



Intel[®] 80321 I/O Processor DMA and AAU Library

APIs and Testbench White Paper

July 2003



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 Intel® Embedded I/O Processors 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>.

Copyright© Intel Corporation, 2003

AlertVIEW, i960, AnyPoint, AppChoice, BoardWatch, BunnyPeople, CablePort, Celeron, Chips, Commerce Cart, CT Connect, CT Media, Dialogic, DM3, EtherExpress, ETOX, FlashFile, GatherRound, i386, i486, iCat, iCOMP, Insight960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel ChatPad, Intel Create&Share, Intel Dot.Station, Intel GigaBlade, Intel InBusiness, Intel Inside, Intel Inside logo, Intel NetBurst, Intel NetStructure, Intel Play, Intel Play logo, Intel Pocket Concert, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel TeamStation, Intel WebOutfitter, Intel Xeon, Intel XScale, Itanium, JobAnalyst, LANDesk, LanRover, MCS, MMX, MMX logo, NetPort, NetportExpress, Optimizer logo, OverDrive, Paragon, PC Dads, PC Parents, Pentium, Pentium II Xeon, Pentium III Xeon, Performance at Your Command, ProShare, RemoteExpress, Screamlane, Shiva, SmartDie, Solutions960, Sound Mark, StorageExpress, The Computer Inside, The Journey Inside, This Way In, TokenExpress, Trillium, Vivonic, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Contents

1.0	Introduction	9
1.1	Demonstrate Libraries Testbench Menu.....	9
2.0	Functional Overview	10
2.1	Library Usage Models.....	10
3.0	Intel® 80321 I/O Processor Component Overview	11
3.1	Data Paths and Components.....	11
3.2	DMA Controller	12
3.2.1	Overview	12
3.2.2	Theory of Operation.....	13
3.3	AAU Controller	14
3.3.1	Overview	14
3.3.2	Theory of Operation.....	15
3.4	Intel® XScale™ Microarchitecture	16
3.4.1	Overview	16
3.4.2	Intel® XScale™ Microarchitecture Memory Management	17
3.4.3	Interrupt Controller.....	17
3.4.4	Data Cache.....	17
4.0	Data Cache Policy Mechanics	18
4.1	Introduction	18
4.2	Page Tables.....	18
4.3	Data Cache and Write Policy	20
5.0	Optimization of Descriptor Processing Software	21
6.0	Library	22
6.1	Ecosystem: Application and AAU/DMA Library APIs.....	22
6.1.1	Flow Sequence Description: AAU and DMA Library.....	23
6.1.1.1	Initialization	23
6.1.1.2	Request.....	23
6.1.1.3	Post Transaction Cleanup.....	23
6.1.1.4	Terminate.....	24
6.2	Memory Map for Intel® 80321 I/O Processor: Virtual Verses Physical for Redboot	25
6.2.1	Intel® 80321 I/O Processor Memory Map	25
6.2.2	Redboot* Intel® IQ80321 Memory Map	26
6.2.3	Redboot Intel® IQ80321 Physical Memory Map - Visual	27
6.2.4	Redboot Intel® IQ80321 Virtual Memory Map - Visual	28
7.0	Library and Test Bench File Organization and Compilation	29
7.1	Folder and File Organization	29
7.1.1	/Lib Files:	29
7.1.2	/Bench Files:	30
7.2	Instruction to Build and Run.....	30

B.2.14	int lib_q_put(AauDma_80321_Type * mgr, void * frame, enum hardware engine).....	47
B.3	Functions Included in fiq_irq_80321.h and fiq_irq_80321.c	48
B.3.1	void intHandlerDetach(void)	48
B.3.2	void callintHandlerAttach(void)	48
B.3.3	void intHandlerAttach(void (*irq)(void),void (*fiq)(void)).....	48
B.3.4	void lib_irq_handler(void)__attribute__((__naked__))	49
B.3.5	void lib_fiq_handler(void)__attribute__((__naked__))	49
B.3.6	void lib_dma_int_setup(void) and void lib_aau_int_setup(void)	49
B.3.7	void lib_dma_int_returnstate(void) and void lib_aau_int_returnstate(void).....	49
C	Testbench: Data Structures	50
C.1	bench.h	50
D	Test Bench Library Function Prototypes	51
D.1	bench.c	51
D.1.1	int main(void);	51
D.1.2	void print_title(enum build b).....	51
D.1.3	void generate_src_dst(void);	51
D.2	lib_demo_cases.c	52
D.2.1	void dma_lib_demo(void) and void dma_lib_demo_witherror(void)	52
D.2.2	void aau_lib_demo(void) and void aau_lib_demo_witherror(void)	53
E	Related Documents	54



Revision History

Date	Revision	Description
July 2003	001	Initial Release.

1.0 Introduction

Intel provides Intel[®] 80321 I/O processor¹ (80321) customers an optimized turnkey Library solution for DMA and AAU applications to provide a fast development ramp. The DMA/AAU Library synergistically combines with existing Intel collateral (See [Section E, “Related Documents”](#) on page 54 for related web documents).

The turnkey Optimized AAU and DMA Library consists of:

- DMA and AAU register set .h files.
- Functions to set Data Cache Policy for specified memory pages.
- Integrated Descriptor Handling.
- Required macros.
- Interrupt handler with setup of Interrupt Controller.
- Rules for optimization.
- Test bench demonstrating Library implementation.

1.1 Demonstrate Libraries Testbench Menu

Below is the Library menu. The testbench provides menu driven, test cases implementing the Libraries. Note that once ECC is turned OFF, the board needs to be rebooted to turn ECC back on.

1. DMA Library DEMO.
2. DMA Library DEMO showing error detection (errors intentionally introduced).
3. AAU Library DEMO.
4. AAU Library DEMO showing error detection (errors intentionally introduced).
5. Turn global ECC OFF, default is ON.

1. ARM* architecture compliant.

2.0 Functional Overview

The Intel® 80321 I/O processor features include Direct Memory Access Control Unit (DMA), Application Accelerator Unit (AAU) and an Intel® XScale™ core. One of the important features of the Intel® XScale™ core, is the ability to customize cache policy by memory page.

The base DMA/AAU 80321 Library provides an hardware interface/memory map:

- DMA Controller Unit.
- Application Accelerator Unit.
- Intel® XScale™ Microarchitecture.

While, the complete Library solution provides:

- Integrated descriptor handling (one set of APIs) for DMA Channel 0 and Channel 1 and the AAU:
 - pre-runtime allocation/alignment of AAU and DMA descriptors.
 - Descriptor Free Stacks and Post Queues.
 - Free descriptors by pointer allocation.
 - Descriptor reclamation from descriptor chain.
- Customized cache policy for descriptor processing and data memory regions.
- Interrupt controller setup for DMA and AAU transactions including chaining in of interrupt handlers.
- Library Demos: Present full implementation of the DMA and AAU Libraries functionality.

2.1 Library Usage Models

The DMA/AAU Libraries can be flexibly used.

- Implemented as a turnkey solution.
- Cafeteria style use of any or all components, code, methodologies and document references.

3.0 Intel® 80321 I/O Processor Component Overview

This section provides an overview of the AAU Controller, DMA Controller and Intel® XScale™ core and Transaction Data Paths.

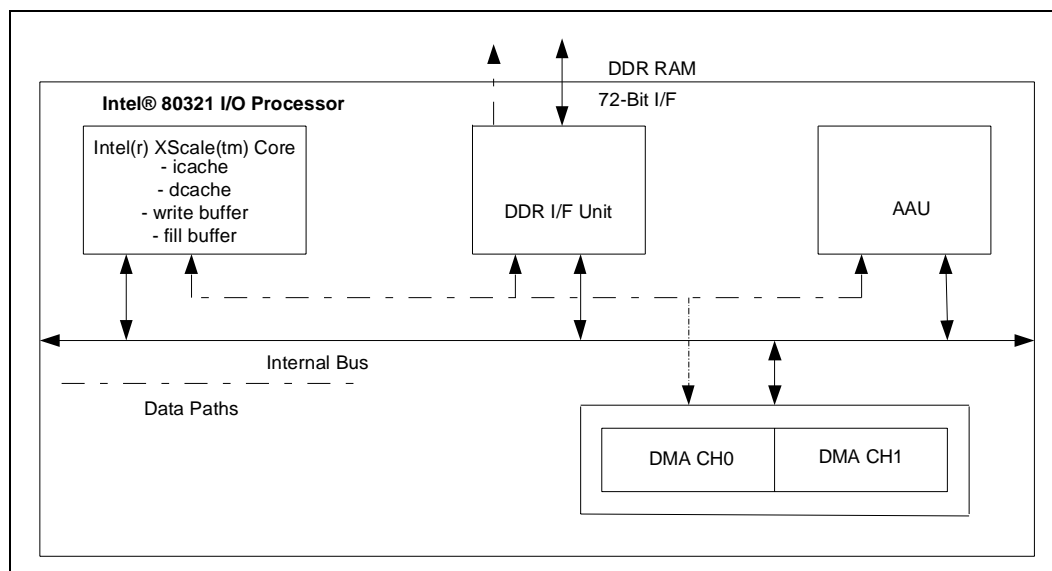
3.1 Data Paths and Components

Figure 1 provides an overview of the 80321 with the data paths and DMA/AAU relevant components: AAU Controller, DMA Controller and Intel® XScale™ core. Here is a description of data flow.

- Intel® XScale™ microarchitecture Descriptor Processing:
 - Code is run from RAM by the Intel® XScale™ microarchitecture using the standard fetch, decode and execute model.
 - Descriptors are written to RAM.
 - Memory mapped AAU/DMA registers written to initiate transaction. DMA or AAU controller, using the pointer to memory, reads the descriptor values from RAM into DMA or AAU Controller registers. Based on descriptor values, the AAU or DMA Controller begins execution.
- AAU or DMA Controller Operations.
 - For a AAU or DMA transfer: the data is transferred first into the AAU or DMA block and then transferred to destination.

