Ready Bus Slave always ignores the signaling presented during the **get2** instruction. The **get2** signaling should be decoded by external logic and used to enable a FIFO in the chained configuration.

Each **get** instruction specifies a read size of 1 to 8 longwords. The Sequencer reads the data in byte increments across the 8-bit data bus, assembles them into a 32-bit value, and writes the value into the GET_CMD memory array. The GET_CMD memory array is 8 entries by 32 bits and is mapped to the GET_CMD register. Each time the GET_CMD register is read by a Microengine thread, a memory array entry is read, the entry is cleared to zero, and a remove pointer is incremented to point at the next item in the array. A Microengine thread reads the

GET_CMD register using the **csr** instruction and can read up to eight entries in a single instruction.

The Ready Bus Slave always drives data onto the Ready Bus data pins (RDYBUS) whenever the RDYCTL# signal specifies a **get1** instruction whether valid data is present in the Send_Cmd memory array or not. Invalid data is defined as any read of the GET_CMD register that returns all zeroes (32 bits) It is up to the Microengine thread in the Ready Bus Master to read the data in the GET_CMD register, detect, and discard invalid data.

When a Ready Bus Slave detects the **get** signaling on the Ready Bus control signals, it drives the data from its SEND_CMD memory array onto the Ready Bus. After data is sent from the Slave SEND_CMD memory array, the memory array entries are cleared to zero. A dedicated signaling mechanism is not provided to the Microengines for determining when the Sequencer gets data from the Ready Bus. However, the RCV_RDY_CNT field in the RCV_RDY_CNT register or the Autopush signal event can be used as described earlier in this section.

The **get2** instruction is used for enabling the external Get FIFO from another Master device in the chained configuration. The **get2** signaling is decoded to enable an external FIFO to drive data onto the Ready Bus. Data is written into this FIFO by a Master IXP1200 on another Ready Bus and read by the Master IXP1200 on the next Ready Bus into its GET_CMD FIFO.

6.6.3.11 Example Ready Bus Sequencer Programs

The following Ready Bus Sequencer example programs show how the instructions are used as well as the number of Ready Bus cycles required for execution.

Example 6-7. Ready Bus Program for a Simple 2 MAC (16, 10/100 Mbps Ports) Single IXP1200 System

Instruction Address	Instruction Code	Instruction	Description	IX Bus Clock Cycles
instr_0	0xD7	**txrdy** MAC 0-1	Fetch the Transmit Ready Bits for MACs 0 through 1.	6
instr_1	0x9C	**TxAutopush**		3
instr_2	0xDB	**rxrdy** MAC 0-1	Fetch the Receive Ready Bits for MACs 0 through 1.	6
instr_3	0x0C	**RxAutopush**		3
instr_4	0x00	**flwctl**	Write flow control MAC mask to the MAC. MAC and mask are specified in the FLW_CTL_MASK register.	6 (valid flow control mask) or 1 (invalid flow control mask).
instr_5, instr_11	0x1F	**NOP1**	Perform no operations.	7 nops x 3 = 21
Total Instruction Cycles (@ 66 MHz).				40 or 45
Desired cycle rate (four times per minimum sized Fast Ethernet packet - 5120 ns).				84 cycles
RDYBUS_SYNC_COUNT_DEFAULT [RDYBUS_TIMER] (Minimum cycles = 83 + 1).				84

Example 6-8. Ready Bus Program for a 4 MAC (32, 10/100Mbps Ports) Dual IXP1200 System

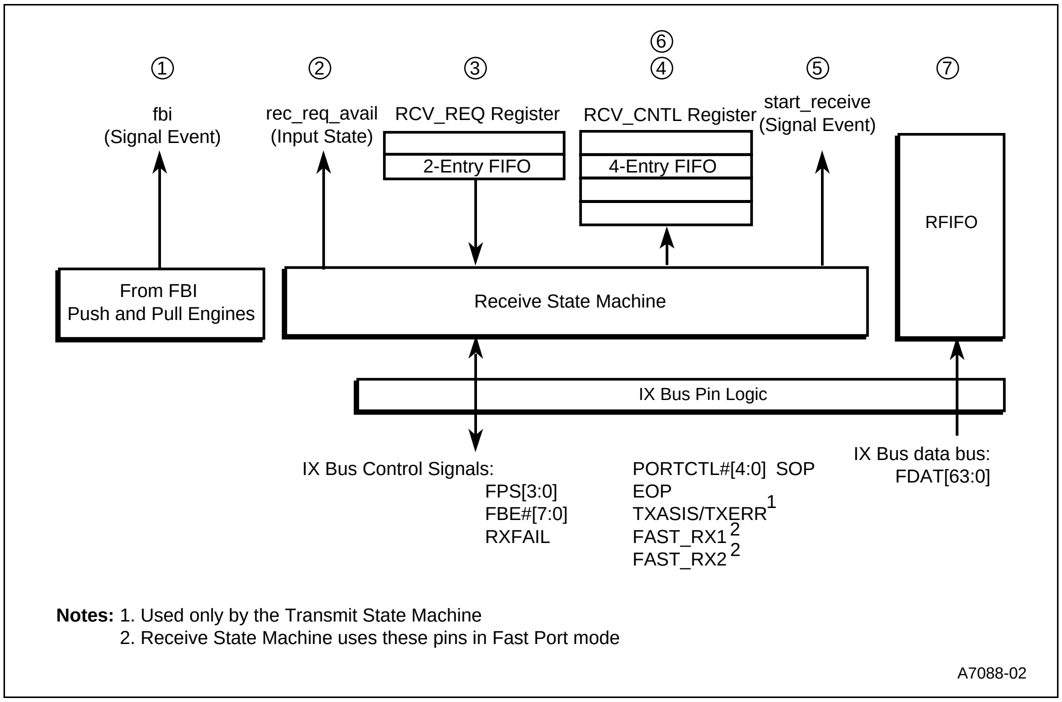
Instruction Address	Instruction Code	Instruction	Description	IX Bus Clock Cycles
instr_0	0x97	**txrdy** MAC 0-3	Fetch the Transmit Ready Bits for MACs 0 through 3.	12
instr_1	0x9C	**TxAutopush**		3
instr_2	0x9B	**rxrdy** MAC 0-3	Fetch the Receive Ready Bits for MACs 0 through 3.	12
instr_3	0x0C	**RxAutopush**		3
instr_4	0x00	**flwctl**	Write flow control MAC mask to the MAC. MAC and mask are specified in the FLW_CTL_MASK register.	6 (valid flow control mask) or 1 (invalid flow control mask).
instr_5	0x3E	**get1_2lw**	Get 4 messages (16 bytes) from device 1.	17
instr_6	0x3D	**send_2lw**	Put 4 messages (16 bytes) out to device 1.	17
instr_7, instr_11	0x1F	**NOP1**	Perform no operations	4 nops x 3 = 12
Total Instruction Cycles (@66Mhz)				77 or 82
Desired cycle rate (4 times per minimum sized Fast Ethernet packet - 5120ns)				84
RDYBUS_SYNC_COUNT_DEFAULT [RDYBUS_TIMER] (Minimum cycles = 83 + 1).				84

6.6.4 Receive State Machine and RFIFO - Receiving Data From IX Bus

The Microengines initiate transfers across the IX Bus, but it is the Receive State Machine that performs the actual read transfer on the IX Bus. The interface to the Receive State Machine is through two FBI registers: RCV_REQ and RCV_CNTL. The steps involved with initiating IX Bus receive transfers are listed below and shown in Figure 6-22. They are described in more detail in later sections.

1. A Microengine thread ensures that any previous Receive Requests have been received into the RCV_REQ FIFO.
2. The Microengine thread determines if there is room in the RCV_REQ FIFO for another Receive Request.
3. Write a Receive Request and signal back when the FBI Unit completes the request.
4. The Receive State Machine processes the request, moves data into the RFIFO, and writes control information to the RCV_CNTL FIFO.
5. The Receive State Machine generates a start_receive signal event to the Microengine thread specified in the Receive Request.
6. The Microengine thread responds to the signal event and reads the control information from the RCV_CNTL register to determine, among other things, where the data is in the RFIFO.
7. The Microengine thread reads the data from the RFIFO on quadword boundaries into its SRAM Transfer Registers or it moves the data directly into SDRAM.

Figure 6-22. Receive State Machine and RFIFO - Receiving Data From IX Bus



Notes: 1. Used only by the Transmit State Machine
2. Receive State Machine uses these pins in Fast Port mode

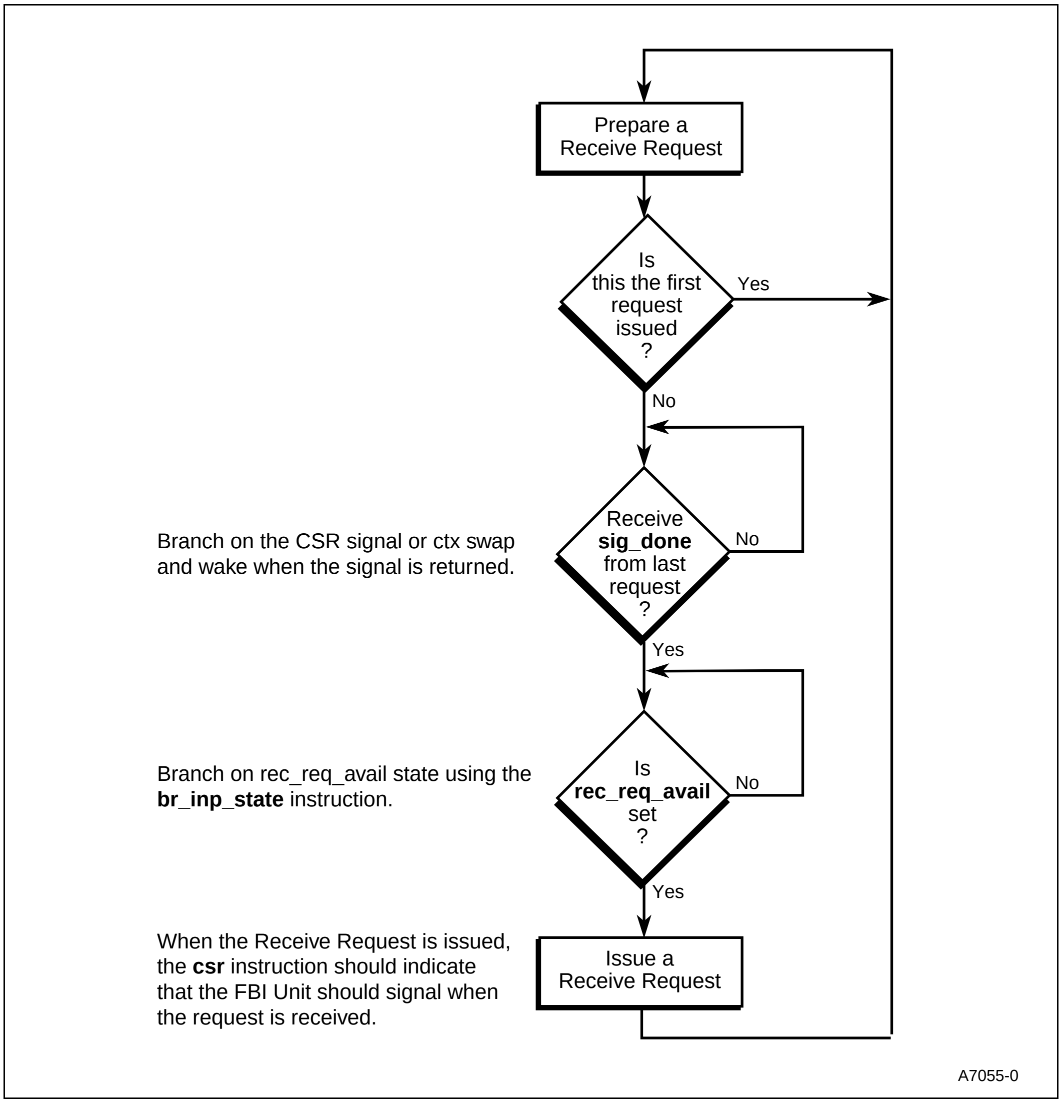
6.6.4.1 Issuing a Receive Request

The RCV_REQ register is used to initiate a receive transfer on the IX Bus. This is mapped to a two-entry FIFO that can be written by the Microengines.

The rate at which Receive Requests are issued to the Receive State Machine must be managed in microcode. To assist in managing the rate at which Receive Requests are issued, the FBI Unit provides status indicating that the RCV_REQ FIFO has room available for another Receive Request. This status is provided via the **rec_req_avail** state. The **rec_req_avail** state is a single signal provided to each Microengine that indicates whether there is room in the RCV_REQ register. A Microengine thread can test this state using the **br_inp_state** instruction.

The **rec_req_avail** state does not indicate whether a Receive Request has been issued and is currently being delivered to the RCV_REQ register. Therefore a Microengine should specify a sig_done or ctx_swap after issuing each Receive Request and then wait for the signal to be returned before testing the **rec_req_avail** state. The following figure illustrates this point.

Figure 6-23. Issuing a Receive Request Flow


Prepare a Receive Request
Is this the first request issued ?
Yes
No
Receive **sig_done** from last request ?
No
Yes
Is **rec_req_avail** set ?
No
Yes
Issue a Receive Request
A7055-0

Branch on the CSR signal or ctx swap and wake when the signal is returned.

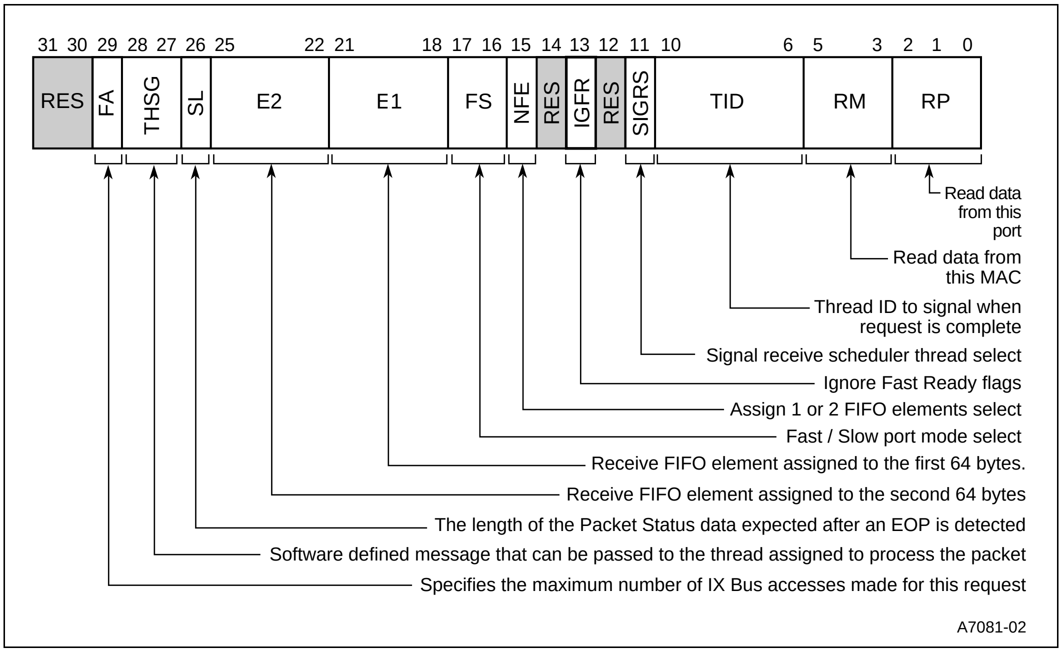
Branch on rec_req_avail state using the **br_inp_state** instruction.

When the Receive Request is issued, the **csr** instruction should indicate that the FBI Unit should signal when the request is received.

6.6.4.2 Receive Request Format (RCV_REQ Register)

The RCV_REQ register instructs the Receive State Machine on how to receive data from the IX Bus. This register is mapped to a two-entry FIFO that is written by a Microengine thread and is read by the Receive State Machine. Figure 6-24 shows the format of the RCV_REQ register.

Figure 6-24. RCV_REQ Register Format


31 30 29 28 27 26 25 22 21 18 17 16 15 14 13 12 11 10 6 5 3 2 1 0
RES
FA
THSG
SL
E2
E1
FS
NFE
RES
IGFR
RES
SIGRS
TID
RM
RP
Read data from this port
Read data from this MAC
Thread ID to signal when request is complete
Signal receive scheduler thread select
Ignore Fast Ready flags
Assign 1 or 2 FIFO elements select
Fast / Slow port mode select
Receive FIFO element assigned to the first 64 bytes.
Receive FIFO element assigned to the second 64 bytes
The length of the Packet Status data expected after an EOP is detected
Software defined message that can be passed to the thread assigned to process the packet
Specifies the maximum number of IX Bus accesses made for this request
A7081-02

FA

Maximum IX Bus Accesses. Indicates the number of times the Receive State Machine should attempt to access the IX Bus. The options are as follows:

- For 64-bit bidirectional IX Bus mode:
  0: Fetch 8 quadwords from IX Bus
  1: Fetch 9 quadwords from IX Bus
- For 32-bit unidirectional IX Bus mode:
  0: Fetch 16 longwords from IX Bus
  1: Fetch 18 longwords from IX Bus

When FA = 1, the last quadword of data is placed into the extended data field of the specified RFIFO element. Cannot be set if the NFE field is set.

TMSG

Thread MeSsaGe. The Microengine thread that issues the Receive Request may pass a 2-bit message through the FBI Unit via this field to the Microengine thread specified in the TID field. The FBI copies the value of this field to a similar 2-bit field in the RCV_CNTL register (RCV_CNTL[31:30]). In Fast Port mode, the value 3 (i.e., both bits set) is reserved.

SL

Status Length. This field is only relevant in 32-bit unidirectional mode and is ignored in 64-bit bidirectional mode. It specifies the size of the IX Bus status information read after an EOP is detected, and therefore specifies how many times the Receive State Machine needs to access the IX Bus to get the status. If this bit is 0, status is 32 bits and only requires one access. If this bit is 1, status is 64 bits and requires two IX Bus accesses.

Note: In 64-bit bidirectional mode, one access is always performed if status is enabled.

E2

Element 2. If the Receive State Machine is instructed to fill two RFIFO elements (see NFE field), the second 64 bytes of data are placed in this element.

E1

Element 1. Specifies the Receive FIFO element where the IX Bus data is placed. If the Receive State Machine is instructed to read two elements of data (see NFE field), the first 64 bytes of data are placed in this element.

FS

Fast or Slow port mode. Specifies whether the Receive State Machine should treat the request as a Fast Port or a Slow port. If Fast Port is specified, this field also specifies which sequence numbers are incremented and assigned to the request, and that the Receive State Machine should perform speculative requests.

NFE

Number of FIFO Elements. Specifies whether the Receive State Machine should attempt to fill one or two RFIFO elements. If status is enabled and two RFIFO elements are specified, the status is always placed in the status field of the first RFIFO element. Cannot be set if the FA field is set.

IGFR

IGnore Fast Ready Flag. If Fast Port mode is selected (see FS), the Receive State Machine ignores the Fast Ready Flag pins (FAST_RX1, FAST_RX2) and processes the request regardless of the state of these pins. Should only be used if it can be guaranteed that there is data available in the MAC port Receive FIFO.

SIGRS

SIGnal Receive Scheduler. If set, the Receive Scheduler thread specified in the RCV_RDY_CTL register, in addition to the thread specified in the TID field, is signaled after the Receive Request is completed. As a result, the data in the RCV_CNTL register must be read twice before the Receive Request data is retired from the RCV_CNTL FIFO.

TID

Thread ID. This field determines which Microengine thread is signaled after the Receive Request is processed.

RM

Receive MAC. During a receive, this field determines which MAC is selected.

RP

Receive Port. During a receive, this field determines which port of the MAC is selected.

6.6.4.3 Processing Receive Requests

The Receive State Machine reads the RCV_REQ register to determine how it should receive data from the IX Bus. This includes how the signaling should be performed on the IX Bus, where the data should be placed into the RFIFO, and which Microengine should be signaled once the data is received.

The Receive State Machine looks for a valid Receive Request in the RCV_REQ FIFO. In 32-bit unidirectional mode, the Receive State Machine processes the Receive Request immediately since it always owns the 32-bit receive IX Bus. In 64-bit bidirectional mode, the Receive State Machine requests ownership of the IX Bus from the IX Bus Arbiter. The IX Bus Arbiter grants access to the IX Bus based on a round robin algorithm between the following resources.

- Receive State Machine (process one Receive Request)
- Transmit State Machine (transmit one valid TFIFO element)

- Another IXP1200 (Shared IX Bus mode only)

When the Receive State Machine is granted access to the IX Bus, it selects the MAC device by asserting the appropriate PORTCTL# signals and it selects the specified port within the MAC by asserting the FPS signals. It then begins receiving data from the MAC device on the FDAT data bus. The Receive State Machine always attempts to read either eight or nine quadwords of data from the MAC device on the IX Bus as specified in the Receive Request. If the MAC device asserts the EOP signal, the Receive State Machine terminates the receive early (before the eight or nine accesses are made).

The Receive State Machine calculates the total bytes received for each Receive Request and reports the value in the RCV_CNTL register. If EOP is received, the Receive State Machine reads the byte enable signals (FBE#) to determine the number of valid bytes in the last receive data cycle.

Note: The FBE# signals are ignored if the EOP signal is not asserted during a receive data cycle.

If status is enabled, the Receive State Machine expects the status data to be provided on the next data cycle following the asserted EOP signal. The status data is placed into the RFIFO status field associated with the RFIFO element.

If the Receive State Machine sees the RXFAIL signal asserted during a receive data cycle, it aborts the receive, and, if the status option is enabled, it does not go back for status. It also indicates in the RCV_CNTL register that RXFAIL was asserted during the receive. The IX Bus data cycles terminate as if EOP were seen (see Section 6.7).

After the FIFO element(s) are filled, the Receive State Machine creates control information about the data in the RFIFO and places it into the Receive Control register (RCV_CNTL). The assigned Microengine thread uses the control information to process the data.

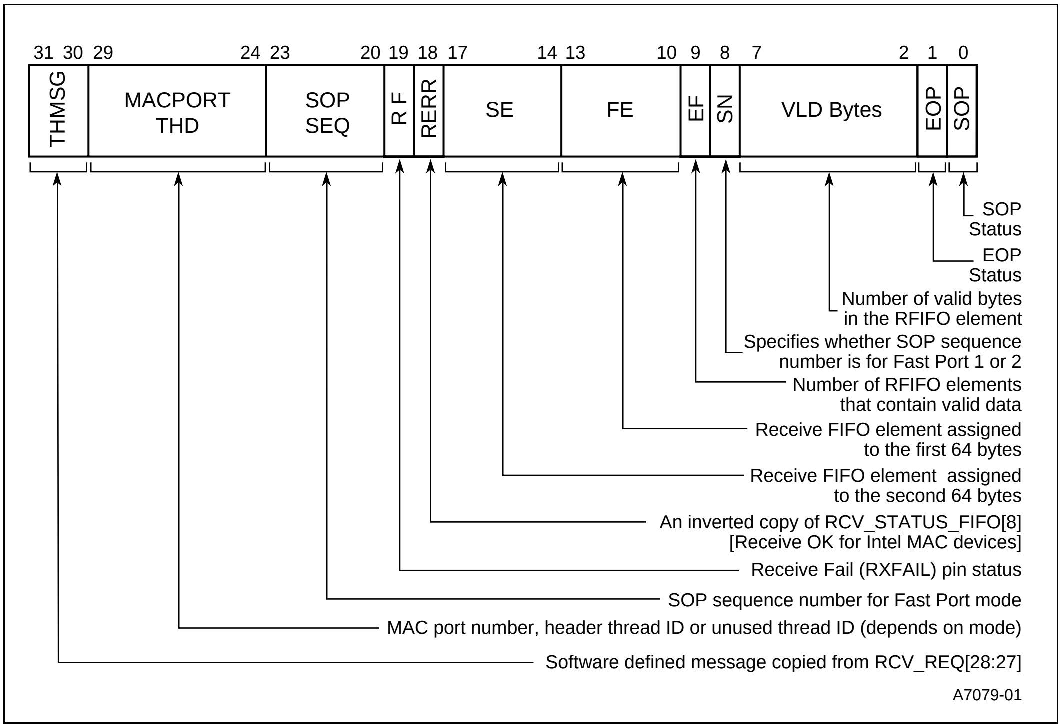
The RCV_CNTL register is mapped to a four-entry FIFO that is written by the Receive State Machine and read by a Microengine thread. The FBI Unit signals the assigned thread when a valid entry reaches the top of the FIFO. When a Microengine thread reads the RCV_CNTL register, the data is popped off the FIFO. If the SIGRS field is set in the RCV_REQ register, the Receive Scheduler thread specified in the RCV_RDY_CTL register is signaled in addition to the thread specified in TID field. In this case, the data in the RCV_CNTL register must be read twice before the Receive Request data is retired from the RCV_CNTL FIFO and the next thread is signaled.

The Receive State Machine writes to the RCV_CNTL register as long as the FIFO is not full. If the RCV_CNTL FIFO is full, the Receive State Machine stalls and stops accepting any more Receive Requests.

6.6.4.4 Receive Data Control Information Format (RCV_CNTL Register)

The RCV_CNTL register provides instruction to the signaled Microengine thread to process the data. This register is mapped to a four-entry FIFO that is written by the Receive State Machine and read by a Microengine thread. Figure 6-25 shows the format of the RCV_CNTL register.

Figure 6-25. RCV_CNTL Register Format


31 30 29 24 23 20 19 18 17 14 13 10 9 8 7 2 1 0
THMSG
MACPORT THD
SOP SEQ
R F
RERR
SE
FE
EF
SN
VLD Bytes
EOP
SOP
SOP Status
EOP Status
Number of valid bytes in the RFIFO element
Specifies whether SOP sequence number is for Fast Port 1 or 2
Number of RFIFO elements that contain valid data
Receive FIFO element assigned to the first 64 bytes
Receive FIFO element assigned to the second 64 bytes
An inverted copy of RCV_STATUS_FIFO[8] [Receive OK for Intel MAC devices]
Receive Fail (RXFAIL) pin status
SOP sequence number for Fast Port mode
MAC port number, header thread ID or unused thread ID (depends on mode)
Software defined message copied from RCV_REQ[28:27]
A7079-01

THMSG

THread MeSsaGe. A 2-bit message may be passed from the thread that issues the Receive Request to the thread that reads the RCV_CNTL register. The thread that issues the Receive Request writes a 2-bit message to RCV_REQ[28:27]. The Receive State Machine copies the message into this field.

