AAU Support Library for the Intel® 80310 I/O Processor Chipset and 80321 I/O Processor

Reference Manual

April 2002

Order Number: 273721-001

Contents

Figures

Tables

Revision History

Date	Revision	Description
April 2002	001	First release of this manual.

Introduction 1

The transfer of data throughout a system is one of the bottlenecks facing system designers today. The Application Accelerator Unit (AAU) of the Intel® 80310 I/O Processor Chipset and 80321 I/O Processor was designed to increase the performance of data transfers within an application.

The objective of this paper and the accompanying source code is to provide a programming example for software developers who are inexperienced with the AAU of the 80310 I/O processor chipset and 80321 I/O processor.

The AAU Library contains a set of APIs, which can be used to access the many features of the AAU. The AAU Library is not intended to be a turnkey solution that should be included in every application. Rather, the library is a template that software developers can use as a guide to help design their own AAU interface.

1.1 Intel® 80310 I/O Processor Chipset AAU Overview

The Intel® 80310 I/O Processor Chipset consists of the Intel® 80200 I/O Processor based on Intel® XScale™ microarchitecture and the Intel® 80312 I/O Companion Chip. The Application Accelerator is located inside the 80312 I/O companion chip.

The Application Accelerator provides low-latency, high-throughput data transfer capability between the AAU and Intel® 80200 processor based on XScale™ microarchitecture (ARM* architecture compliant) local memory. It executes data transfers to and from Intel® 80200 processor local memory and also provides the necessary programming interface.

The Application Accelerator performs the following functions:

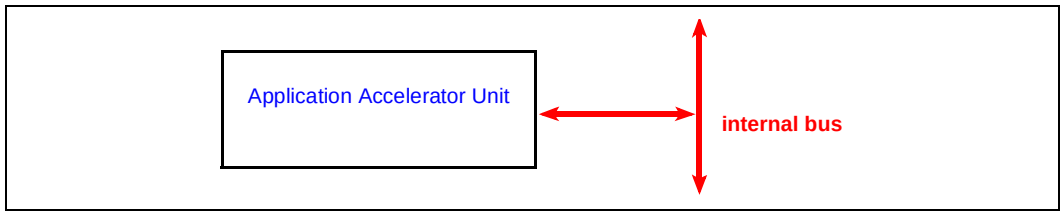
- Transfers data (read) from memory controller
- Performs an optional Boolean operation (XOR) on read data
- Transfers data (write) to memory controller

The AAU features:

- 1-Kbyte, arranged as 8-byte x 128-deep store queue
  - — Configurable to a 512-byte, arranged as 8-byte x 64-deep store queue
- Utilization of the Intel® 80312 I/O companion chip memory controller interface
- 2^{32} addressing range on the Intel® 80200 processor local memory interface
- Hardware support for unaligned data transfers for the internal bus
- Fully programmable from the Intel® 80200 processor
- Support for automatic data chaining for gathering and scattering of data blocks

Figure 1 shows a simplified connection of the Application Accelerator to the Intel® 80312 I/O companion chip internal bus.

Figure 1. Application Accelerator Unit

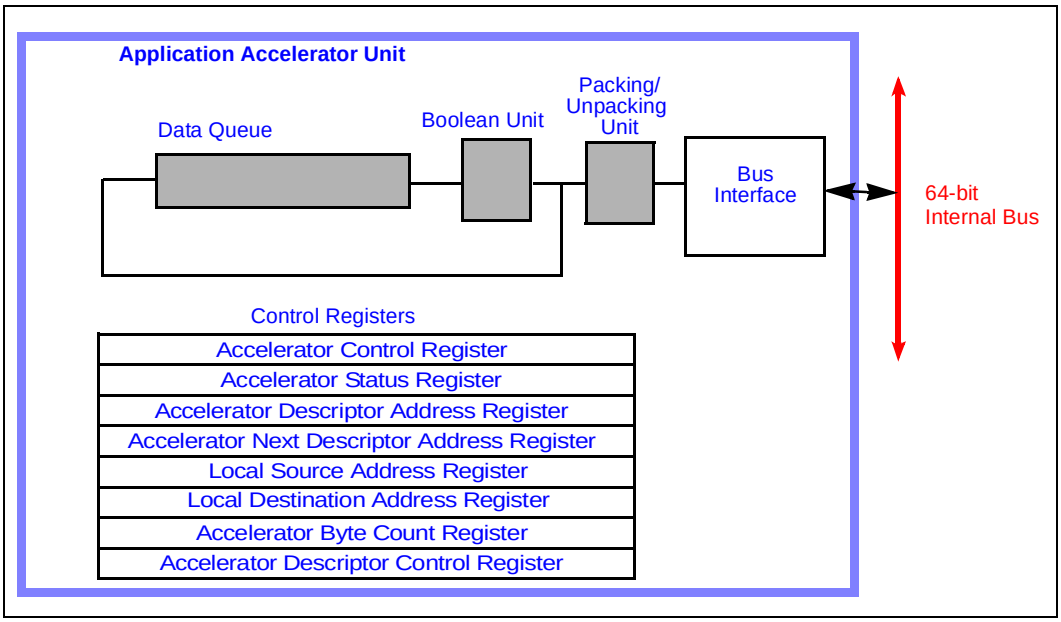


1.2 Intel® 80310 I/O Processor Chipset AAU Theory of Operation

The Application Accelerator is a master on the internal bus and performs data transfers to and from local memory. It does not interface to either the primary PCI or secondary PCI bus. The AAU uses direct addressing for the memory controller.

The Application Accelerator Unit implements the XOR algorithm in hardware. It performs the XOR operation on multiple blocks of source (incoming) data and stores the result back in Intel® 80200 processor local memory. The source and destination addresses are specified through chain descriptors resident in Intel® 80200 processor local memory. Figure 2 shows the block diagram of the AAU. The AAU can also be used to perform memory-to-memory transfers of data blocks controlled by the Intel® 80312 I/O companion chip memory controller unit.

Figure 2. 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 the *Intel® 80321 I/O Processor Developer's Manual* (273517).

The Application Accelerator unit contains a hardware data packing and unpacking unit. This unit enables data transfers from and to unaligned addresses in Intel® 80200 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 1K-byte holding queue (arranged as an 8-byte x 128-deep queue). Data to be written back to Intel® 80200 processor 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 Intel® 80200 processor local memory to form a linked list.

1.3 Differences between the Intel® 80312 I/O Companion Chip and the Intel® 80321 I/O Processor AAU

The Intel® 80321 I/O Processor allows for a XOR-transfer with up to 32 source blocks of data versus only eight for the Intel® 80312 I/O Processor. This has resulted in the addition of Source Address Registers (SAR) 9 – 32.

The Intel® 80321 I/O Processor also includes two new operations for the AAU:

- **Memory Block Fill** of up to 16 MB of local memory with a 32-bit constant
- **Parity Check Operation**

  The AA confirms that the XOR of all the bytes of the parity stripe — with their associated bytes in the streams of source data — result in 00H for the entire byte count. The source data streams and the parity data stream can be up to 16 MB long.

