AMD

PCI Bus Software Support for Alchemy™ Au1500™ Processor from AMD

Application Note

Revision: **1.3**
Issue Date: **August 2002**

Contacts

www.amd.com pcs.support@amd.com

Trademarks

1. Introduction

The Au1500 System-On-a-Chip, SOC[1], features an integrated 33/66MHz PCI 2.2 compliant bus for connecting to a variety of external peripherals. This document describes software techniques for supporting the Au1500's integrated PCI bus.

This document focuses on the software support and considerations that are needed for an Au1500 integrated PCI controller configured as a host bridge.

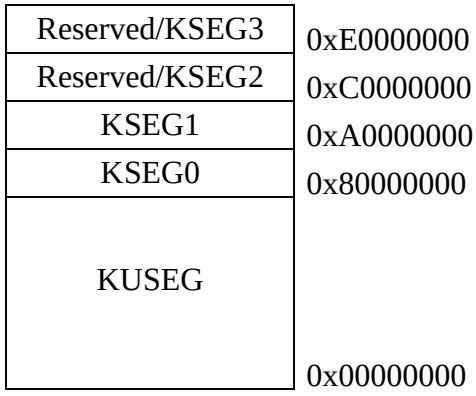
For information pertaining to the Au1500 configured as a PCI satellite, see "Satellite Mode" on page 17.

This document assumes the reader is familiar with the PCI Local Bus Specification version 2.2 [2].

2. MIPS32™ Architecture Memory Map

In the MIPS architecture, all addresses (instruction fetches, data loads and data stores) are virtual addresses [3]. As a result, address translation is always performed on program instruction fetches and data accesses. The type of address translation depends upon the upper bits of the program address. The MIPS architecture defines the KUSEG, KSEG0 and KSEG1 regions according to these upper bits of the program's virtual address. The program's 32-bit memory space is thus divided:

Figure 1: MIPS 32-bit Memory Map



The KUSEG region extends from 0x00000000 to 0x7FFFFFFF, a 2GB space which uses translation look-a-side buffers, TLBs, to determine the corresponding physical address. The KUSEG region is accessible while the CPU is in either user mode or kernel mode.

The KSEG0 region extends from 0x80000000 to 0x9FFFFFFF, a 512MB space which has a direct correlation to a physical address. In addition, the KSEG0 region is inherently cacheable; meaning that both instruction and data caching is occuring for references to this area. The KSEG0 region is only accessible while the CPU is in kernel mode.

The KSEG1 region extends from 0xA0000000 to 0xBFFFFFFF, a 512MB space which also has a direct correlation to a physical address. However, the KSEG1 region is inherently non-cacheable; meaning that any instruction or data reference will bypass the cache and directly access physical memory. The KSEG1 region is only accessible while the CPU is in kernel mode.

For the KSEG0 and KSEG1 regions, the corresponding physical address is bits 28:0 of the virtual address with address bits 31:29 zero. That is, KSEG0 and KSEG1 map directly onto the first 512MB of physical memory. For example, KSEG0 address 0x80000000 and KSEG1 address 0xA0000000 both map directly onto physical address 0x00000000. The KSEG0 and KSEG1 regions provide two views of physical memory; one cacheable and one non-cacheable.

The address translation mechanism of the MIPS architecture always presents a physical address to the memory controllers (and other address decode logic).