In terms of optimization, the primary goal is to minimize traffic on the bus. The programmer can support this objective by minimizing traffic for descriptor processing between RAM and the Intel® XScale™ core. The bus transactions created by the DMA and AAU engines are handled by the controllers.

Figure 1. Intel® 80321 I/O Processor Memory-to-Memory DMA and AAU Transfers Diagram



3.2 DMA Controller

3.2.1 Overview

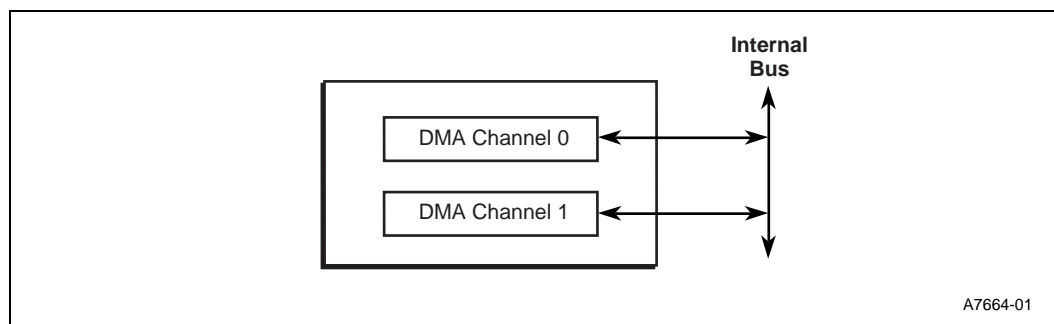
The DMA Controller provides low-latency, high-throughput data transfer capability. The DMA Controller optimizes block transfers of data between the PCI bus and the local processor memory. The DMA is an initiator on the Internal bus with burst capabilities providing a maximum throughput of 1064 Mbytes/sec. when the PCI bus is operating in 64-bit/133 MHz PCI-X mode.

The DMA Controller hardware is responsible for executing data transfers and for providing the programming interface. The DMA Controller features:

- Two Independent Channels
- Two 1-KByte queues in Ch0 and Ch1
- Utilization of the Intel® 80321 I/O processor (80321) Memory Controller Interface
- 2^{32} addressing range on the Internal Bus interface
- 2^{64} addressing range on the PCI interface by using PCI Dual Address Cycle (DAC)
- Hardware support for unaligned data transfers for both the PCI bus and the Internal Bus
- Up to 1064 Mbytes/s burst support for the PCI bus in the PCI-X mode and 1600 Mbytes/s for the Intel® 80321 I/O processor internal bus
- Direct addressing to and from the PCI bus
- Memory to Memory DMA transfer mode
- Fully programmable directly from the internal bus
- Support for automatic data chaining for gathering and scattering of data blocks
- 64-bit/200 MHz Intel® 80321 I/O processor internal bus interface.

Figure 2 shows the connections of the DMA channels to the Internal Bus.

Figure 2. DMA Controller

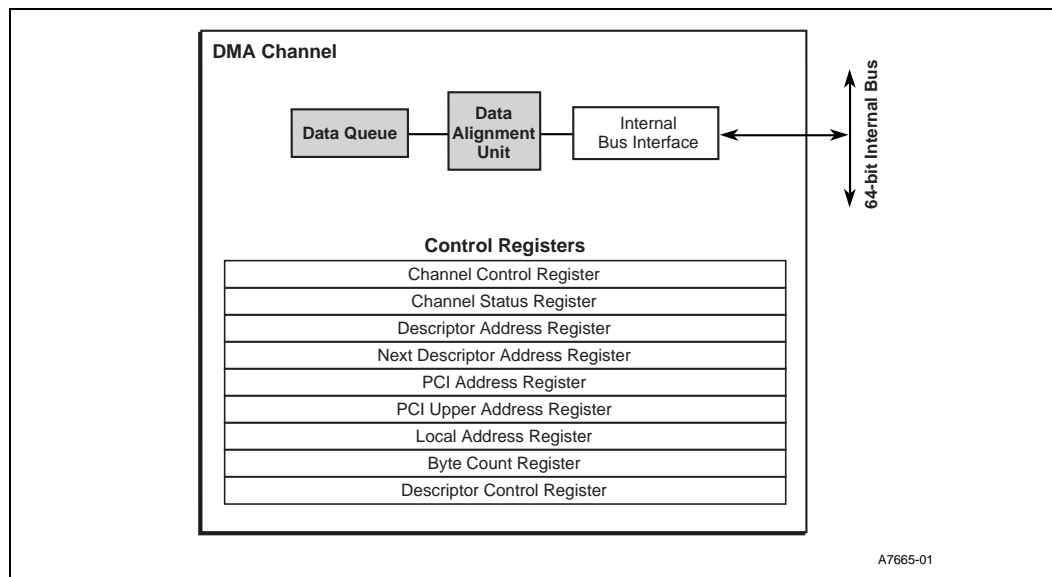


3.2.2 Theory of Operation

The DMA Controller provides two channels of high throughput PCI-to-Memory or Memory-to-Memory transfers. Channels 0 and 1 transfer blocks of data between the PCI bus and Intel® 80321 I/O processor local memory.

The two DMA channels operate identically. Each channel has an internal bus interface. Figure 3 shows the block diagram for one channel of the DMA Controller.

Figure 3. DMA Channel Block Diagram



Each DMA channel uses direct addressing for both the PCI bus and the internal bus. It supports data transfers to and from the full 64-bit address range of the PCI bus. This includes 64-bit addressing using the PCI DAC command. The channel provides a special register which contains the upper 32 address bits for the 64-bit address. The DMA channels do not support data transfers that cross a 32-bit address (4 GByte) boundary.

When Memory-to-Memory transfer mode is enabled, the DMA can be programmed to transfer data across the entire 4 Gbyte memory space on the Internal Bus. This includes but is not limited to transferring data to and from the Intel® 80321 I/O processor’s Memory Controller (MCU), and from the Peripheral Bus Interface (PBI) to the MCU.

Both the PCI interface and the internal bus interface support large burst lengths up to 4 KBytes.

The channel programming interface is accessible from the internal bus through a memory-mapped register interface. Each channel is programmed independently and has its own set of registers. A normal DMA transfer is configured by writing the source address, destination address, number of bytes to transfer, and various control information into a chain descriptor in Intel® 80321 I/O processor local memory.

Each channel supports chaining. Chain descriptors that describe one DMA transfer each can be linked together in Intel® 80321 I/O processor local memory to form a linked list. Each chain descriptor contains all the necessary information for transferring a block of data in addition to a pointer to the next chain descriptor. The end of the chain is indicated when the pointer is zero.

Each channel contains a hardware data alignment unit. This unit enables data transfers from or to unaligned addresses in either the PCI address space or the I/O processor local address space. All combinations of unaligned data are supported with the data alignment unit.

3.3 AAU Controller

This section describes the integrated Application Accelerator (AA) Unit. The operation modes, setup, external interface, and implementation of the AA unit are detailed in this chapter.

3.3.1 Overview

The Application Accelerator provides low-latency, high-throughput data transfer capability between the AA unit and 80321 local memory. It executes data transfers to and from 80321 local memory, check for all-zero result across local memory blocks, perform memory block fills, and provides the necessary programming interface. The Application Accelerator performs the following functions:

- Transfers data (read) from memory controller.
- Performs an optional boolean operation (XOR) on read data.
- Transfers data (write) to memory controller.
- Check for All-zero result across local memory blocks.
- Perform memory block fills.

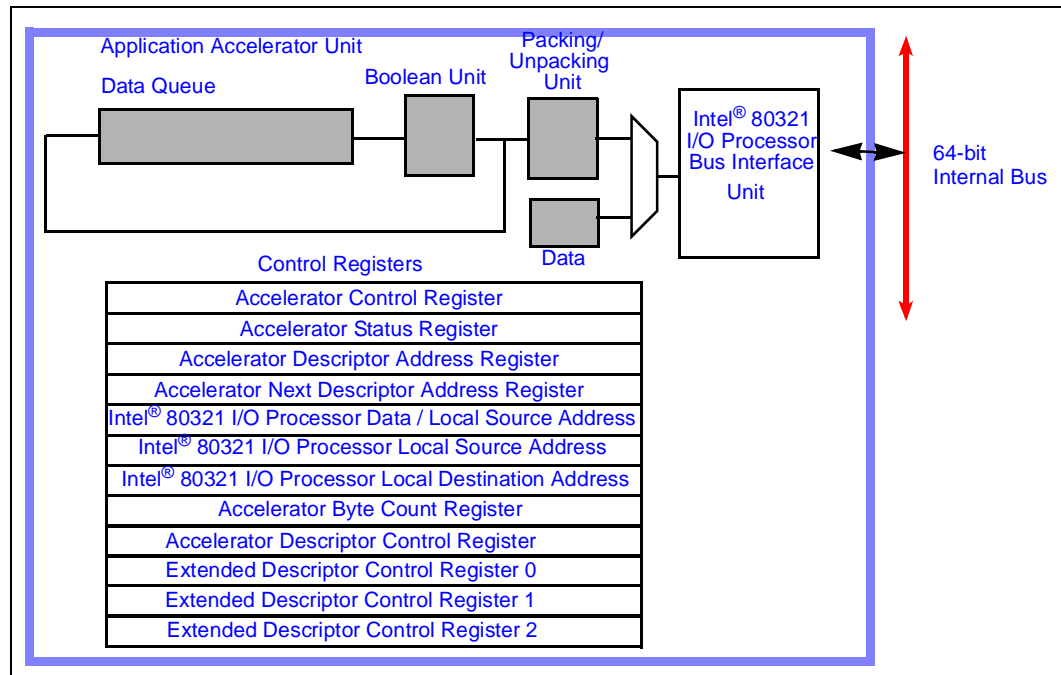
The AA unit features:

- 1Kbyte/512-Byte store queue.
- Utilization of the 80321 memory controller Interface.
- 2^{32} addressing range on the 80321 local memory interface.
- Hardware support for unaligned data transfers for the internal bus.
- Fully programmable from the 80321.
- Support for automatic data chaining for gathering and scattering of data blocks.
- Support for writing a constant value to a memory block (block fill).
- Support for writing descriptor status to local memory.