  The result of the check is written back to the Descriptor Control Word in local memory.

  Note: The Parity Check operation is not currently enabled. For more information, refer to the *Intel® 80321 I/O Processor Specification Update* (273519).

1.3.1 Extended Descriptor Control Registers (EDCR)

Three new registers have been added to the Intel® 80321 I/O Processor: EDCR0, EDCR1, and EDCR2. See Table 1 for details.

These registers are loaded when a chain descriptor — that requires a minimum of 16 Source Addresses — is read from memory. The values in the EDCRx define the command/control values as follow:

- EDCR0: SAR16 – SAR9
- EDCR1: SAR24 – SAR17
- EDCR2: SAR32 – SAR25

Table 1. New or Changed AAU Registers in the Intel® 80321 I/O Processor

Register Name	Internal Bus Address	Default Value	Register Description
SAR9-32	FFFF E840 – FFFF E8A4H	0000 0000H	Source Address Register9-32 – Each of these registers is loaded with blocks of data to be operated upon by the AAU
EDCR0	FFFF E83CH	0000 0000H	Extended Descriptor Control Register 0 – The values in EDCR0 define the command/control value for SAR16 – SAR9.
EDCR1	FFFF E860H	0000 0000H	Extended Descriptor Control Register 1 – The values in EDCR1 define the command/control value for SAR24 – SAR17.
EDCR2	FFFF E884	0000 0000H	Extended Descriptor Control Register 2 – The values in EDCR2 define the command/control value for SAR32 – SAR25.

1.3.2 The Accelerator Descriptor Control Register (ADCR)

Three ADCR bits have been added to the Intel® 80321 I/O Processor: bits 28, 29, and 30. For details see Table 2.

Table 2. New or Changed AAU Bit Functions in the Intel® 80321 I/O Processor

Bit Position	Internal Bus Address	Default Value	Bit Function
ADCR.26,25	FFFF E828H	00	Supplemental Block Control Interpreter – This bit field specifies the number of data blocks on which the operation is executed.
ADCR.28	FFFF E828H	0	Transfer Complete – This bit is set when the AA completes the processing of the descriptor.
ADCR.29	FFFF E828H	0	Parity Error – This bit is set when the bit wise parity computed across the data blocks specified by the SARx registers results in the detection of bad parity.
ADCR.30	FFFF E828H	0	Parity Enable – When this bit is set the AA computes the parity across the data blocks specified by the SARx registers.
NOTE: For more information, see Chapter 6 of the *Intel® 80321 I/O Processor Developer's Manual* (273517).			

1.4 Provided by the AAU Library

The following items and capabilities are provided by the AAU Library:

- A set of APIs that can be used to initialize and access the various features of the AAU
- The ability to support both the Intel® 80321 I/O Processor and the Intel® 80310 I/O Processor chipset.
- The entire source code of the AAU Library
- The ability to build the library with either the ARM* compiler or GNUPro* Compiler
- Project files for the ATI* Codelab|EDE and ARM ADS software toolchains
- Makefiles for the GNUPro and ARM toolchains
- A testbench that provides examples of how to initialize, execute, and benchmark the AAU Library

1.5 Future Enhancements

While the AAU Library is intended as a template for software developers, there are some areas that could be enhanced before implementing it in an application. These enhancements would include:

- Increase the error-handling capabilities

  Currently, the AAU Library clears the AAU Status Register (ASR) and marks a descriptor as executed in the ISR. The AAU Library currently will not handle an internal bus or other error.

- Benchmark using a 512-byte, versus a 1-KB, data buffer per application to obtain the highest performance

  This can be accomplished by setting the appropriate bit in the Accelerator Control Register (ACR).

- Allow an entire chain of descriptors to be filled out before executing the AAU operation

  Currently, each descriptor needs to be submitted before another descriptor is requested. If this enhancement were made, the descriptor-reclaiming function (CleanChain) also would need to be enhanced.

- Implement the full suite of Intel® XScale™ optimization strategies

  See *Coding Tips for Developers Targeting I/O Processors Based on the Intel® XScale™ Microarchitecture* (273618)

- Modify the AAU Library to be re-entrant, to support a multi-tasking environment

  The AAU Library was not created for a multi-tasking environment. It was created to give an example of how to utilize the functions of the AAU.

1.6 Related Documents

Intel® 80321 I/O Processor *Developer's Manual* (273517)

Intel® IQ80321 Evaluation Platform Board Manual (273521)

Intel® 80312 I/O Companion Chip Developer's Manual (273410)

Intel® IQ80310 Evaluation Platform Board Manual (273431)

Coding Tips for Developers Targeting I/O Processors Based on the Intel® XScale™ Microarchitecture (273618)

Intel® 80310 I/O Processor Chipset AAU Coding Techniques (273649)

Basic Requirements 2

2.1 Memory Allocation

The AAU Library uses the C Standard Library functions malloc and free to create and release all of the descriptors used throughout the AAU Library. The Testbench uses malloc and free to create and release all of the source and destination data used to test the AAU Library APIs. Therefore, malloc and free must be available in the environment that the AAU Library is being used.

2.2 Interrupt Handling/Debug Monitors

The user has the option, at compile time, to have the AAU Library set the interrupt-enable bit to trigger interrupts upon completion of an AAU operation. If this feature is enabled, the AAU Library inserts its own interrupt handler by chaining its interrupt handler in front of the debug monitor's interrupt handlers.

This feature is only available when compiling with GNUPro* and using Red Hat* Red Boot.

2.3 Memory Map

The AAU operates on physical addresses. The AAU Library uses a macro "VIRT_TO_PHY" that translates virtual addresses into physical addresses. Another macro "PHY_TO_VIRT" is used to translate back, from physical addresses into virtual addresses.

These two macros must be modified per platform to generate the proper physical and virtual addresses.

2.4 Compiling for 80321 Versus 80310

The default compilation for the AAU Library and Testbench is for the Intel® 80310 I/O Processor chipset. If the Intel® 80321 I/O Processor is going to be used, the AAU Library must be compiled with a "-D_USE_80321" switch. This will allow the preprocessor to define the appropriate functions, variables, and descriptor alignment for the 80321 AAU enhancements.

2.5 Compiler Options

Compiler switches for the AAU Library and Testbench are listed in Table 3.