Note: If the port serviced is a Fast Port, the Receive State Machine overwrites any message with the value 3 if the speculative Receive Request is cancelled.

MACPORT/THD

MAC Port Number/Header Thread ID.

- If the Fast Port mode specified in RCV_RDY_CNTL[14:13] = 00 (single thread mode), and if the SOP field is cleared, then this field is the thread ID of the receive processing thread that was not used. Or, if the SOP field is set, this field is the MAC port ID.
- If the Fast Port mode specified in RCV_RDY_CNTL[14:13] = 01 (header/body thread mode), and if the SOP bit is cleared, then this

field is the header thread ID. Or, if the SOP field is set, this field is the MAC port ID.

- If the Fast Port mode specified in RCV_RDY_CNTL[14:13] = 10 (Explicit Thread mode), this field specifies the MAC port ID.

SOP SEQ

Start of Packet Sequence Number. Relevant for Fast Ports. If the SOP field is set, specifies the sequence number assigned to this request. The SN field specifies whether the sequence number is for Fast Port 1 or 2. If the SOP field is cleared, specifies the MAC packet (MPKT) sequence number assigned to this request.

RF

Receive Fail. If set, the RXFAIL pin was asserted during receive, indicating a packet was received with errors, the Receive Request was aborted, and status is not transferred.

RERR

Receive ERRor. Relevant if status is enabled. Provides an inverted copy of bit 8 of the RFIFO element status field. The Intel 21440 Multiport 10/100 Mbps Ethernet Controller MAC provides Receive OK status in this bit position that indicates no error occurred while receiving this data.

Note: If the RF field is set or status is not enabled, status is not received and this field is irrelevant.

SE

Second Element. A copy of the E2 field in the RCV_REQ register. If the EF field is set, this specifies the RFIFO element that contains the second 64 bytes of data.

FE

First Element. Specifies the RFIFO element that contains the first 64 bytes of data.

EF

Elements filled. Specifies whether one or two RFIFO elements were filled during the Receive Request.

SN

Sequence Number. Only relevant for Fast Port modes. Specifies whether the sequence number assigned in the SOP_SEQ field is for Fast Port 1 or Fast Port 2.

VLD BYTES

Valid Bytes. If EOP is set, specifies the number of valid bytes in the RFIFO element. Valid bytes are contiguous beginning at the first byte in the RFIFO element. (Actual Valid Bytes = Valid bytes + 1). If EOP is not set, this field should be ignored.

EOP

End Of Packet. If set, the EOP signal was asserted during an IX Bus receive cycle.

SOP

Start Of Packet. If set, the SOP signal was asserted during a receive data cycle. If the Receive Request indicated that the port was a Fast Port and this field is set, the following additional functions are performed.

The sequence number for the specified Fast Port is incremented.

- Header Body Mode:
  If this field is asserted, the Receive State Machine signals the thread specified in the RCV_REQ TID field (header thread). If this field was asserted on the previous Receive Request, the Receive State Machine signals the thread specified in the RCV_REQ TID field (body thread), latches this TID internally and uses the latched

TID for all future Receive Requests to this port until the SOP pin is asserted once again.

- Single Thread Mode:
If SOP is asserted during this Receive Request, the Receive State Machine uses the TID field in the RCV_REQ register and latches it internally. If SOP is not asserted during a Receive Request, the Receive State Machine uses the latched TID.

6.6.4.5 Interpreting the Byte Enable Pins (FBE#[7:0])

Eight byte enable signals are provided to indicate which bytes are valid on the IX Bus.

The Receive State Machine uses the FBE#[7:0] signals to calculate the number of bytes received for each Receive Request. The Receive State Machine reports the byte count in the REC_CNTL register. The number of bytes read for each receive data cycle is assumed to be eight unless the EOP signal is asserted during the data cycle. In this case, the Receive State Machine reads the byte enable signals (FBE#) to determine the number in the last receive data cycle. The FBE# signals are ignored if the EOP signal is not asserted during a receive data cycle.

The Transmit State Machine asserts the FBE# signals to indicate which bytes are valid during an IX Bus transmit cycle. A Microengine thread writes the number of valid quadwords and the number of valid bytes in the last quadword to the control field of the TFIFO. The Transmit State Machine always asserts the correct byte enable signals regardless of whether the EOP signal is asserted.

For bidirectional IX Bus mode, FBE#[7:0] is a single 8-bit bus. For unidirectional mode, FBE#[7:0] is partitioned into two 4-bit buses where bits [3:0] are used for receive and bits [7:4] are used for transmit.

The IX Bus Endian bit in the RDYBUS_TEMPLATE_CTL register determines how the Transmit and Receive State Machines interpret the byte enable signals based on either big endian or little endian data. The IXP1200 interprets endian format based on longword boundaries as shown below.

Table 6-23. 64-bit Mode Endian Format

<table>
<tr><th>63</th><th>56</th><th>55</th><th>48</th><th>47</th><th>40</th><th>39</th><th>32</th><th>31</th><th>24</th><th>23</th><th>16</th><th>15</th><th>8</th><th>7</th><th>0</th></tr>
<tr><td colspan="2">FDAT [63:56]</td><td colspan="2">FDAT [55:48]</td><td colspan="2">FDAT [47:40]</td><td colspan="2">FDAT [39:32]</td><td colspan="2">FDAT [31:24]</td><td colspan="2">FDAT [23:16]</td><td colspan="2">FDAT [15:8]</td><td colspan="2">FDAT [7:0]</td></tr>
<tr><td colspan="16"><b>Big Endian</b></td></tr>
<tr><td colspan="2">Byte 5</td><td colspan="2">Byte 6</td><td colspan="2">Byte 7</td><td colspan="2">Byte 8</td><td colspan="2">Byte 1</td><td colspan="2">Byte 2</td><td colspan="2">Byte 3</td><td colspan="2">Byte 4</td></tr>
<tr><td colspan="16"><b>Little Endian</b></td></tr>
<tr><td colspan="2">Byte 8</td><td colspan="2">Byte 7</td><td colspan="2">Byte 6</td><td colspan="2">Byte 5</td><td colspan="2">Byte 4</td><td colspan="2">Byte 3</td><td colspan="2">Byte 2</td><td colspan="2">Byte 1</td></tr>
</table>

Table 6-24. Byte Enable Signaling Based On Endian Format

Valid Bytes	FBE#[7:0]	
	Big Endian Mode	Little Endian Mode
1	0xF7	0xFE
2	0xF3	0xFC
3	0xF1	0xF8
4	0xF0	0xF0
5	0x70	0xE0
6	0x30	0xC0
7	0x10	0x80
8	0x00	0x00

Table 6-25. 32-bit Mode Endian Format

63	56	55	48	47	40	39	32	
FDAT [63:56]		FDAT [55:48]		FDAT [47:40]		FDAT [39:32]		Transmit
FDAT [31:24]		FDAT [23:16]		FDAT [15:8]		FDAT [7:0]		Receive
Big Endian								
Byte 1		Byte 2		Byte 3		Byte 4		
Little Endian								
Byte 4		Byte 3		Byte 2		Byte 1		

Table 6-26. Byte Enable Signaling Based On Endian Format

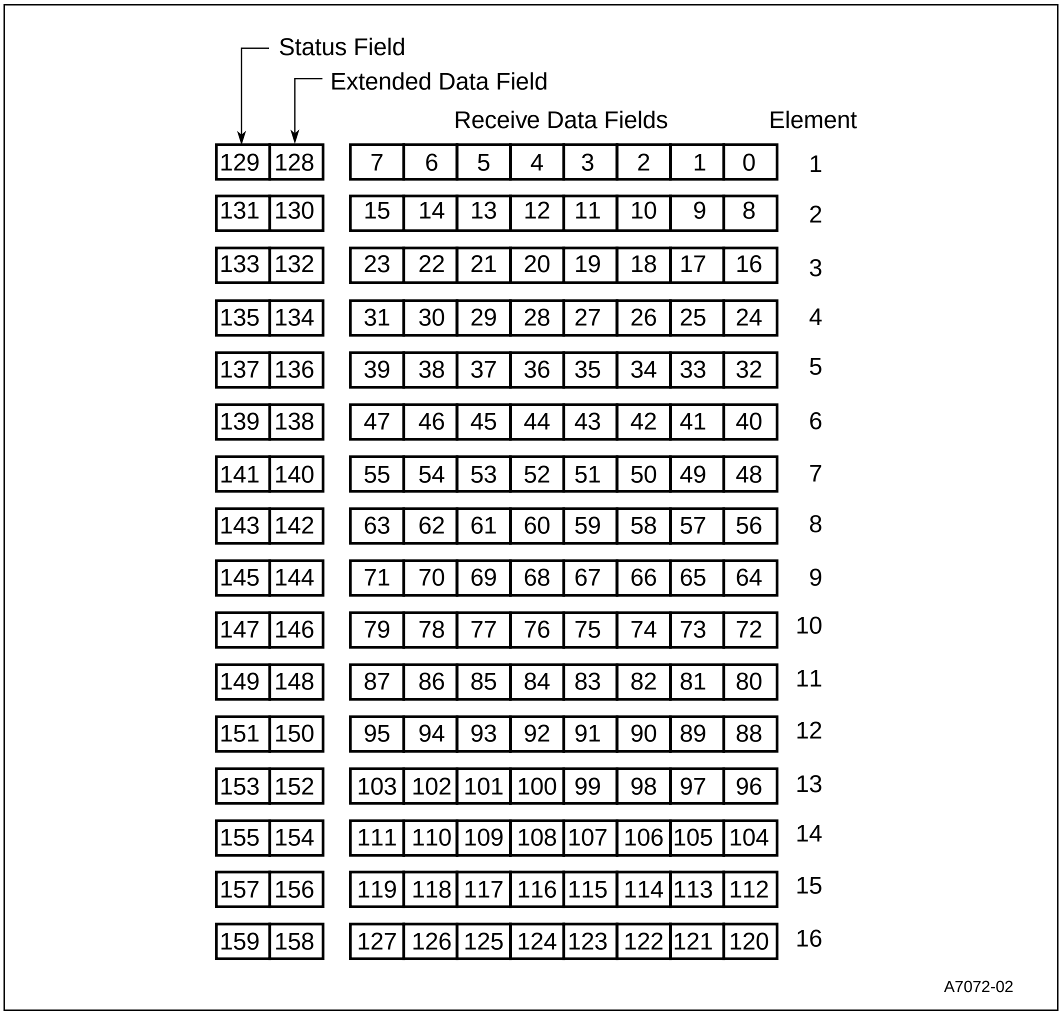
Valid Bytes	Big Endian Mode	Little Endian Mode
1	0x7	0xE
2	0x3	0xC
3	0x1	0x8
4	0x0	0x0

6.6.4.6 Reading the RFIFO

The Receive FIFO (RFIFO) is implemented as a memory array that can be written on quadword boundaries. Each element of the RFIFO consists of ten quadwords and contains both Receive Data and Extended Data/Status fields.

The RFIFO is always read on quadword boundaries and can be read by the Microengines into the SRAM Transfer Registers using the **r_fifo_rd** instruction. Or, the Microengines can instruct the SDRAM Unit to move data from the RFIFO into SDRAM using the **sdram** instruction. Figure 6-26 shows the quadword addresses for the RFIFO.

Figure 6-26. Quadword Addresses for RFIFO



Status Field	Extended Data Field	Receive Data Fields								Element
129	128	7	6	5	4	3	2	1	0	1
131	130	15	14	13	12	11	10	9	8	2
133	132	23	22	21	20	19	18	17	16	3
135	134	31	30	29	28	27	26	25	24	4
137	136	39	38	37	36	35	34	33	32	5
139	138	47	46	45	44	43	42	41	40	6
141	140	55	54	53	52	51	50	49	48	7
143	142	63	62	61	60	59	58	57	56	8
145	144	71	70	69	68	67	66	65	64	9
147	146	79	78	77	76	75	74	73	72	10
149	148	87	86	85	84	83	82	81	80	11
151	150	95	94	93	92	91	90	89	88	12
153	152	103	102	101	100	99	98	97	96	13
155	154	111	110	109	108	107	106	105	104	14
157	156	119	118	117	116	115	114	113	112	15
159	158	127	126	125	124	123	122	121	120	16

- RFIFO Data Field

  The data field contains the data that is read from the IX Bus by the Receive State Machine. The number of valid bytes in the data field is reported in the RCV_CNTL register.

- RFIFO Extended Data and Receive Status Field

  A Receive Request can instruct the Receive State Machine to gather nine quadwords of data - one more than the data element can hold. It can also be programmed via the

RDYBUS_TEMPLATE_CTL register to gather one or two quadwords of reception status information after an EOP is detected. In both cases, the data is placed into the Extended Data/ Status field associated with the data.

The extended data option is supported in all modes except in 32-bit unidirectional IX Bus mode when 64-bit status is specified. The data placed into the extended data/status field is presented in two formats based on the mode. See Figure 6-27 and Figure 6-28.

Figure 6-27. Format For 32-bit Unidirectional IX Bus Mode with 64-bit Status

Status

63 32	31 0
RESERVED	STATUS (LONGWORD 2)

Extended Data

63 32	31 0
RESERVED	STATUS (LONGWORD 1)

Figure 6-28. Format for All Other Modes

Status

63 32	31 0
STATUS (LONGWORD 2)	STATUS (LONGWORD 1)

Extended Data

63 32	31 0
EXTENDED DATA (LONGWORD 2)	EXTENDED DATA (LONGWORD 1)

The extended data option is enabled on a per-Receive-Request basis by setting the FA bit in the RCV_REQ register. The Microengine threads that read the RFIFO know extended data is available based on the number of quadwords reported in the RCV_CNTL register.

Some Intel IX Bus MAC devices provide status on the IX Bus on the next read after an EOP is asserted. The Receive State Machine can be enabled to automatically read this status and place it into the RFIFO status field when it detects an EOP. For 64-bit bidirectional IX Bus mode, one quadword is always read and placed into this field. For 32-bit unidirectional IX Bus mode, the Receive Request can specify whether the status length is one longword or one quadword. The format of the status is dependent on the MAC device. The only thing the IXP1200 does with the status is to copy bit 8 to the Receive OK bit in the RCV_CNTL register. This eliminates the need to read the status field from the RFIFO.

The Intel 21440 Multiport 10/100 Mbps Ethernet Controller is an example of a device that provides 32 bits of status on the IX Bus. Figure 6-29 shows its status format.

Figure 6-29. Intel 21440 Multiport 10/100 Mbps Ethernet Controller Status Format

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LEN																RES					MLT	BRD	ROK	FLW	RES	MER	RTL	RNT	DRB	CRC	OVF

Bits	Field	Description
31:16	LEN	Packet length
15:11	RES	RESERVED
10	MLT	Multicast packet
9	BRD	Broadcast packet
8	ROK	Receive OK
7	FLW	Flow-control packet
6	RES	RESERVED
5	MER	MII error
4	RTL	Too long packet
3	RNT	Runt packet
2	DRB	Alignment error
1	CRC	CRC error
0	OVF	Receive FIFO overflow (if set, LEN field is not valid)

6.6.4.7 Receive Scheduler Thread

In general, it is preferable to have a single Microengine thread issue Receive Requests than to have them issued by multiple Microengine threads. If a single thread issues Receive Requests, the following advantages are achieved:

- It is easier to manage RCV_REQ FIFO overflows.
- It is easier to maintain which RFIFO elements are available.
- It is easier to determine which Microengine threads are available.

A single Microengine thread that issues Receive Requests is referred to as the Receive Scheduler thread. The microcode developer can implement the Receive Scheduler to meet specific application requirements, but, in general, the Receive Scheduler performs the following functions:

- Determines which MAC receive ports need to be serviced.
- Determines which Microengine thread to assign to a Receive Request.
- Determines which RFIFO elements to assign to a Receive Request.
- Issues Receive Requests to the FBI Receive State Machine.

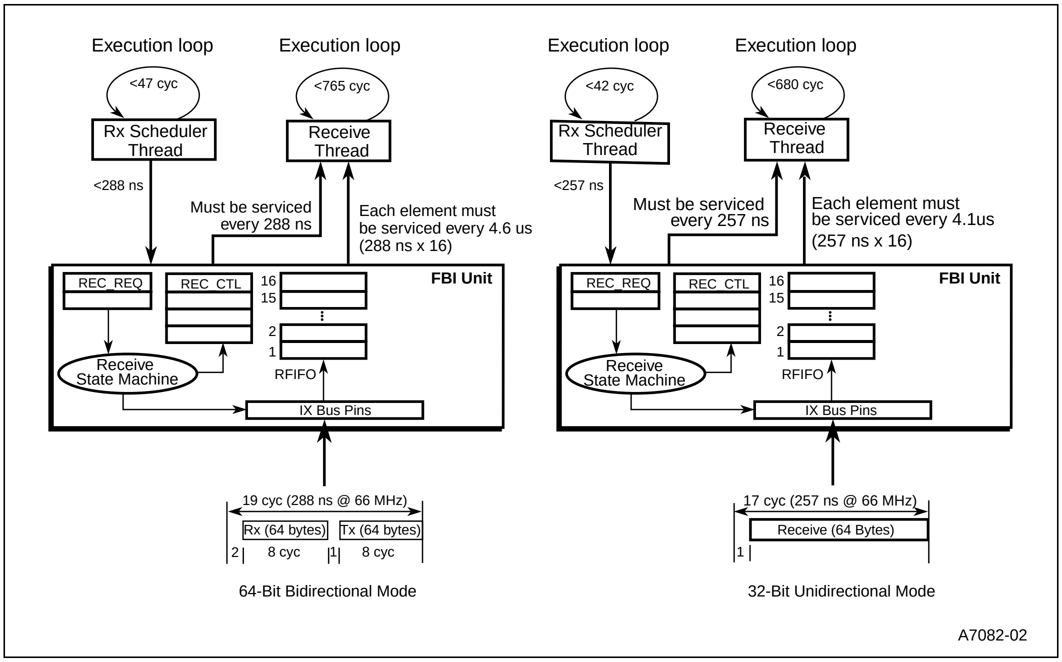
To maintain full utilization of the IX Bus, the Receive Scheduler must perform its tasks as quickly as possible to ensure that the Receive State Machine is always busy. The Receive Scheduler thread must have real-time execution properties and is typically the only thread running on a Microengine since it can not allow another thread to block its ability to perform its task. The tasks required of a Receive Scheduler thread include:

1. Determining which ports need to be serviced by reading the RCV_RDY_HI, RCV_RDY_LO, and RCV_RDY_CNT registers. The Receive Scheduler must also determine which Receive Ready flags are new and which are old using either the Receive Request Count or the Receive Ready Count in the RCV_RDY_CNT register.
2. Tracking the thread processing status of the other Microengine threads by reading the THD_DONE_REG0 and THD_DONE_REG1 registers.
3. Determining which RFIFO elements are not in use. Since it is the Receive Scheduler that assigns Receive threads to process the data in the RFIFO elements, and it also knows the thread processing status from the THREAD_DONE_REG0 and THREAD_DONE_REG1 registers, it can determine which RFIFO elements are currently available. This is because a Receive FIFO element assigned to a thread is not available as long as the thread is busy. If the thread has finished processing, the element is free to be used again.
4. Issuing Receive Requests to the IX Bus Receive State Machine by writing to the RCV_REQ register.

Timing Considerations When Issuing Receive Requests

To utilize the peak bandwidth of the IX Bus, the Receive Scheduler must issue Receive Requests to the Receive State Machine fast enough such that a Receive Request is always available for it to process. Figure 6-30 shows the rate at which Receive Requests must be issued in order to take advantage of the complete bandwidth of the IX Bus. The time period of the decision loop is in the range of 42 to 47 Core clock cycles.

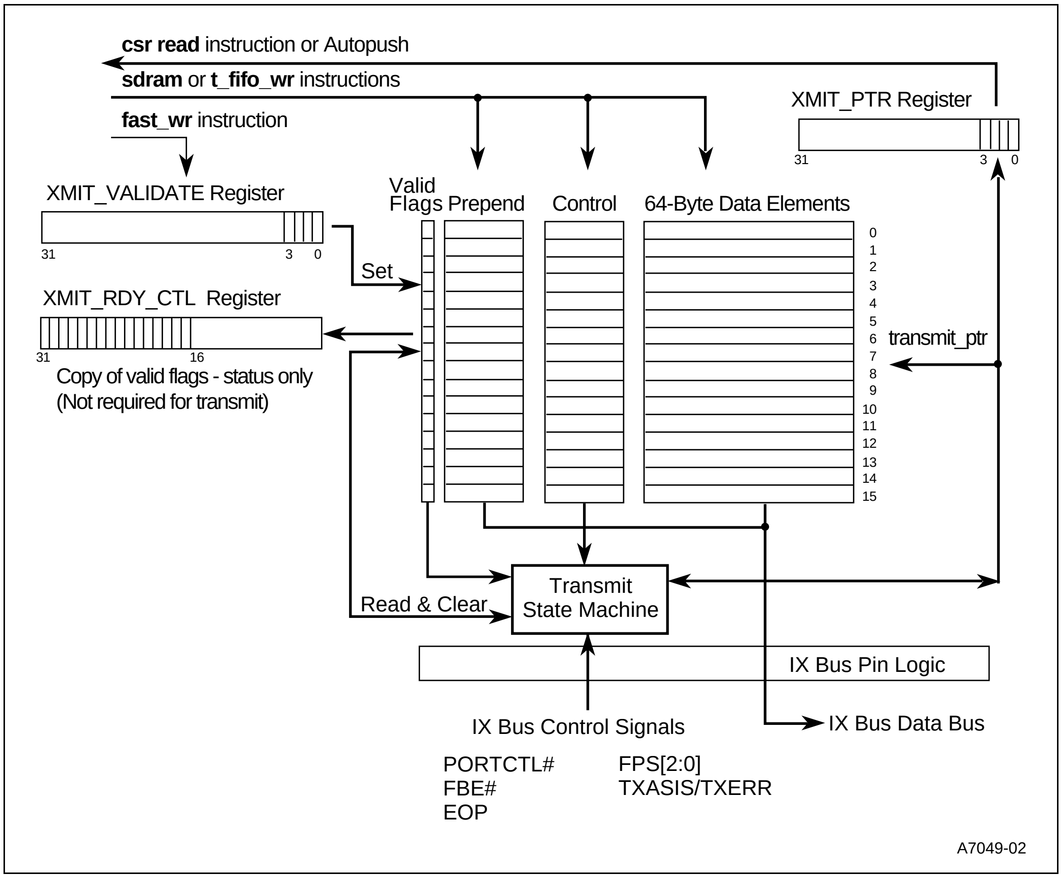
Figure 6-30. Receive Scheduler Decision Time



6.6.5 Transmit State Machine and TFIFO - Transmitting Data on the IX Bus

Microengines initiate transfers across the IX Bus while the Transmit State Machine actually performs the data transfers on the IX Bus. The Microengine interface to the Transmit State Machine is through the TFIFO elements and the XMIT_VALIDATE FBI register. The steps involved with IX Bus transmit transfers are listed below. These steps are described in more detail throughout this section.

Figure 6-31. IX Bus Transmit Transfer Steps



The steps involved with initiating an IX Bus transmit transfers are the following:

1. The Microengine must manage which TFIFO elements have been written with valid data and which elements the Transmit State Machine has transmitted. A Microengine reads the remove pointer from the XMIT_PTR register to determine which elements have been sent and updates the TFIFO element status maintained in microcode.
2. The Microengine writes the data to the data field of the next TFIFO element.
3. The Microengine writes the control information to the control field.
4. The Microengine sets the Valid flag for the element using the **fast_wr** instruction to the XMIT_VALIDATE register and updates the TFIFO element status maintained in microcode.

In the default mode, the Microengine should set the valid flag once the SDRAM reference is complete and the data is in the TFIFO data element. This ensures that the Transmit State Machine will not attempt to transmit that TFIFO element until the data is present.

In Dual Valid Bit mode, an additional flag is used for each TFIFO element. The first valid flag is set by the Microengine thread and the second is set by the SDRAM Unit once the memory reference is complete. In this case, the Microengine does not need to wait for the data to be in the TFIFO before setting its valid bit. Instead, it sets the first valid bit once the control information is written and then issues an **sdram[t_fifo_wr,...], sigdone** instruction. When the SDRAM Unit completes that data write to the TFIFO, it will signal the Microengine as well as set the second valid bit. (Note that the sig_done token must be used for this to occur.)

The steps involved with transmitting data onto the IX Bus are the following:

1. The Transmit State Machine checks the Valid flag(s) of the current element specified in XMIT_PTR register.
2. If the Valid flag(s) is set, the Transmit State Machine reads the Control field and transmits first the data in the Prepend field (if enabled by bit [16] of the Control field) and then transmits the data in the Data field. If the Valid flag(s) is cleared, the Transmit State Machine waits at that element until it is set.
3. At the beginning of the transfer of the TFIFO element across the IX Bus, the transmit pointer is incremented by one and the process is repeated.

6.6.5.1 Initiating a Transmit Request (TFIFO Format)

From a Microengine perspective, the Transmit FIFO (TFIFO) is implemented as a memory array that can be written on quadword boundaries. From the Transmit State Machine perspective, the elements are implemented as a circular buffer of 16 elements.

Each element of the TFIFO consists of ten quadwords and contains four fields: Transmit Data, Prepend, Control, and Valid.

- The Transmit Data field is a 64-byte TFIFO that holds the Transmit data.
- The Prepend field is 8 bytes and holds data that is sent before the Data field when the Prepend option is enabled in the Control field.
- The Control field is 8 bytes and instructs the Transmit State Machine on how to transmit the data.
- The Valid flag(s) indicates to the Transmit State Machine that the Data, Prepend, and Control fields are valid and the element can be transmitted over the IX Bus.