3.3.2 Theory of Operation

The Application Accelerator is a master on the internal bus and performs data transfers to and from local memory. It does not interface to the PCI bus. AA uses direct addressing for memory controller. The AA implements XOR algorithm in hardware. It performs XOR operation on multiple blocks of source (incoming) data and stores result back in 80321 local memory. The source and destination addresses are specified through chain descriptors resident in 80321 local memory. The AA can also check for all-zero result across local memory blocks or fill a memory block with arbitrary data. Figure 4 shows a block diagram of the AA unit. The AA can also perform memory-to-memory transfers of data blocks controlled by 80321 memory controller unit.

Figure 4. Application Accelerator Block Diagram



The Application Accelerator programming interface is accessible from the internal bus through a memory-mapped register interface. Data for the XOR operation is configured by writing the source addresses, destination address, number of bytes to transfer, and various control information into a chain descriptor in local memory. Chain descriptors are described in detail in Section 6.3.2, “Chain Descriptor Format (Four Source Addresses)” in the *Intel® 80321 I/O Processor Developer’s Manual*. The Application Accelerator unit contains a hardware data packing and unpacking unit. This unit enables data transfers from and to unaligned addresses in Intel® 80321 I/O processor local memory. All combinations of unaligned data are supported with the packing and unpacking unit. Data is held internally in the Application Accelerator until ready to be stored back to local memory. This is done using a 1KByte/512Byte holding queue. Data to be written back to 80321 local memory can either be aligned or unaligned.

Each chain descriptor contains the necessary information for initiating an XOR operation on blocks of data specified by the source addresses. The Application Accelerator unit supports chaining. Chain descriptors that specify the source data to be XORed can be linked together in 80321 local memory to form a linked list.

Similar to XOR operations, AA can be programmed to check for All-zero result across multiple memory blocks specified by chain descriptors. In addition, AA can also be used to perform memory block fills.

3.4 Intel® XScale™ Microarchitecture

3.4.1 Overview

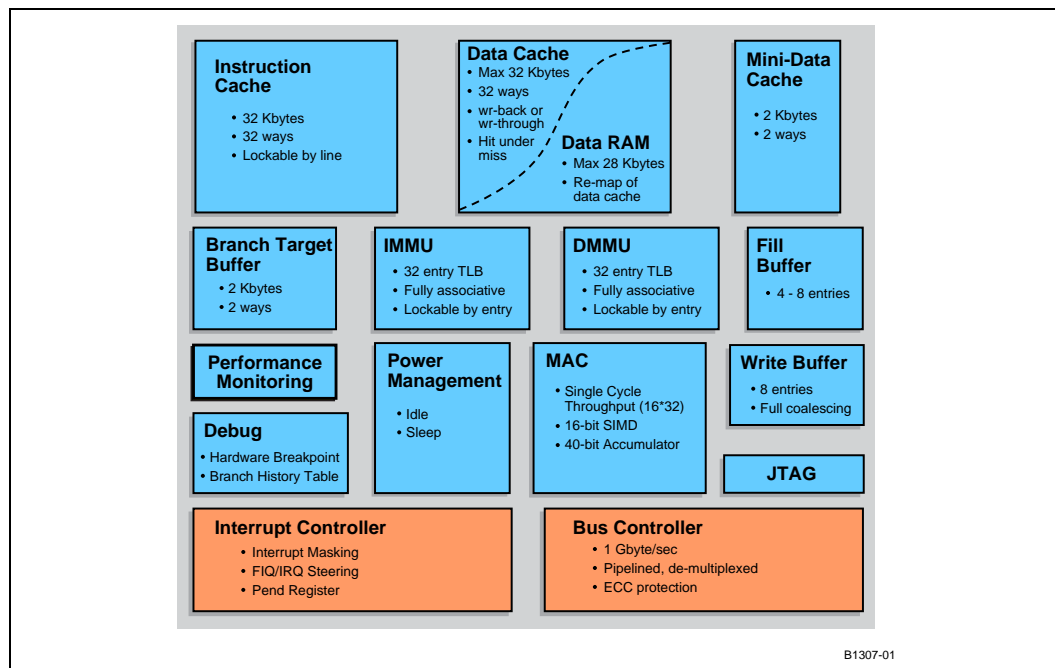
The Intel® XScale™ microarchitecture (compliant with ARM* Architecture V5TE), is designed for high-performance and low-power; leading the industry in mW/MIPs. The Intel® XScale™ microarchitecture integrates a bus controller and an interrupt controller around a core processor, with intended embedded markets such as: handheld devices, networking, remote access servers, etc. This technology is ideal for internet infrastructure products such as network and I/O processors, where ultimate performance is critical for moving and processing large amounts of data quickly.

The Intel® XScale™ microarchitecture incorporates an extensive list of architecture features that allows it to achieve high performance. This rich feature set allows programmers to select the appropriate features that obtains the best performance for their application. Many of the architectural features added to Intel® XScale™ microarchitecture help hide memory latency which often is a serious impediment to high-performance processors. This includes:

- the ability to continue instruction execution even while the data cache is retrieving data from external memory.
- a write buffer.
- write-back caching.
- various data cache allocation policies which can be configured different for each application.
- cache locking.
- and a pipelined external bus.

All these features improve the efficiency of the external bus.

Figure 5. Intel® 80200 Processor based on Intel® XScale™ Microarchitecture Features



3.4.2 Intel® XScale™ Microarchitecture Memory Management

The Intel® XScale™ microarchitecture implements the Memory Management Unit (MMU) Architecture specified in the *ARM Architecture Reference Manual*. The MMU provides access protection and virtual to physical address translation.

The MMU Architecture also specifies the caching policies for the instruction cache and data memory. These policies are specified as page attributes and include:

- identifying code as cacheable or non-cacheable.
- selecting between the mini-data cache or data cache.
- write-back or write-through data caching.
- enabling data write allocation policy.
- and enabling the write buffer to coalesce stores to external memory.

See the *Intel® XScale™ Microarchitecture Programmer's Reference Manual* for more detail.

3.4.3 Interrupt Controller

An interrupt controller is implemented on the Intel® XScale™ microarchitecture that provides masking of interrupts and the ability to steer interrupts to FIQ or IRQ. It is accessed through Coprocessor 13 registers. See the *Intel® XScale™ Microarchitecture Programmer's Reference Manual* for more detail.

3.4.4 Data Cache

The Intel® XScale™ microarchitecture implements a 32-Kbyte, a 32-way set associative data cache and a 2-Kbyte, 2-way set associative mini-data cache. Each cache has a line size of 32 bytes, supports write-through or write-back caching.

The data/mini-data cache is controlled by page attributes defined in the MMU Architecture and by coprocessor 15.

See the *Intel® XScale™ Microarchitecture Programmer's Reference Manual* discusses all this in more detail.

4.0 Data Cache Policy Mechanics

4.1 Introduction

Data cache policies can be customized for each memory page allowing different software operations to interface with a tailored Data Cache Policy. Data Cache policies are setup and controlled using the Intel® XScale™ Microarchitecture page tables. Also see *ARM Architecture Reference Manual* for a description of the Page Table Translation process.

4.2 Page Tables

The Intel® XScale™ microarchitecture extends the page attributes defined by the C and B bits in the page descriptors with an additional X bit. This bit allows four more attributes to be encoded when X=1. These new encodings include allocating data for the mini-data cache and write-allocate caching. A full description of the encodings can be found in the *Intel® XScale™ Microarchitecture Programmer's Reference Manual*.

The Intel® XScale™ microarchitecture retains ARM definitions of the C and B encoding when X = 0, which is different than the first generation Intel® StrongARM products. The memory attribute for the mini-data cache has been moved and replaced with the write-through caching attribute.

When write-allocate is enabled, a store operation that misses the data cache (cacheable data only) generates a line fill. When disabled, a line fill only occurs when a load operation misses the data cache (cacheable data only).

Write-through caching causes all store operations to be written to memory, whether they are cacheable or not cacheable. This feature is useful for maintaining data cache coherency.

These attributes are programmed in the translation table descriptors, which are highlighted in [Table 1, “First-level Descriptors” on page 19](#), [Table 2, “Second-level Descriptors for Coarse Page Table” on page 19](#) and [Table 3, “Second-level Descriptors for Fine Page Table” on page 19](#). Two second-level descriptor formats have been defined for Intel® XScale™ microarchitecture, one is used for the coarse page table and the other is used for the fine page table.

Note: P bit is not implemented.

Table 1. First-level Descriptors

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																				
SBZ														0	0					
Coarse page table base address										P	Domain			SBZ		0	1			
Section base address					SBZ			TEX		AP		P	Domain			0	C	B	1	0
Fine page table base address										SBZ		P	Domain			SBZ		1	1	

Table 2. Second-level Descriptors for Coarse Page Table

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																				
SBZ														0	0					
Large page base address						TEX			AP3		AP2		AP1		AP0		C	B	0	1
Small page base address								AP3		AP2		AP1		AP0		C	B	1	0	
Extended small page base address							SBZ		TEX			AP		C	B	1	1			

Table 3. Second-level Descriptors for Fine Page Table

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																				
SBZ														0	0					
Large page base address						TEX			AP3		AP2		AP1		AP0		C	B	0	1
Small page base address								AP3		AP2		AP1		AP0		C	B	1	0	
Tiny Page Base Address							TEX			AP		C	B	1	1					

The P bit controls ECC.

Note: P bit is not implemented.

The TEX (Type Extension) field is present in several of the descriptor types. In the Intel® XScale™ microarchitecture, only the LSB of this field is used; this is called the X bit.