Table 3. AAU Library and Testbench Compiler Switches

Option	Description
-D_USE_80321	Activates the AAUMemset function. The Testbench will able to perform Parity and Memset tests. The AAU Descriptors will be aligned on 64-word boundaries. The AAUDescriptors will have the extended register definitions for the Intel® 80321 I/O Processor.
-D_USE_INTERRUPTS	Links the interrupt handlers into the Debug Monitors' interrupt handlers. The Testbench will set the "Interrupt Enable" bit in each descriptor that it creates. The CleanChain procedure will look for descriptors as marked (from the ISR), before reclaiming them for use. This feature is only available when compiling with GNUPro* for the Red Hat* Red Boot monitor.
-DDEBUG_PRINTF	Implements printf statements, throughout the AAU Library and Testbench, for reporting progress, errors, etc. This is useful when first bringing up the AAU Library to watch how the AAU Library functions.
-D_USING_ARMCC_	Implements a set of macros and attribute definitions that are compatible with the ARM* Compiler. If the AAU Library and Testbench are being built with the ARM* ADS* tool suite, this option must be used to avoid errors.
-SPECS=<FILE>.SPECS	For GNUPro* only: Overrides the built-in specs file with the board specific specs file. For the IQ80310 platform set `<file>` to `iq80310`. For the IQ80321 platform, set <file> to redboot. Use this option as a compiler switch for the AAU LIbrary and as a compiler and linker switch for the Testbench.

2.6 User Defines

The following option is available in the "userDefsAAULib.h" file for the AAU Library:

Table 4. 'User Define' for AAU Descriptors

Option	Description
NUM_DESCRIPTORS	Sets the number of descriptors that the AAU Library will allocate. Increasing the number of descriptors will increase the amount of memory used. However, increasing the number of descriptors will reduce the time between the cleaning and reclaiming of descriptors. The user should benchmark the optimum number of descriptors to obtain the maximum memory and performance balance.

AAU Library Data Structures 3

The AAU Library relies on two main data structures to execute. These are the AAUDescriptor and the DescriptorTracker.

3.1 AAUDescriptor Structure

The AAUDescriptor contains the variables that are needed to build a proper AAU Descriptor in memory. Figure 3 gives an example of how to use these variables.

Figure 3. AAUDescriptor Structure

```
struct AAUDescriptor {
     void           * nda;           /* Next Descriptor Address */
     void           * sar[4];        /* Source Addresses        */
     void           * dar;           /* Destination Address     */
     unsigned int     bc;            /* Byte Count              */
     unsigned int     dc;            /* Descriptor Control      */
     void           * esar0[4];      /* Extended Src. Addresses0*/
#ifdef _USE_80321
     unsigned int     edc0;          /* Extended Desc. Cntrl. 0  */
     void           * esar1[8];      /* Extended Src. Addresses1 */
     unsigned int     edc1;          /* Extended Desc. Cntrl. 1  */
     void           * esar2[8];      /* Extended Src. Addresses2 */
     unsigned int     edc2;          /* Extended Desc. Cntrl. 2  */
     void           * esar3[8];      /* Extended Src. Addresses3 */
     unsigned int     pad[21];       /* 64-word aligned for 80321 */
#endif
     unsigned short   mark;          /* Used by ISR */
     unsigned short   asr;           /* Used by ISR */
           unsigned int     pad2[3];       /* 8-word  aligned for 80310 */
};
```

Notice that the AAUDescriptor definition contains a preprocessor check for the _USE_80321 definition. If this variable is defined, the preprocessor will include the extended registers used by the Intel® 80321 I/O Processor.

Examples of AAU Descriptors for an 80310 and 80321 processor are shown in Figure 4 and Figure 5, respectively. The reason for the padding in each descriptor is to properly align the descriptor on the proper word boundaries, as required by the AAU.

- For the 80310, the required boundary is 8-word aligned, if all 8 source addresses are used.
- For the 80321, the required word boundary is 64-word aligned, if all 32 source addresses are used.

The structures are setup to align on the maximum required word boundary, regardless of whether the maximum number of source addresses are used.

Figure 4. AAU Descriptor for Intel® 80310 I/O Processor

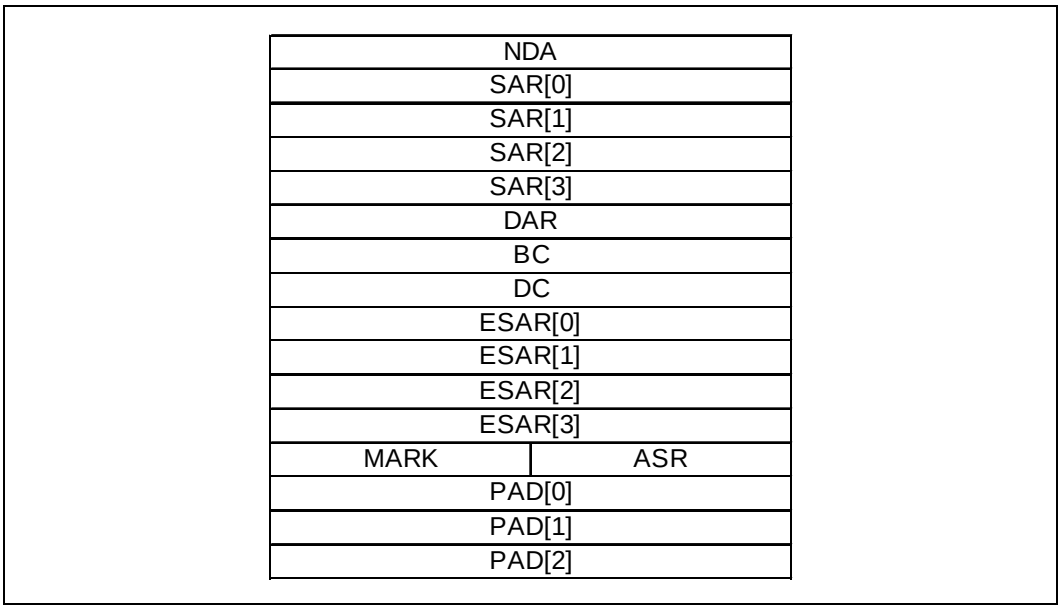
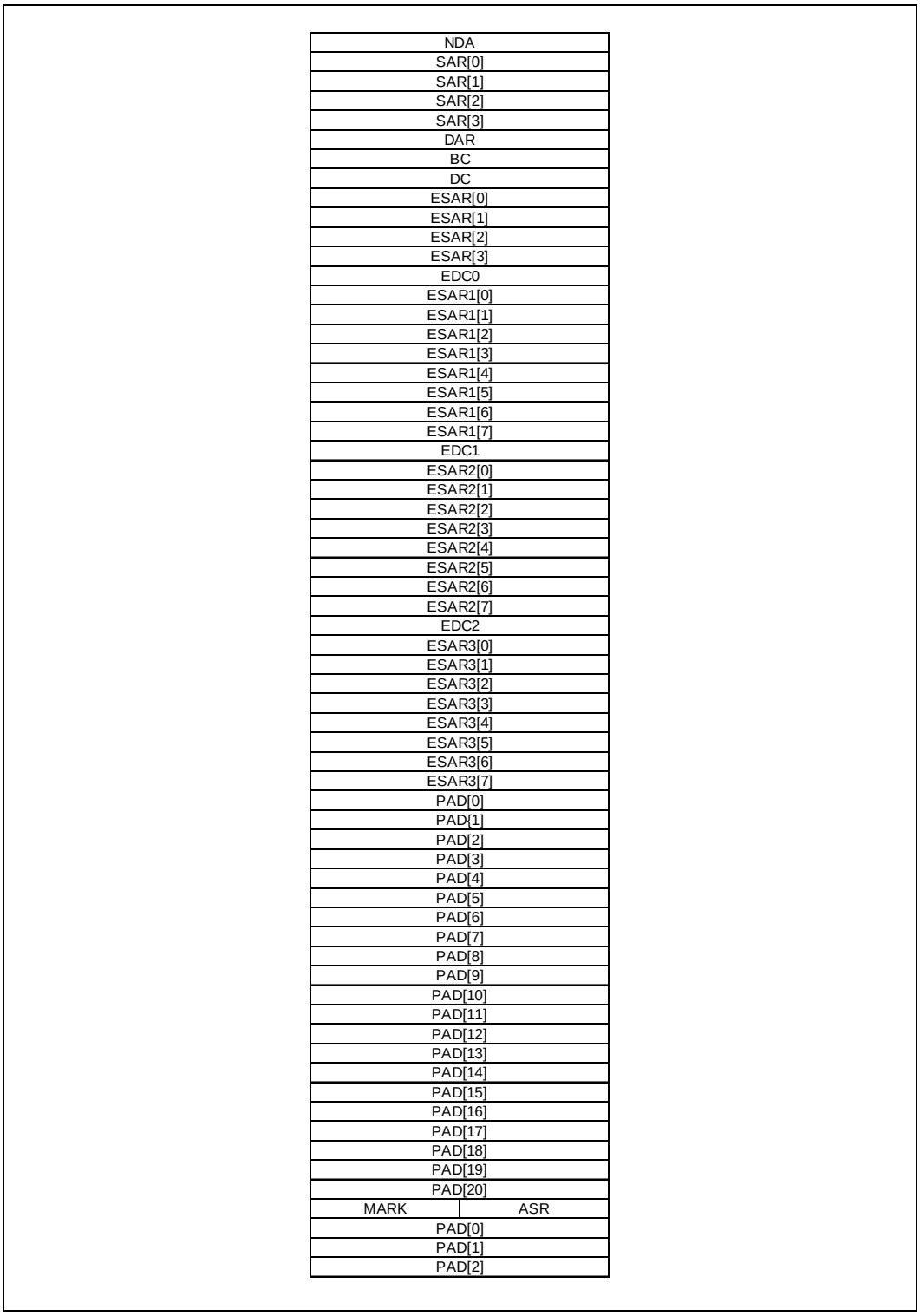