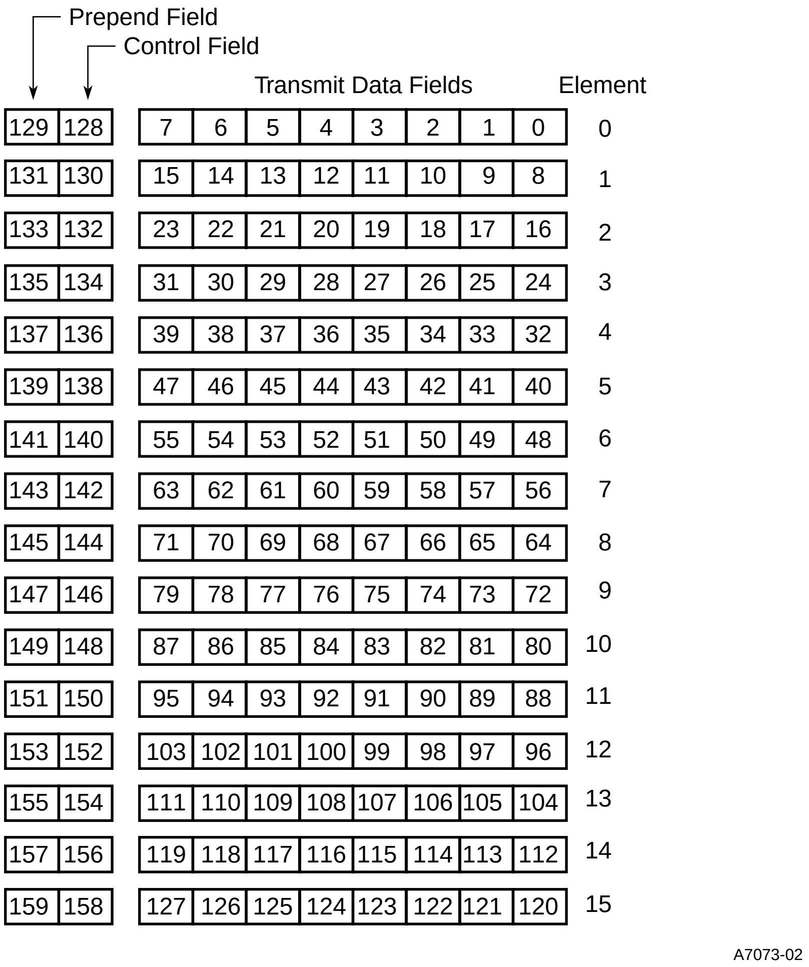
Data is written to the TFIFO on quadword boundaries. A Microengine writes data from its SRAM Transfer Register using the **t_fifo_wr** instruction. The **sdram** instruction can also be used to instruct the SDRAM Unit to move data directly from SDRAM into the TFIFO. A Microengine can write to the TFIFO elements in any order, however, the Transmit State Machine always services the elements contiguously. If the Transmit State Machine does not detect a Valid flag set for an element, it stops transmitting until the Valid flag for that element is set.

The Valid flag should only be set after the data, control, and prepend fields are written. A Microengine sets the Valid flag by writing to the XMIT_VALIDATE register using the **fast_wr** instruction. Since the **fast_wr** instruction requires immediate data to be specified when the microcode is assembled, the indirect reference is typically used to specify a 4-bit element number

(0-15). The FBI Unit reads the element number and automatically sets the Valid flag for the appropriate TFIFO element. The current state of the Valid flags for each of the elements can be read through the XMIT_RDYCTL register.

The format of the TFIFO elements as well as the quadword addresses are shown in Figure 6-32.

Figure 6-32. TFIFO Format and Quadword Word Address



Prepend Field	Control Field	Transmit Data Fields								Element
129	128	7	6	5	4	3	2	1	0	0
131	130	15	14	13	12	11	10	9	8	1
133	132	23	22	21	20	19	18	17	16	2
135	134	31	30	29	28	27	26	25	24	3
137	136	39	38	37	36	35	34	33	32	4
139	138	47	46	45	44	43	42	41	40	5
141	140	55	54	53	52	51	50	49	48	6
143	142	63	62	61	60	59	58	57	56	7
145	144	71	70	69	68	67	66	65	64	8
147	146	79	78	77	76	75	74	73	72	9
149	148	87	86	85	84	83	82	81	80	10
151	150	95	94	93	92	91	90	89	88	11
153	152	103	102	101	100	99	98	97	96	12
155	154	111	110	109	108	107	106	105	104	13
157	156	119	118	117	116	115	114	113	112	14
159	158	127	126	125	124	123	122	121	120	15

Transmit Data Field — The 64-byte transmit data fields contain the data that is sent onto the IX Bus. The data is placed on the bus beginning at the lower quadword address.

Control Field — The 8-byte control field instructs the Transmit State Machine how to transmit the data. The format of the control field is as follows:

63:19	18	17	16	15:13	12:10	9	8	7	6	5:3	2:0
RES	TXERR	TXASIS	PREPEND	VLD QWORDS	VLD BYTES	EOP	SOP	ERR	RES	MAC _NO	PORT _NO

TXERR
TX Error. If set, the TXASIS/TXERR pin is asserted during the last cycle of the data transfer.

TXASIS
TX As Is. If set, the TXASIS/TXERR pin is asserted during the first cycle of the data transfer.

PREPEND
If set, the data in the prepend field is transmitted on the IX Bus before the data field.

VLD QWORDS
Valid Quadwords. Specifies the number of valid quadwords of data in the TFIFO element. Valid quadwords are contiguous beginning at the first quadword in the TFIFO element. The number of valid quadwords transmitted equals VLD QWORDS + 1.

VLD BYTES
Valid Bytes. Specifies the number of valid bytes using the FBE# signals in the last valid quadword of data. The number of valid bytes transmitted equals VLD BYTES + 1.

EOP
End Of Packet. If set, the EOP pin is asserted during the last transmit data cycle that contains valid data.

SOP
Start Of Packet. If set, the SOP pin is asserted during the first transmit data cycle. Note that if the prepend option is enabled, SOP is asserted coincident with the prepend data and not the packet data.

ERR
Error. If set, the Transmit State Machine does not send this element and skips to the next element. The valid bit for the element must be set for the ERR bit to take effect.

MAC_NO
MAC Number. This field, along with the fact that this is associated with an IX Bus transmit, determines how the PORTCTL# signals are asserted.

PORT_NO
Port Number. Determines how the FPS signals are asserted during the IX Bus transfer.

Prepend Field
The 8-byte Prepend field is transmitted when the prepend bit in the control field is set. The pupose of the prepend data is defined by the user. It can, for example, be used as control information to a switching fabric.

6.6.5.2 Transmitting TFIFO Data

The Transmit State Machine checks the Valid flags for the TFIFO element associated with the transmit pointer. If the Valid flag is set, it reads the control information from the control field and transmits the data in the Data field and (if enabled) the data in the Prepend field. After the data is transmitted, the Valid flag is cleared, the remove pointer is incremented and the Transmit State Machine looks at the next Valid flag. The Transmit State Machine services the TFIFO elements sequentially, wrapping around to element 0 after it services element 15. If the remove pointer points to an element that does not have a Valid flag set, it stalls the Transmit State Machine which waits for the flag to be set.

6.6.6 Transmit Scheduler Thread

The tasks involved with determining when and how to issue transmit requests indicate that it would be best to have a single Microengine thread perform these functions. This thread is referred to as the Transmit Scheduler thread. The microcode developer can implement the Transmit Scheduler to meet specific application requirements, but fundamentally it performs the following functions:

- Manages which transmit queues or the MAC ports have data available to transmit.
- Manages which MAC transmit ports can accept data.
- Manages which TFIFO elements are currently being used and which are available.
- Manages which Microengine thread to assign to issue the transmit request.
- Assigns a Microengine thread to transmit data to a specified port using a specified TFIFO element.

The concept of a Transmit Scheduler thread is extended to the hardware from the point of view of the TxAutopush operation. The TxAutopush operation enables the XMIT_RDY_LO, XMIT_RDY_HI, and XMIT_PTR registers to be autopushed to a Microengine thread specified in the XMIT_RDY_CTL register. It is expected that this thread perform the Transmit Scheduler functions described above.

6.6.6.1 Assigning a Transmit Thread - Examples

Once the Transmit Scheduler decides which port to service and which Transmit thread to assign to that port, it issues a Transmit Request to a Transmit thread. The transmit algorithm is programmable. This section provides two examples.

Example 6-9. Assignment Using Inter-Thread Communications

This example illustrates the assignment of a Transmit Request to a Transmit thread using inter-thread communications. Inter-thread communications may be implemented by allocating memory as mailboxes and having the Transmit Scheduler write the Transmit Requests to a mailbox. Once the request is written to the mailbox, the Transmit Scheduler signals the assigned Transmit thread by writing to the Transmit Thread ID to the INTER_THD_SIG register. This generates a signal event to the specified Microengine thread. The Transmit thread responds to the Interthread signal event by reading the mailbox and processing the Transmit Request.

Example 6-10. Assignment Using Messages

This example takes a different approach. It is based on the premise that all the Transmit threads reside on a single Microengine. The idea is for the Transmit Scheduler to issue Transmit Requests in the form of a message that is written to a circular buffer kept within Scratchpad memory. The Transmit Scheduler maintains an insert pointer into the circular buffer and the Transmit threads maintain a remove pointer. A typical message format is shown below.

31	30 16	15 12	11 8	7 0
VLD	Unused	Element count	Element number	Transmit Queue

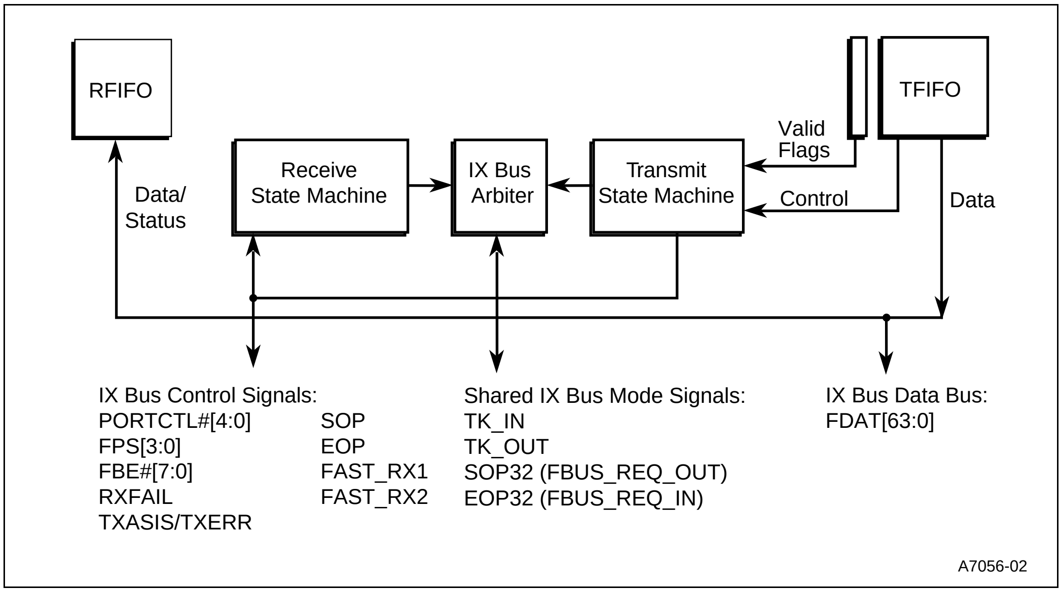
VLD	Indicates whether the message is valid (1 = message is valid)
Element count	Indicates number of TFIFO elements assigned (0 = no assignment)
Element number	Identifies the TFIFO element number
Transmit Queue	Identifies the transmit queue

The Transmit threads use absolute addressing to share a common general-purpose register (GPR) that contains the remove pointer. When each Transmit thread wakes up, it uses the remove pointer to read the task assignment message and checks the valid bit. If the message is valid, it increments the task assignment pointer in the global GPR and processes the request. The Transmit threads can calculate the packet location based on the port, queue, and the base address of the transmit descriptor queue base address. When the task is complete, the thread task is repeated.

6.6.7 IX Bus Arbiter

The IX Bus Arbiter is used only in 64-bit bidirectional mode. It selects whether the IX Bus Interface Logic is owned by either the Receive State Machine, Transmit State Machine, or, if enabled, another IXP1200 on a shared IX Bus. The shared IX Bus mode bit in the RDYBUS_TEMPLATE_CTL register determines whether the IXP1200 is configured for single IXP1200 mode or shared IX Bus mode.

Figure 6-33. IX Bus Arbiter Flow



For single IXP1200 systems, the IX Bus arbitration policy is to switch between granting ownership to either the Receive State Machine or the Transmit State Machine. When the Receive State Machine is granted ownership, it completes one Receive Request and then the arbitration moves to the Transmit State Machine. When the Transmit State Machine is granted ownership it may transmit one TFIFO element. If one of the state machines does not require access to the bus, the other state machine is granted another turn.

For shared IX Bus configurations, the IX Bus arbitration policy is to switch between granting ownership to the Receive State Machine, the Transmit State Machine, and another IXP1200. When the Receive State Machine is granted ownership, it may complete one Receive Request. When the Transmit State Machine is granted ownership, it transmits one TFIFO element. The IX Bus Arbiter then looks at the EOP32 pin (which has the alternative function of IX Bus Request Input) to determine if another IXP1200 is requesting ownership of the IX Bus. If so, the IX Bus Arbiter deasserts the TK_OUT pin, relinquishing IX Bus ownership. Otherwise IX Bus ownership is retained in the current IXP1200 for another set of transmits and receives. Whenever a Receive Request is available or the next TFIFO element is validated, the IX Bus Arbiter requests IX Bus ownership by asserting the SOP32 pin (which has the alternative function of IX Bus Request Output). The IX Bus Arbiter then waits for its TK_IN pin to be deasserted, indicating that IX Bus ownership has been granted.

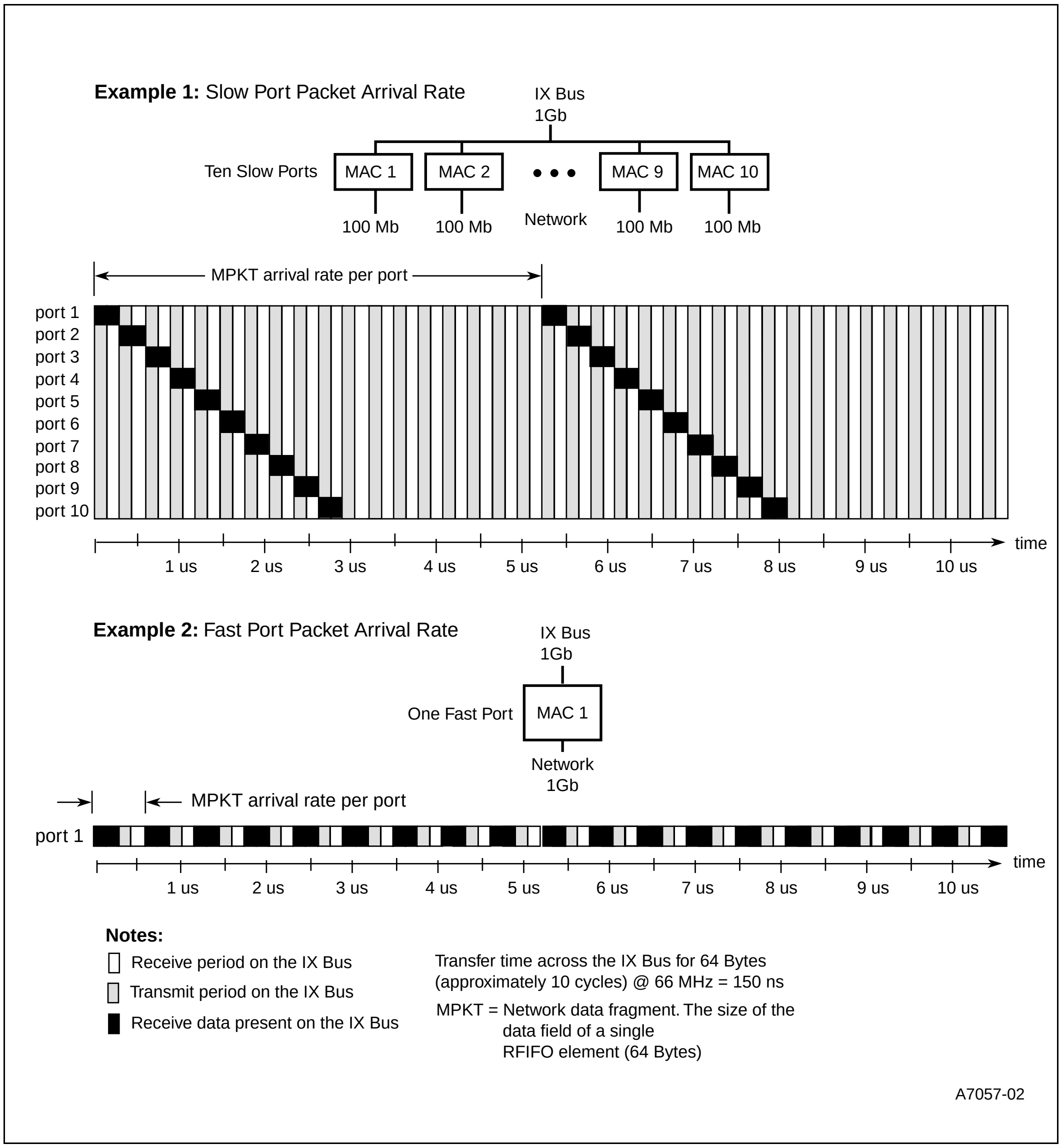
6.6.8 Slow Ports and Fast Ports

The concept of Slow Ports and Fast Ports is relevant only to how packets are received on the IX Bus. The type of port used is defined by software on a per-port basis every time a request is issued. One factor that could determine which type of port to use is bandwidth. Figure 6-34 illustrates the differences between Fast and Slow Ports. Both examples show a total IX Bus bandwidth of 1 Gbps for 64-bit bidirectional IX Bus mode. The first example shows ten 100 Mbps ports, while the second shows a single 1 Gbps port. Even though they require the same bandwidth, the data on the Fast Port must be read ten times as fast as the Slow Port. Because of this, the IXP1200 addresses two issues:

1. How to maintain the order in which packets are received from MAC ports.
2. How to issue Receive Requests such that a MAC port Receive FIFO does not overflow.

The following sections describe how this is done.

Figure 6-34. Slow Ports vs. Fast Ports


Example 1: Slow Port Packet Arrival Rate
IX Bus
1Gb
Ten Slow Ports
MAC 1
MAC 2
MAC 9
MAC 10
100 Mb
100 Mb
Network
100 Mb
100 Mb
MPKT arrival rate per port
port 1
port 2
port 3
port 4
port 5
port 6
port 7
port 8
port 9
port 10
time
1 us
2 us
3 us
4 us
5 us
6 us
7 us
8 us
9 us
10 us
Example 2: Fast Port Packet Arrival Rate
IX Bus
1Gb
One Fast Port
MAC 1
Network
1Gb
MPKT arrival rate per port
port 1
time
1 us
2 us
3 us
4 us
5 us
6 us
7 us
8 us
9 us
10 us
Notes:
Receive period on the IX Bus
Transmit period on the IX Bus
Receive data present on the IX Bus
Transfer time across the IX Bus for 64 Bytes
(approximately 10 cycles) @ 66 MHz = 150 ns
MPKT = Network data fragment. The size of the
data field of a single
RFIFO element (64 Bytes)
A7057-02

6.6.8.1 Maintaining Packet Order

Slow Ports and Fast Ports handle packet ordering differently due to the rate at which data arrives at the port and the rate at which a Microengine thread can process the data. The Receive State Machine reads packets from the MAC ports in fragments equal in size to one or two IXP1200 RFIFO elements. This document refers to the data in a single RFIFO element as a MAC Packet (MPKT). The amount of processing required for an MPKT may include header processing (header modification, forward lookup, etc.) or simply moving a packet body fragment to memory.

Port Blocking - Slow Ports

Port blocking maintains packet order ensuring that each MPKT is processed serially. In other words, each Receive Request is processed by a Microengine thread and placed into a buffer in memory before another Receive Request is issued to the same port. The following steps outline the tasks required for port blocking:

1. The Receive Scheduler thread determines which ports need to be serviced (via the Receive ready Flags) and which Microengine threads are available to process a Receive Request (via thread done or self destruct registers).
2. The Receive Scheduler thread maintains a record of thread to port assignments.
3. The Receive Scheduler thread issues a Receive Request and does not issue any more Receive Requests to that port until the Receive Request has completed (i.e. the port is blocked).
4. Receive State Machine processes the Receive Request and wakes the assigned Microengine thread
5. Assigned Microengine thread processes data, places data in memory, and signals the Receive Scheduler that the Receive Request is complete.
6. Repeat these steps for subsequent MPKTs.

Port Blocking requires the Receive Scheduler Thread maintain a record of which threads it assigns to which ports. After the Microengine threads assigned to the port completes the Receive Request, it must notify the Receive Scheduler Thread that the Receive Request is complete so that the port can be unblocked.

The IXP1200 provides two mechanisms that allow the assigned Microengine thread to indicate when the Receive Request has been completed: thread done and self destruct. Both methods require that the assigned Microengine thread signal the Receive Scheduler by writing a message to an FBI CSR (THD_DONE or SELF_DESTRUCT) and that the Receive Scheduler poll the CSR periodically to determine the status.

The THREAD_DONE and SELF_DESTRUCT registers are described in Section 6.5.6.

The SELF_DESTRUCT CSR support a one-bit message. The assigned Microengine Thread may set a bit within the SELF_DESTRUCT register (using the **fast_wr** instruction) to indicate that the Receive Request is complete. For example, Thread 12 may set bit 12 in the SELF_DESTRUCT register. When the Receive Scheduler reads the SELF_DESTRUCT register, the register value is returned and all the bits in the SELF_DESTRUCT register are automatically cleared. The Receive Scheduler can then look at the returned value to determine which threads have completed their assignments and then update its thread/port assignment list.

The THREAD_DONE CSRs supports a two-bit message for each Microengine thread. The assigned Microengine Thread may write a two-bit message to the THREAD_DONE register (using the **fast_wr** instruction) to indicate when it has completed its task. Each time a message is written to the THREAD_DONE register, the current message is logically ORed with the new message. The bit values in the THREAD_DONE registers are cleared by writing a “1” so the Scheduler thread may clear the messages in the THREAD_DONE register by writing the data read back to the THREAD_DONE register. An example of four message types is shown below.

Example 6-11. Four Message Types

Message	Meaning	Description
00	Thread is busy.	The thread is not available for another assignment.
01	Thread is idle - MPKT assignment complete.	The thread has completed its assigned Receive Request, but it did not see an EOP.
10	Thread is idle - MPKT assignment complete and the last packet contained an EOP.	The thread has completed its assigned Receive Request and an EOP was detected.
11	RFIFO Element is free.	The thread has removed all the data from the RFIFO so it can be assigned to another Receive Request. However, the Thread can still be busy working on a Receive Request.

The assigned threads write their status to the THREAD_DONE register whenever it changes. For example, a Microengine thread may immediately write 00 to THREAD_DONE after the Receive State Machine signals the assigned thread. When the Receive Scheduler reads the THREAD_DONE register, it can look at the returned value to determine the status of each thread and then update its thread/port assignment list.

Sequence Numbers - Fast Ports

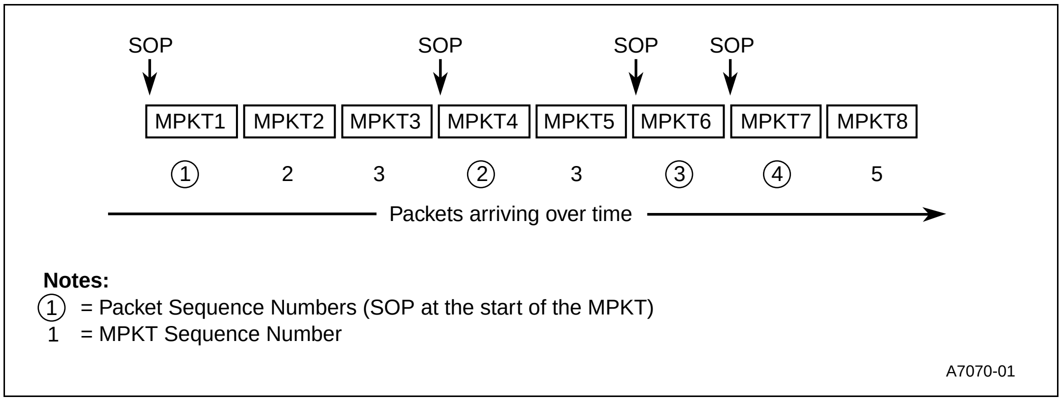
The packet rate of a Fast Port is such that the rate at which the Receive State Machine reads MPKTs from a single MAC port is so fast that a Microengine thread may NOT be able to process an MPKT before the Receive State Machine brings in another MPKT from the same port. A Fast Port assumes that the data arriving at a single MAC port is placed into multiple RFIFO elements and multiple Microengine threads are assigned to process the data in the RFIFO elements. For example, a single Fast Port may require eight RFIFO elements and six Microengine threads to maintain full line rate.

Fast Ports preclude the use of port blocking since MPKTs are processed in parallel and by different threads. The IXP1200 uses sequence numbers to ensure packet ordering for Fast Ports. Two sets of sequence numbers are supported, one per Fast Port. Each set provides a packet sequence number, an MPKT sequence number, and an enqueue sequence number. These sequence numbers are maintained as 4-bit counters within the FBI Unit and both automatically roll over to zero after a count of fifteen. The sequence numbers can only be incremented and can not be assigned any specific value.

A packet sequence number is incremented by the Receive State Machine once per Fast Port Receive Request and placed into the RCV_CNTL register. A copy of the sequence number field is also maintained in the SOP_SEQ1 (for Fast Port 1) or SOP_SEQ2 (for Fast Port 2) register. The Receive State Machine increments the packet sequence numbers in a manner that allows the microcode to track not only the sequence of packets, but also the sequence of the individual MPKTs. If the SOP signal is detected during a Receive Request, the packet sequence number provides a sequence number based on a packet (hereafter referred to as an SOP sequence number). If the SOP signal is not detected during a Receive Request, the packet sequence number is based on an MPKT packet (hereafter referred to as an MPKT sequence number). The Receive threads can determine the type of packet sequence number since the RCV_CNTL register contains both the packet sequence number and SOP status.

Both packet sequence numbers are implemented as two separate 4-bit counters. An SOP sequence number counter is incremented each time an SOP is detected. An MPKT sequence number counter inherits the SOP sequence number whenever the SOP signal is asserted, and is incremented once per Receive Request when the SOP signal is not detected. The figure below shows an example of how the sequence numbers are incremented.