A Small Page descriptor does not have a TEX field. For these descriptors, TEX is implicitly zero; that is, they operate as when the X bit had a '0' value.

The X bit, when set, modifies the meaning of the C and B bits. Description of page attributes and their encoding can be found in [Chapter 3, "Memory Management"](#).

4.3 Data Cache and Write Policy

All of these descriptor bits affect the behavior of the Data Cache and Write Buffer.

When the X bit for a descriptor is zero, the C and B bits operate as mandated by the ARM architecture. This behavior is detailed in [Table 4](#).

When the X bit for a descriptor is one, the C and B bits' meaning is extended, as detailed in [Table 5](#).

Table 4. Data Cache and Buffer Behavior when X = 0

C B	Cacheable?	Bufferable?	Write Policy	Line Allocation Policy	Notes
0 0	N	N	-	-	Stall until complete ^a
0 1	N	Y	-	-	
1 0	Y	Y	Write Through	Read Allocate	
1 1	Y	Y	Write Back	Read Allocate	

- a. Normally, the processor continues executing after a data access when no dependency on that access is encountered. With this setting, the processor stalls execution until the data access completes. This guarantees to software that the data access has taken effect by the time execution of the data access instruction completes. External data aborts from such accesses are imprecise (but see [Section 2.3.4.4](#) for a method to shield code from this imprecision).

Table 5. Data Cache and Buffer Behavior when X = 1

C B	Cacheable?	Bufferable?	Write Policy	Line Allocation Policy	Notes
0 0	-	-	-	-	Unpredictable -- do not use
0 1	N	Y	-	-	Writes do not coalesce into buffers ^a
1 0	(Mini Data Cache)	-	-	-	Cache policy is determined by MD field of Auxiliary Control register ^b
1 1	Y	Y	Write Back	Read/Write Allocate	

- a. Normally, bufferable writes can coalesce with previously buffered data in the same address range
 b. See [Section 7.2.2](#) for a description of this register

5.0 Optimization of Descriptor Processing Software

Performance benefits can be gained by tuning the software performing descriptor processing. Central to the tuning process is using the feature of the Intel[®] XScale[™] microarchitecture that allows separate Data Cache policies for individual memory pages.

When tuning custom applications keep in mind:

- In performance experiments, better performance was achieved by prebuilding linked list verses using the append resume function. Constraints of custom applications may or may not allow the prebuilding of lists.
- When append is used, the uncached unbuffered memory mapping for the last descriptor is used to append the next descriptor. This eliminates a 16 byte cache flush (since there are two dirty bits for each cache line each for 16 bytes).
- Descriptor processing is open to software optimization while DMA and AAU transfer runtimes are the same for all approaches. Also, the DMA and AAU engine run concurrently with the Intel[®] XScale[™] core.
- When multiple data cache policies are comparable in runtime performance, choosing the policy that has data caches OFF is preferable, since uncached memory regions do not allocate cachelines and impact costly cacheline evictions to ram.
- GNUPro Toolset Compiler optimization Level -O2 is best for performance in the performance test cases.
- For all performance cases, increasing the descriptor byte count field does not alter the Policy providing the best performance.
- In a custom application, when caches are ON for a memory region and the preload can be that placed in a compute bound section of code, preload provides significant benefit.
- Custom applications performing 80321 DMA and AAU descriptor processing may show different comparative results, when different concurrent applications produce different Data Bus and Data Cache interactions. Examples:
 - High usage of data cache by application could result in cacheline evictions. In this case uncached descriptors may show better performance.
 - High bus traffic by a concurrent application may alter the comparative performance of cached Policies
 - Interleaving of data bus transactions while building descriptors in RAM may reduce opportunities for coalescing.
- Typically data regions are set as Policy '000'. This eliminates synchronizing the source and destination data with the Data Cache.

6.0 Library

6.1 Ecosystem: Application and AAU/DMA Library APIs

APIs provide the following hardware and software interfaces.

80321 Hardware Interfaced:

- Intel® XScale™ microarchitecture data cache.
- Intel® XScale™ microarchitecture Page Tables.
- DMA Controller Unit.
- AAU Controller Unit.
- Interrupt Controller.

Library Software Grouped by Category:

- Initialization
 - Hardware
 - Descriptor Management Data Structures
- Operations
 - AAU and DMA Descriptor Management: requests, descriptor reclamation
 - DMA and AAU transaction initialization (posting)
 - Interrupt handling
- Termination
 - Free Memory
 - Return interrupt handlers to pre-library state
 - Return interrupt controller to pre-library state

6.1.1 Flow Sequence Description: AAU and DMA Library

6.1.1.1 Initialization

lib_mem_map_malloc()	Allocates separate memory regions for AAU descriptors, DMA descriptors and Data. Records memory map to data structure Memmap_Type mem_map ;
lib_set_xcb_mem_range()	Sets xcb bits for address range. This is used in conjunction with global variable mem_map to set descriptor and data address ranges.
lib_new_mgr()	Allocates memory for AauDma_80321_Type data structure.
lib_dma_init() and lib_aau_init()	Initializae AauDma_80321_Type data structure . The Free Stacks and Post Queues are initialized.
callintHandlerAttach()	Chains interrupt handlers lib_irq_handler() and lib_fiq_handler() into interrupt vectors.
lib_dma_int_setup() and lib_aau_int_setup()	Called to unmask and route interrupts for DMA and AAU.
lib_set_coalescing_global()	Turns on Intel® XScale™ microarchitecture global coalescing. When global coalescing is already set, it remains on with function call.

6.1.1.2 Request

lib_stack_pop() is called to get either a AAU or DMA Ch0 or DMA Ch1 descriptor from Free Stack.

In the case of an append, the descriptor values are written. Then, when the cache policy for the memory region used by the descriptors is either 010, 011 or 111 (caches on), the descriptor needs to be flushed to RAM using a macro. Function **dmalib_postq_appnd_resume()** or **aaulib_postq_appnd_resume()** are called to append the descriptor to the specified chain, place in post queue, records to global variable chainTailAAUDMA[], then sets resume to start the transaction.

6.1.1.3 Post Transaction Cleanup

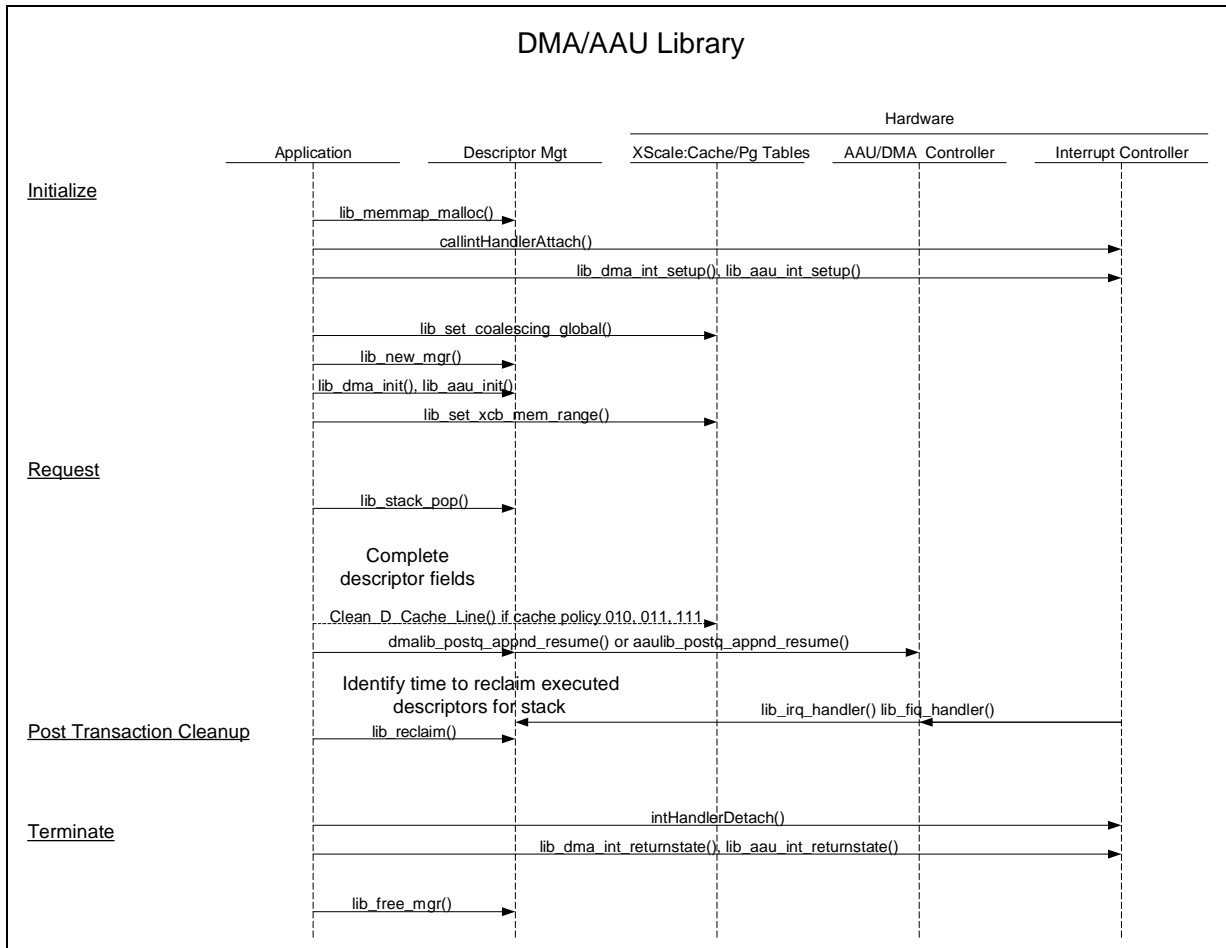
At some point descriptors for completed transactions must be reclaimed from the Post Queues and returned to the Free Stack (when append is used). This is accomplished by calling **lib_reclaim()**.