Figure 5. AAU Descriptor for Intel® 80321 I/O Processor



3.2 DescriptorTracker Structure

The DescriptorTracker structure keeps track of all the descriptors that have been allocated for the AAU Library. The number of descriptors allocated is dependent on the variable **NUM_DESCRIPTORS**, which is defined in the file **userDefsAAULib.h**.

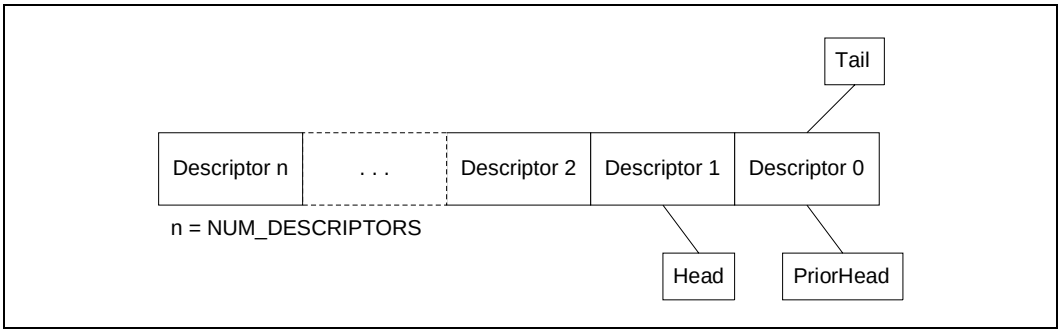
Figure 6 shows the definition of the DescriptorTracker structure.

Figure 7 shows how the pointers are assigned to the block of descriptors. As each descriptor is issued, the head pointer will move towards the higher-number descriptor, with the prior head pointer always trailing by one descriptor. This allows the AAU Library to chain descriptors together. The tail pointer is fixed at the first descriptor in the chain.

Figure 6. DescriptorTracker Structure

```
struct DescriptorTracker {
   volatile unsigned int  * acr_ptr;   /* Pointer to ACR Register */
   volatile unsigned int  * asr_ptr;   /* Pointer to ASR Register */
   volatile unsigned int  * adar_ptr;  /* Pointer to ADAR Register */
   struct AAUDescriptor   * head;      /* front of list */
   struct AAUDescriptor   * tail;      /* tail of list */
   struct AAUDescriptor   * priorhead; /* one from head - chaining */
   unsigned int           * MemtoFree; /* Keep track of mem to free */
   unsigned int             stacksize; /* Number descriptors issued */
};
```

Figure 7. DescriptorTracker: Assigning Pointers to the Block of Descriptors



AAU Library Flow 4

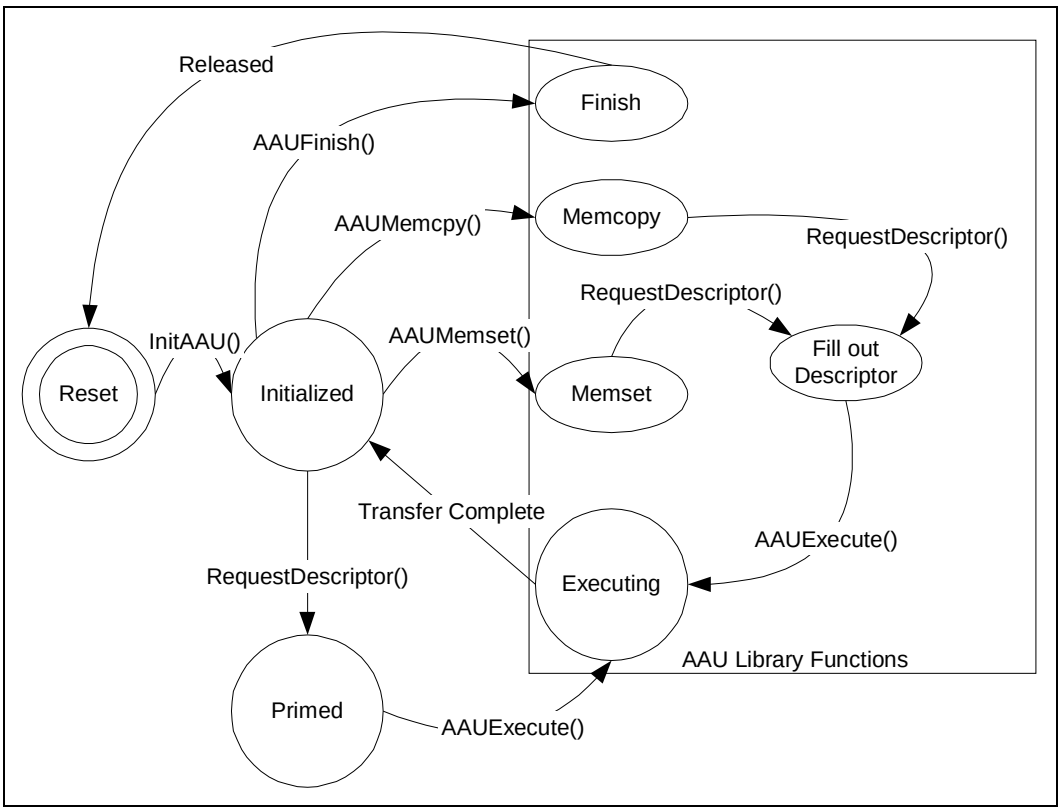
The AAU Library has four basic states, described in Table 5.