3. Au1500 36-Bit Physical Addresses

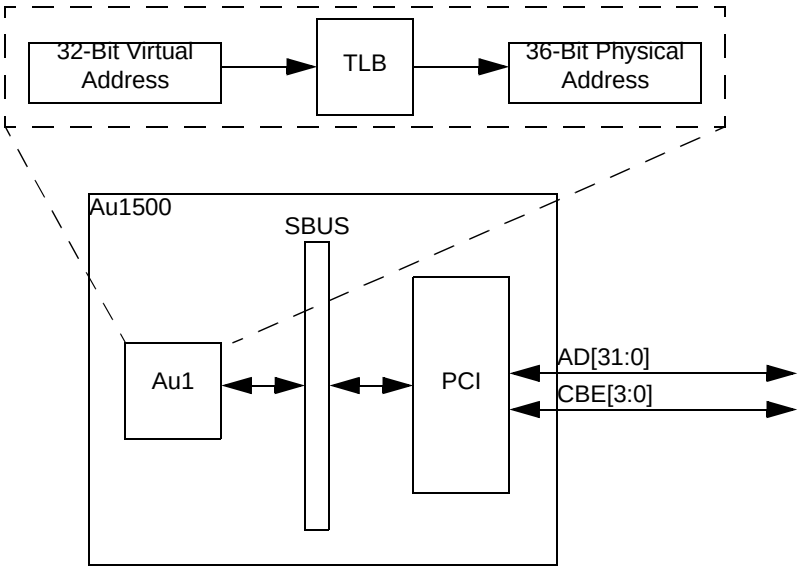
From the information/memory map in “MIPS 32-bit Memory Map” on page 3, it is apparent that there is no room to locate a 32-bit PCI address space directly within the MIPS 32-bit memory map! Fortunately, the MIPS32 architecture specifies a 36-bit physical address space to accomodate large address spaces, such as the PCI bus. The Au1500 takes advantage of the 36-bit physical address and locates the PCI address space as such:

Table 1. Au1500 PCI Address Space Mapping

36-Bit Physical Address	PCI Function
0x4 XXXXXXXX	PCI Memory Space
0x5 XXXXXXXX	PCI I/O Space
0x6 XXXXXXXX	PCI Configuration Space

Since 36-bit physical addresses are not directly visible to the processor (i.e. through the KSEG0 or KSEG1 regions), the PCI space must be mapped into the system using a TLB and then accessed using virtual address pointers (a 32-bit pointer which is translated by a TLB into a 36-bit physical address). The address translation steps are depicted in “Au1500 PCI Address Translation” on page 5.

Figure 2: Au1500 PCI Address Translation



The PCI controller decodes the upper bits of the 36-bit system bus physical address to determine if the cycle is a configuration, memory or I/O cycle. The lower 32-bits of the 36-bit system bus physical address are connected to the PCI bus AD[31:0].

4. Software Support

The technique for supporting the PCI bus is largely dependent upon the driver model for the software run-time system (i.e. operating system). For example, drivers in the Linux operating system run in “kernel” mode (i.e. privileged and likely in either KSEG0 or KSEG1 regions), where as drivers in the Windows CE operating system run in “user” mode (i.e. each driver is in its own thread with its own address space).

In Linux, drivers run at the kernel privileged mode and typically execute from either KSEG0 or KSEG1 space. That is, drivers expect the registers and/or memory of a peripheral to be directly visible in the KSEG0 or KSEG1 regions. Enabling PCI bus support in this environment is challenging since, at the time of this writing, the drivers do not have a readily-available facility for mapping a 36-bit address into a 32-bit address which can be accessed by the driver. Nonetheless, a scheme is in place to support Au1500 PCI in Linux.

In Windows CE, drivers run in a non-privileged mode as a thread of the I/O process, DEVICE.EXE. Each device driver has its own address space, and must explicitly map in the the physical address(es) of the peripheral’s resources. Device drivers utilize the MmMapIoSpace() function call which does support 36-bit physical addresses and greatly facilitates PCI bus support on the Au1500.

Many real-time operating systems, RTOS, do not use virtual memory, but rather execute entirely from the KSEG0 and/or KSEG1 regions. Device drivers for these operating systems may expect the peripheral resources to be directly memory mapped. Without virtual memory, supporting the PCI bus

is more challenging, but in most cases can be easily solved with the use of static or “wired” TLB entries to directly map regions on the PCI bus of interest to the RTOS.

Most other operating systems or applications without operating systems can use one or more of the techniques described herein.

NOTE: This document assumes a little-endian core configuration. Since PCI is inherently little-endian, with both the Au1 core and PCI in little-endian, each has the same “view” of memory. Utilizing the Au1 core in big-endian mode is possible, and the issues are discussed in “Big-Endian Considerations” on page 17.