When the interrupt bit is set for a AAU or DMA descriptor, a interrupt is initiated when the descriptor transaction completes. The interrupt handler identifies the source of the interrupt as a AAU or DMA transaction, the csr value is recorded to the Frame and the interrupt is cleared. When a error has occurred for the descriptor then the Frame csr value is non-zero.

6.1.1.4 Terminate

When choosing to terminate the Library, then call `intHandlerDetach()` to return the interrupt vectors to the pre-chained in state, call `lib_dma_int_returnstate()` and `lib_aau_int_returnstate()` to mask DMA and AAU interrupts and call `lib_free_mgr()` to free memory.

Figure 6. DMA/AAU Library Flow Diagram



6.2 Memory Map for Intel[®] 80321 I/O Processor: Virtual Verses Physical for Redboot

Examples provided in this section are based on the Red Hat* Redboot software running on the Intel[®] IQ80310 Evaluation Platform Board (IQ80321).

6.2.1 Redboot* Intel® IQ80321 Memory Map

The virtual memory maps use a C, B, and X column to indicate the caching policy for the region.

X	C	B	Description
0	0	0	Un-cached/Un-buffered
0	0	1	Un-cached/Buffered
0	1	0	Cached/Buffered Write Through, Read Allocate
0	1	1	Cached/Buffered Write Back, Read Allocate
1	0	0	Invalid -- not used
1	0	1	Un-cached/Buffered No write buffer coalescing
1	1	0	Mini D-Cache - Policy set by Auxiliary Control Register
1	1	1	Cached/Buffered Write Back, Read/Write Allocate

Physical Address Range	Description
0x0000 0000 - 0x7FFF FFFF	ATU Outbound Direct Window
0x8000 0000 - 0x900F FFFF	ATU Outbound Translate Windows
0xa000 0000 - 0xBFFF FFFF	SDRAM
0xf000 0000 - 0xF080 0000	FLASH (PBIU ^a CS0 ^b)
0xfe80 0000 - 0xFE80 0FFF	UART (PBIU CS1)
0xfe84 0000 - 0xFE84 0FFF	Left 7-segment LED (PBIU CS3)
0xfe85 0000 - 0xFE85 0FFF	Right 7-segment LED (PBIU CS2)
0xfe8d 0000 - 0xFE8D 0FFF	Rotary Switch (PBIU CS4)
0xfe8f 0000 - 0xFE8F 0FFF	Battery Status (PBIU CS5)
0xffff 0000 - 0xFFFF FFFF	Intel® 80321 I/O Processor Memory Mapped Registers

- a. PBIU: Intel® 80321 I/O processor Peripheral Bus Interface Unit.
b. CS: Chip-Select for the PBIU on Intel® 80321 I/O processor.

Default Virtual Map	X	C	B	Description
0x00000000 - 0x1ffffff	1	1	1	SDRAM
0x20000000 - 0x9ffffff	0	0	0	Outbound Direct Window
0xa0000000 - 0xb0ffffff	0	0	0	Outbound Translate Windows
0xc0000000 - 0xdffffff	0	0	0	Un-cached alias for SDRAM
0xe0000000 - 0xe0ffffff	1	1	1	Cache flush region (no phys memory)
0xf0000000 - 0xf0800000	0	1	0	Flash (PBIU CS0)
0xfe800000 - 0xfe800fff	0	0	0	UART (PBIU CS1)
0xfe840000 - 0xfe840fff	0	0	0	Left 7-segment LED (PBIU CS3)
0xfe850000 - 0xfe850fff	0	0	0	Right 7-segment LED (PBIU CS2)
0xfe8d0000 - 0xfe8d0fff	0	0	0	Rotary Switch (PBIU CS4)
0xfe8f0000 - 0xfe8f0fff	0	0	0	Battery Status (PBIU CS5)
0xffff0000 - 0xfffffff	0	0	0	Intel® 80321 I/O processor Memory Mapped Registers

6.2.2 Redboot Intel® IQ80321 Physical Memory Map - Visual

Figure 7. Redboot Intel® IQ80321 Physical Memory Map

0000 0000h - 7FFF FFFFh	ATU Outbound Direct Addressing Window		
8000 0000h - 9001 FFFFh	ATU Outbound Translation Window		
9002 0000h - FFFF DFFFh Code/Data External Memory	<table border="1"> <tr> <td>A000 0000h to size of the DIMM</td> <td>SDRAM (DDR)</td> </tr> </table>	A000 0000h to size of the DIMM	SDRAM (DDR)
	A000 0000h to size of the DIMM	SDRAM (DDR)	
	<table border="1"> <tr> <td>F000 0000h - F080 0000h</td> <td>FLASH (8 Meg)</td> </tr> </table>	F000 0000h - F080 0000h	FLASH (8 Meg)
	F000 0000h - F080 0000h	FLASH (8 Meg)	
	<table border="1"> <tr> <td>FE80 0000h</td> <td>UART</td> </tr> </table>	FE80 0000h	UART
	FE80 0000h	UART	
	<table border="1"> <tr> <td>FE84 0000h</td> <td>7-segment 0 (W)</td> </tr> </table>	FE84 0000h	7-segment 0 (W)
FE84 0000h	7-segment 0 (W)		
<table border="1"> <tr> <td>FE85 0000h</td> <td>7-segment 1 (W)</td> </tr> </table>	FE85 0000h	7-segment 1 (W)	
FE85 0000h	7-segment 1 (W)		
<table border="1"> <tr> <td>FE8D 0000h</td> <td>Rotary Switch (R)</td> </tr> </table>	FE8D 0000h	Rotary Switch (R)	
FE8D 0000h	Rotary Switch (R)		
<table border="1"> <tr> <td>FE8F 0000h</td> <td>Battery Status (R)</td> </tr> </table>	FE8F 0000h	Battery Status (R)	
FE8F 0000h	Battery Status (R)		
FFFF E000h - FFFF E8FFh	Peripheral Memory-Mapped Registers		
FFFF E900h - FFFF FFFFh	Intel® 80321 IO Processor Reserved		

6.2.3 Redboot Intel® IQ80321 Virtual Memory Map - Visual

Figure 8. Redboot Intel® IQ80321 Virtual Memory Map

0x00000000 - 0x1fffffff	SDRAM (DDR)
0x20000000 - 0x9fffffff	ATU Outbound Direct Addressing Window
0xa0000000 - 0xb00ffffff	ATU Outbound Translation Window
0xc0000000 - 0xdfffffff	Un-cached alias for SDRAM
0xe0000000 - 0xe00ffffff	Cache flush region (no physical memory)
0xF0000000 - 0xF0800000	FLASH (8 Meg)
0xFE800000	UART
0xFE840000	7-segment 0 (W)
0xFE850000	7-segment 1 (W)
0xFE8D0000	Rotary Switch (R)
0xFE8F0000	Battery Status (R)
0xFFFFE000 - 0xFFFFE8FF	Peripheral Memory-Mapped Registers

7.0 Library and Test Bench File Organization and Compilation

Testbench runs with Redhat* monitor. When another OS is used, modifications may be major depending on the system architecture (i.e., Linux*).

7.1 Folder and File Organization

Folders:

- Lib.....The DMA/AAU Library.
- Bench.....Test bench illustrating an implementation of Library.

7.1.1 /Lib Files:

- lib_headers.h:
 - all #include files for Library.
- 80200.h:
 - 80200 macros per *Intel® XScale™ Microarchitecture Programmer's Reference Manual*.
- AAU8021.h:
 - 80321 AAU: memory map, macros and global variables for append and reclaim.
- DMA80321.h:
 - 80321 DMA: memory map, macros and global variables for append and reclaim.
- AAUDMADescMgr.h, AAUDMADescMgr.c:
 - Functions, data structures and macros to malloc memory for data and descriptor processing and sets cache policy for memory map of both descriptors and data.
- fiq_irq_80321.h, fiq_irq_80321.c:
 - 80321 interrupt controller memory map, function calls to: attach/detach irq and fiq interrupt handlers, mask/unmask and route fiq and irq.
- xscale.h, xscale.c:
 - Setup memory map with Intel® XScale™ microarchitecture xcb cache policy, set global coalescing, global data cache clean.
- Makefile:
 - Creates: xscale.o, fiq_irq_80321.o and aaudmadescmgr.o.

7.1.2 /Bench Files:

- bench.h, bench.c:
 - Provides menu to call test cases and present output as stdio.
- lib_demo_cases.c:
 - Demonstration of AAU and DMA Library.
- Headers.h:
 - Combine all #include references.
- makefile:
 - Creates bench_321.elf. Library object files: xscale.o, fiq_irq_80321.o and aaudmadescmgr.o.

7.2 Instruction to Build and Run

- Unzip into directory.
- Setup GNUPro Toolset in directory path.
- Open DOS window and go to directory above Lib and Bench.
- Type b at command prompt to call b.bat file:
 - b.bat file erases existing .o and .elf files, build required .o files and bench_80321.elf.
- To run call run321 to call run321.bat file.
- When debugger comes up:
 - Select: File>Target setting, enter baud rate 115200, COM port, target>Ok.
 - Open console window.
 - Select: Run>Download > Continue.
 - Click mouse in Console window.
 - Enter menu option > <enter>.

8.0 General Notes

8.1 GNUPro Toolset Compiler Optimization

The GNUPro toolset offers compiler setting of -O0 through -O3. Each level has its unique characteristic in terms of level of optimization and object size. Level -O2 was found to provide the best performance for descriptor processing.

8.2 Reclamation of Descriptors

The implementation in the test case waits until the channel is not busy, then calls `lib_reclaim()`. Function `lib_reclaim()` traverses a descriptor chain starting with `chainHeadAAUDMA[3]` through the current descriptor as identified by the AAU Accelerator Descriptor Address Register (ADAR) or DMA Descriptor Address Register (DAR). All descriptors between are identified as completed and returned to the Free stack for that channel. Note when any AAU or DMA engine error occurs, the descriptor is marked with `csr` value `!= 0x0` inside the interrupt handler. Non-zero `csr` values are identified inside the `lib_reclaim()`.