Table 5. Basic States of AAU Library

State	Description
Reset	The state entered on Reset or by calling the **AAUFinish** routine.
Initialized	The state entered by calling **InitAAU** from the Reset state or when the AAU Library finishes executing a descriptor.
Primed	The state entered when a descriptor is requested from the AAU Library.
Executing	The state when the AAU Library is processing a descriptor and/or the AAU is executing an AAU operation.

Figure 8 gives a state-diagram representation of the AAU Library.

Figure 8. AAU Library State Diagram



AAU Library APIs 5

The AAU Library APIs can be separated into the four basic states in which they relate:

- Reset
- Initialized
- Primed
- Executing

5.1 Reset

5.1.1 InitAAU

Item	Description
Prototype	int InitAAU(void);
Input	None
Output	• Success — (0) • Fail — nonzero
Purpose	To setup interrupt handling, allocate and align the AAU Descriptors, and perform a 0-byte AAU transfer, in order to facilitate chaining.
Operation	• Disable AAU • Clear ASR • If 80321, — Setup AAU Interrupts • Attach the IRQ/FIQ Handler • Call Quick Init (function to perform a 0-byte transfer) — Return Error if fail • Allocate NUM_DESCRIPTORS using malloc — Return Error if fail • Align the descriptors — If 80321, align on 64-word boundary — If 80310, align on 8-word boundary • Fill out the DescriptorTracker based on allocated descriptors

5.1.2 SetupAAUInterrupts (only used for 80321)

Item	Description
Prototype	void SetupAAUInterrupts(void);
Input	None
Output	None
Purpose	To setup the Interrupt Control and Interrupt Steering register of the 80321. **NOTE:** The AAU only supports chaining interrupts when compiling with GNUPro* for Red Hat* Red Boot.
Operation	• Set INTSTR to steer AAU Interrupts to IRQ • Set INTCTL to unmask the AAU end-of-chain (EOC) and end-of-transfer (EOT) interrupts

5.1.3 IntHandlerAttachAAU (if interrupts enabled)

Item	Description
Prototype	void intHandlerAttachAAU(void (*irq)(void), void (*fiq)(void));
Input	Pointer to AAU IRQ handler, Pointer to AAU FIQ handler
Output	None
Purpose	To install the IRQ and FIQ handler by chaining the handlers in front of the current Boot Monitor handlers. **NOTE:** The AAU only supports interrupts when compiling with GNUPro for Red Hat Red Boot.
Operation	• Read current value of IRQ/FIQ Vector (pointer to function) • Write IRQ and FIQ values over the current values • Set next_irq_service and next_fiq_service as the original values – to allow non-AAU interrupts to be serviced

5.1.4 QuickInit

Item	Description
Prototype	int QuickInit(void);
Input	None
Output	• Success — (0) • Fail — nonzero
Purpose	To allocate a single descriptor, fill it out for a 0-byte transfer, and submit it to the AAU. This allows for all future AAU Descriptors to be chained.
Operation	• Allocate single descriptor • Align it to an 8-word boundary • Fill out the descriptor • Start the AAU

5.1.5 SizeAAU

Item	Description
Prototype	int SizeAAU(void);
Input	None
Output	The size of the AAU Structure multiplied by NUM_DESCRIPTORS
Purpose	To return the number of bytes that the AAU Library has to allocate for all the descriptors requested.
Operation	Multiply size of AAUStruct by NUM_DESCRIPTORS

5.1.6 IntHandlerDetachAAU (if interrupts enabled)

Item	Description
Prototype	void intHandlerDetachAAU(void);
Input	None
Output	None
Purpose	To release the AAU FIQ and IRQ handlers from the interrupt chain. **NOTE:** The AAU only supports chaining interrupts when compiling with GNUPro* for Red Hat* Red Boot.
Operation	Sets the FIQ and IRQ vector back to their original state, before IntHandlerAttachAAU was called.

5.1.7 FlushDrainFullCache

Item	Description
Prototype	void FlushDrainFullCache(void);
Input	None
Output	None
Purpose	To drain and flush the data cache.
Operation	• Allocate 1,024 data cache lines (entire Dcache) • Invalidate the Data Cache

5.2 Initialized

5.2.1 DescriptorRequest

Item	Description
Prototype	struct AAUDescriptor * DescriptorRequest(void);
Input	None
Output	A pointer to an AAU Descriptor or NULL, if fail
Purpose	To return a pointer to an allocated AAU Descriptor that can be filled out and submitted for an AAU operation.
Operation	Return the head of the descriptor chain • If all descriptors allocated, call CleanChain

5.2.2 AAUMemcpy

Item	Description
Prototype	int AAUMemcpy (void * dest, void * src, unsigned int count);
Input	Pointer to destination data block, pointer to source data block, byte count.
Output	• Success — (0) • Fail — FAIL_DESCRIPTOR
Purpose	To execute a memory block copy, using the AAU, from source to destination of byte-count bytes.
Operation	• Request a descriptor • Fill out the descriptor with the source, destination, and byte count submitted • Set the descriptor control for a direct fill and write-enable operation • Flush the cache line that the descriptor is on • Turn on the AAU

5.2.3 AAUMemset (only used for 80321)

Item	Description
Prototype	int AAUMemset (void * src, unsigned int data, unsigned int count);
Input	Pointer to destination data block, data, byte count
Output	• Success — (0) • Fail — FAIL_DESCRIPTOR
Purpose	To execute a memory-block fill, using the AAU, of byte-count bytes to the destination address, using the source data.
Operation	• Request a descriptor • Fill out the descriptor with the destination, source data, and byte count submitted • Set the descriptor control for a block fill and write-enable operation • Flush the cache line that the descriptor is on • Turn on the AAU