5. Au1500 PCI Host Bridge Configuration

The Au1500 integrated PCI controller must be configured before it is utilized. This document assumes configuration of a host bridge (via the PCI_CFG pin). The steps necessary to configure the controller are as follows.

1. Enable the PCI bus 33 and/or 66 MHz clock source.
2. Take the Au1500 PCI controller and PCI bus out of reset.
3. Configure the Au1500 PCI controller registers.

The PCI controller requires these steps, in order, to ensure proper operation.

5.1 Enable PCI Bus Clock

The 33 or 66 MHz PCI clock can be generated either internally or externally, depending upon the application/board-design. For issues pertaining to PCI clock generation, see the “PCI Clock Generation” applications note.

If the PCI clock is generated internally, the clock generator registers must be programmed to generate the PCI clock.

If the PCI clock is generated externally, it must be enabled and running.

5.2 Enable PCI controller

The PCI bus signal RST# must be asserted for at least 100 microseconds after the PCI clock has stablized. It then must be negated for a minimum of 5 PCI clocks before accessing the PCI bus. Additional reset timing parameters are in the PCI 2.2. specification: “4.2.3.2. Timing Parameters” and “4.3.2. Reset”.

The PCI bus RST# signal also resets the Au1500 PCI controller. As such, RST# must be driven low and then high according to the PCI reset timing to reset both the PCI bus and the Au1500 integrated PCI controller. The Au1500 PCI controller must be taken out of reset and 5 PCI clocks elapsed **before** any of its configuration registers can be accessed.

On the Au1500, GPIO200 defaults to an output that drives low (a zero). This pin is intended to drive the PCI bus RST# signal. If the board-design utilizes GPIO200 in this capacity, then software must drive GPIO200 according to the PCI bus reset timing. The following code snippet de-asserts RST# by driving GPIO200 high (no need to drive it low since it defaults to low).

```
li t0, 0xB1700000 /* base address of GPIO2 */
li t1, 3          /* gpio2_enable[MR=1,CE=1]
sw t1, 0x0010(t0) /* gpio2_enable, provide clocks */
li t1, 1          /* gpio2_enable[MR=0,CE=1]
sw t1, 0x0010(t0) /* gpio2_enable, take away reset */
li t1, 1          /* gpio2_dir[GPIO200=1], output
sw t1, 0x0000(t0) /* gpio2_dir, set GPIO200 as output */
li t1, 0x00010001 /* gpio2_output[GPIO200ENA=1,GPIO200=1] */
sw t1, 0x0008(t0) /* gpio2_output, GPIO200 = 1 */
```

Note: *NOTE: If GPIO200 (or other GPIO) is not used to control PCI RST#, then PCI RST# must also be tied to the Au1500 RSTIN signal. See the Au1500 Specification Update for additional information.*

5.3 Configure the PCI Controller

The PCI controller contains configuration registers which are located at physical address 0x0 14005000 (KSEG1 address 0xB4005000). A simple host bridge setup is provided in Table 2 on page 7.

Table 2. Host Bridge Configuration

Register	KSEG1 Address	Value
pci_cmem	0xB4005000	0x00000000
pci_config	0xB4005004	0x0000000F
pci_b2bmask_cch	0xB4005008	0x00000000
pci_b2bbase0_venid	0xB400500C	0x00000000
pci_b2bbase1_id	0xB4005010	0x00000000
pci_mwmask_dev	0xB4005014	0xE0000000
pci_mwbase_rev_ccl	0xB4005018	0x00000000
pci_err_addr	0xB400501C	-
pci_spec_intack	0xB4005020	-
pci_id	0xB4005100	0x00001755
pci_statcmd	0xB4005104	0x02A00356
pci_classrev	0xB4005108	0x00000000

Table 2. Host Bridge Configuration (Continued)

Register	KSEG1 Address	Value
pci_hdrtype	0xB400510C	0x00000000
pci_mbar	0xB4005110	0x00000008
pci_timeout	0xB4005140	0x00000080

The Au1500 host bridge is enabled for operation when pci_statcmd[BUS_MASTER], bit 2, is set. The bridge will not generate PCI cycles until this bit is set. Furthermore, pci_statcmd[MEMORY_SPACE], bit 1, must be set if the Au1500 memory window (see next section) is to be utilized.