There are alternate methods to select the time when to reclaim executed descriptors. These include:

- Wait until a request for a free descriptor returns a null pointer indicating there are no free descriptors remaining in that channel.
- Wait till slack time in processing and call `lib_reclaim()`.
- Set timer and call in intervals.

8.3 Extending Library to Run with Multiple Threads

The Library is single threaded. However, all functions take a pointer to the data structure `AauDma_80321_Type`. This feature simplifies porting to multiple threads applications. The issue becomes mapping the DMA and AAU controllers to multiple threads.

8.4 When Using Append / Resume Sequence

`DMA_APPEND(LAST,NEXT)`

- `LAST` should be the uncached/unbuffered address of the last descriptor for that channel. This eliminates a cache clean.
- `NEXT` should be the physical address of the descriptor appended.

```
DMA_APPEND(LAST,NEXT) , DMA_RESUME_CHO_REG();
```

- The append followed by the resume operation should only be separated by `","` operator to eliminate possibility of instruction reordering by compiler.



8.5 DMA or AAU Descriptors Variables should be Volatile

By making variables volatile, this eliminates the potential for compiler optimization altering the variable. An example would be the global variables used for append:

```
volatile void *      chainHeadAAUDMA[3];//Global variable used for append  
volatile void *      chainTailAAUDMA[3];//Global variable used for append  
volatile AauDma_80321_Type *desc_mgr;
```



9.0 Conclusion

The purpose of the 80321 DMA/AAU Library is a fast ramp for developers by providing a turnkey optimized solution that synergistically combines with exiting Intel collateral (See [Appendix E](#), “[Related Documents](#)” for related web documents).

Appendix A Library Data Structures

A.1 DMA Descriptor

```
//DMA Data Structure

typedef struct dma {
    unsigned long    nda;
    unsigned long    pad;
    unsigned long    puad;
    unsigned long    lad;
    unsigned long    bc;
    unsigned long    dc;
}Dma_Type;
```

A.2 AAU Descriptors

```
//AAU data structure for 4 sources

typedef struct aau_4src_desc {
    unsigned long        nda;
    unsigned long        sar1;
    unsigned long        sar2;
    unsigned long        sar3;
    unsigned long        sar4;
    unsigned long        dar;
    unsigned long        bc;
    unsigned long        dc;
}Aau_4src_Type;

//AAU data structure for 8 sources

typedef struct aau_8src_desc {
    unsigned long        nda;
    unsigned long        sar1;
    unsigned long        sar2;
    unsigned long        sar3;
    unsigned long        sar4;
    unsigned long        dar;
    unsigned long        bc;
    unsigned long        dc;
    unsigned long        sar5;
    unsigned long        sar6;
    unsigned long        sar7;
    unsigned long        sar8;
}Aau_8src_Type;

//AAU data structure for 16 sources

typedef struct aau_16src_desc {
    unsigned long        nda;
    unsigned long        sar1;
    unsigned long        sar2;
    unsigned long        sar3;
    unsigned long        sar4;
    unsigned long        dar;
    unsigned long        bc;
    unsigned long        dc;
    unsigned long        sar5;
    unsigned long        sar6;
    unsigned long        sar7;
    unsigned long        sar8;
    unsigned long        edc0;
    unsigned long        sar9;
    unsigned long        sar10;
    unsigned long        sar11;
}
```



```
        unsigned long sar12;
        unsigned long sar13;
        unsigned long sar14;
        unsigned long sar15;
        unsigned long sar16;
    }Aau_16src_Type;
```

```
//AAU data structure for 32 sources
```

```
typedef struct aau_32src_desc {
    unsigned long nda;
    unsigned long sar1;
    unsigned long sar2;
    unsigned long sar3;
    unsigned long sar4;
    unsigned long dar;
    unsigned long bc;
    unsigned long dc;
    unsigned long sar5;
    unsigned long sar6;
    unsigned long sar7;
    unsigned long sar8;
    unsigned long edc0;
    unsigned long sar9;
    unsigned long sar10;
    unsigned long sar11;
    unsigned long sar12;
    unsigned long sar13;
    unsigned long sar14;
    unsigned long sar15;
    unsigned long sar16;
    unsigned long edc1;
    unsigned long sar17;
    unsigned long sar18;
    unsigned long sar19;
    unsigned long sar20;
    unsigned long sar21;
    unsigned long sar22;
    unsigned long sar23;
    unsigned long sar24;
    unsigned long edc2;
    unsigned long sar25;
    unsigned long sar26;
    unsigned long sar27;
    unsigned long sar28;
    unsigned long sar29;
    unsigned long sar30;
    unsigned long sar31;
    unsigned long sar32;
}Aau_32src_Type;
```

A.3 Descriptor Management: Both AAU and DMA Descriptors (AAUDMADescMgr.h)

```
// Used to reference Descriptor Management data structures
enum hardware{      DMA_CH0, //DMA Channel 0
                    DMA_CH1, //DMA Channel 1
                    AAU      }; //AAU

volatile void *    chainHeadAAUDMA[3]; //Global variable used for
append and reclaim
volatile void *    chainTailAAUDMA[3]; //Global variable used for
append and reclaim

//Below are Frames which include additional information to DMA or AAU data structure required
//for Library.

typedef struct dma_frame{
    Dma_Type      d;          /* descriptor      */
    short         pid;       /* pid            */
    short         tid;       /* tid            */
    short         csr;       /* CSR value      */
    short         mark;      /* If != 0 then executed*/
}Dma_Frame_Type;

typedef struct aau_frame{
    Aau_32src_Type d;
    short         pid;       /* pid            */
    short         tid;       /* tid            */
    short         csr;       /* CSR value      */
    short         mark;      /* If != 0 then executed*/
    unsigned long pad[23];   //Now 256 bytes or 64 byte
                                //for 64 word alignment.
}Aau_Frame_Type;

/* Container designed to be multiples of cacheline (32 bytes).*/
//Used for Post Queue

typedef struct queue{
    int           CircQ_Front;
    int           CircQ_Length;
    int           CircQ_Limit;
    unsigned long qty_marked;    //Quantity marked
    unsigned long * pad[4];
    void          * CircQ[QUEUESIZE]; //Channel size s/b
mult of 8
}Queue_Type;

//Used for Free Stack

typedef struct stack{
    int           Stack_Length;
    int           Stack_Limit;
    int           pad[6];        /* To stay on cacheline. */
    void          * Stack[STACKSIZE];
}Stack_Type;
```



```
/* Structure mapping two Channel_DMAs and one AAU to Controllers*/
typedef struct aaudma_80321{
    //AAU and DMA Channels
    Stack_Type      FreeStack[3]; //2 DMA Channels+1 AAU stack
    Queue_Type      Queue[3];    //2 DMA Channels + 1 AAU
    //Put whatever malloced here
    void *          toFreeDMADesc;
    void *          toFreeDMAData;
    void *          toFreeAAUDesc;
    void *          toFreeAAUData;
    void *          toFreedma_80321;
}AauDma_80321_Type;
```

A.4 Intel® XScale™ Microarchitecture Page Tables and Library Memory Map (xscale.h)

```
//Memory map recorded

typedef struct memmap{
    //Memory map virtual addresses
    //Descriptors
    unsigned longaaau_desc_lower_va; //va = virtual address
    unsigned longaaau_desc_upper_va;
    unsigned longaaau_desc_num_pages;
    unsigned longaaau_desc_xcb;

    unsigned longdma_desc_lower_va;
    unsigned longdma_desc_upper_va;
    unsigned longdma_desc_num_pages;
    unsigned longdma_desc_xcb;

    //Data region
    unsigned longdata_lower_va;
    unsigned longdata_upper_va;
    unsigned longdata_num_pages;
    unsigned longdata_xcb;

    unsigned longlad;
    unsigned longpad;

    int        page_size;
    unsigned longpage_boundry_1st;

    //memory malloced
    void *      toFree;
    int        size_malloced;
}Memmap_Type;

//Information returned for a memory page

typedef struct page{
    //Level 1
    unsigned long    pt_base;
    unsigned long    virtadd;
    unsigned int     type_lv11;
    unsigned int     type_lv12;
    unsigned long    *lv11_des_ptr;
    unsigned long    lv11_des_val;
    unsigned int     xcb_lv11_before;
    unsigned int     xcb_lv11_after;
    //Level 2
    unsigned long    *lv12_des_ptr;
    unsigned long    lv12_des_val;
    unsigned long    baseloc;
    int              input_p;
    int              input_x;
    int              input_c;
    int              input_b;
    unsigned int     xcb_input;
    unsigned int     xcb_lv12_before;
    unsigned int     xcb_lv12_after;
    int              page_size;
    unsigned int     page_xcb;
}Page_Type;
```

Appendix B Library Function Prototypes

B.1 Functions Included in xscale.h and xscale.c

B.1.1 void lib_set_coalescing_global(void)

Item	Description
Prototype	<code>void lib_set_coalescing_global(void);</code>
Input	None
Output	None
Purpose	Turns on coalescing for Intel® XScale™ microarchitecture write buffer.
Operation	Call function to turn on write coalescing globally.

B.1.2 void lib_flush_data_cache(void)

Item	Description
Prototype	<code>void lib_flush_data_cache(void);</code>
Input	None
Output	None
Purpose	Cleans and invalidates 32 Kb data cache.
Operation	Selects 32 Kb memory region reserved for this clean operation, allocates cache lines to evict dirty data to ram. Then invalidates full data cache.