5.2.4 AAUFinish

Item	Description
Prototype	void AAUFinish(void);
Input	None
Output	None
Purpose	To release the memory allocated by the AAU Library, and detach the interrupt handlers, if they were installed.
Operation	• Call free for the initial descriptor allocated for the 0-byte transfer • Call free for the block of descriptors allocated for the DescriptorTracker • Call IntHandlerDetach, if interrupts were turned on for the AAU Library

5.2.5 CleanChain

Item	Description
Prototype	int CleanChain(void);
Input	None
Output	• Success — (0) • Success — CLEAN_FAIL
Purpose	To reclaim descriptors that have been executed already by the AAU.
Operation	• Start at the tail of the descriptor chain and look for a descriptor whose address matches the ADAR register – which means it is the current descriptor — If using interrupts – look for descriptors that have 0xabcd in their mark field as well • For each descriptor that has been executed, add one back to the STACKSIZE field of the DescriptorTracker — If using interrupts, clear the mark field of the descriptor

5.2.6 FlushCacheline

Item	Description
Prototype	void FlushCacheline(struct AAUDescriptor *descriptor);
Input	Pointer to a descriptor that needs to be flushed from the data cache.
Output	None
Purpose	To flush cache lines to write out a descriptor to memory, since the AAU only operates on local memory, not cache.
Operation	Call the cache function to clean a cache line.

5.3 Primed

5.3.1 AAUExecute

Item	Description
Prototype	void AAUExecute(struct AAUDescriptor *aaudescriptor);
Input	Pointer to a descriptor to be executed.
Output	None
Purpose	To link a descriptor to the chain, flush the cache line of the previous descriptor, and start the AAU.
Operation	• Set the ANDAR of the previous descriptor to the current descriptor to be executed • Flush the cache line of the previous descriptor, so the ANDAR setting is saved to memory • Set the RESUME and ENABLE bit in the ACR to start the AAU Operation

5.4 Executing

5.4.1 IRQHandlerAAU (if interrupts enabled)

Item	Description
Prototype	void irqHandlerAAU(void);
Input	None
Output	None
Purpose	To clear the AAU Interrupt and mark a descriptor as executed. **NOTE:** The AAU only supports interrupts when compiling with GNUPro* for Red Hat* Red Boot.
Operation	Read the IRQISR (80321) or FIQISR (80310) to verify the AAU caused the interrupt • If so, write 0xFFFF to the ASR and write 0xABCD to the mark field of the current descriptor • If not, call the next IRQ interrupt service routine

5.4.2 FIQHandlerAAU (if interrupts enabled)

Item	Description
Prototype	void fiqHandlerAAU(void);
Input	None
Output	None
Purpose	To clear the AAU Interrupt and mark a descriptor as executed. **NOTE:** The AAU only supports chaining interrupts when compiling with GNUPro* for Red Hat* Red Boot.
Operation	Read the IRQ1ISR (80310) or FIQISR (80321) to verify the AAU caused the interrupt • If so, write 0xFFFF to the ASR and write 0xABCD to the mark field of the current descriptor • If not, call the next FIQ interrupt service routine

5.4.3 AAUSuspend

Item	Description
Prototype	void AAUSuspend (void);
Input	None
Output	None
Purpose	To suspend the operation of the AAU.
Operation	Clear the AA Enable bit of the ACR.

5.4.4 AAUResume

Item	Description
Prototype	void AAUResume(void);
Input	None
Output	None
Purpose	To resume operation of the AAU when it has been suspended.
Operation	Set the AA Enable bit of the ACR.

Conclusion 6

The AAU provides low-latency, high-throughput data transfer functions for I/O processing. The AAU can also perform an XOR operation in hardware on source data blocks to calculate parity. The AAU of the 80321 I/O processor has the ability to fill a block of memory with a 32-bit constant.

The AAU Library provides a set of APIs that can be used to take advantage of the features of the AAU. The Intel® 80310 I/O Processor Chipset and the Intel® 80321 I/O Processor can be supported by the AAU Library.

The AAU Library is best suited for a developer looking for a programming template from which to create their own AAU support functions for their application.

AAU Library Testbench A

The AAU Library Testbench is used to test and benchmark the AAU Library APIs. The main component of the Testbench is an experiment data structure that is used to define the experiments to run against the AAU Library.

A.1 Provided by the Testbench

- A set of APIs that is used to invoke the APIs of the AAU LIbrary
- An experiment data structure that can be easily modified to support specific test cases
- A verify procedure that validates an AAU Operation was executed correctly
- The entire source code of the Testbench
- The ability to build the Testbench with either the ARM* compiler or GNUPro* Compiler
- Project files for the ATI* Codelab|EDE and ARM* ADS* software toolchains
- A Makefile for the GNUPro* and ARM* toolchain

A.2 Basic Requirements

A.2.1 Memory Allocation

The Testbench uses the C Standard Library functions malloc and free to create and release all of the source and destination data blocks used to test the AAU Library. Therefore, malloc and free must be available in the environment for which the AAU Testbench is being used.

A.2.2 Memory Map

The AAU operates on physical addresses. The AAU Testbench uses a macro — “VIRT_TO_PHY” — that translates virtual addresses into physical addresses. Another macro — “PHY_TO_VIRT” — is used to translate back, from physical addresses into virtual addresses.

These two macros must be modified per platform to generate the proper physical and virtual addresses.

A.2.3 Compiling for Intel® 80321 I/O Processor versus Intel® 80310 I/O Processor

The default compilation for the Testbench is for the Intel® 80310 I/O Processor chipset. If the Intel® 80321 I/O processor is going to be used, the Testbench must be compiled with a “-D_USE_80321” switch. This will allow the preprocessor to define the appropriate functions, variables, and descriptor alignment for the Intel® 80321 I/O processor AAU enhancements.

A.3 Compiler Options

Table A-1 gives definitions for controlling how the Testbench is compiled. These are options that should be passed to the compiler to control how the Testbench is built.

Table A-1. Testbench Compiler Options

Option	Effect of Using the Option
-D_USE_80321	The AAUMemset function will be activated. The Testbench will able to perform Parity and Memset tests. The AAUDescriptors will be aligned on 64-word boundaries. The AAUDescriptors will have the extended register definitions for the Intel® 80321 I/O Processor.
-D_USE_INTERRUPTS	The Testbench will set the “Interrupt Enable” bit in each descriptor that it creates. **NOTE:** This option is only available when compiling GNUPro* for Red Hat* Red Boot.
-DDEBUG_PRINTF	This option will use printf statements throughout the Testbench to report progress, errors, etc. Printf statements will still be generated by the Testbench if this option is not enabled, but no debugging information will be output.
-D_USING_ARMCC_	This option will use a different set of macros and attribute definitions to be compatible with the ARM* Compiler. **NOTE:** If the AAU Library and Testbench are being built with the ARM* ADS* tool suite, this option must be activated to avoid errors.
-SPECS=<FILE>.SPECS	For GNUPro only: Overrides the built-in specs file with the board specific specs file. For the IQ80310 platform set `<file>` to `iq80310`. For the IQ80321 platform, set <file> to redboot. Use this option as a compiler switch for the AAU LIbrary and as a compiler and linker switch for the Testbench.