5.3.1 Au1500 Memory Window

The registers pci_mwmask_dev, pci_mwbase_rev_ccl and pci_mbar provide a window into the Au1500 memory. The window size can vary, as illustrated in Table 3 on page 8.

Table 3. Au1500 Memory Windows

Window Size	mwmask	Window Size	mwmask
64KB	0xFFFF	16MB	0xFF00
128KB	0xFFFE	32MB	0xFE00
256KB	0xFFFC	64MB	0xFC00
512KB	0xFFF8	128MB	0xF800
1MB	0xFFF0	256MB	0xF000
2MB	0xFFE0	512MB	0xE000
4MB	0xFFC0	1GB	0xC000
8MB	0xFF80	2GB	0x8000

The example PCI controller configuration above creates a 512MB window starting at Au1500 physical address 0x0000000, which is visible in PCI memory space starting at PCI address 0x00000000. The configuration creates a simple 1:1 mapping of Au1500 memory space into the PCI space, and vice-versa. The 512MB window intentionally corresponds to the same size as the MIPS KSEG0/KSEG1 region which covers all SDRAM and static bus memories.

5.3.2 Memory Window Considerations

Software likely needs at least these bus address translation routines:

1. Au1 KSEG0/1 address to PCI address,
2. PCI address to Au1 KSEG0/1 address,

3. Au1 physical address to PCI address, and
4. PCI address to Au1 physical address

The bus address translation routines are necessary since a MIPS virtual address is not [necessarily] equivalent to a PCI physical address. More specifically, the pci_mwbase and pci_mbar registers permit mapping a window of Au1500 memory anywhere in PCI memory space. These bus address translation routines know how to compute, for a given memory location, a PCI address from an Au1 address, and vice versa. In particular, device drivers for PCI devices should use these bus address translation routines when handling PCI memory addresses, particularly for DMA pointers/buffers.

For example, with the simple host bridge configuration above, the bus address translation routines merely manipulate the three most significant bits of the address corresponding to the MIPS KSEG0/ KSEG1 designations. In more complicated schemes, the bus address translation routines must include the pci_mwbase and/or pci_mbar value in the address calculations.

In determining the Au1500 memory window size and location, the following items should be taken into consideration:

1. PCI accesses into Au1500 memory must occur to memory that is pre-fetchable.
2. The Au1 MIPS vector table may be exposed in PCI space, so it may be possible to damage the vector table.
3. The Au1500 memory window is located in PCI space via pci_mbar. This register is not visible to the PCI auto-configuration routine.

PCI-initiated memory accesses to Au1500 memory must be to pre-fetchable memory. When a PCI-initiated memory access occurs to Au1500 memory, up to 8 words are transferred. As a result, it is not possible to access the integrated peripherals or other memory locations (such as FIFOs, registers, etc.) that have side-effects. In most instances, only the Au1500 SDRAM should be accessed from PCI.

Depending upon how the Au1500 memory window is configured, the Au1 core MIPS vector table in RAM (virtual address 0x80000000, physical address 0x00000000) may be exposed in the PCI space. Depending upon the application and/or the development status of the associated software, this may not be desirable. If the Au1 core vector table must be protected, then the pci_mwbase register must be changed, along with the bus address translation routines, to avoid exposing the MIPS vector table in PCI space.

During the PCI bus auto-configuration, PCI device MBARs are programmed with address ranges that do not conflict with other PCI devices. However, the Au1500 itself is not visible during a PCI bus auto-configuration, so the Au1500 pci_mbar register is programmed independent of the standard PCI auto-configuration code. See “Auto-Configuration Considerations” on page 11 for more information.