Figure 6-35. Sequence Number Assignment for One Fast Port



Notes:
① = Packet Sequence Numbers (SOP at the start of the MPKT)
1 = MPKT Sequence Number

The enqueue sequence numbers are used by the Receive threads to determine whether it is their turn to place a packet onto a transmit queue. When an entire network packet has been received, the Receive thread reads the enqueue sequence number from the appropriate ENQUEUE_SEQ register. If the enqueue sequence number matches the SOP sequence number assigned to the packet, the Receive thread can place the packet onto a transmit queue. If the enqueue sequence number does not match, the Receive thread goes to sleep and waits for a “sequence number change” signal event to occur. When the event occurs, the Microengine thread reads the enqueue sequence number again and checks for a match. If a match occurs, the packet may be placed onto a transmit queue.

After a packet is placed on a transmit queue, the Receive thread increments the enqueue sequence number. The enqueue sequence numbers are incremented by writing to either the ENQUEUE_SEQ1 or ENQUEUE_SEQ2 register using the **fast_wr[incr_enq_num1]** or **fast_wr[incr_enq_num2]** instruction, respectively. A Microengine thread may choose to write its processing status to the THREAD_DONE register as well as incrementing the enqueue sequence number. This is accomplished with a single **fast_wr** instruction to the THREAD_DONE_INCR1 register or the THREAD_DONE_INCR2 register.

6.6.8.2 Issuing Receive Requests

A Microengine thread must manage the rate at which it issues Receive Requests to ensure that it does not issue more Receive Requests to a port than is required, but that it also issues enough Receive Requests so that the MAC port Receive FIFO does not overflow.

Slow Ports - Guaranteed Availability

When using Slow Ports, a Microengine thread reads the MAC receive FIFO Ready Flags for multiple ports, determines which ports have data available, and always issues Receive Requests based on the knowledge that data is available in the MAC Receive FIFO. Since it reads multiple Receive FIFO Ready flags each time, it can issue multiple Receive Requests before it has to read the flags again.

The fundamental rule for Slow Ports is that each Receive Request is issued based on the knowledge that there is data waiting in the MAC receive FIFO. In other words, the Receive State Machine is provided a guarantee that data is available at the MAC device when it starts processing the Receive Request.

Fast Ports - Speculative Requests

Since Fast Ports operate at a much higher data rate than Slow Ports, the rate at which a single MAC port is required to be serviced precludes the concept of issuing Receive Requests only when data is known to be available in a MAC port Receive FIFO. The reason guaranteed availability can not be supported for Fast Ports is because the latencies associated with the following tasks may be greater than that packet arrival rate:

- Reading the Receive FIFO Ready flags from a MAC device.
- Reading the Receive FIFO Ready flags into a Microengine thread.
- Microengine thread making a decision on whether to issue a Receive Request.
- Writing the Receive Request to the FBI Unit.
- Receive State Machine Processing the Receive Request.

Fast Ports support these higher data rates by supporting speculative requests. Speculative requests allow a Microengine thread to issue multiple Receive Requests to a port based on the speculation that there is data available in the MAC receive FIFO. At the time the Receive State Machine processes each Receive Request, the Receive State Machine determines if data is available at the MAC port and either processes the request or it is canceled.

The Receive State Machine determines whether there is data available at either of the two Fast Ports by reading the IXP1200s fast receive ready pins (FAST_RX1 and FAST_RX2). These pins provide a direct connection to the MAC ports Receive FIFO Ready flags. The MAC ports should assert these signals when the receive FIFO threshold level is reached. The Receive State Machine will sample the fast ready pins immediately before processing a Receive Request (RCV_REQ) from a Fast Port. Therefore, the fast ready pins need to be valid at that point in time.

If a fast ready pin is not asserted, the Receive State Machine cancels a request and writes a cancel message (binary 11) into the RCV_CNTL register’s message field. It then signals the assigned Receive thread. The Receive thread should be programmed to read the RCV_CNTL register, interpret the cancel message correctly and indicate to the Receive Scheduler thread that it is available for other tasks.

The state of the two fast ready pins is reflected in the RCV_RDY_CNT register. A Receive Scheduler can read this register to determine whether it should issue a Receive Request to a Fast Port. There are two methods for issuing Speculative Requests: Blind Speculative Requests and Smart Speculative Requests.

The IXP1200 supports two Fast Ports. The following is required to support Fast Ports.

a. The MAC port must support a fast ready pin, and it must be tied to one of the IXP1200s fast ready pins (FAST_RX1 and FAST_RX2).

b. One of three Fast Port modes must be selected in the RCV_RDY_CTL register.

c. The Receive Request must have the Fast Port bits set to indicate to the Receive State Machine that the port is a Fast Port and which sequence number is assigned. Sequence numbers are described in Figure 6.6.8.1.

Blind Speculative Requests

A Receive Scheduler can blindly issue Receive Requests to the Receive State Machine without knowledge of whether there is data available. Because of the time it takes for the Receive State Machine to cancel a request, Blind Speculative Requests affect IX Bus performance by introducing a three IX Bus cycle penalty for each cancelled request.

Smart Speculative Request

Smart Speculative Requests have the advantage of incurring less cancel penalties than the blind Speculative Requests. For Smart Speculative Requests, the Receive Scheduler thread reads the fast Ready flags from the RCV_RDY_CNT register on a periodic basis to determine when it should issue Receive Requests. Then it issues enough Receive Requests to cover the data that might have arrived in the MAC port since the last time it read the Fast Ready flags.

6.6.8.3 Fast Port Modes

The Receive State Machine supports three Fast Port modes that determine how Receive threads are assigned to process packet data in the RFIFO. A packet is defined as beginning with the assertion of SOP and ending with the assertion of EOP. These Fast Port modes are referred as Single Thread, Header/Body Thread and Explicit Thread modes. When selecting modes, consider the following:

— The Receive Scheduler thread needs to know which threads are available to assign threads to process each Receive Request.

— Order must be maintained by using a SOP sequence number. An MPKT sequence number may also be needed depending on the mode selected.

— The execution time for the Receive thread is variable. For example, the processing of a header would take longer than the processing of subsequent data pieces.

Single Thread Mode

The single thread mode assigns a single thread to each packet when using Speculative Requests.

The Receive State Machine supports the single thread assignments in the following manner. When the Receive State Machine detects an SOP, it signals the thread specified in the RCV_REQ register and saves the thread number in the header field of the REC_FASTPORT_CTL register. If it does not detect an SOP, the Receive State Machine ignores the thread ID presented in the RCV_REQ register and signals the thread specified in the REC_FASTPORT_CTL register. The Receive State Machine writes the unused thread ID to the RCV_CNTL register MACPORT/THD field. The unused thread ID must be returned to the Receive Scheduler thread so it knows that the thread as available for processing. There are several methods for returning the unused thread ID. Here are three possible methods:

— The Receive State Machine signals the Receive thread when the Receive Request is complete and the Receive thread passes the unused thread ID to the Receive Scheduler thread using inter-thread communications.

— The Receive Scheduler can request that it be signaled as well as the Receive thread after the Receive State Machine completes the Receive Request. In this case RCV_CNTL must be read twice before data is removed from the 4-entry RCV_REQ FIFO. In most cases, the Receive thread reads it once and the Receive Scheduler thread also reads it once. If two reads are not performed to the RCV_CNTL FIFO, it becomes blocked and the Receive State Machine stalls.

— The Receive State Machine signals the Receive thread when the Receive Request is complete, and the Receive thread sets the bit corresponding to the unused thread ID in the SELF_DESTRUCT register. The Receive Scheduler thread periodically reads the self destruct register to determine which threads are available.

Header/Body Threads Mode

The Header/Body thread mode assigns two threads per packet when using Speculative Requests. The idea behind the Header/Body threads is as follows. The first thread is the

Header thread and it is responsible for processing the header to determine how to forward the packet. The second thread is the Body thread. It's responsible for moving the remainder of the packet to memory. When the Body threads completes its task, it can use inter-thread signaling to notify the header thread where the body of the packet is located. The header thread can then place the packet onto a transmit queue.

The Receive State Machine supports header and body threads in the following manner. When the Receive State Machine detects an SOP, it signals the thread specified in the RCV_REQ register and saves the thread number in the header field of the REC_FASTPORT_CTL register. When the Receive State Machine processes the next request, it signals the thread specified in the RCV_REQ register and saves the thread number in the body field of the REC_FASTPORT_CTL register.

From this point forward, the Receive State Machine ignores the thread ID presented in the RCV_REQ register and signals the body thread specified in the REC_FASTPORT_CTL register. The Receive State Machine writes the unused thread ID to the RCV_CNTL register's MACPORT/THD field. As with the single thread mode, the unused thread ID must be returned to the Receive Scheduler thread so it knows that the thread is available for processing.

Explicit Thread Mode

The Explicit thread mode is identical to the Slow Port mode in the way thread assignments are made. In other words, the Receive State Machine always uses the thread assignment in the Receive Request. In this mode, the MPKT sequence number is provided to ensure that MPKTs are queued in the correct order.

6.6.8.4 Timing Considerations for Back-to-Back Reads

The timing considerations outlined below are relative only when the Receive State Machine performs two consecutive requests to the same port. This occurs for Fast Port mode where Speculative Requests are issued to the same port and not for Slow Ports. As mentioned earlier, Receive Requests should only be issued to Slow Ports only when there is data available in the Receive FIFO of the MAC port. This implies that two consecutive requests to the same port never occurs. Unpredictable results occur if two consecutive Receive Requests are made in Slow Port mode.

- Waiting for EOP

  A 4-cycle delay on the IX Bus is introduced for back-to-back requests to the same port since the Receive State Machine must wait to determine if an EOP is present in the data for the current Receive. If an EOP is present, it automatically reads the status for the port, if enabled, before it begins the next Receive Request.

- Programmable FP_READY_WAIT Delay

  The FP_READY_WAIT (Fast Port Ready Wait) FBI register specifies two 4-bit count values (one for each Fast Port) that determine how many IX Bus clock cycles the Receive State Machine waits between the time a Receive Request is complete, and the time the Receive State Machine samples the Fast Receive Ready pins (FAST_RX1 and FAST_RX2) to begin the next receive. A Receive Request is complete based on the following:

  — If MAC status is not enabled: After the data cycle where the EOP signal is asserted.

  — If MAC status is enabled: When status is requested.

  — If Rxfail is asserted: After the data cycle when the Rxfail signal is asserted.

  The FP_READY_WAIT delay is provided for the flexibility to support MAC devices that provide the fast ready flag at different times. This delay is programmable and has a reset value of 6 IX Bus clock cycles.

6.7 FBI Error Specifications

The sections that follow summarize the FBI states following a cancel , a receive fail, or a receive error.

6.7.1 Cancel

A cancel occurs when a Fast Port speculative request is cancelled.

Table 6-27. Cancel States

State	Value
rec_ctl<msg>	3
rec_ctl<macport/thd>	rec_req<recport> (note - no thread stuffing)
rec_ctl<sopseq#>	Undefined
rec_ctl<rxfail>	0
rec_ctl<erroroccur>	0
rec_ctl<elem2>	rec_req<elem2>
rec_ctl<elem1>	rec_req<elem1>
rec_ctl<elems_filled>	Undefined
rec_ctl<seqnum>	Undefined
rec_ctl<validbytes>	Undefined
rec_ctl<eop>	Undefined
rec_ctl<sop>	0
thread signaled	rec_req<thread_id> (Note - no header/body, etc.)
sched signaled	As specified by rcv_rdy_ctl<5> (sig_sched)
SOP seq num	Not incremented
collection count	Increment

6.7.2 Receive Fail

A receive fail occurs when the RXFAIL pin is asserted.

Table 6-28. Receive Fail States

State	Value
rec_ctl<msg>	rec_req<msg>
rec_ctl<macport/thd>	Based on rcv_rdy_ctl<14:13>: 00 (many_threads_mode) - rec_req<recport> 01 (2_threads_mode) - rec_req<recport> 10 (1_thread_mode) - rec_req<recport> if rec_ctl<sop> rec_req<thread_id> if !rec_ctl<sop> (thread_stuffing)
rec_ctl<sopseq#>	If rec_ctl<sop>, then SOP seq num
rec_ctl<rxfail>	1
rec_ctl<erroroccur>	0
rec_ctl<elem2>	rec_req<elem2>
rec_ctl<elem1>	rec_req<elem1>
rec_ctl<elems_filled>	Undefined
rec_ctl<seqnum>	rec_req<17> (Fast Port sel)
rec_ctl<validbytes>	Undefined
rec_ctl<eop>	IX Bus<eop>
rec_ctl<sop>	IX Bus<sop>
thread signaled	Based on rcv_rdy_ctl<14:13>: 00(many_threads_mode) - rec_req<thread_id> 01(2_threads_mode) - header/body 10(1_thread_mode) - rec_req<thread_id> if rec_ctl<sop> header_thread if !rec_ctl<sop>
sched signaled	As specified by rcv_rdy_ctl<5> (sig_sched)
SOP seq num	Incremented if rec_ctl<sop> = 1
collection count	Increment

6.7.3 Receive Error

A receive error occurs when status is enabled and bit 8 of the status signals that an error has occurred.

Table 6-29. Receive Error States

State	Value
rec_ctl<thmsg>	rec_req<msg>
rec_ctl<macport/thd>	Based on rcv_rdy_ctl<14:13>: 00(many_threads_mode) - rec_req<recport> 01(2_threads_mode) - rec_req<recport> 10(1_thread_mode) - rec_req<recport> if rec_ctl<sop> rec_req<thread_id> if !rec_ctl<sop> (thread_stuffing)
rec_ctl<sopseq#>	If rec_ctl<sop>, then SOP seq num
rec_ctl<rxfail>	0
rec_ctl<erroroccur>	1
rec_ctl<elem2>	rec_req<elem2>
rec_ctl<elem1>	rec_req<elem1>
rec_ctl<elems_filled>	IX Bus
rec_ctl<seqnum>	rec_req<17> (fast port sel)
rec_ctl<validbytes>	IX Bus
rec_ctl<eop>	IX Bus<eop>
rec_ctl<sop>	IX Bus<sop>
thread signaled	Based on rcv_rdy_ctl<14:13>: 00(many_threads_mode) - rec_req<thread_id> 01(2_threads_mode) - header/body 10(1_thread_mode) - rec_req<thread_id> if rec_ctl<sop> header_thread if !rec_ctl<sop>
sched signaled	As specified by rcv_rdy_ctl<5> (sig_sched)
SOP seq num	Incremented if rec_ctl<sop> = 1
collection count	Increment

SDRAM Unit 7

7.1 Overview

The SDRAM Unit provides an interface to up to 256 Mbytes of synchronous DRAM (SDRAM), with excellent burst performance. The external SDRAM Bus operates at ½ the Core frequency and contains a 64-bit data bus, a 14-bit address bus, and control signals. An active memory optimization feature allows the SDRAM controller to obtain high memory performance from standard SDRAM devices by eliminating bank precharge latencies whenever possible.

Read and write operations to SDRAM can be performed by the Microengines, StrongARM* core, PCI bus masters (including I_2O accesses), and the PCI DMA channels. Configuration registers in the SDRAM Unit allow the user to set the timing characteristics of the SDRAM Bus.

Recommended (but not required) use of SDRAM would be to:

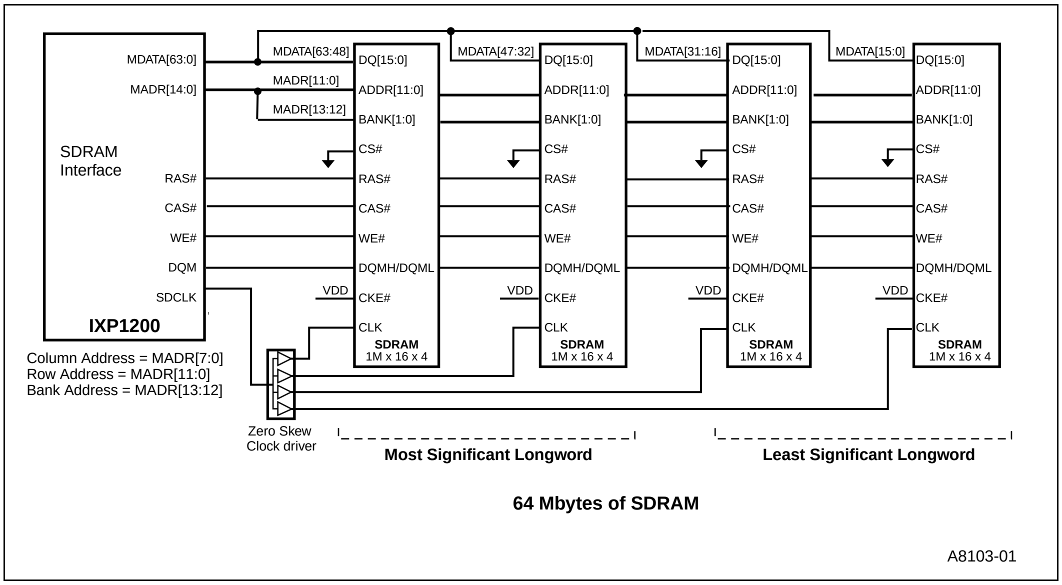
1. Store large data structures such as packet/cell/frame data and forwarding table information.
2. Hold StrongARM* core instruction code during runtime.

7.2 SDRAM Bus Configurations

Figure 7-1 shows typical connections between an IXP1200 and SDRAM. The StrongARM* core must be programmed to configure the SDRAM Unit with the SDRAM CSRs before the SDRAM devices can be accessed. This programmability allows the SDRAM Unit to support a wide variety of SDRAM devices. Refer to Section 4.5 in the *IXP1200 Network Processor Programmer's Reference* for a summary of the SDRAM CSRs.

The number of row address pins (from 11 to 13) and column address pins (from 8 to 10) are programmable through the use of the SDRAM_MEMCTL0 register. Two bank bits, supporting up to four banks, are provided and are automatically assigned by the SDRAM Unit to the address pins immediately following the row address bits. The SDRAM MDATA[63:0] pins are designed to drive a single load. The SDRAM clock is generated by the IXP1200 and an external clock driver is typically required to drive the SDRAM devices to minimize clock skew. Figure 7-1 shows an example of a 64 Mbyte SDRAM configuration using 1 Mbit x 16-bit x 4 bank devices.

Figure 7-1. SDRAM Interface External Connections



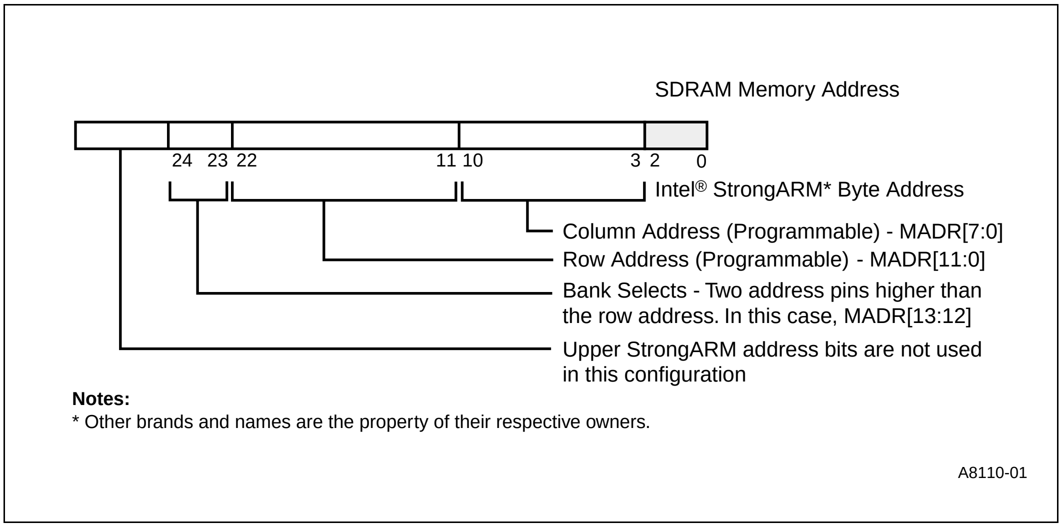
7.2.1 Bank, Row, and Column Pin Assignments

Bank, row, and column address pin assignments are based on an internal SDRAM address. The StrongARM* core generates a byte address while the Microengines generate a quadword address. The number of external address pins used for the row and column addresses are programmed through the SDRAM_CSR register. Table 7-1 shows the configurations supported by the SDRAM Unit and Figure 7-2 shows the relationship between the internal address and the external addresses for a 32 Mbyte SDRAM configuration that uses 1 Mbit x 16 bit x 4 bank devices (i.e., the configuration example illustrated in Figure 7-1).

Table 7-1. SDRAM Configurations

Total Memory	# of SDRAM Devices	Size of SDRAM Devices	Configuration (per bank)	Internal Banks	Bank Bits	RAS Bits	CAS Bits
8 Mbytes	4	16 Mbit	512 K x 16-bit	2	MADR[11]	MADR[10:0]	MADR[7:0]
16 Mbytes	8	16 Mbit	1 M x 8-bit	2	MADR[11]	MADR[10:0]	MADR[8:0]
32 Mbytes	4	64 Mbit	2 M x 16-bit	2	MADR[13]	MADR[12:0]	MADR[7:0]
64 Mbytes	8	64 Mbit	4 M x 8-bit	2	MADR[13]	MADR[12:0]	MADR[8:0]
32 Mbytes	4	64 Mbit	1 M x 16-bit	4	MADR[13:12]	MADR[11:0]	MADR[7:0]
64 Mbytes	8	64 Mbit	2 M x 8-bit	4	MADR[13:12]	MADR[11:0]	MADR[8:0]
64 Mbytes	4	128 Mbit	2 M x 16-bit	4	MADR[13:12]	MADR[11:0]	MADR[8:0]
128 Mbytes	8	128 Mbit	4 M x 8-bit	4	MADR[13:12]	MADR[11:0]	MADR[9:0]
128 Mbytes	4	256 Mbit	4 M x 16-bit	4	MADR[14:13]	MADR[12:0]	MADR[8:0]
256 Mbytes	8	256 Mbit	8 M x 8-bit	4	MADR[14:13]	MADR[12:0]	MADR[9:0]

Figure 7-2. SDRAM Addressing


SDRAM Memory Address
24 23 22
11 10
3 2
0
Intel® StrongARM* Byte Address
Column Address (Programmable) - MADR[7:0]
Row Address (Programmable) - MADR[11:0]
Bank Selects - Two address pins higher than the row address. In this case, MADR[13:12]
Upper StrongARM address bits are not used in this configuration
A8110-01

Notes:
* Other brands and names are the property of their respective owners.

7.2.2 Initializing the SDRAM Interface

Before accessing the SDRAM devices after a system reset, the StrongARM* core must initialize the SDRAM Unit and the SDRAM devices. The SDRAM Unit is configured by the StrongARM* core through the use of four SDRAM CSRs: SDRAM_CSR, SDRAM_MEMCTL0, SDRAM_MEMCTL1, and SDRAM_MEMINIT. The use of these registers allows the SDRAM Unit to support a wide variety of SDRAM devices. The parameters that are programmable are listed in Table 7-2.

Table 7-2. Programmable SDRAM Registers

Programmable Values	Description
Refresh Count	The rate at which SDRAM refresh cycles are generated.
BURST Length	The maximum number of SDRAM accesses between CAS cycles.
CAS Latency	Expressed as a number of SDRAM clock cycles.
Row Address Width	Specifies the number of address pins used for the row address.
Column Address Width	Specifies the number of address pins used for the column address.
tRWT	Read/Write Turnaround Time.
tDPL	Data In to Precharge Time.
tDQZ	DQM Data Out Disable Latency.
tRC	Bank Cycle Time.
tRRD	Bank-to-Bank Delay Time.
tRCD	RAS-to-CAS Delay.
tRASmin	Active Command Period.
tRP	Precharge Time.

Table 7-2. Programmable SDRAM Registers (Continued)

Programmable Values	Description
tRSC	A new command may be issued following the mode register set command once a delay equal to tRSC has elapsed. tRSC = (calculated time - 1), where the minimum number of clock cycles is calculated by dividing the minimum time specified by the SDRAM manufacturer by the SDRAM clock cycle time (1/2 the IXP1200 Core clock frequency) and then rounding off to the next higher integer.
INIT_RFRSH	Initialization Refresh. Specifies the number of times to issue a refresh during initialization.
INIT_DLY	Initialization Delay. Specifies the number of SDRAM cycles to wait after an INIT before accessing SDRAM.

The SDRAM initialization sequence begins by programming the SDRAM CSRs to specify the timing and configuration parameters for the specific SDRAM devices in the system.

Note: The last register to be programmed must be the SDRAM_CSR since it contains the INIT bit that causes the other register settings to be loaded to the SDRAM Controller. Setting the INIT bit also causes the SDRAM Unit to initialize the SDRAM devices by writing to their Mode Register.

Some SDRAM devices require that the SDRAM Unit wait for a given number of clock cycles after the power supply and SDRAM clocks have stabilized before they can be accessed for data. This timing delay value is programmable via the INIT_DLY parameter in the SDRAM_MEMINIT register. After the initialization delay has completed, the SDRAM Unit issues a Precharge All command, which puts all of the devices into the “all banks idle” state. This is the only time that the Precharge All command is used. All other precharge operations are performed exclusively on a bank-by-bank basis.

After the banks have been put into an idle state, the Load Mode Register command is executed. The Load Mode Register command programs the burst length, burst type, CAS latency, operating mode and write burst mode into the SDRAM devices.

After the Load Mode Register command is complete, the SDRAM devices require a series of Auto Refresh commands to be executed by the SDRAM Unit. The number of Auto Refresh commands performed is determined by the INIT_REFRSH field in the SDRAM_MEMINIT register. After the specified number of Auto Refresh commands have been executed, the SDRAM devices are ready to accept other commands.

During initialization, the Command Service Priority Logic holds off all accesses to SDRAM memory space. The arbiter is under control of this logic.

7.2.2.1 Configuration Registers (SDRAM_MEMCTL0)

The IXP1200 supports a sequential burst type and programmed write burst lengths. The burst length and CAS latency for the SDRAM device is selected by programming the appropriate values into the SDRAM_MEMCTL0 CSR register. Table 7-3 provides more details on sequential bursts.