B.1.3 int lib_memmap_malloc(int dma_desc_Mbs, int aau_desc_Mbs, int data_Mbs)

Item	Description
Prototype	<code>int lib_memmap_malloc(int dma_desc_Mbs, int aau_desc_Mbs, int data_Mbs)</code>
Input	dma_desc_Mbs: Enter the number of megabytes to be allocated for DMA descriptors aau_desc_Mbs: Enter the number of megabytes to be allocated for AAU descriptors data_Mbs: enter the number of megabytes to be entered for data. Transaction source and destination data.
Output	SUCCESS == 0 FAIL == non-zero
Purpose	Allocates and records memory regions on page boundaries for DMA descriptors, AAU descriptors and data area.
Operation	Function mallocs region of size totaling the number of megabytes requested for DMA, AAU and data descriptors plus one page. The memory regions for each are recorded to the global data structure Memmap_Type mem_map.

B.1.4 void lib_set_xcb_mem_range(unsigned int xcb, void * virt_addr_base, int size_in_bytes)

Item	Description
Prototype	<code>void lib_set_xcb_mem_range(unsigned int xcb, void * virt_addr_base, int size_in_bytes);</code>
Input	<u>xcb</u> : xcb bit values. Example x=1, c=0, b=1 would be 0x5; While x=1, c=1, b=1 would be 7 <u>virt_addr_base</u> : Enter virtual address of start of range in which has the xcb bits set for all pages within the range. <u>size_in_bytes</u> : number of bytes to establish the upper bound for the cache policy change (xcb bits)
Output	None
Purpose	To set the data cache policy (xcb bits) for the memory range after cleaning and invalidates the data cache.
Operation	<ul style="list-style-type: none"> • Cleans and invalidates the full data cache • Interacted through the pages within the address range • For each page sets the data policy per input parameter xcb • Invalidates the data TLBs

B.1.5 Page_Type lib_get_page_attributes(unsigned long virt_addr)

Item	Description
Prototype	<code>Page_Type lib_get_page_attributes(unsigned long virt_addr);</code>
Input	Virtual memory address
Output	Page_Type that provides: page_size, page_xcb = current cache policy (xcb bits), base_loc = page boundary prior to virt_addr
Purpose	Obtain attributes of page that includes the input parameter address. Attributes include page size, lower page boundary and cache policy.
Operation	Function calls lib_set_page_xcb with input parameter that does not change the state of the page tables

B.1.6 Page_Type lib_set_page_xcb(unsigned long base, unsigned int xcb)

Item	Description
Prototype	<code>Page_Type lib_set_page_xcb(unsigned long base, unsigned int xcb);</code>
Input	base: address on page in which xcb: cache policy. Bit 2==x bit, bit 1: c== bit and bit 0==b bit
Output	Page_Type that provides: page_size, page_xcb = current cache policy (xcb bits), base_loc = page boundary prior to virt_addr
Purpose	Sets the cache policy for the page including the input parameter address. <ul style="list-style-type: none"> • Before calling: data cache should be cleaned and invalidated • After calling: Data TLB needs to be invalidated.
Operation	Function sets the xcb bits in the page table using the Translation Process algorithm per the ARM Architecture Manual page b3-6 to and the bit definitions per the Intel® XScale™ Microarchitecture Programmer's Reference Manual.

B.2 Functions included in AAUDMADescMgr.h and AAUDMADescMgr.c

B.2.1 AauDma_80321_Type * lib_new_mgr(void)

Item	Description
Prototype	AauDma_80321_Type * lib_new_mgr(void);
Input	None
Output	None
Purpose	Allocates memory and instantiates data structure AauDma_80321_Type.
Operation	Allocate memory and cache align for AauDma_80321_Type. Note the lib_dma_init() is passed memory for the AAU and DMA descriptor buffers and initializes the stack and queue data structures.

B.2.2 int lib_dma_buffersize(void)

Item	Description
Prototype	int lib_dma_buffersize(void)
Input	None
Output	Returns the number of bytes required for DMA Buffers
Purpose	Specifies the number of bytes required for DMA buffers including amount for alignment.
Operation	Returns (sizeof(Dma_Frame_Type) * DMASTACKSIZE) + sizeof(Dma_Frame_Type)

B.2.3 int lib_aau_buffersize(void)

Item	Description
Prototype	int lib_aau_buffersize(void);
Input	None
Output	Returns the number of bytes required for AAU Buffers
Purpose	.Specifies the number of bytes required for DMA buffers including amount for alignment.
Operation	

**B.2.4 void lib_dma_init(AauDma_80321_Type * mgrt,
void * dma_desc_baseaddr)**

Item	Description
Prototype	<code>void lib_dma_init(AauDma_80321_Type * mgrt, void * dma_desc_baseaddr);</code>
Input	<code>mgrt</code> - pointer to <code>AauDma_80321_Type</code> allocated by <code>lib_new_mgr()</code> . <code>dma_desc_baseaddr</code> - Base address of memory region to be used by DMA descriptors
Output	None
Purpose	To take memory address allocated for DMA buffers and initialize stack and queues. After this function call DMA stacks and queues are initialized and DMA Section of the Library is read for use.
Operation	<ul style="list-style-type: none"> Initialize DMA free stack Initialize DMA channel 0 and 1 post queues Execute dummy descriptors for DMA channels 0 and 1. The initialize <code>chainHeadDMA[channel]</code> and <code>chainTailDMA[channel]</code> to allow append/resume.

**B.2.5 void lib_aau_init(AauDma_80321_Type * mgrt,
void * aau_desc_baseaddr)**

Item	Description
Prototype	<code>void lib_aau_init(AauDma_80321_Type * mgrt, void * aau_desc_baseaddr);</code>
Input	<code>mgrt</code> - pointer to <code>AauDma_80321_Type</code> allocated by <code>lib_new_mgr()</code> . <code>aau_desc_baseaddr</code> - Base address of memory region to be used by AAU descriptors
Output	None
Purpose	To take memory address allocated for AAU buffers and initialize stack and queues. After this function call AAU stacks and queues are initialized and AAU Section of the Library is read for use.
Operation	

B.2.6 void lib_free_mgr(AauDma_80321_Type * mgr)

Item	Description
Prototype	<code>void lib_free_mgr(AauDma_80321_Type * mgr);</code>
Input	Pointer to <code>AauDma_80321_Type</code> data structure to free
Output	None
Purpose	Frees all memory associated with <code>AauDma_80321_Type</code> and sets to NULL.
Operation	Frees all memory associated with <code>AauDma_80321_Type</code> and sets to NULL.

B.2.7 `void * lib_stack_pop(AauDma_80321_Type * mgr, enum hardware engine)`

Item	Description
Prototype	<code>void * lib_stack_pop(AauDma_80321_Type * mgr, enum hardware engine)</code>
Input	AauDma_80321_Type being used) Engine: either DMA Channel 0, DMA Channel 1 or AAU
Output	Pointer to Frame when successful or NULL when stack empty
Purpose	To provide a DMA or AAU frame from a free stack. When a cacheable memory regions is used for descriptors, using stack increases the likelihood that stack is still in cache.
Operation	Gets DMA or AAU frame from the top of the channel specific Free stack and changes stack state to reflect removal.

B.2.8 `Bool lib_stack_push(AauDma_80321_Type * mgr, void * frame, enum hardware engine)`

Item	Description
Prototype	<code>Bool lib_stack_push(AauDma_80321_Type * mgr, void * frame, enum hardware engine)</code>
Input	mgr: AauDma_80321_Type pointer to data structure being accessed. frame: pointer to frame being returned to the stack Engine: either DMA Channel 0, DMA Channel 1 or AAU
Output	Success == 0 Fail == non-zero
Purpose	Place frame on stack for future reuse.
Operation	Place DMA Channel 0, DMA Channel 1 or AAU frame on top of free stack and change stack state to show a new top of stack.

B.2.9 `void * lib_top_of_stack(AauDma_80321_Type * mgr, enum hardware engine)`

Item	Description
Prototype	<code>void * lib_top_of_stack(AauDma_80321_Type * mgr, enum hardware engine)</code>
Input	Pointer to AauDma_80321_Type being used. Engine: either DMA Channel 0, DMA Channel 1 or AAU
Output	Pointer to TOS frame or NULL when stack is empty
Purpose	Get a pointer to top frame on the stack (Either DMA Ch0, Ch1 or AAU). This function does not change the state of the stack. Could be used for preload.
Operation	See purpose

B.2.10 inline int dmalib_postq_appnd_resume(AauDma_80321_Type * mgr, void * frame, enum hardware engine)

Item	Description
Prototype	inline int dmalib_postq_appnd_resume (AauDma_80321_Type * mgr, void * frame , enum hardware engine)
Input	mgr: AauDma_80321_Type accessing frame: DMA frame to be appended Engine: either DMA Channel 0, DMA Channel 1
Output	SUCCESS == 0 FAIL == non-zero
Purpose	To post frame to post queue, append frame to a channel specific chain of DMA descriptors and set channel resume to initiate transfer. NOTE: When using cached memory, flush descriptor to RAM before calling function.
Operation	<ul style="list-style-type: none"> • Post to queue • Append to DMA channel chain • Reset chain chainTailAAUDMA[3]pointer • Set DMA channel resume

B.2.11 inline int aaulib_postq_appnd_resume(AauDma_80321_Type * mgr, void * frame)

Item	Description
Prototype	inline int dmalib_postq_appnd_resume (AauDma_80321_Type * mgr, void * frame)
Input	mgr: AauDma_80321_Type accessing frame: AAU frame to be appended
Output	SUCCESS == 0 FAIL == non-zero
Purpose	To post frame to post queue, append frame to a channel specific chain of DMA descriptors and set channel resume to initiate transfer. NOTE: When using cached memory, flush descriptor to RAM before calling function.
Operation	<ul style="list-style-type: none"> • Post to AAU queue • Append to DMA channel end of chain • Reset chain chainTailAAUDMA[3]pointer • Set AAU channel resume