A.3.1 Testbench Data Structures

The Testbench AAU_Experiment structure is the main component in the Testbench. It is used to define all the experiments to be executed on the AAU Library.

This component is located in “**AAUTestbench.h**” and can be easily modified to remove or add any experiments desired.

Figure 9. Testbench’s ‘AAU_Experiment’ Component

```
struct AAU_Experiment {
    int     run_test;       /* 1=Run Test, 0=Skip                */
    enum    test_type test; /* 0=xor, 1=copy, 2=fill, 3=parity, 4=C-lib memcpy */
    PU32    src_address[NUM_ADDRESS];  /* Total source addresses */
    PU32    dest_address;   /* Only one destination needed       */
    U32     address_used;   /* Total addresses used (src only)   */
    U32     bytecount;      /* Number of bytes                   */
    U32     data;           /* Used for Fill Operations          */
    int     PassFail;       /* 0=PASS, Nonzero=FAILCODE          */
    U32     cycles;         /* # Cycles to complete AAU function */
};

enum       test_type  {xor=0, copy, fill, parity, libcpy};
```

A.3.2 Testbench APIs

A.3.2.1. Main

Item	Description
Prototype	int main(void);
Input	None
Output	• Success — (0) • Fail — Nonzero
Purpose	To initialize the AAU, and call all the Testbench functions.
Operation	• Call InitAAU from the AAU Library • For Each Experiment — Call CreateBlocks to allocate source and destination data blocks — Call WriteDataToMemory to generate source data — Call RunExperiment to execute the AAU Operation — Call VerifyResults to validate the AAU Operation passed • Call AAUFinish to Release the AAU Library • Call PrintSummary to show a summary of pass/fail and cycle count for each experiment

A.3.2.2. CreateBlocks

Item	Description
Prototype	int Create_Blocks(int index);
Input	Index to what experiment is being executed.
Output	• Success — (0) • Fail — Nonzero
Purpose	To allocate source and destination data blocks for use by the Testbench.
Operation	• Allocate Source block(s) – based on address_used element in the AAU_Experiment data structure used by this experiment. • Allocate a Destination block

A.3.2.3. WriteDataToMemory

Item	Description
Prototype	int WriteDataToMemory(int index);
Input	Index to what experiment is being executed.
Output	• Success — (0) • Fail — Nonzero
Purpose	To write source data to the block(s) of source addresses that were allocated in CreateBlocks.
Operation	For every source byte allocated, write a semi-unique data value to that address.

A.3.2.4. RunExperiment

Item	Description
Prototype	int Run_Experiment(int index);
Input	Index to what experiment is being executed
Output	• Success — (0) • Fail — Nonzero
Purpose	To execute the experiment, based on the experiment type.
Operation	• If copy test – call AAUMemcpy with source, destination, byte count for experiment • If xor test – call DescriptorRequest, fill out descriptor, and call AAUExecute • If libcpy test – call the C Library Memcpy with source, destination, byte count for experiment • If fill test (80321 only) – call AAUMemset with source, data, byte count for experiment

A.3.2.5. VerifyResults

Item	Description
Prototype	int VerifyResults(int index);
Input	Index to what experiment is being executed.
Output	• Success — (0) • Fail — Nonzero
Purpose	To verify the experiment was completed correctly.
Operation	• If copy test – Verify the destination values match the source values. • If xor test – Verify a software xor of all source values match the destination values. • If libcpy test – Verify the destination values match the source values. • If fill test (80321 only) – Verify the destination values match the data.

A.3.2.6. PrintSummary

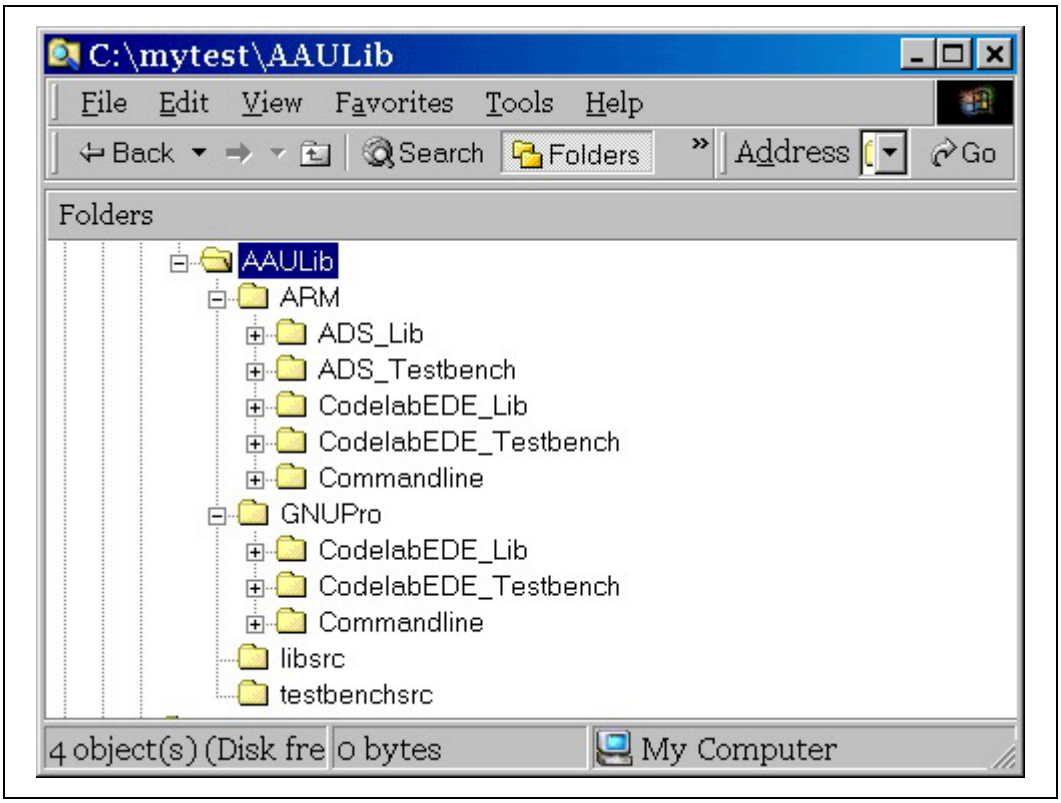
Item	Description
Prototype	void PrintSummary(void);
Input	None
Output	None
Purpose	To print a summary of the experiments that were executed.
Operation	For each experiment that was executed, print the Pass/Fail result, and cycles it took to execute the AAU operation.

Building the AAU Library and TestbenchB

B.1 Directory Structure

Figure B-1 shows the directory structure of the AAU Library. The displayed folders are contained in the AAULib.zip file, available on the corporate Web site.