Table 7-3. Sequential Bursts

Burst Length	Starting Column Address	Order of access within a burst (Sequential)	Burst Length	Starting Column Address	Order of access within a burst (Sequential)
2	**A0**		8	**A2 A1 A0**	
	0	0-1		0 0 0	0-1-2-3-4-5-6-7
	1	1-0		0 0 1	1-2-3-4-5-6-7-0
4	**A1 A0**			0 1 0	2-3-4-5-6-7-0-1
	0 0	0-1-2-3		0 1 1	3-4-5-6-7-0-1-2
	0 1	1-2-3-0		1 0 0	4-5-6-7-0-1-2-3
	1 0	2-3-1-0		1 0 1	5-6-7-0-1-2-3-4
	1 1	3-0-1-2		1 1 0	6-7-0-1-2-3-4-5
				1 1 1	7-0-1-2-3-4-5-6

Whenever a boundary of the block is reached within a given sequence above, the following access wraps within the block.

The SDRAM_MEMCTL0 burst length field (bits [15:12]) must be written to the value that corresponds to the desired burst size. Interleaved memory is not supported.

Memory reads and writes are done in bursts of one quadword (64 bits). Transfers can be from one quadword up to the programmed burst length. For a read that requires less than one quadword (e.g., a memory read from PCI), the PCI Unit discards the unused data. For a write that requires less than one quadword, the PCI uses the DQM (DQ Mask) pins to inhibit writing to some bytes. The DQM pins also inhibit writing to unoccupied bytes. The row/column multiplexer mode is the value programmed into the SDRAM address and size register (see Section 5.4.3 of the *IXP1200 Network Processor Datasheet*).

Descriptions of the SDRAM_CSR, SDRAM_MEMCTL0, SDRAM_MEMCTL1 and SDRAM_MEMINIT registers can be found in the *IXP1200 Network Processor Programmer's Reference*.

7.2.3 SDRAM Bus Commands

The IXP1200 connects to the SDRAM devices via two buses - a control bus and a data bus. The signals on these buses are sampled synchronously on the rising edge of a master controlled clock source (SDCLK) provided by the IXP1200.

Data returned from the SDRAM devices is sampled with a gated version of the Core clock. This signal is nearly synchronous to SDCLK. Because the two clocks are pseudo-synchronous, the zero delay clock buffer is a necessity.

The control bus consists of the RAS#, CAS#, WE# and DQM signals. In general, RAS# (Row Address Strobe) is used to indicate that the address being driven is a row address, CAS# (Column Address Strobe) indicates a column address, and WE# (Write Enable) indicates the direction (i.e., write/read) of the transfer request. The DQM signal masks a data write when asserted on a write access, and forces the SDRAM devices off the data (DQ) bus when asserted during a read access. Different combinations of the above signals are used to indicate other SDRAM commands such as Load Mode Register, Precharge, and Self Refresh.

When the control bus does not contain a valid command, it is driven with a NOP command, in which all of its pins are in the inactive state. The IXP1200 does not support Chip Select pins, so the commands are always valid. Table 7-4 lists the SDRAM commands and the state of the pins for each command.

Table 7-4. SDRAM Commands and Pin States

NAME (FUNCTION)	RAS#	CAS#	WE#	DQM	ADDR	DQ[15:0]
NO OPERATION (NOP)	H	H	H	X	X	X
ACTIVE (Select bank and activate row)	L	H	H	X	Bank/Row	X
READ (Select bank and column, and start Read burst)	H	L	H	L/H	Bank/Col	X
WRITE (Select bank and column, and start Write burst)	H	L	L	L/H	Bank/Col	Valid
BURST TERMINATE	H	H	L	X	X	Active
PRECHARGE (Deactivate row in bank or banks)	L	H	L	X	Code	X
AUTO REFRESH	L	L	H	H	X	X
LOAD MODE REGISTER	L	L	L	X	Opcode	X

Note: H= high, L= Low, X=Don’t Care

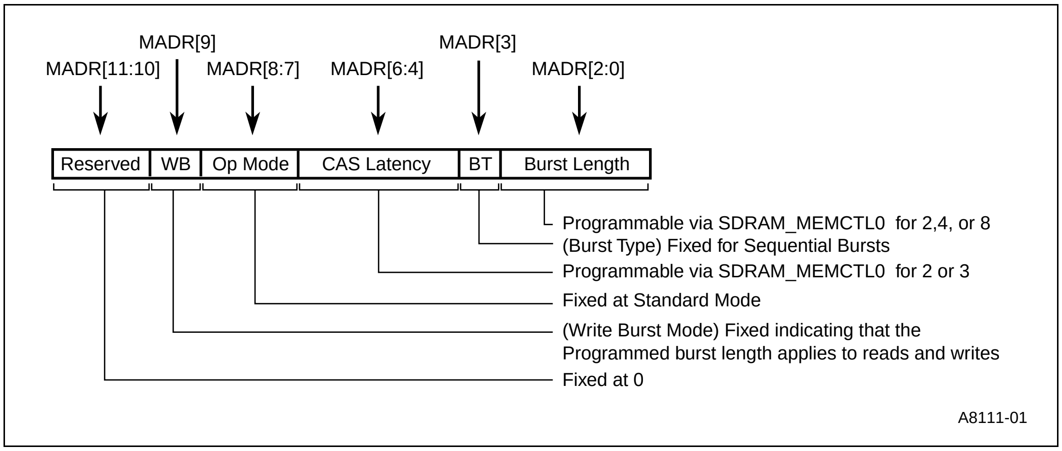
7.2.3.1 No Operation (NOP)

The No Operation (NOP) command is used to perform a NOP to an SDRAM device that is selected (CS# is LOW). This prevents unwanted commands from being registered during idle or wait states. Other commands already in progress within the SDRAM devices are not affected by a NOP.

7.2.3.2 Load Mode Register

The mode register data is loaded through the MADR[11:0] address pins. The Load Mode Register command is performed automatically by the SDRAM Unit when the StrongARM* core sets the INIT bit in the SDRAM_CSR. Figure 7-3 shows the format of the data on the address pins during a Load Mode Register command.

Figure 7-3. Load Mode Register Command


MADR[9]
MADR[3]
MADR[11:10]
MADR[8:7]
MADR[6:4]
MADR[2:0]
Reserved
WB
Op Mode
CAS Latency
BT
Burst Length
Programmable via SDRAM_MEMCTL0 for 2,4, or 8
(Burst Type) Fixed for Sequential Bursts
Programmable via SDRAM_MEMCTL0 for 2 or 3
Fixed at Standard Mode
(Write Burst Mode) Fixed indicating that the Programmed burst length applies to reads and writes
Fixed at 0
A8111-01

7.2.3.3 Active

The Active command is used to open (or activate) a row in a particular bank for a subsequent access. The value on the highest two address pins selects the bank, and the remaining address pins select the row. This row remains active (or open) for accesses until a Precharge command is issued to that bank.

A Precharge command is issued automatically by the SDRAM Unit before opening a different row in the same bank.

7.2.3.4 Read

The Read command is used to initiate a burst read access to an active row. The value on the highest two address pins select the bank. Depending on the number of column address bits that are programmed into the SDRAM_CSR register, the starting column location may be MADR[9:0], MADR[8:0], or MADR[7:0]. Read data appears on the DQ pins based on the logic level of the DQM pins two clocks earlier. If any DQM pin is registered high, its corresponding DQ will be in the high impedance state two clocks later. If the DQM pin is registered LOW, the DQ pins will provide valid data after two clocks.

7.2.3.5 Write

The Write command is used to initiate a burst write access to an active row. The value on the highest two address pins selects the bank. Depending on the number of column address bits that are programmed into the SDRAM_CSR register, the starting column location may be MADR[9:0], MADR[8:0], or MADR[7:0]. Data appearing on the DQ pins is written to the memory array based on the logic level of the DQM pins coincident with the data. If a given DQM signal is registered low, the corresponding data will be written to memory; if the DQM signal is registered high, the corresponding data inputs will be ignored, and a Write will not be executed to that byte/column location.

7.2.3.6 Burst Terminate

The Burst Terminate command is used to truncate fixed-length bursts. The Burst Terminate command is issued **X** cycles before the clock edge at which the last desired data element is valid, where **X** equals the programmable CAS latency minus one.

Note: CAS latency is specified by the SDRAM device manufacturer.

7.2.3.7 Self Refresh

The SDRAM Unit supports the Self Refresh function.

7.2.3.8 Precharge

The Precharge command is used to deactivate the open row in a particular bank or the open row in all banks. The bank(s) will be available for a subsequent row access a specified time (tRP) after the Precharge command is issued. The SDRAM Unit indicates via MADR[10] whether one or all banks are to be precharged, and in the case where only one bank is to be precharged, inputs MADR[13:12] select the bank. Once a bank has been precharged, it is in the idle state and must be activated prior to any Read or Write commands being issued to that bank.

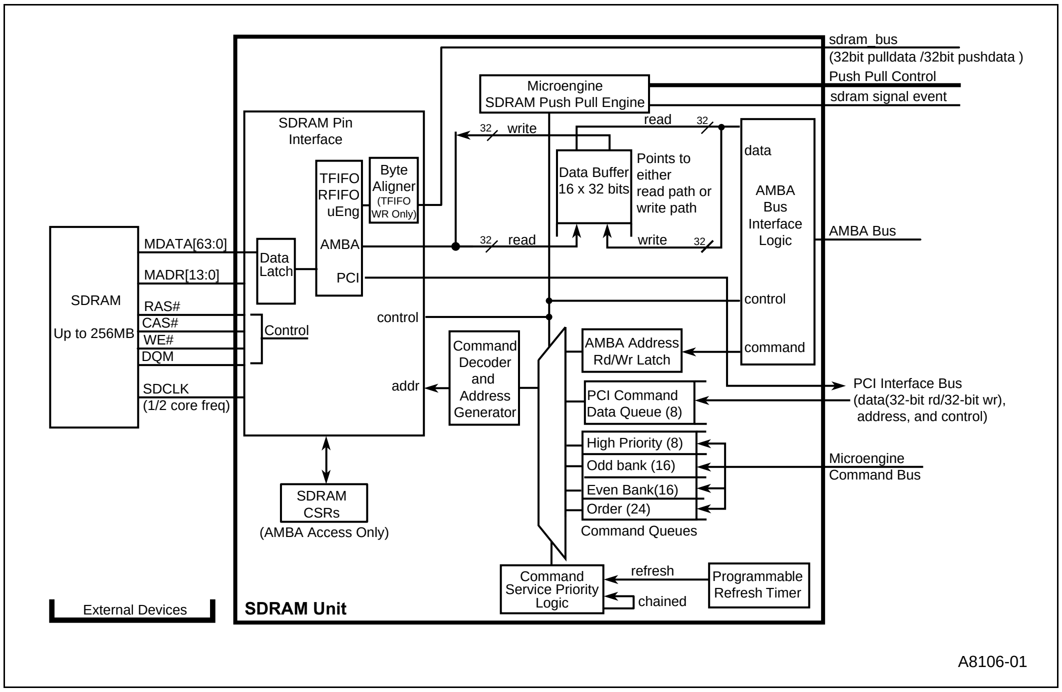
7.2.3.9 Unsupported SDRAM Commands

The Command Inhibit, Auto Precharge, and Auto Refresh SDRAM commands are not supported by the IXP1200 SDRAM Unit.

7.3 Interfacing to the SDRAM Unit

Figure 7-4 illustrates the internal components of the SDRAM Unit. This section describes how the StrongARM* core, PCI Unit, IX Bus Unit and the Microengines interface to the SDRAM Unit and its internal components.

Figure 7-4. SDRAM Interfacing



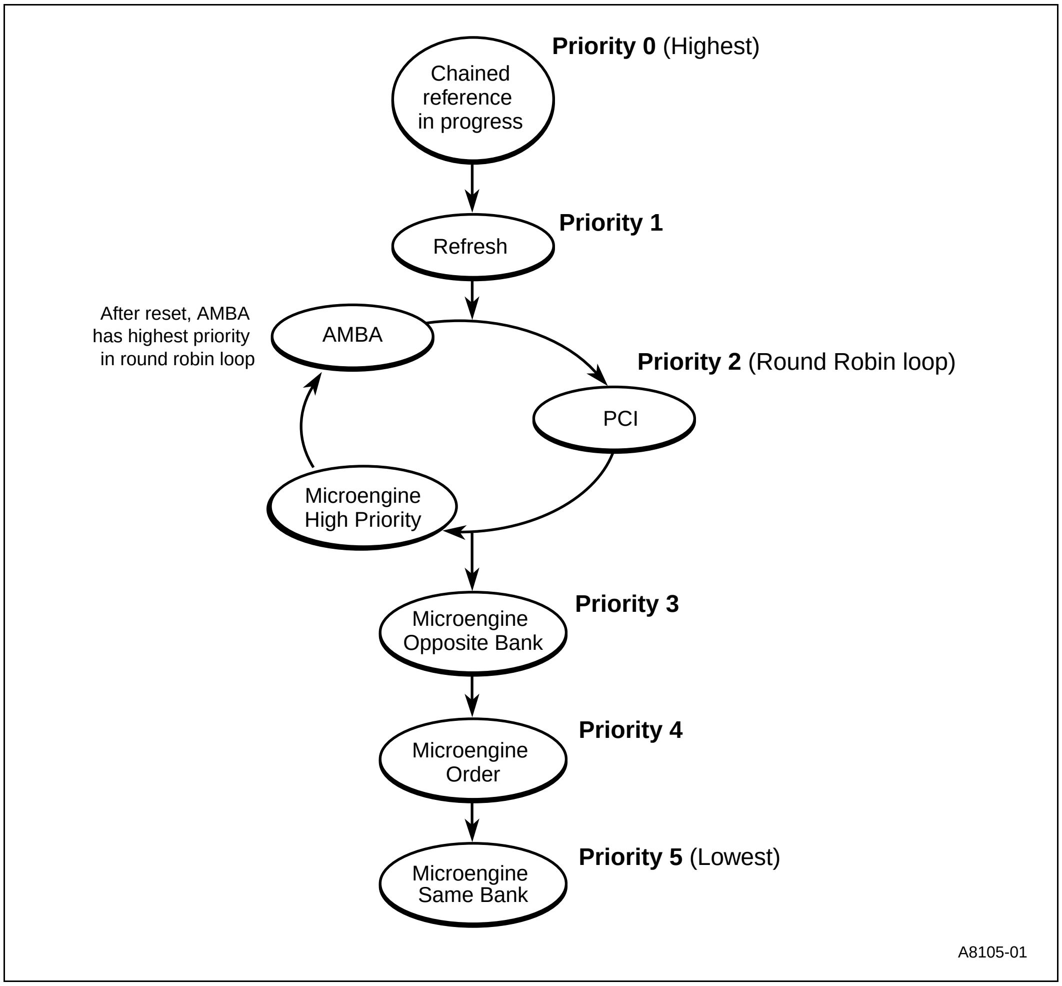
7.3.1 SDRAM Command Service Priority Logic

The SDRAM Unit handles memory references from six sources:

- StrongARM* core
- PCI Unit
- Four Microengine Command Queues: (Odd Bank, Even Bank, Order, and Priority)

The SDRAM Unit contains Command Service Priority Logic that determines when each of the references to memory from the sources gains access to the SDRAM Bus. The arbitration policy exists at two levels: major units (Microengines, StrongARM* core, PCI Unit, and SDRAM Refresh) and among the different types of Microengine commands. The arbitration policy used by the Command Service Priority Logic is shown in Figure 7-5.

Figure 7-5. Command Service Priority Logic Arbitration Policy


Priority 0 (Highest)
Chained reference in progress
Priority 1
Refresh
After reset, AMBA has highest priority in round robin loop
AMBA
Priority 2 (Round Robin loop)
PCI
Microengine High Priority
Priority 3
Microengine Opposite Bank
Priority 4
Microengine Order
Priority 5 (Lowest)
Microengine Same Bank
A8105-01

7.3.1.1 Priority 0: Chained Referenced

The highest priority is given to chained references that are currently in progress. A chained reference enters a command queue and is serviced at a priority level equal to the command queue in which it is placed. When the chained reference is serviced, the command queue from which it was taken will have the highest priority until the chain is terminated. The command bus arbiter will not service any more commands from the other Microengine Command Queues until the chain is complete.

Note: Executing an abnormally long sequence of chained requests might affect the rate at which SDRAM refreshes occur. The Refresh Controller is designed to generate requests such that the average refresh rate is maintained, but chained accesses may have some bearing on the actual timing of the execution of the Refresh commands.

7.3.1.2 Priority 1: Refresh Requests

Requests for SDRAM refresh cycles have the next priority after chained references, so that SDRAM refreshes occur as close as possible to the rate at which they were programmed.

7.3.1.3 Priority 2: Round Robin Requests

The next three requests are grouped together in a Round Robin prioritization scheme. These requests are granted accesses in a rotating priority fashion depending upon which interface last received a grant.

At reset, an AMBA request will have the highest priority of the group, followed by PCI requests and then the Microengines High Priority Queue, until a grant is generated to one of these three units. After a grant is generated to one of the three units, the granted interface will have the lowest priority of the three, while the remaining two interfaces move up in priority. This enables each unit to have fair access to the SDRAM Bus. The Round Robin priority of the three units changes only after a grant has been generated to one of the units and is static otherwise.

7.3.1.4 Priority 3 through 5: The Remaining Microengine Requests

The remaining priorities are based on the Order, Odd bank, and Even Bank Microengine command queues. While the Microengine Order Queue request is static, the Odd and Even Bank command queues are dynamic, depending upon the bank of the last grant that was generated.

When the Command Service Priority Logic services a command queue, the bank of the beginning quadword address is noted and registered by the Command Service Priority Logic. The next command queue that is serviced is based on this bank information. If the bank of the previous access was to the Even bank, the Command Service Priority Logic will assign priority 3 to the Microengine Odd Bank command queue (the opposite bank), and priority 5 to the Microengine Even Bank command queue (same bank). If the previous SDRAM access was to an odd bank, the Even Bank command queue is assigned to priority 3 while the Odd Bank Queue (opposite bank) is assigned priority 5. This allows the SDRAM Unit to optimize the performance of the SDRAM bus. Refer to Section 7.3.4 for more information on active memory optimization.

7.3.2 Read-Modify-Write

The Read-Modify-Write function allows individual bytes to be written to the SDRAM devices. To accomplish this, the SDRAM Unit firsts reads the data from a quadword address, modifies the specified bytes, and then writes the modified quadword back to the SDRAM device. These three steps are performed atomically. The SDRAM Unit supports Read-Modify-Write operations from the Microengines, StrongARM* core, and PCI Unit.

The PCI requires byte write capability for byte lane support and single longword write operations. The StrongARM* core requires byte write capability to support load and store byte and word instructions. The Read-Modify-Write operations are transparent to the StrongARM* core and the PCI Unit

The Microengines can explicitly request that a Read-Modify-Write operation be performed at an SDRAM quadword address using the indirect reference option. The reference count must be a value of 1 and the data that is written is specified in an SDRAM write transfer register. A mask in the **indirect_ref** specifies which bytes are to be modified.

7.3.3 Chained References

Chained references allow a source to issue multiple commands and be assured that the SDRAM Unit will execute the commands as long as the source keeps the chain going.

The SDRAM Unit will automatically set the chained bit if it receives two or more consecutive commands from the PCI Unit that request access to consecutive SDRAM longword addresses.

The Microengines issue chained references by specifying the chain_ref within the **sdram** instruction. As long as the Microengine issues **sdram** instructions specifying the **chain_ref** optional token, the chain is kept alive. From a Microengine perspective, the chained references have three implications. First, the Command Bus Arbiter will not allow any other Microengine to issue SDRAM commands until the chain has been completed. A chained request is considered complete when all of the commands associated with the request have been entered into the SDRAM Unit queuing structure, not when the actual memory access is complete.

Secondly, it indicates that the SDRAM Unit should process the SDRAM command from the same command queue from which the chained reference was serviced and continue to service this command queue until the chain is complete.

The third implication only occurs if the **sdram** instruction specifies both the **t_fifo_wr** command and the **chain_ref** optional token. In this case it also causes the byte aligner to hold data left over from the preceding non-aligned access in a residue latch within the SDRAM Unit, so that the data can be used on the next **t_fifo_wr** command that is part of the chain. This improves bandwidth during chained **t_fifo_wr** operations.

The chained references can only be submitted to the Priority or Order command queues. The **chain_ref** optional token can not be used with the **optimize_mem** optional token.

Note: When using a chained reference, if an interface generates a chained request and fails to follow it up immediately with another request, it will cause the Arbiter to stop until a following request is generated. It is precisely this function which allows the SDRAM interface to hold off requests from all interfaces until the SDRAM initialization sequence has been completed.

7.3.4 Active Memory Optimization

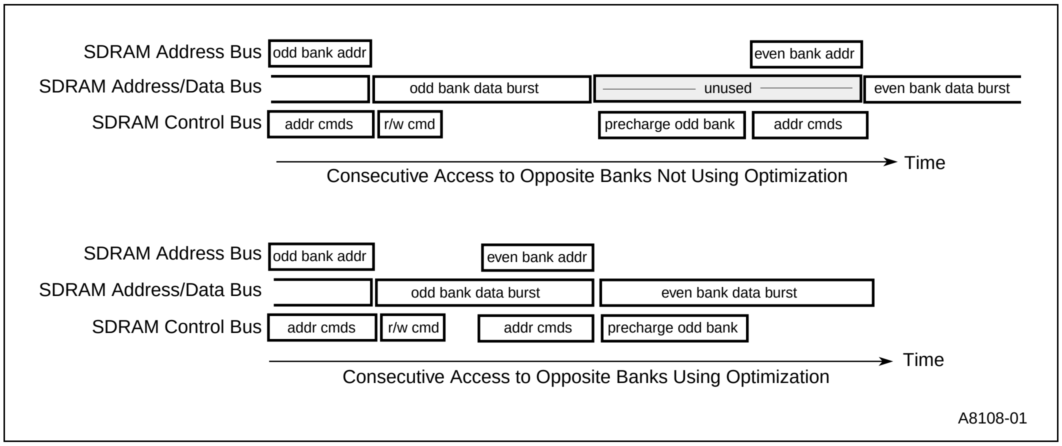
SDRAMs are burst-based memories requiring only a single address to be specified to transfer data from several addresses. This means that the control bus is unused in many cases when burst transfers are being performed. SDRAMs also have multiple banks, but data can only be transferred from a single bank at a time. However, the other banks may be addressed via the control bus in preparation for the next data transfer. Each time SDRAM accesses cross a bank boundary, the current bank must be closed and precharged. This requires use of the control bus and can introduce latencies associated with reading data from a different bank.

The SDRAM Unit can eliminate the latencies associated with bank switching by taking advantage of the multi-bank nature of SDRAM devices, and the unused control bus cycles to optimize SDRAM data transfers. It accomplishes this in two ways:

1. By making use of cycles on the control bus that would otherwise be NOPs to prepare and have waiting an access to another bank.
2. By making use of cycles on the control bus that would otherwise be NOPs to precharge the bank that was closed while an access to another bank is in progress.

This allows the IXP1200 to eliminate precharge latency and increase the effective bandwidth of the SDRAM Bus. Figure 7-6 illustrates how bank switching improves performance.

Figure 7-6. Bank Switching



The SDRAM Unit in the IXP1200 provides an additional advanced feature that increases the ability of the SDRAM Unit to exploit bank switching. This feature is implemented as separate command queues for references to the Odd Banks and the Even Banks. The SDRAM Unit can then alternate between servicing commands from these queues.

The programmer decides whether a Microengine SDRAM reference should be sorted into the Odd Bank or Even Bank Command Queue by specifying the **optimize_mem** optional token within the **sdram** instruction. The SDRAM Unit automatically sorts the command based on the SDRAM address. The commands submitted to the Odd and Even Bank Command Queues are not guaranteed to be completed in the order in which they where issued. If order is important, the **optimize_mem** option should not be used (in which case the command goes into the Order command queue).

Note: It is assumed that a majority of memory accesses will use the **optimize_mem** token. Continual assignment of memory references to the Order or High Priority Queues defeats the purpose of the memory optimization performance gain, and in some instances leads to less than desired performance.

7.4 Microengine SDRAM Transactions

7.4.1 Microengines Command Queues

The Microengines issue commands to the SDRAM Unit which in turn places them into one of four Command Queues within the SDRAM Unit. An optional token within the instruction specifies into which of the four queues a command is submitted. The four Command Queues, the queue sizes and the optional tokens are listed in Table 7-5.

Table 7-5. Command Queues

Microengine Command Queue	Queue Size	Instruction optional token
Odd Bank	16	optimize _mem
Even Bank	16	optimize _mem
Order (default)	16	order (or no optional token)
High Priority Queue	16	priority

The Command Queues are sized so that in typical operation, the queues should not fill completely. A back-pressure mechanism will signal the Microengines Command Bus arbiter to cease allowing SDRAM commands to be sent to the SDRAM Unit when any of the queues have only six entries available. This ensures there is room for any SDRAM commands that have already been granted access to the SDRAM Unit by the Microengine Command Bus arbiter.

When the SDRAM Unit processes an **sdram** instruction, the SDRAM Push-Pull Engine moves the data between the SDRAM Unit and the Microengine SDRAM write/read transfer registers via an internal 64-bit SDRAM data bus. The SDRAM bus is divided into a 32-bit pull and 32-bit push bus and data is moved across these buses at the Core frequency.

If a Microengine thread chooses to be signaled upon completion of a command, the **sig_done** or **ctx_swap** optional token should be specified in instruction.

7.4.2 SDRAM Byte Aligner

The SDRAM Byte Aligner allows data to be read from SDRAM memory on non-quadword aligned addresses, align the data and present it to the TFIFO (in the IX Bus Interface Unit) on aligned quadword boundaries.

Note: The Byte Aligner can only be used by the Microengines using the **sdram** instruction with the **t_fifo_wr** command.

The SDRAM Unit aligns and merges data that is read from two consecutive quadword addresses to create the non-aligned quadword. The four-bit byte alignment value specified in the indirect reference data defines the alignment.

The alignment value can range from 0 to 7 and indicates the byte offset into the first quadword of data. The remaining data comes from the next consecutive address. When more than one quadword is specified in the **sdram** instruction, the unused data from the second address will be merged with the data from the next address. Depending on burst size, memory accesses from SDRAM which occur on non-byte aligned boundaries, the **chain_ref** optional token can be used to eliminate the need for specifying the byte alignment between consecutive **sdram** instructions.