B.2.12 `int lib_reclaim(AauDma_80321_Type * mgr, enum hardware engine)`

Item	Description
Prototype	<code>int lib_reclaim(AauDma_80321_Type * mgr, enum hardware engine)</code>
Input	mgr: AauDma_80321_Type data structure using Engine: either DMA Channel 0, DMA Channel 1 or AAU
Output	the number of descriptors returned
Purpose	Return Frames completed frames to the Free Stack
Operation	<ul style="list-style-type: none"> • Get channel DAR value to identify last descriptor used for DMA transfer • Get <code>chainHeadAAUDMA[3]</code> value • Traverse DMA channel chain until starting at the <code>chainHeadAAUDMA[3]</code> value until == DAR value. • For all descriptors in chain, remove from Channel queue and place in Free Stack • Adjust <code>chainTailAAUDMA[3]</code> to DAR • Return number of descriptors returned

B.2.13 `void * lib_q_get(AauDma_80321_Type * mgr, enum hardware engine)`

Item	Description
Prototype	<code>void * lib_q_get(AauDma_80321_Type * mgr, enum hardware engine)</code>
Input	mgr: The Aau_Dma_80321_Type used Engine: either DMA Channel 0, DMA Channel 1 or AAU
Output	Return a pointer to the Frame removed from the channel queue
Purpose	Retrieve frame on FIFO sequence from queue.
Operation	<ul style="list-style-type: none"> • Get Frame using FIFO • Adjust state of queue to reflect removed frame

B.2.14 `int lib_q_put(AauDma_80321_Type * mgr, void * frame, enum hardware engine)`

Item	Description
Prototype	<code>int lib_q_put(AauDma_80321_Type * mgr, void * frame, enum hardware engine)</code>
Input	mgr: AauDma_80321_Type operating on frame: Frame returning to Queue Engine: either DMA Channel 0, DMA Channel 1 or AAU
Output	SUCCESS == 0 FAIL == non-zero
Purpose	To post frame to channel queue.
Operation	<ul style="list-style-type: none"> • Post to queue • Adjust queue to reflect state change • Return SUCCESS or FAIL

B.3 Functions Included in `fiq_irq_80321.h` and `fiq_irq_80321.c`

B.3.1 `void intHandlerDetach(void)`

Item	Description
Prototype	<code>void intHandlerDetach(void);</code>
Input	None
Output	None
Purpose	To remove <code>lib_fiq_handler</code> and <code>lib_irq_handler</code> handlers from chain. Returns state of interrupt vectors to pre- <code>callintHandlerAttach()</code> state.
Operation	Restore pre- <code>callintHandlerAttach()</code> vector values.

B.3.2 `void callintHandlerAttach(void)`

Item	Description
Prototype	<code>void callintHandlerAttach(void);</code>
Input	None
Output	None
Purpose	Chains in <code>lib_fiq_handler</code> and <code>lib_irq_handler</code> to interrupt vectors.
Operation	Calls function <code>intHandlerAttach(lib_irq_handler, lib_fiq_handler)</code> .

B.3.3 `void intHandlerAttach(void (*irq)(void),void (*fiq)(void))`

Item	Description
Prototype	<code>void intHandlerAttach(void (*irq)(void),void (*fiq)(void));</code>
Input	<i>irq</i> : <i>irq</i> handler to be chained into interrupt vector <i>fiq</i> : <i>fiq</i> handler to be chained into interrupt vector
Output	None
Purpose	To chain functions into interrupt vectors.
Operation	<ul style="list-style-type: none"> Get location pointed to by interrupt vectors Save contents at location to global variable used to restore state Record address of function being chained in Function chained in jump to prior vector if interrupt is not AAU or DMA related.

B.3.4 void lib_irq_handler(void)__attribute__((__naked__))

Item	Description
Prototype	void lib_irq_handler(void)__attribute__((__naked__));
Input	None
Output	None
Purpose	IRQ handler for DMA and AAU initiated interrupts. Naked attribute eliminate function prolog and epilog
Operation	Test for DMA and AAU irq interrupt per IINTSRC

B.3.5 void lib_fiq_handler(void)__attribute__((__naked__))

Item	Description
Prototype	void lib_fiq_handler(void)__attribute__((__naked__));
Input	None
Output	None
Purpose	FIQ handler for DMA and AAU error condition interrupts. Naked attribute eliminate function prolog and epilog.
Operation	Test for DMA and AAU <i>fiq</i> interrupt per IINTSRC

B.3.6 void lib_dma_int_setup(void) and void lib_aau_int_setup(void)

Item	Description
Prototype	void lib_dma_int_setup(void) void lib_aau_int_setup(void)
Input	None
Output	None
Purpose	Steer interrupts and unmask
Operation	<ul style="list-style-type: none"> • asm ("mcr\tp6, 0, %0, c4, c0, 0" :: "r" (intstr)); • asm ("mcr\tp6, 0, %0, c0, c0, 0" :: "r" (intctl));

B.3.7 void lib_dma_int_returnstate(void) and void lib_aau_int_returnstate(void)

Item	Description
Prototype	void lib_dma_int_returnstate(void) void lib_aau_int_returnstate(void)
Input	None
Output	None
Purpose	.Return the state of interrupt controller to pre-lib_dma_int_setup(void)
Operation	<ul style="list-style-type: none"> • Get pre--lib_dma_int_setup(void) or pre--lib_aau_int_setup(void) values stored in global variables intstr_val and intctl_val. Then call: • asm ("mcr\tp6, 0, %0, c4, c0, 0" :: "r" (intstr_val)); • asm ("mcr\tp6, 0, %0, c0, c0, 0" :: "r" (intctl_val));Ta

Appendix C Testbench: Data Structures

C.1 bench.h

```
//Global Data structure to record experiment state

typedef struct experiment{
    enum test          test; //AAU_TEST or DMA_TEST or UNDEF_TEST

    //Data memory map. Source and destination
    int                max_buf_size;
    unsigned long      lad_lower;
    unsigned long      lad_upper;
    unsigned long      pad_lower;
    unsigned long      pad_upper;

    //Experiment characteristics
    char                xcb[40]; //xcb bits
}Experiment_Type;
```

Appendix D Test Bench Library Function Prototypes

D.1 bench.c

D.1.1 int main(void);

Item	Description
Prototype	<code>int main(void);</code>
Input	None
Output	None
Purpose	Testbench initialization and control of test cases run based on keyboard input from menu selection
Operation	Initialization <ul style="list-style-type: none"> Allocates descriptor manager data structure Allocated memory for descriptor processing and data and records to global structure Chains in interrupt handlers Enables and routes irq and fiq interrupts Menu Selection <ul style="list-style-type: none"> Infinite while loop that call test cases based on keyboard input from Menu

D.1.2 void print_title(enum build b)

Item	Description
Prototype	<code>void print_title(enum build b)</code>
Input	enum build identifies the test case.
Output	None
Purpose	To print the test case title to stdio and file bnech.out
Operation	Based on input parameter C switch statement selects corresponding printf statements.

D.1.3 void generate_src_dst(void);

Item	Description
Prototype	<code>void generate_src_dst(void);</code>
Input	None
Output	None
Purpose	Reset state of memory before each test case is run.
Operation	Resets state of descriptor and data memory regions to 0. Then writes test data to source locations.



D.2 lib_demo_cases.c

D.2.1 void dma_lib_demo(void) and void dma_lib_demo_witherror(void)

Item	Description
Prototype	<pre>void dma_lib_demo(void) void dma_lib_demo_witherror(void)</pre>
Input	None
Output	None
Purpose	To demonstrate functionality of DMA Library including error detection using interrupt handler
Operation	<p>Demonstrated functionality includes (this is commented and handled by main()):</p> <ul style="list-style-type: none"> • Allocate memory for descriptor manager • malloc the memory map • chain in the interrupt handler • set global coalescing <p>Called by case:</p> <ul style="list-style-type: none"> • initialize the DMA Library data structure inside descriptor manager (setup descriptor Free Stack and Post Queues, initialize DMA engines for appends) • set descriptor processing memory range as 001 • stat data memory range as 000 • execute DMA descriptor transactions • use append and reclaiming descriptors • reclaim reports any DMA transfer error since an error causes a interrupt to occur and the interrupt handler records the csr value to the frame. For an error the csr non-zeros. The non-zero csr value is identified by reclaim.

D.2.2 void aau_lib_demo(void) and void aau_lib_demo_witherror(void)

Item	Description
Prototype	<pre>void aau_lib_demo(void) void aau_lib_demo_witherror(void)</pre>
Input	None
Output	None
Purpose	To demonstrate functionality of AAU Library including error detection using interrupt handler
Operation	<p>Demonstrated functionality includes (this is commented and handled by main()):</p> <ul style="list-style-type: none"> • Allocate memory for descriptor manager • malloc the memory map • chain in the interrupt handler • set global coalescing <p>Called by case:</p> <ul style="list-style-type: none"> • initialize the AAU Library data structure inside descriptor manager (setup descriptor Free Stack and Post Queues, initialize AAU engine for appends) • set descriptor processing memory range as 001 • stat data memory range as 000 • execute AAU descriptor transactions • use append and reclaiming descriptors • reclaim reports any DMA transfer error since an error causes a interrupt to occur and the interrupt handler records the csr value to the frame. For an error the csr non-zeros. The non-zero csr value is identified by reclaim.

Appendix E Related Documents

- *Intel® 80200 Processor based on Intel® XScale™ Microarchitecture Developer's Manual*
<http://developer.intel.com/design/iio/manuals/273411.htm>.
- *Intel® 80321 I/O Processor Developer's Manual (273517)*
<http://developer.intel.com/design/iio/manuals/273517.htm>.
- *Intel® XScale™ Microarchitecture Programmer's Reference Manual*
<http://developer.intel.com/design/intelxscale/273436.htm>.
- *Intel® 80321 I/O Processor Developer's Manual*
<http://developer.intel.com/design/iio/manuals/273517.htm>.
- *Data Access Performance Optimization on the Intel® 80321 I/O Processor White Paper*
<http://developer.intel.com/design/iio/papers/273872.htm>.
- *ARM Architecture Reference Manual*, Edited by David Seal, Addison-Wesley

Many other Application Notes and tools:

- <http://www.intel.com/design/intelxscale/>.