Figure B-1. AAU Library Directory Structure



- ARM subdirectory contains project files for ATI* Codelab|EDE and ARM* ADS. It also contains a directory for building with a Makefile.
- GNUPro subdirectory contains project files for ATI Codelab|EDE. It also contains a directory for building with a Makefile.
- Libsrc subdirectory contains all of the source and header files for the AAU Library.
- Testbenchsrc subdirectory contains all of the source and header files for the Testbench.

B.2 Building the AAU Library and Testbench with the ARM* Compiler

B.2.1 Using ATI* Codelab|EDE

- Open the project file “CodelabEDE_Testbench.dsw” located under AAULib->ARM->CodelabEDE_Testbench
  - The Testbench project file contains a dependency on the AAU Library, which means that both projects can be built from the Testbench project file.
- Adjust the project settings for the AAU Library and the Testbench to have the desired command-line options, as defined in Section A.3, “Compiler Options” on page A-28
- Select Build->Rebuild All to rebuild the AAU Library and Testbench
- The Testbench executable file, “CodelabEDE_Testbench.axf” will be located under AAULib->ARM->CodelabEDE_Testbench->Output
- This file can be loaded and run under ARM* Angel on the IQ80310 platform or the IQ80321 platform, depending on the compiler options selected

The AAU Library can be compiled without the Testbench by using the project file “CodelabEDE_Lib.dsw” located under AAULib->ARM->CodelabEDE_Lib. The AAU Library output file, “aau.lib” is located under AAULib->ARM->CodelabEDE_Lib->Output.

B.2.2 Using ARM* ADS

- Open the project file “ADS_Lib.mcp” located under AAULib->ARM->ADS_Lib
- Open the project file “ADS_Testbench.mcp” located under AAULib->ARM->ADS_Testbench
- The 80310 and 80321 are the two targets available in these project files
- Select the desired target
- Adjust the compiler settings for the AAU Library and Testbench to have the desired command-line options, as defined in Section A.3, “Compiler Options” on page A-28
- Select Make in each project to build the AAU Library and Testbench
- The Testbench executable file, “ADS_Testbench.axf” will be located under AAULib->ARM->ADS_Testbench->Output
- This file can be loaded and run under ARM Angel on the IQ80310 platform or the IQ80321 platform, depending on the compiler options selected

The AAU Library, “ADS_Lib.a” is compiled separately from the Testbench and will be located under AAULib->ARM->ADS_Lib->Output

B.2.3 Using Makefiles

- There are four Makefiles for the AAU Library and Testbench located under AAULib->ARM->Commandline
  - 80310_Lib_Makefile – This file will create the AAU Library for the IQ80310 platform
  - 80310_Testbench_Makefile – This file will create the Testbench for the IQ80310 platform
  - 80321_Lib_Makefile – This file will create the AAU Library for the IQ80321 platform
  - 80321_Testbench_Makefile – This file will create the Testbench for the IQ80321 platform
- To create the AAU Library and Testbench for the IQ80310 platform
  - Edit the 80310_Lib_Makefile and 80310_Testbench_Makefile for the desired compiler options, as defined in Section A.3, “Compiler Options” on page A-28
  - Enter “make clean –f 80310_Lib_Makefile”
  - Enter “make clean –f 80310_Testbench_Makefile”
  - Enter “make –f 80310_Lib_Makefile”
  - Enter “make –f 80310_Testbench_Makefile”
  - The output files, “aau.lib” and “aautestbench.axf” will be located in AAULib->ARM->Commandline->Output
- To create the AAU Library and Testbench for the IQ80321 platform
  - Edit the 80321_Lib_Makefile and 80321_Testbench_Makefile for the desired compiler options, as defined in Section A.3, “Compiler Options” on page A-28
  - Enter “make clean –f 80321_Lib_Makefile”
  - Enter “make clean –f 80321_Testbench_Makefile”
  - Enter “make –f 80321_Lib_Makefile”
  - Enter “make –f 80321_Testbench_Makefile”
  - The output files, “aau.lib” and “aautestbench.axf” will be located in AAULib->ARM->Commandline->Output

B.3 Building the AAU Library and Testbench with the GNUPro* Compiler

B.3.1 Using ATI* Codelab|EDE

- Open the project file “CodelabEDE_Testbench.dsw” located under AAULib->GNUPro->CodelabEDE_Testbench
  - The Testbench project file contains a dependency on the AAU Library, which means that both projects can be built from the Testbench project file.
- Adjust the project settings for the AAU Library and the Testbench to have the desired command-line options, as defined in Section A.3, “Compiler Options” on page A-28
- Select Build->Rebuild All to rebuild the AAU Library and Testbench
- The Testbench executable file, “CodelabEDE_Testbench.elf” will be located under AAULib->GNUPro->CodelabEDE_Testbench->Output
- This file can be loaded and run under Red Hat* Red Boot on the IQ80310 platform or the IQ80321 platform, depending on the compiler options selected
- The AAU Library can be compiled without the Testbench by using the project file “CodelabEDE_Lib.dsw” located under AAULib->GNUPro->CodelabEDE_Lib. The AAU Library output file, “aau.lib” is located under AAULib->GNUPro->CodelabEDE_Lib->Output.

B.3.2 Using Makefiles

- There are four Makefiles for the AAU Library and Testbench located under AAULib->GNUPro->Commandline
  - 80310_Lib_Makefile – This file will create the AAU Library for the IQ80310 platform
  - 80310_Testbench_Makefile – This file will create the Testbench for the IQ80310 platform
  - 80321_Lib_Makefile – This file will create the AAU Library for the IQ80321 platform
  - 80321_Testbench_Makefile – This file will create the Testbench for the IQ80321 platform
- To create the AAU Library and Testbench for the IQ80310 platform
  - Edit the 80310_Lib_Makefile and 80310_Testbench_Makefile for the desired compiler options, as defined in Section A.3, “Compiler Options” on page A-28
  - Enter “make clean –f 80310_Lib_Makefile”
  - Enter “make clean –f 80310_Testbench_Makefile”
  - Enter “make –f 80310_Lib_Makefile”
  - Enter “make –f 80310_Testbench_Makefile”
  - The output files, “aau.lib” and “aautestbench.elf” will be located in AAULib->GNUPro->Commandline->Output
- To create the AAU Library and Testbench for the IQ80321 platform
  - Edit the 80321_Lib_Makefile and 80321_Testbench_Makefile for the desired compiler options, as defined in Section A.3, “Compiler Options” on page A-28
  - Enter “make clean –f 80321_Lib_Makefile”
  - Enter “make clean –f 80321_Testbench_Makefile”
  - Enter “make –f 80321_Lib_Makefile”
  - Enter “make –f 80321_Testbench_Makefile”
  - The output files, “aau.lib” and “aautestbench.elf” will be located in AAULib->GNUPro->Commandline->Output