How the SDRAM Unit will perform the alignment is based on the Endian bit in the SDRAM_CSR register. The endian format is based on bytes that are on longword boundaries (Little Endian format: 8765 4321, Big Endian format: 5678 1234). The following example shows the results of different alignment values for both Big and Little Endian modes. N is the address specified in the **sdram[t_fifo_wr]** instruction or the last data read from the previous read in a chained reference.

The data at quadword addresses N and N+1 are as follows:

Table 7-6. Data at Quadword Address N and N+1

SDRAM Address	Data
N+1	1234 5678
N	ABCD EFGH

The data at these two consecutive SDRAM quadword addresses are aligned and merged and written to the TFIFO on quadword boundaries according to the different align values and the endian mode specified. Table 7-7 shows the different possible alignment results.

Table 7-7. Alignment Results

Align value	Little Endian Mode	Big Endian Mode
0	ABCD EFGH	ABCD EFGH
1	8ABC DEFG	BCD5 FGHA
2	78AB CDEF	CD56 GHAB
3	678A BCDE	D567 HABC
4	5678 ABCD	5678 ABCD
5	4567 8ABC	6781 BCD5
6	3456 78AB	7812 CD56
7	2345 678A	8123 D567

7.5 PCI SDRAM Transactions

Since the PCI Unit operates with longword (32 bit) data while the SDRAM interface to the physical memory is quadword (64 bit), there must be a coalescing of data from the PCI Unit before presenting it to the SDRAM devices. The SDRAM Unit performs this function.

The PCI Unit presents a command and the data (if a write is to take place) to the SDRAM Unit for each PCI bus transfer. When performing a burst of consecutive commands that will address contiguous memory locations in SDRAM, the SDRAM Unit merges the commands into a single chained command if the read or write are to adjacent memory addresses. The merging of the commands result in a single 64-bit access.

The SDRAM-to-PCI and PCI-to-SDRAM transfer operations are described in Chapter 5 (PCI Unit) of the *IXP1200 Network Processor Hardware Reference Manual*.

Note: It is possible that the SDRAM Unit will prefetch more data than was requested by a PCI host device. If so, this data is buffered in the PCI Unit but will not persist past the deassertion of FRAME#.

7.6 StrongARM* Core SDRAM Transactions

The StrongARM* core issues references to the SDRAM Unit via the AMBA Bus. These requests can be generated directly by the StrongARM* core Processor or by Icache, Dcache (main and mini), StrongARM* core Read Buffer, or StrongARM* core Write Buffer.

StrongARM* core references to non-cached areas cause the StrongARM* core to stall until the SDRAM Unit provides data at the AMBA Data Buffer. For write operations, the StrongARM* core will stall until it gains access to the AMBA Bus, which is shared by the caches and buffers. Once access is granted, the StrongARM* core will write data to the AMBA Bus Logic which translates the AMBA bus signaling into a Microengine-like command and the data is written into the AMBA Data Buffer so that the StrongARM* core may continue to execute. When the SDRAM command arbiter grants access to the AMBA Bus interface, the AMBA Bus command is completed and the data is read from the AMBA Data Buffer. The SDRAM Unit does not queue AMBA transactions and therefore only one AMBA transaction can be in progress at a time: a read (that can prefetch) or two writes. The AMBA Write Buffer can hold 16 longwords, up to two cache lines at a time. The AMBA Data buffer is a FIFO, which is used for one type of transaction at a time, reading or writing. For example, after reading data from SDRAM the buffer must be emptied before a write to SDRAM can utilize the buffer for the write data.

7.6.1 StrongARM* Core and Microengine SDRAM Address Space

Table 7-8 shows the Memory Map for the SDRAM address space. Notice that the StrongARM* core addresses this space using byte addressing while the Microengines access this space using the **sdram** instruction and longword addressing.

Table 7-8. Memory Map for SDRAM Address Space

Memory Area Description	StrongARM Address Space (Byte Addressing)	Microengine sdram Instruction command (Longword addressing)
SDRAM Control Registers	FF00 0000 - FF00 0014	Not supported
SDRAM Prefetch Memory (256 Mbytes)	D000 0000 - DFFF FFFF	Not supported
SDRAM non-Prefetch Memory (256 Mbytes)	C000 0000 - CFFF FFFF	read and write 0x000 0000 - 0x1FF FFFF
Cache Flush Area (16 Kbytes)	A000 0000 - A000 4000	Not supported

7.6.1.1 SDRAM CSRs

The SDRAM Unit must be configured before it can be used. Only the StrongARM* core has access to the four SDRAM CSRs (the SDRAM_CSR, SDRAM_MEMCTL0, SDRAM_MEMCTL1 and SDRAM_MEMINIT registers). The SDRAM CSRs allow the bus timing and row and column address pin assignments to be configured for a variety of SDRAM devices. The last register to be programmed must be the SDRAM_CSR since it contains the INIT bit that causes the other register settings to be loaded by the SDRAM Controller.

7.6.1.2 SDRAM non-Prefetch Memory (256 Mbytes)

Located in the C000 0000 - CFFF FFFF address space, access to memory at these addresses will not cause an automatic fetch of the next eight longwords from memory.

7.6.1.3 SDRAM Prefetch Memory (256 Mbytes)

Located in the D000 0000 - DFFF FFFF Address Space, access to memory at these addresses will automatically fetch the next 8 longwords from memory to reduce SDRAM latency to the StrongARM* core. Only StrongARM* core accesses support prefetch. Microengine access to this area will not activate the prefetch logic.

7.6.1.4 Cache Flush Area (16 Kbytes)

The cache flush area is accessible only from the StrongARM* core. This area returns zeros when read and is intended to facilitate rapid cache flushing by returning zero data without requiring an external memory access. This space is used for read and write bursts. Non-bursted writes to this space have no effect. Refer to Section 3.2.4.5 for more information.

Note: If the cache is disabled then the StrongARM* core will not perform a burst.

7.7 SDRAM and the IX Bus Interface

The transfer of data between the IX Bus Interface Unit and the SDRAM interface is an important part of the IXP1200. Data from external devices attached to the IX Bus is stored in a 16-element receive FIFO (RFIFO) located in the IX Bus Interface Unit. Data that is to be transmitted to devices connected to the IX Bus is stored in a separate 16-element transmit FIFO (TFIFO) also located in the IX Bus Interface Unit. A separate datapath exists in the IXP1200 for transfer of data between SDRAM and these FIFOs.

7.7.1 SDRAM to TFIFO Operation

Moving data from SDRAM to the TFIFO is done by the Microengines using the **sdram[t_fifo_wr, ...]** instruction with the **indirect_ref** token.

Note: The **indirect_ref** token must be used with this command.

With this command, up to 16 quadwords (128 bytes) of data can be moved from SDRAM to the specified TFIFO element(s). Each TFIFO element holds up to eight quadwords of data (64 bytes), so a 16 quadword reference will span multiple TFIFO elements. When issued by a Microengine, this command is placed in the Order queue. Once the SDRAM Service Priority logic services the command, a data “push” to the IX Bus Interface Unit will result. Byte alignment is possible and is specified in the **indirect_ref** portion of the command. The IX Bus Interface Unit can be optionally signaled by the SDRAM Unit when the data transfer is complete.

7.7.2 Receive FIFO to SDRAM Operation

Moving data from RFIFO to SDRAM is done by the Microengines using the **sdram[r_fifo_rd, ...]** instruction with the **indirect_ref** token.

Note: The **indirect_ref** token must be used with this command.

With this command, up to 16 quadwords (128 bytes) can be moved from the RFIFO element(s) to SDRAM. Each RFIFO element can hold up to eight quadwords of data (64 bytes), so a 16 quadword reference will empty two RFIFO elements. When issued by a Microengine, this command is placed in the Order queue. Once the SDRAM Service Priority logic services the command, a data “pull” from the IX Bus Interface Unit will result. Byte alignment **is not possible** when moving data from the RFIFO to SDRAM.

7.7.3 SDRAM and IX Bus Data Path Operation

While the datapath is separate for transferring data to and from the IX Bus FIFOs, it should be noted that the command path and thereby the Order queue is shared with the Microengines. The internal datapath to and from the RFIFO and TFIFO elements is 32 bits wide. Therefore, half the data (32 bits) will be held for one cycle before sending it on the bus.

Transfers of data from the SDRAM Unit to the TFIFO out of the area marked as prefetchable will not cause a prefetch operation to occur as this is only for the StrongARM* core.

SRAM Unit 8

8.1 Overview

The SRAM Unit provides access to three types of devices through a common bus interface:

- **SSRAM**. Up to 8 Mbytes of either Flow Thru or Pipelined Synchronous SRAM devices.
- **BootROM**. Flash or EPROM. This is where the StrongARM* core begins executing instruction from after a reset.
- **SlowPort**. Other devices such as Lookup CAMs, encryption devices, and control and status interfaces to MAC/PHY devices can be placed onto the SRAM Bus.

Each of the three types of devices are mapped to different address spaces and each address space provides unique timing that is programmed via SRAM configuration registers. The devices on the SRAM Bus can be accessed by the Microengines and the StrongARM* core. The external interface consists of a 32-bit data bus, 19-bit address bus, and control signals. The bus is clocked at half the core frequency.

The SSRAM address space supports advanced functionality beyond simple reads and writes. These functions are designed to offload the StrongARM* core and Microengines of common operations that might otherwise require additional instructions that perform multiple memory operations. These commands provide the following functionality.

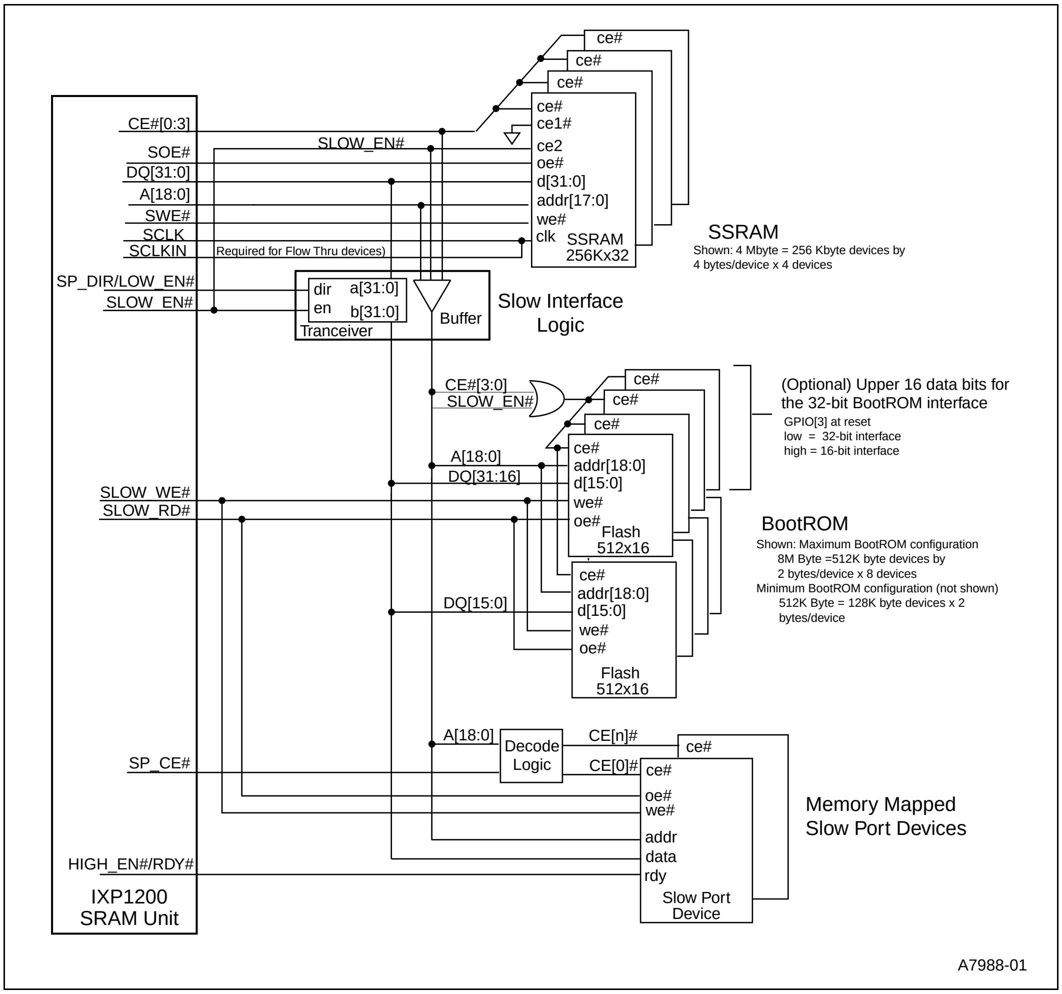
- Eight Push-pop Registers for buffer management
- Bit Test, Set, and Clear commands for atomic bit operations
- Eight-entry Content-Addressable Memory (CAM) for locking eight separate memory areas
- Journal pointer management support.

The StrongARM* core addresses the four address spaces in the SRAM Unit using byte addressing while the Microengines use longword addressing.

8.2 SRAM Bus Configurations

The SRAM Bus supports a 32-bit data bus, 19-bit address bus, and control signals. The bus is clocked at half the core frequency and three different address spaces support three different bus timings. Figure 8-1 illustrates an example of external connections on the SRAM interface. The sections that follow provide additional information about these address spaces.

Figure 8-1. SRAM External Connections



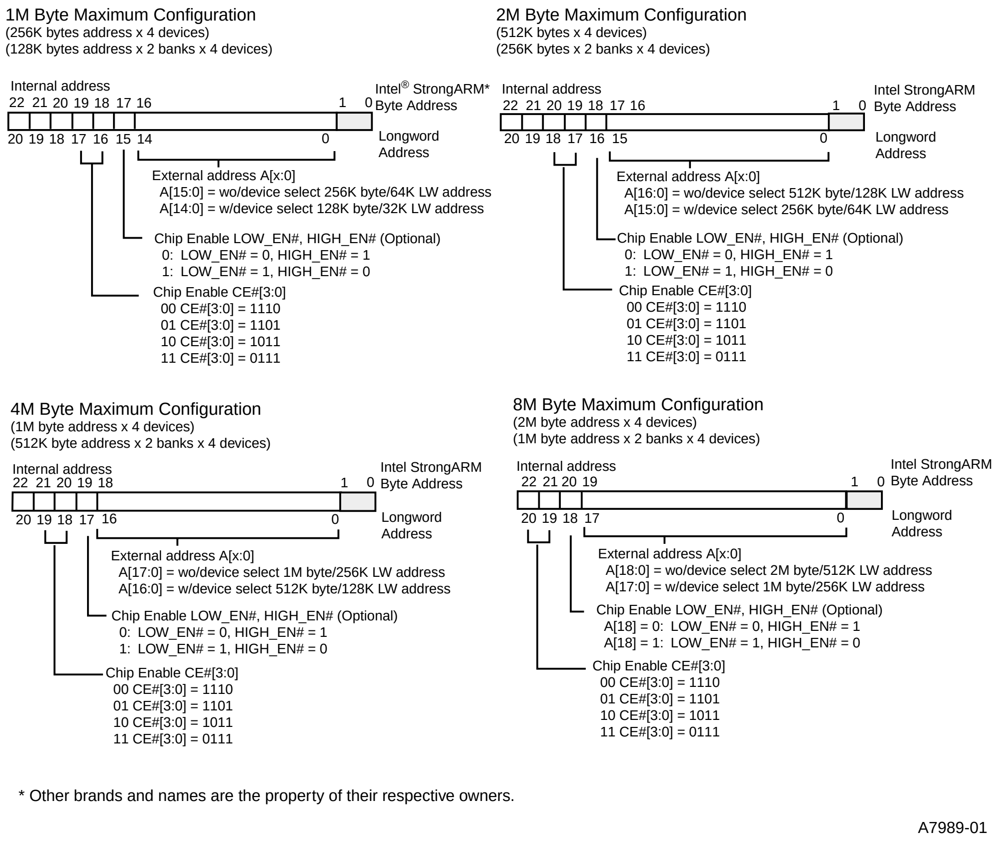
8.3 SSRAM Address Space

The SSRAM provides a memory source with lower access latency than the SDRAM interface. The SSRAM is intended to contain lookup tables and buffer free-lists used by the processors when moving data through the IXP1200.

The SRAM Unit provides a 32-bit interface to 8 Mbytes of Synchronous SRAM (SSRAM). Both Flow Thru devices and Pipelined Burst SSRAM devices are supported. Internally, the Microengines always addresses the SRAM Unit using longword addressing while the StrongARM* core uses byte addressing. However, the external physical memory is always addressed using longword addresses.

The internal address is used to generate the external: longword address (A[18:0] - maximum of 2 Mbytes), chip selects (CE#[3:0] - maximum 4 devices), and two additional chip selects which are optional (HIGH_EN# and LOW_EN#). The external address pins A[18:0] always reflect the state of the low 19-bits of the internal longword address. The primary chip selects (CE#[3:0]), and the two additional chip selects (HIGH_EN# and LOW_EN#) pins are also based on the internal address and can be configured via the SRAM CSRs to support the four memory configurations illustrated in Figure 8-2.

Figure 8-2. Memory Configurations


1M Byte Maximum Configuration
(256K bytes address x 4 devices)
(128K bytes address x 2 banks x 4 devices)
Internal address
22 21 20 19 18 17 16
1 0
Intel® StrongARM* Byte Address
20 19 18 17 16 15 14
0
Longword Address
External address A[x:0]
A[15:0] = wo/device select 256K byte/64K LW address
A[14:0] = w/device select 128K byte/32K LW address
Chip Enable LOW_EN#, HIGH_EN# (Optional)
0: LOW_EN# = 0, HIGH_EN# = 1
1: LOW_EN# = 1, HIGH_EN# = 0
Chip Enable CE#[3:0]
00 CE#[3:0] = 1110
01 CE#[3:0] = 1101
10 CE#[3:0] = 1011
11 CE#[3:0] = 0111
2M Byte Maximum Configuration
(512K bytes x 4 devices)
(256K bytes x 2 banks x 4 devices)
Internal address
22 21 20 19 18 17 16
1 0
Intel StrongARM Byte Address
20 19 18 17 16 15
0
Longword Address
External address A[x:0]
A[16:0] = wo/device select 512K byte/128K LW address
A[15:0] = w/device select 256K byte/64K LW address
Chip Enable LOW_EN#, HIGH_EN# (Optional)
0: LOW_EN# = 0, HIGH_EN# = 1
1: LOW_EN# = 1, HIGH_EN# = 0
Chip Enable CE#[3:0]
00 CE#[3:0] = 1110
01 CE#[3:0] = 1101
10 CE#[3:0] = 1011
11 CE#[3:0] = 0111
4M Byte Maximum Configuration
(1M byte address x 4 devices)
(512K byte address x 2 banks x 4 devices)
Internal address
22 21 20 19 18
1 0
Intel StrongARM Byte Address
20 19 18 17 16
0
Longword Address
External address A[x:0]
A[17:0] = wo/device select 1M byte/256K LW address
A[16:0] = w/device select 512K byte/128K LW address
Chip Enable LOW_EN#, HIGH_EN# (Optional)
0: LOW_EN# = 0, HIGH_EN# = 1
1: LOW_EN# = 1, HIGH_EN# = 0
Chip Enable CE#[3:0]
00 CE#[3:0] = 1110
01 CE#[3:0] = 1101
10 CE#[3:0] = 1011
11 CE#[3:0] = 0111
8M Byte Maximum Configuration
(2M byte address x 4 devices)
(1M byte address x 2 banks x 4 devices)
Internal address
22 21 20 19
1 0
Intel StrongARM Byte Address
20 19 18 17
0
Longword Address
External address A[x:0]
A[18:0] = wo/device select 2M byte/512K LW address
A[17:0] = w/device select 1M byte/256K LW address
Chip Enable LOW_EN#, HIGH_EN# (Optional)
A[18] = 0: LOW_EN# = 0, HIGH_EN# = 1
A[18] = 1: LOW_EN# = 1, HIGH_EN# = 0
Chip Enable CE#[3:0]
00 CE#[3:0] = 1110
01 CE#[3:0] = 1101
10 CE#[3:0] = 1011
11 CE#[3:0] = 0111
A7989-01

* Other brands and names are the property of their respective owners.

The CE#[3:0] chip select signals allow up to four physical devices to be selected without external decode of the address pins. If required, the CE#[3:0] chip select signals can be qualified with the HIGH_EN# and LOW_EN# pins to allow a total of eight banks to be selected without external decode logic.

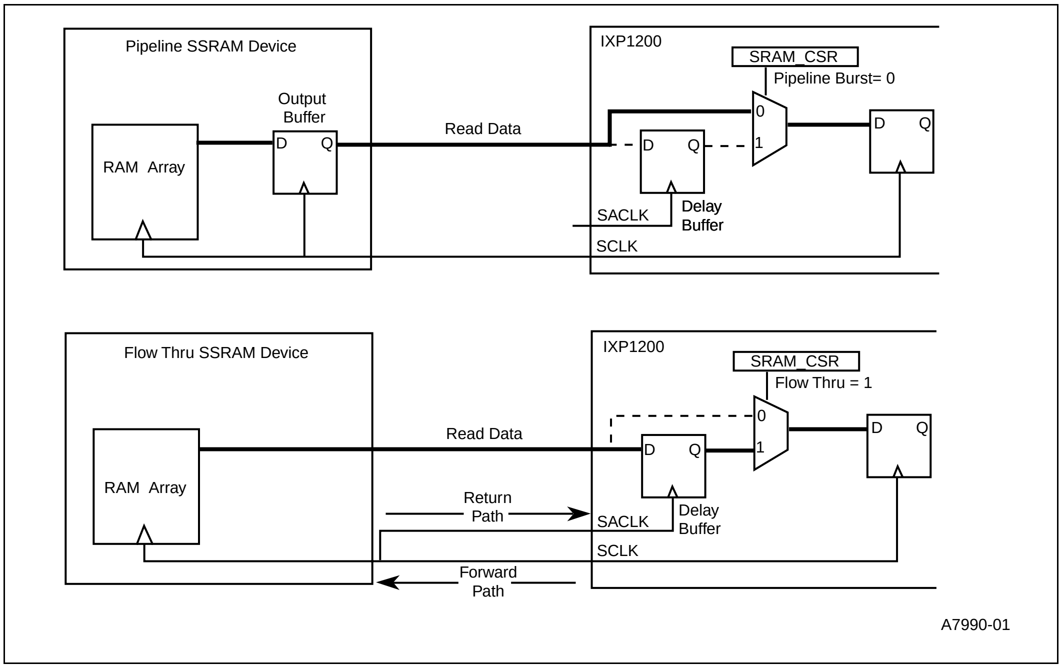
The SSRAM interface also allows a wait state to be inserted whenever the address switches between the physical devices on the bus. This is set via the SRAM CSRs and must be set when using SSRAM devices that have a slower output disable time than output enable time. The wait state is asserted whenever an access crosses a chip selects (CE#) or bank select (HIGH_EN# or LOW_EN#) boundary.

8.3.1 Pipelined vs. Flow Thru SRAM Devices

The SSRAM Interface can be configured via the SRAM_CSR to support either Flow Thru devices, which have a pipeline delay of 2 clock cycles, or Pipelined Burst devices that support Dual Cycle Deselect, which have a pipeline delay of 3 clock cycles. Other types of SSRAM devices, including single cycle deselect, are not supported. Figure 8-3 shows the IXP1200 internal logic associated with the Pipelined and Flow Thru SSRAM devices and the internal logic of the IXP1200 SRAM Unit.

The Pipelined SRAM devices include an internal output buffer that latches the data while Flow Thru devices do not. The IXP1200 supports the timing differences between the two types of SRAM devices by adding an internal buffer that adds an extra clock cycle delay for the Flow Thru device. This extra delay assures that the SRAM data arrives on the same clock cycle whether it comes from Pipelined or Flow Thru devices.

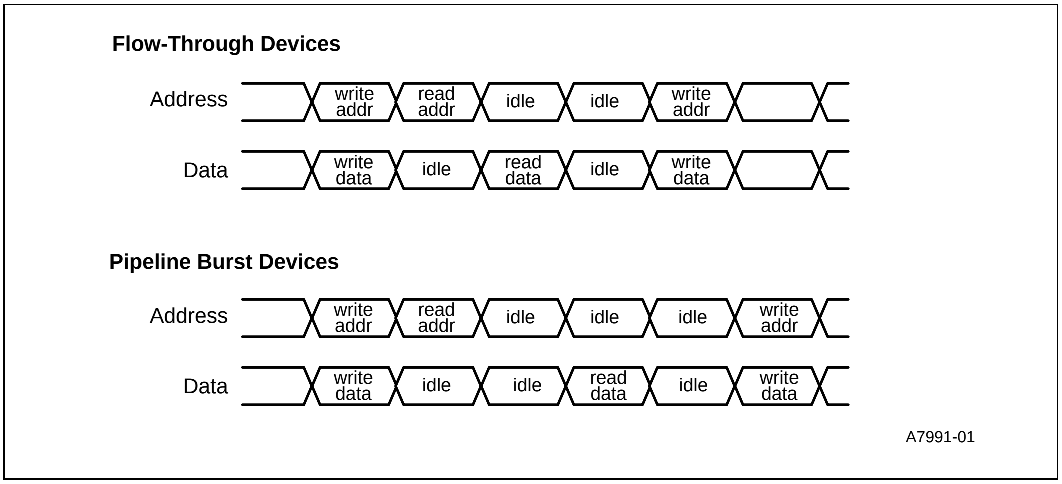
Figure 8-3. Pipelined vs. Flow Thru SSRAM Device



The internal output buffer of the Pipelined SRAM is clocked by the IXP1200 SCLK signal. This signal enters the SRAM device at the end of the trace resulting in a skew between the SCLK at the IXP1200 and the SCLK at the SRAM device where the output buffer is actually clocked. This skew is compensated for in the IXP1200 Flow Thru configuration by clocking the delay buffer in the IXP1200 with an SCLK signal that traverses the board with an etch length equal to that of the forward and return path to the SRAM device. The SCLKIN pin is used as the input pin for the return path signal that clocks the IXP1200 delay buffer.

The Flow Thru devices provide better performance than Pipelined devices for the case when a read operation is followed by a write operation, since the IXP1200 can begin the write operation one cycle earlier. Figure 8-4 illustrates the performance advantage by showing the timing for a write-read-write operation for both Flow Thru and Pipelined devices.

Figure 8-4. Flow Thru Device



8.4 BootROM Address Space

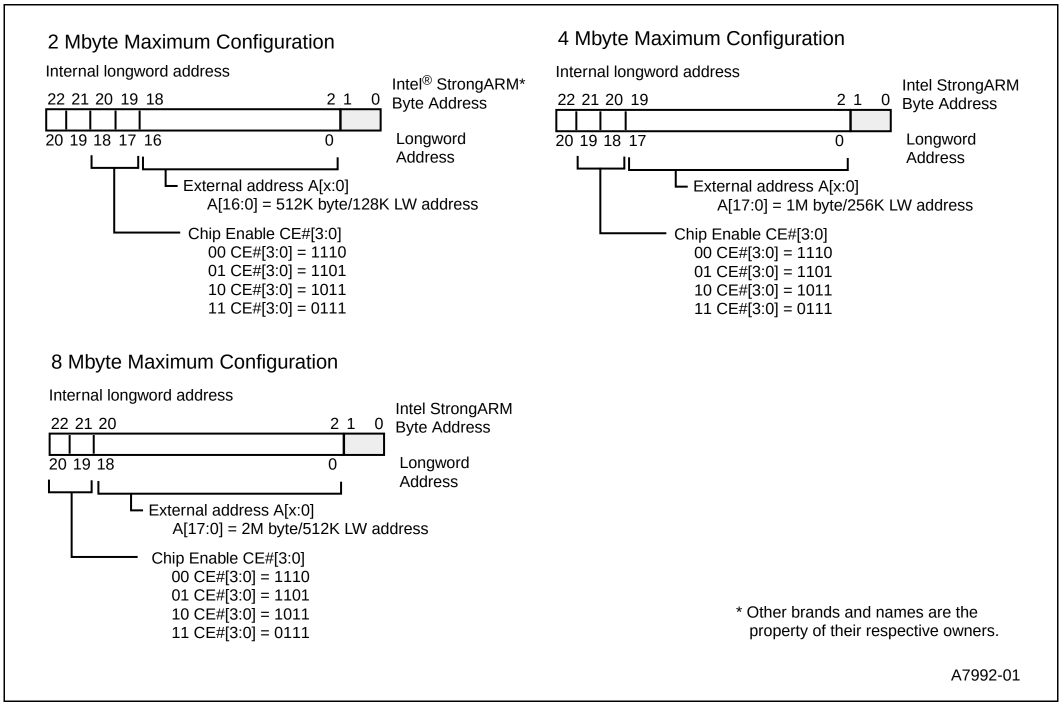
The BootROM address space supports up to 8 Mbytes of Flash or EPROM. This address space can be accessed by the StrongARM* core and the Microengines. The BootROM is mapped to the StrongARM* core physical address 0. After a reset, the StrongARM* core begins fetching instructions from this address space.

Note: The StrongARM* core must boot from address 0000 0000h, which is mapped to the BootROM. It cannot boot from PCI or any other devices in its address map.

BootROM address space can be configured to support either a 16-bit or 32-bit data bus. The bus size is determined at reset and is based on the state of the GPIO[3] pin (GPIO[3] pulled high = 16-bit bus and GPIO[3] is pulled low = 32-bit bus).

The BootROM address space shares the four external chip select pins (CE#[3:0]) with the SSRAM interface. These signals are asserted based on a 21-bit internal longword address. Three CE# configurations are supported and are configured via the SRAM CSRs (Figure 8-5). As shown in Figure 8-1, CE# signals should be gated with the SLOW_EN# signal to ensure that the BootROM devices are not selected when the SSRAM asserts the CE# signals.

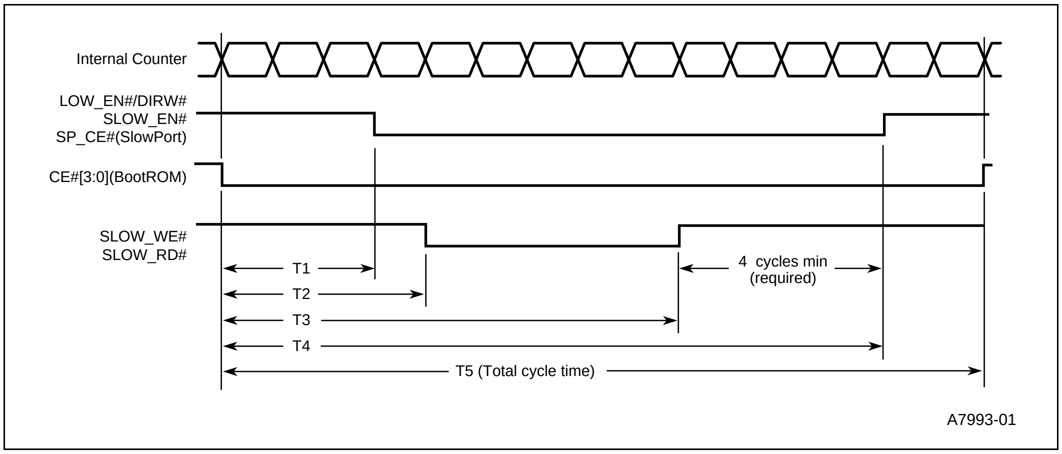
Figure 8-5. BootROM Addressing



* Other brands and names are the property of their respective owners.

Figure 8-6 shows the timing characteristic for the BootROM interface. The programmable timing values are listed in the figure (T1 to T5). All the time values are based on an internal counter that is decremented each Core clock cycle. The BootROM timing parameters are programmed via the SRAM_SLOW_CONFIG and SRAM_BOOT_CONFIG SRAM CSRs.

Figure 8-6. BootROM Timing



For applications where a small amount of BootROM is required, a 16-bit data bus mode may be selected. This mode is intended to allow the StrongARM* core to Boot from 16-bit BootROM devices. When this mode is selected, whenever the StrongARM* core reads data from the BootROM address space, the SRAM Unit reads the data pins DQ[15:0], merges them into a single

32-bit longword and delivers them to the StrongARM* core. This only occurs during read operations by the StrongARM* core. When the StrongARM* core performs a write operation, it must specify a longword address and place valid data only on the lower 16 bits.

The 16-bit data bus mode is not supported for the Microengines. When the 16-bit data bus mode is selected, the Microengines always read and write 32-bit data and software must recognize that valid data only resides in the lower 16 bits.

BootROM address space can be configured to support a 16-bit data bus or a 32-bit bus. The bus size is determined at reset and is based on the state of the GPIO[3] pin (GPIO[3] pulled high = 16-bit bus and GPIO[3] is pulled low = 32-bit bus).

8.5 SlowPort Address Space

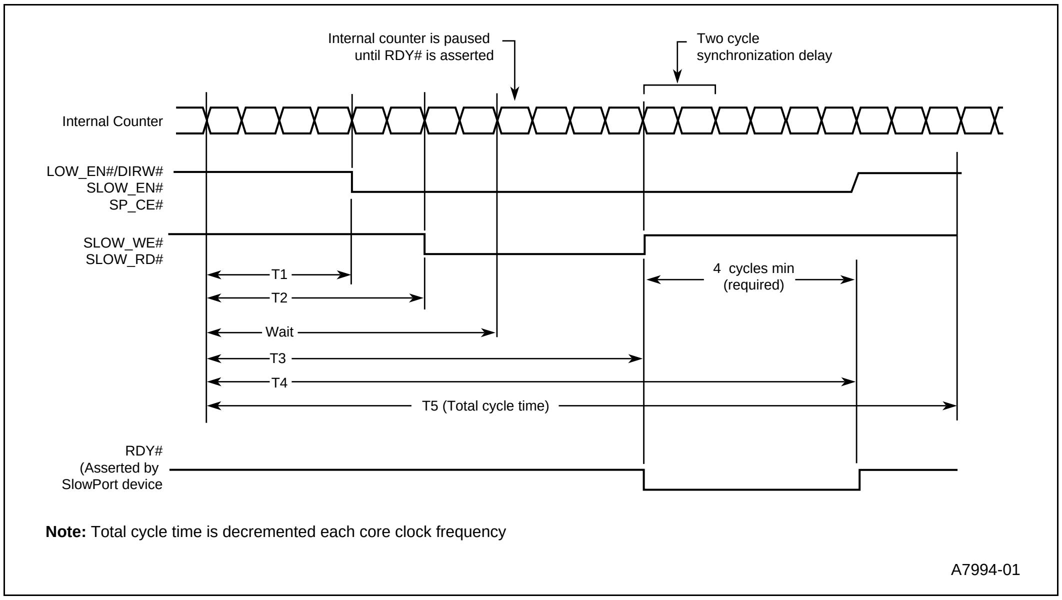
The SRAM Unit provides a 2 Mbyte address space (512K longword addresses) to access SlowPort devices. The SlowPort address space allows peripheral devices to be placed onto the SRAM interface. The SRAM SlowPort memory space has timing characteristics similar to the BootROM memory space, although these two address spaces have their own programmable timing values that are set via the SRAM CSRs. Both the StrongARM* core and the Microengines have access to the SRAM SlowPort.

The CE# signals are not supported when the SlowPort address space is accessed. Instead, the SlowPort interface supports one chip select signal (SP_CE#) that is asserted whenever the SlowPort address space is accessed. The SP_CE# signal can be qualified with the address pins to derive more chip select signals.

The SlowPort interface supports a Ready indication (RDY#) which can be asserted by a SlowPort device to indicate that either data is stable on the data bus during read access or that data was latched by the SlowPort device during write access. This allows IXP1200 to dynamically vary the bus timing to provide higher performance when devices with different timing characteristics are placed on SlowPort interface. The SlowPort interface should be programmed to meet the timing requirements of the slowest device on the bus that does not support a RDY# pin. If all the SlowPort devices support a RDY# pin, the SlowPort interface timing should be programmed to support the fastest device on the bus.

Figure 8-7 shows the timing characteristic for the SlowPort interface. The programmable timing values are listed in the figure (T1 to T6). All the time values are based on an internal counter that is decremented each Core clock cycle. If the HIGH_EN#/RDY# pin is configured as a RDY# pin (via the SRAM_CSR) the internal counter is paused at T6 until the RDY# signal is asserted by the SlowPort device. Once the RDY# signal is asserted, a two cycle synchronization delay occurs before the counter resumes decrementing. The SlowPort timing parameters are programmed via the SRAM_SLOW_CONFIG and SRAM_SLOWPORT_CONFIG SRAM CSRs.

Figure 8-7. SlowPort Timing



Note: Total cycle time is decremented each core clock frequency

A7994-01

8.6 Slow Interface Logic

The BootROM and SlowPort devices are isolated from the SSRAM devices by the Slow Interface Logic shown in Figure 8-1. The isolation allows the buses to meet the electrical loading requirements specified in the IXP1200 datasheet and reduce etch run lengths thereby reducing transmission line effects on the bus.

8.7 SRAM CSRs

The SRAM Control and Status Registers (CSRs) reside in the SRAM Unit and can be accessed by both the Microengines and the StrongARM* core. These registers are described in detail in the *IXP1200 Network Processor Programmer's Reference* and are used for the following purposes:

- Configure the SSRAM timing and banks
- Configure the BootROM timing and banks
- Configure the Slow Port timing
- Configure and write to the SRAM Journal Area

When changing the SlowPort and BootROM timing, the following sequence should be followed to ensure proper operation.

- To shorten a timing parameter change the assert and deassert times specified in the SRAM_BOOT_CONFIG/SRAM_SLOWPORT_CONFIG registers before changing the total cycle time specified in the SRAM_SLOW_CONFIG register.

- To lengthen a timing parameter, change the total cycle time specified in the SRAM_SLOW_CONFIG register before changing the assert and deassert times specified in the SRAM_BOOT_CONFIG/SRAM_SLOWPORT_CONFIG registers.

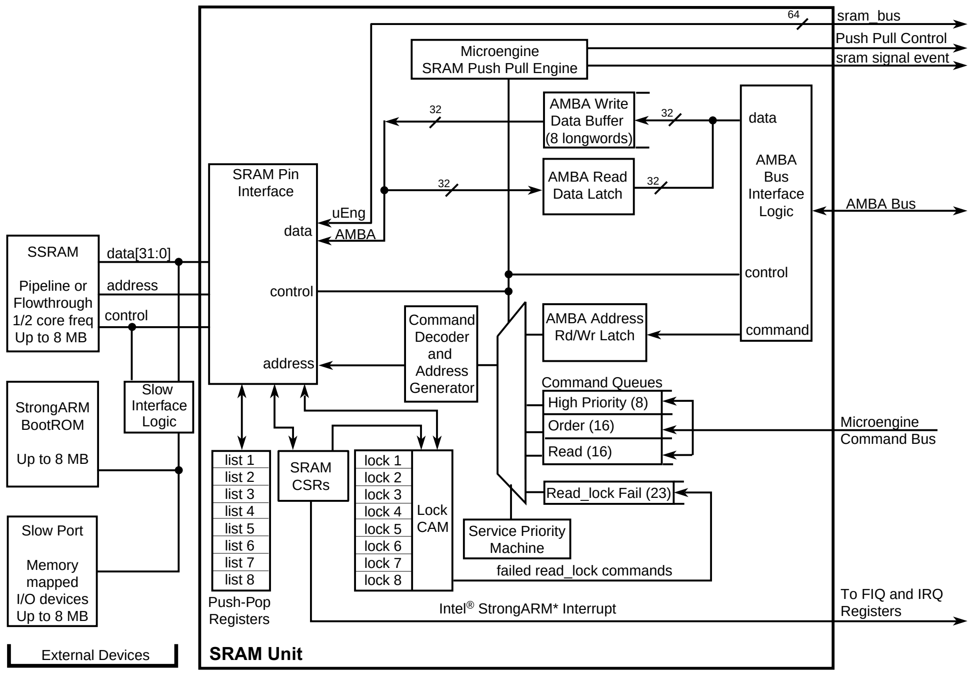
8.8 Advanced SRAM Commands

The SRAM Unit supports advanced functionality beyond simple read and write operations to the SSRAM space. These commands are designed to offload the StrongARM* core and Microengines of managing common operations that might otherwise be managed in software requiring multiple memory operations. These commands provide the following functionality.

- Eight Push-Pop Registers for buffer management
- Bit Test, Set, and Clear commands for atomic bit operations
- Eight entry CAM for locking eight separate memory areas
- Journal pointer management support.

Figure 8-8 is a block diagram of the SRAM Unit.

Figure 8-8. SRAM Unit Block Diagram



Note: Other brands and names are the property of their respective owners.

A7995-02

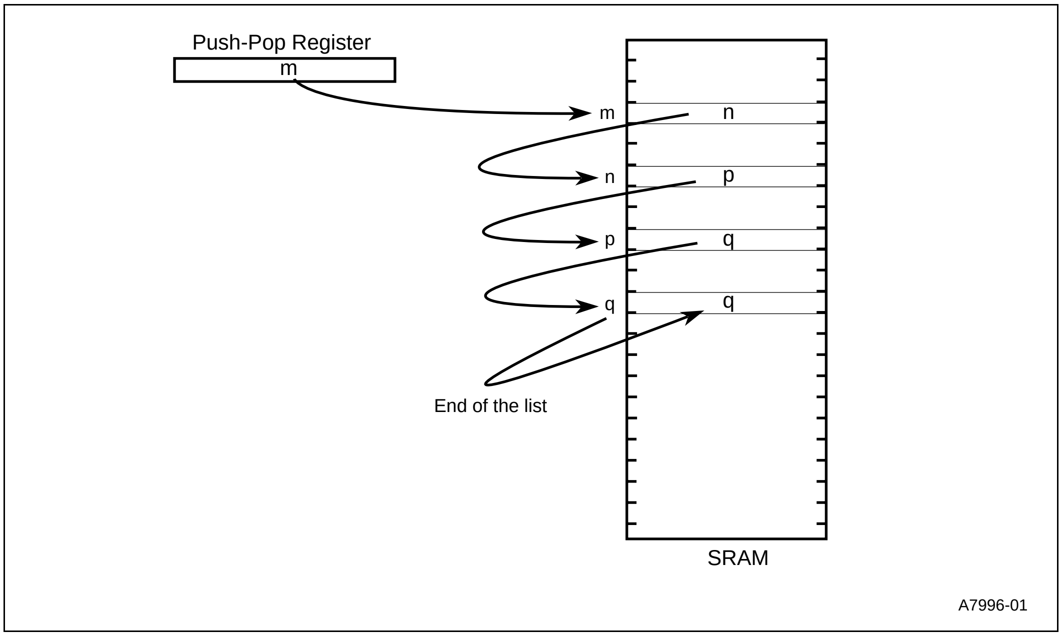
8.8.1 Push Pop Registers

Much of the activity that the SSRAM is intended to support is accessing linked lists. Each packet that is received results in one or more free (or empty) data buffers and/or buffer descriptors being pushed and/or popped from linked lists. To assist in this operation, the SRAM Unit has eight Push-Pop registers.

Each Push-pop register can be programmed to contain a pointer to the head of a linked list. When the register is read, the pointer is returned (popped) and SRAM Unit updates the register with the new pointer to the head of the list. When the register is written with a pointer (pushed), and SRAM Unit updates the register and the linked list in SSRAM. The Push-Pop Registers takes what would normally be three separate memory references (push = read-write-write, pop = read-read-write) and translate it into a single operation for the StrongARM* core and Microengines. These registers can be read and written by the StrongARM* core and any of the Microengines.

A system software architecture can be designed such that an application program executing on the StrongARM* core builds up to eight linked lists in memory that may be used as buffers and descriptors. The elements within the linked list must be at least one longword and the first longword must contain a pointer to the next element in the linked list. The linked list must be built such that the last item in the linked list contains a pointer to itself rather than a next item. The Push-Pop hardware will then pop the same pointer each time a pop operation is performed when the list is empty. Software must distinguish when there are no items on the linked list. One method might be to force the last item on the list (i.e the first item popped) to a value of 0 and then write a value of 0 to address 0. Then whenever a buffer is popped off the list, a simple compare of the immediate data equal to 0 would indicate if the buffer list has been exhausted.

Figure 8-9. Push-Pop Operations



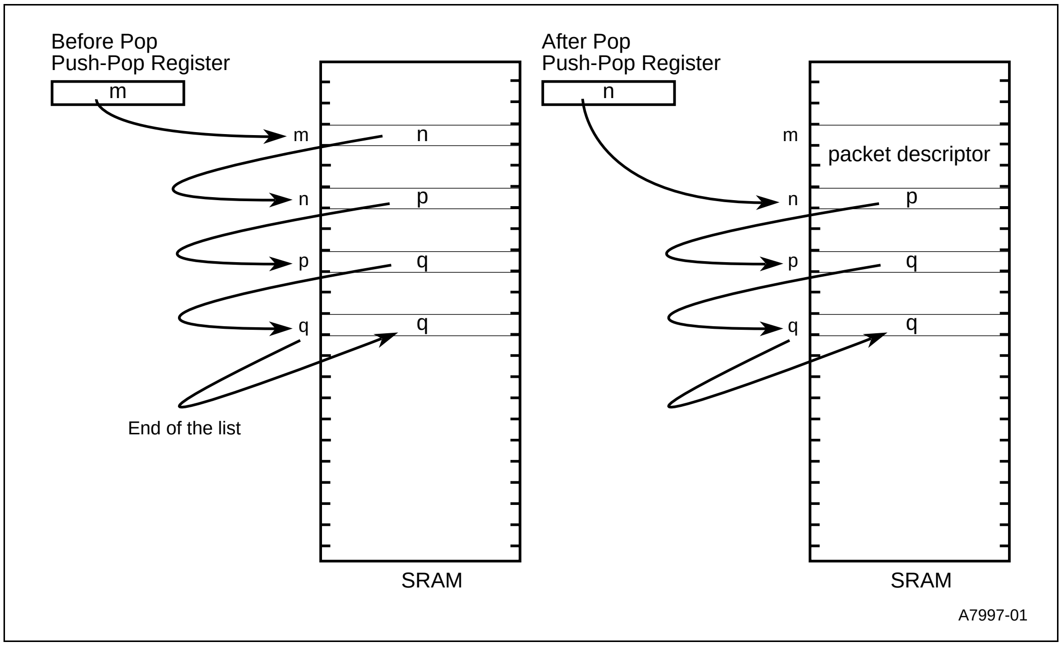
The Pop operation is described below and shown in Figure 8-10.

1. A Microengine thread or the StrongARM* core reads the Push-Pop Register value (m). A pop operation appears to the Microengines and the StrongARM* core as a single read operation to the register. The Microengines explicitly indicate one of eight Push-Pop registers in the

sram[pop] instruction. The StrongARM* core specifies the Push-Pop register by reading from one of eight unique addresses. The SRAM Unit returns the “popped” pointer to the Microengine SRAM read transfer register or on the AMBA bus as data.

2. The SRAM Unit reads the value at address m (n) and writes it to the Push-Pop Register.

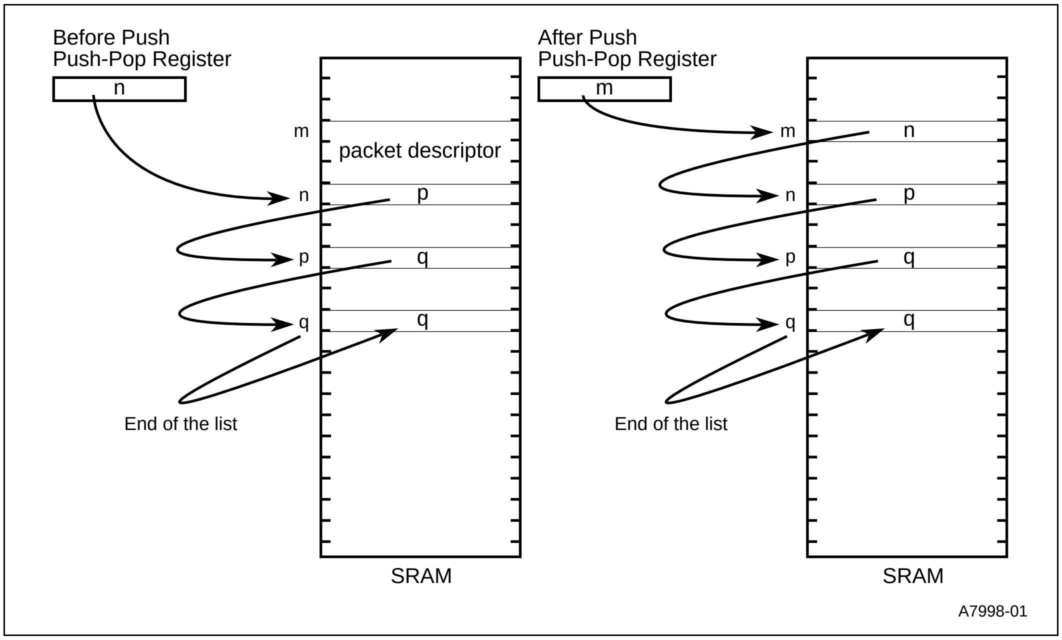
Figure 8-10. Pop Operation



The Push operation is described below and illustrated in Figure 8-11.

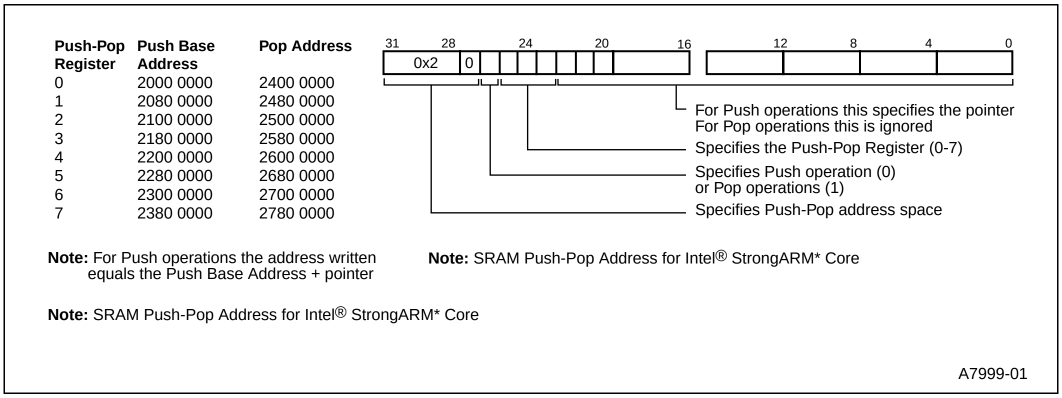
1. Microengine or StrongARM* core writes the new pointer to the Push-Pop Register value (m). A Microengine specifies the pointer in the address fields of the sram[push] instruction. The StrongARM* core specifies both the Push-Pop register and the pointer in the address (rather than the data). The eight Push-Pop registers are located at unique base addresses. The StrongARM* core adds the pointer and the base address to form the write address, and executes a longword write instruction. The data written is discarded by the SRAM Unit. A push operation appears to the Microengines and the StrongARM* core as a single write operation to the register.
2. The SRAM Unit writes the address in the Push-Pop register (n) to the address being pushed (m).

Figure 8-11. Push Operation



As noted in the procedures above, the StrongARM* core address is used to specify both the push or pop operation. Pop operations read a unique address for each queue. Push operations write to an address equal to the base address of the Push-Pop list plus the pointer to be popped onto the Push-Pop list. Figure 8-12 shows how the address is formed for Push and Pop operations.

Figure 8-12. Push-Pop Addressing



Push-Pop Register	Push Base Address	Pop Address
0	2000 0000	2400 0000
1	2080 0000	2480 0000
2	2100 0000	2500 0000
3	2180 0000	2580 0000
4	2200 0000	2600 0000
5	2280 0000	2680 0000
6	2300 0000	2700 0000
7	2380 0000	2780 0000

31
28
24
20
16
12
8
4
0
0x2
0
For Push operations this specifies the pointer
For Pop operations this is ignored
Specifies the Push-Pop Register (0-7)
Specifies Push operation (0)
or Pop operations (1)
Specifies Push-Pop address space
A7999-01

Note: For Push operations the address written equals the Push Base Address + pointer

Note: SRAM Push-Pop Address for Intel® StrongARM* Core

Note: SRAM Push-Pop Address for Intel® StrongARM* Core

From the StrongARM* core perspective, the Push-Pop Registers should not be marked as a cacheable area. A pop operation may be performed using the Read Buffer, however the data should be invalidated once it is read from the Read Buffer. Push operations may be performed through the write buffer.

8.8.2 SRAM Lock CAM

The SRAM controller maintains an 8-entry CAM. This CAM is used to protect an area in SRAM from being accessed by two or more processes (StrongARM* core and Microengine threads) at the same time. If a process wishes to protect against write access to an area in memory, it can take out a lock on the memory area. The Microengines access the Lock CAM function using the **sram** instruction while the StrongARM* core maps these command into an 8 Mbyte address space within the SRAM Unit address space (Figure 3-1).

The read_lock command performs two operations. First, it reads the address specified in the command and second, it locks the memory location. The memory location is unlocked using either the unlock command or the write_unlock command. The write_unlock command performs two operations. First is write data to the address specified in the command and second it unlocks the memory location.

When the SRAM Unit processes a read_lock command, it checks its eight entry CAM to determine if a lock has already been taken on the address specified in the command. If not, the SRAM Unit places the address into one of the eight CAM entries, read the data at the address, and returns the data to the requestor.

A read_lock command can fail under two conditions:

- A lock has already been taken on the address specified in the command
- All eight Lock CAM entries are occupied

If the read_lock command fails, the command is placed into the Read_lock Fail queue.

When an unlock or write_unlock command is serviced by the SRAM Unit, the address specified in the command is removed from the Lock CAM. In the case of the write_unlock command, the data specified by the command is also written to SSRAM. After the unlock or write_unlock command is completed, the next command serviced by the SRAM Unit is taken from the head of the Read_lock Fail queue. If the first entry in the Read_lock fail queue specifies an address that is no longer in the Lock CAM, the command is serviced. Commands continue to be taken from the Read_lock fail queue until a head of the Read_lock fail queue specifies an address that is currently in the CAM or the queue is empty. If the head of the queue specifies an address that is currently in the Lock CAM, the SRAM Unit will discontinue processing the Read_lock Fail queue and return to servicing the other command queues. Since only the first entry in the Read_lock queue is checked when an unlock occurs, a locked entry may block unlocked entries in the Read_lock Fail queue from completing (see Section 8.8.2.3).

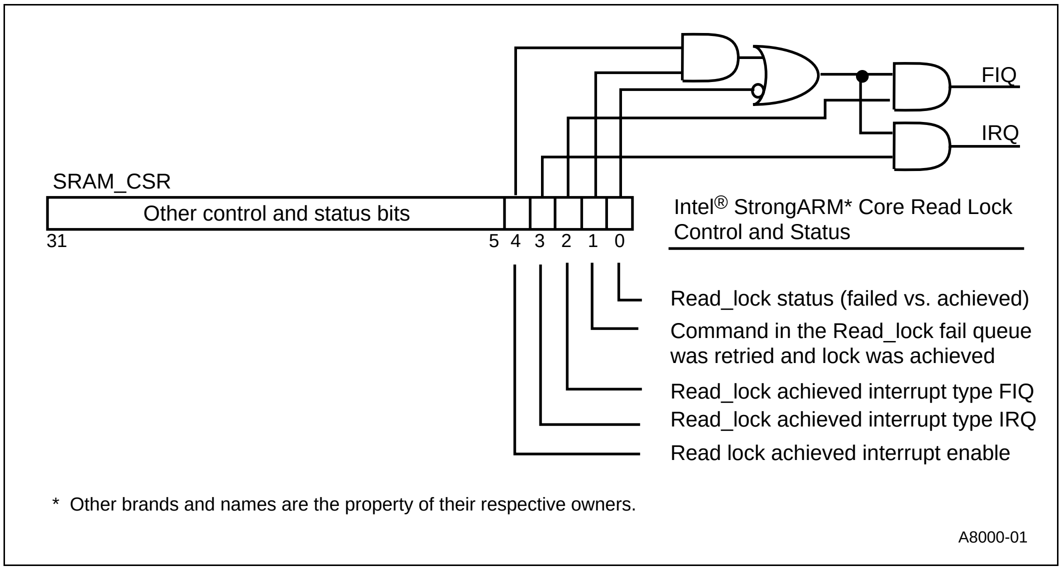
8.8.2.1 ReadLock and Microengines

The Microengines access the Read_lock CAM using the **sram** instruction and can read multiple longwords using a single read_lock command. Each thread can only have one read_lock request pending at a time. To enforce this hardware requirement, the Microengine Assembler requires the user to perform a context swap when issuing a read_lock. If the SRAM controller places a read_lock command in the Read_lock Fail queue, the thread remains swapped out until the read_lock is removed from the Read_lock fail queue and placed into the CAM. Thus, a thread is swapped in only when a read_lock is achieved.

8.8.2.2 ReadLock and StrongARM* Core

The StrongARM* core maps the read_lock, unlock, and write_unlock commands into three separate 8 Mbyte address spaces (refer to Figure 3-1). If a read_lock operation fails, the SRAM Unit always returns the read data, and a status bit is set in the SRAM_CSR to indicate that the read_lock has failed. If enabled, an interrupt can be generated when a read_lock fails, or the StrongARM* core may read the status bit immediately after reading from the read_lock address space. Data is always returned to the StrongARM* core so that the StrongARM* core can continue processing while the failed read_lock command is in the Read_lock Fail queue. When the SRAM Unit retries a failed read_lock command from the Read_lock Fail queue and the lock is achieved, another status bit is set in the SRAM_CSR register indicating that the lock was achieved. The StrongARM* core may poll this status bit or, if enabled, an interrupt can be generated when the bit is set. Once a re-tried read_lock is achieved, the StrongARM* core must read (not read_lock) the lock location once more to get the data. Figure 8-13 illustrates the logic.

Figure 8-13. Read_Lock Logic


FIQ
IRQ
SRAM_CSR
Other control and status bits
31
5 4 3 2 1 0
Intel® StrongARM* Core Read Lock Control and Status
Read_lock status (failed vs. achieved)
Command in the Read_lock fail queue was retried and lock was achieved
Read_lock achieved interrupt type FIQ
Read_lock achieved interrupt type IRQ
Read lock achieved interrupt enable
* Other brands and names are the property of their respective owners.
A8000-01

From the StrongARM* core perspective, the read_lock, write_unlock, and unlock address spaces should not be marked as a cacheable area. A read_lock operation may be performed using the Read Buffer, however the StrongARM* core must read the read_lock achieved bit and the data should be invalidated once it is read from the Read Buffer. Write_unlock and unlock operations may be performed through the Write Buffer.

A CAM entry holds a single address and does not hold any information on the size of the memory that is being locked. If a range of memory is to be locked, software needs to manage and enforce this semantic.

8.8.2.3 Maintaining Read_lock Order

The SRAM_CSR register contains a read_lock order bit (RLK) that ensures that read_locks are completed in the order in which they where issued. If this bit is set, whenever a Read_lock fails, the command is placed into the Read_lock Fail queue and all subsequent Read_lock commands are also placed into the Read_lock Fail queue regardless of whether or not the address resides in the CAM.

There is no hardware protection preventing any of the processes from writing to a locked area, so vigilance must be maintained in software to ensure a Read_lock is performed on memory areas designated as protected via the Lock CAM before the memory area is accessed.

8.8.2.4 Filling the Read_Lock Fail Queue

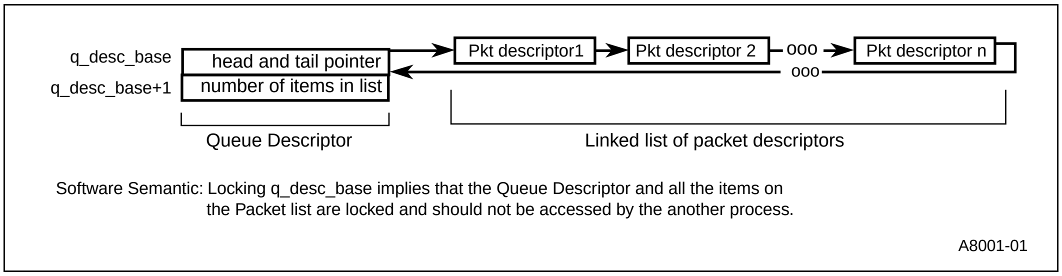
The SRAM Read_lock Fail queue holds up to a maximum of 23 failed read_lock requests that are processed as described in Section 8.8.2. If a read_lock command is issued, either from the Microengines or the StrongARM* core, and the Read_lock Fail queue is full, the commmand is placed in the first entry of the Read_lock Fail queue, overwriting any address that is currently there.

In order to prevent the wrapping of the Read_lock Fail queue, software must be written so that it monitors and controls the number of read_locks commands outstanding at any point in time.

8.8.2.5 Application Example: Read_lock

The following example describes how the read_lock and write_unlock commands can be used in a practical application. The example describes how a designer might implement a transmit queue for each physical network port. These transmit queues are maintained within SRAM and each transmit queue is defined by a queue descriptor. The queue descriptor contains a pointer to the first packet on the queue, a pointer to the last packet on the queue, and the number of packets on the queue. The packets themselves are maintained as a linked list of packet descriptors. Any of the Microengine threads and the StrongARM* core must first lock the transmit queue before modifying the linked list. This ensures two or more processes do not make changes to the list at the same time and possibly corrupt the integrity of the transmit queue. The Read_lock command can be used to read the queue descriptor data and lock the transmit queue in a single instruction. After the linked list is modified, the Write_unlock command can be used to update the queue descriptor and unlock the queue in a single instruction.

Figure 8-14. Read_Lock Application



Software Semantic: Locking q_desc_base implies that the Queue Descriptor and all the items on the Packet list are locked and should not be accessed by the another process.

8.8.3 Bit Test & Set / Bit Test & Clear

The SRAM Unit supports an SRAM bit write command that allows individual bits to be set or cleared and may also return the state of the bits prior to setting or clearing. The SRAM Unit accomplishes this by performing a Read-Modify-Write operation at the address specified based on the bit mask provided as write data.

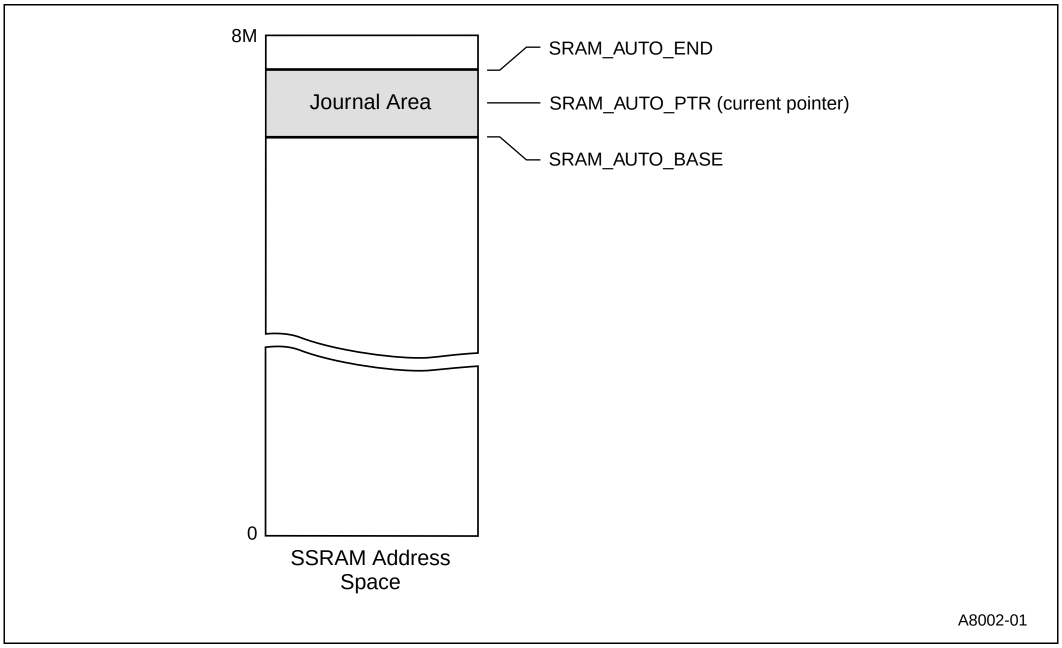
When a Microengine thread issues a SRAM bit write operation reference, the command is placed into the write/order command queue and the bit mask is provided in an SRAM write transfer register as write data. If the test operation is also specified, the same transfer register is used to deliver the pre-modified test value back to the Microengine thread for test operations.

When the StrongARM* core issues a bit operation reference, a write operation is performed on the AMBA bus and the write data contains the bit mask. The SRAM Unit performs the Read-Modify-Write operation and writes the test value to the SRAM_TEST_MOD SRAM CSR. The StrongARM* core can read this register immediately after performing the Read-Modify-Write operation since the SRAM Unit stalls the StrongARM* core until the Read-Modify-Write operation is complete. Since it is the SRAM Unit that performs bit operations, bit operations should only be performed on memory areas that are non-cacheable. To avoid coherency problems, test operations should not be performed on memory areas marked as bufferable. The reasoning is that the StrongARM* core software would not know when it can read the test value since it does not know when a write operation is completed by the write buffer.

8.8.4 SRAM Journaling

The SRAM Unit provides hardware support for maintaining a journal within SRAM. This allows the Microengines and StrongARM* core to write debug or other data to a fixed location (the SRAM_AUTO_BASE register). The SRAM Unit places the data in SRAM in a user-defined journal area. The SRAM Unit assists the Microengines in creating a journal by maintaining the pointers to the journal area in SRAM. There are three pointers that specify the start (base), end, and current position of the journal area. These are initialized via the SRAM_AUTO_BASE, SRAM_AUTO_END, and SRAM_AUTO_PTR SRAM registers. The SRAM registers are accessible by the StrongARM* core and the Microengines. When specifying the pointers, the entire address space from SRAM_AUTO_BASE to SRAM_AUTO_END must fall within the physical SSRAM address space. Unpredictable results will occur if any of this address range falls within the BootROM, Slowport, or SRAM register address spaces.

Figure 8-15. SRAM Journaling Address Space



The Journal registers should be initialized in the following order to ensure proper operation.

SRAM_AUTO_END	Writing this register clears the sram_auto_ptr register
SRAM_AUTO_BASE	Writing this also writes base address to the sram_auto_ptr register

Once these SRAM registers are initialized, a write to the SRAM_AUTO_PTR register will cause the following data to be written to SRAM at the address in the pointer register:

[31:29] Microengine ID/StrongARM* core ID
[28:27] Thread ID
[26:0] User defined data

The SRAM Unit will supply the Microengine ID and thread ID if the source is a Microengine. Otherwise, the StrongARM* core should write the entire 32 bits using an ID of 7 and a thread ID of 0 (bits [31:27] = 11100). After each write, the current pointer (SRAM_AUTO_PTR) is incremented. The SRAM Unit will automatically wrap around to the base address when the pointer reaches the end of the journal area. There is no hardware support to determine when the journal has filled and wrapped around to the beginning.

Software can determine the current pointer by reading the SRAM_AUTO_PTR register. However, each time the register is read, the current pointer is incremented.

8.9 Interfacing to the SRAM Unit

This section describes how the StrongARM* core and Microengines interface to the SRAM Unit.

8.9.1 SRAM Map

Table 8-1 shows the memory map for the SRAM address space. Notice that the StrongARM* core addresses this space using byte addressing while the Microengines access this space using the **sram** instruction and longword addressing.

Table 8-1. Memory Map for SRAM Address Space

Physical Device Space	Function	StrongARM Address Space (Byte Addressing)	Microengine sram Instruction Command	Microengine Address Space (Longword Addressing)
SlowPort	SlowPort	3840 0000 - 385F FFFF	read/write	70 0000 - 7F FFFF
SRAM CSRS	SRAM CSRs	3800 0000 - 3800 0013	read/write	60 0000 - 60 0080
SRAM	Pop Command	2400 0000 - (see Figure 8-4)	pop	00 0000 - 1F FFFF
SRAM	Push Command	2000 0000 - (see Figure 8-4)	push	00 0000 - 1F FFFF
SRAM	Bit Test & Set	1980 0000 - 19FF FFFF	bit_wr (test_and_set_bits)	00 0000 - 1F FFFF
SRAM	Bit Test & Clear	1900 0000 - 197F FFFF	bit_wr (test_and_clear_bits)	00 0000 - 1F FFFF
SRAM	Bit Set	1880 0000 - 18FF FFFF	bit_wr (set_bits)	00 0000 - 1F FFFF
SRAM	Bit Clear	1800 0000 - 187F FFFF	bit_wr (clear_bits)	00 0000 - 1F FFFF

Table 8-1. Memory Map for SRAM Address Space

Physical Device Space	Function	StrongARM Address Space (Byte Addressing)	Microengine sram Instruction Command	Microengine Address Space (Longword Addressing)
SRAM	Unlock	1600 0000 - 167F FFFF	unlock	00 0000 - 1F FFFF
SRAM	Write Unlock	1400 0000 - 147F FFFF	write_unlock	00 0000 - 1F FFFF
SRAM	Read Lock	1200 0000 - 127F 7777	read_lock	00 0000 - 1F FFFF
SRAM	Read/Write	1000 0000 - 107F FFFF	read/write	00 0000 - 1F FFFF
BootROM	BootROM	0000 0000 - 007F FFFF	read/write	20 0000 - 3F FFFF

8.9.2 Microengine SRAM Transactions

The SRAM Unit provides three command queues: Priority, Write/Order, and Read. There are three optional tokens specified by the Microengine sram instruction which determine the command queue in which the commands are submitted. The three optional tokens are priority, optimize_mem, and ordered.

The Priority Queue has a higher priority than the Read and Write/Order queues. The Write/Order queue is the default queue when an optional token is not specified. It can also be explicitly specified using the ordered optional token. Both read and write commands are placed into the Write/Order queue where they are assured to be serviced in the order in which they were issued.

The optimize_mem token causes an sram command to be placed into either the Read or Write/ Order Queue based on the type of reference. The reason commands are grouped into the Read and Order/Write queues is to improve performance by reducing the number of read-to-write bus turnaround cycles on the SRAM bus. This is accomplished by servicing multiple commands in the Read queue before switching to the Order/Write queue and visa versa. The following table shows which commands queues the sram commands are delivered to when the optimize_mem optional token is used in the instruction.

The optimize_mem token cause an SRAM command to be place into either the Read or Write/ Order queue based on the type of reference. The reason commands are grouped into the Read and Order/Write queues is to improve performance by reducing the number of read-to-write bus turnaround cycles on the SRAM bus. This is accomplished by servicing multiple commands in the Read queue before switching to the Order/Write queue and visa versa. The only commands that are segregated into the Read queue are the SRAM[Read], SRAM[Read_Lock], and SRAM[Pop] commands. All other commands are explicitly placed into the Write/Order queue.

The three command queues, the queue sizes and the optional token are listed in the Table 8-2.

Table 8-2. SRAM Command Queue Sizes

Microengine Command Queue	Queue Size	Instruction Optional Token
Read only Queue	16	optimize _mem
Order and Write Queue	24	optimize _mem or no optional token
High Priority Queue	8	priority

The command queues are sized so that in typical operation, the queues should not fill. If any of the command queues fill, a back-pressure signal notifies the Command Bus Arbiter to cease allowing SRAM commands to be sent to the SRAM Unit. Backpressure is applied when any of the queues have only six entries available. This ensures there is room in the queues for any SRAM commands that have already been granted access to the command bus by the command bus arbiter.

When the SRAM Unit processes an SRAM instruction, the SRAM Push-Pull Engine moves the data between the SRAM Unit and the SRAM transfer registers via an internal 64-bit SRAM bus. The 64-bit bus is divided in half so that 32 bits connect to SRAM Read transfer registers and 32 bits connect to SRAM Write transfer registers. If a Microengine thread chooses to be signaled upon completion of a command, the sig_done or ctx_swap optional token should be specified in the instruction.

8.9.3 StrongARM* Core SRAM Transactions

The StrongARM* core issues references to the SRAM Unit via the AMBA Bus. These requests can be generated directly by StrongARM* core Processor or the Icache, Dcache (main and mini), StrongARM* core Read Buffer , StrongARM* core Write buffer (Note that in most cases the Icache would fetch instructions from the SDRAM Unit).

The programmer should be aware that StrongARM* core references to non-cached areas will cause the StrongARM* core to stall until the SRAM Unit provides data at the AMBA read data latch.

For write operations, the StrongARM* core will stall until is gains access to the AMBA Bus which is shared by the caches and buffers. Once access is granted, the StrongARM* core will write data to the AMBA Bus Logic which translates the AMBA bus signaling into a Microengine-like command and the data is written into the AMBA Write Data Buffer so that the StrongARM* core may continue to execute. When the SRAM Command Arbiter grants access to the AMBA Bus interface, the AMBA Bus command is completed and the data is read from the AMBA Write Data Buffer. The SRAM Unit does not queue AMBA transaction and therefore only one AMBA transaction can be in progress at a time. The AMBA Write Data Buffer can hold 8 longwords, allowing it to hold up to one cache line at a time.

For the best performance, it is recommended that the programmer take advantage of the Dcaches, Read Buffer and Write Buffer. Table 8-3 lists the address spaces and the recommendations for marking these areas as cacheable or bufferable.

Table 8-3. Cacheable and Bufferable Address Spaces

Physical Device Space	Function	Dcache	Read Buffer	Write Buffer
SlowPort	SlowPort	Yes[1]	Yes[1]	Yes
SRAM CSRS	SRAM CSRs	No	Yes[2]	Yes
SRAM	Pop Command	No	Yes[2]	--
SRAM	Push Command	No	--	Yes
SRAM	Bit Test & Set	No	--	No
SRAM	Bit Test & Clear	No	--	No
SRAM	Bit Set	No	--	Yes
SRAM	Bit Clear	No	--	Yes
SRAM	Unlock	No	--	Yes
SRAM	Write Unlock	No	--	Yes

Table 8-3. Cacheable and Bufferable Address Spaces

Physical Device Space	Function	Dcache	Read Buffer	Write Buffer
SRAM	Read Lock	No	Yes[2]	--
SRAM	Read/Write	Yes	Yes	Yes
BootROM	BootROM	Yes		

[1] The designer should evaluate the ramifications based on the specific SlowPort device.

[2] Data should be invalidated immediately after the StrongARM* core reads the data from the read buffer.

8.9.4 SRAM Burst Count

A burst count is provided to the SRAM Unit by the following IXP1200 components:

- Microengines
- StrongARM* core
- StrongARM* core Icache and Dcache (main and mini)
- StrongARM* core Read Buffer
- StrongARM* core Write buffer

The Microengines specify the burst count in the instruction. Burst counts generated by the StrongARM* core are accepted by the AMBA Bus Logic which generates a single Microengine-like command that includes the burst count. Table 8-4 lists the burst sizes generated by the Microengines and StrongARM* core.

Table 8-4. Burst Sizes

Source	Operation	Longword Burst Size	Comments
Microengines	read or write	1 to 8	sram instruction
	read or write	1 to 16	sram instruction using indirect_ref
StrongARM* core	read or write	1	A burst size of 1 that specifies byte or word writes requires the SRAM Unit to perform a Read-Modify-Write operation
Dcache	read	8	Line fills
	write	4 or 8	Line and half line evictions
Icache	read	8	Line fills (Icache fills are typically from SDRAM)
Read Buffer	read	1,4,8	Controlled via the Coprocessor 15 READ_BUFFER_OPERATIONS register (Register 9)
Write Buffer	write	1,2,3,4,8	A burst size of 1 that specifies byte or word writes requires the SRAM Unit to perform a Read-Modify-Write operation Write buffer performs burst counts greater than 1 only for the StrongARM store multiple instructions

8.9.5 SRAM Command Service Priority

The SRAM Unit services SRAM reference commands from five sources:

- StrongARM* core
- Three Microengine Queues (Read, Write/Order, Priority)
- Read_lock Failed Queue

The Service Priority Machine decides from which source to get the next command. The decision policy is based on the table below. The priority is dependent on the command that was previously serviced.

Table 8-5. SRAM Command Service Priority

Priority	Previous Requested SRAM Instruction		
	Write-Unlock or Unlock	**Read, Read-Lock, or Pop**	**Write, Bit_Set, Bit_Clr, or Push**
1	Read Lock Fail queue	AMBA requests (StrongARM* core)	AMBA requests (StrongARM* core)
2	AMBA requests (StrongARM* core)	High priority	High priority
3	High priority	Read queue[1]	Write/Order queue[1]
4	Write/Order queue[1]	Write/Order queue[1]	Read queue[2]
5	Read queue[2]		

[1] Services four commands from the Write/Order queue regardless of whether it is a read or a write. After four order commands are serviced, continue to service commands in the Write/Order queue until either a read command is serviced (in which case it switches to the Read queue) or a maximum of eight commands are serviced from the Write/Order queue (in which case it switches to the Read queue regardless of the last command serviced).

[2] Services eight commands from the Read queue then it switches to the Write/Order queue

If the previous command was a write_unlock or unlock command, the Read_lock Fail queue has the highest priority. This queue holds the highest priority until a lock cannot be granted or until the queue is drained. Note that the Service Priority Machine only checks the head of the queue for a locked entry.

A timeslot exists between two consecutive AMBA bus transactions such that a pending Microengine command may be serviced between two consecutive AMBA Bus commands. Note that StrongARM* core commands that generate burst counts are treated single commands. Refer to Section 8.9.4 for more on StrongARM* core burst counts.

8.9.6 Read-Modify-Write

The Read-Modify-Write function allows individual bytes to be written to SRAM devices. To accomplish this, the SRAM Unit firsts reads data from a longword address, modifies the specified bytes, and then writes the modified longword back to the SRAM device. These three steps are performed atomically. The SRAM Unit supports Read-Modify-Write operations for the Microengines and the StrongARM* core.

The StrongARM* core performs Read-Modify-Write operations for bit write operations (bit test, set, and clear) and byte and word writes to non-cached memory. Read-Modify-Write operations are performed automatically by the SRAM Unit.

The Microengines perform Read-Modify-Write operations on the SRAM longword addresses using the indirect reference option. The reference count specified in the instruction must be a value of 1 and the data to be written is specified in an SRAM write transfer register. A mask specified by the indirect reference determines which bytes are modified at the longword address.