5.3.3 Application-specific Values

The registers pci_b2bmask_cch, pci_b2bbase0_venid, pci_b2bbase1_id, pci_b2bbase1_id, pci_mwmask_dev, pci_id, pci_classrev all contain bitfields that correlate to values in the Au1500 PCI controller configuration space header. The values that are appropriate for the PCI configuration space header are determined by the application. Consult “Appendix D: Class Codes” of the PCI 2.2 specification, as well as the PCI Special Interest Group, PCISIG, for appropriate values.

6. PCI Configuration Cycles

PCI configuration cycles are used to configure devices that are attached to the PCI bus. The configuration space is normally scanned by software to identify and more importantly configure the devices for proper behavior on the PCI bus.

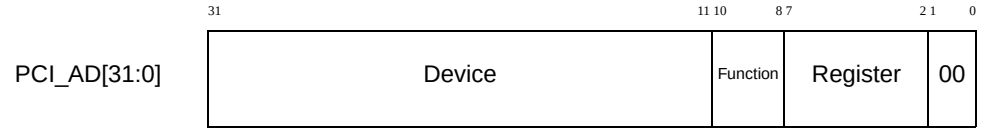
The PCI configuration space address is derived from the Au1 36-bit physical address: bits [35:32]=6 indicate PCI configuration space, bit [31] indicates a Type 0 or Type 1 configuration cycle, and bits [30:2] are copied directly to the PCI_AD[30:2].

Note: *NOTE: The Au1500 PCI controller does not respond to a configuration cycle from itself. As such it will not be detected by a PCI bus scan.*

6.1 Type 0 Configuration Cycles

Type 0 configuration cycles are the most common. The address phase of a Type 0 configuration cycle is the following:

Figure 3: Type 0 Configuration Cycle



Bit 31 of the 36-bit physical address is a zero and causes the PCI controller to emit “00” on PCI_AD[1:0]. During the address phase, only one bit is permitted to be set in the Device number field. The single “1” bit denotes the IDSEL of a PCI target device for the configuration cycle. A total of 20 target devices is supported. The following table enumerates the 20 possible configuration spaces.

Table 4. Au1500 Type 0 Configuration Cycle Base Addresses

Device Number	IDSEL	36-Bit Physical Address	Device Number	IDSEL	36-Bit Physical Address
0	PCI_AD[11]	0x6 00000800	10	PCI_AD[21]	0x6 00200000
1	PCI_AD[12]	0x6 00001000	11	PCI_AD[22]	0x6 00400000
2	PCI_AD[13]	0x6 00002000	12	PCI_AD[23]	0x6 00800000
3	PCI_AD[14]	0x6 00004000	13	PCI_AD[24]	0x6 01000000
4	PCI_AD[15]	0x6 00008000	14	PCI_AD[25]	0x6 02000000
5	PCI_AD[16]	0x6 00010000	15	PCI_AD[26]	0x6 04000000
6	PCI_AD[17]	0x6 00020000	16	PCI_AD[27]	0x6 08000000

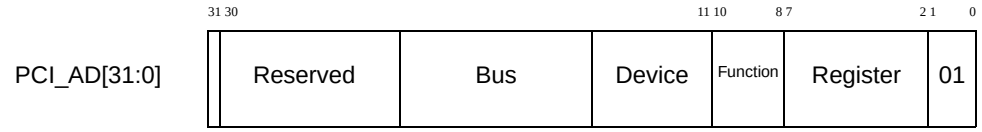
Table 4. Au1500 Type 0 Configuration Cycle Base Addresses (Continued)

Device Number	IDSEL	36-Bit Physical Address	Device Number	IDSEL	36-Bit Physical Address
7	PCI_AD[18]	0x6 00040000	17	PCI_AD[28]	0x6 10000000
8	PCI_AD[19]	0x6 00080000	18	PCI_AD[29]	0x6 20000000
9	PCI_AD[20]	0x6 00100000	19	PCI_AD[30]	0x6 40000000

6.2 Type 1 Configuration Cycles

Type 1 configuration cycles are used to configure PCI devices on a remote PCI bus. The address phase of a Type 1 configuration cycle is the following:

Figure 4: Type 1 Configuration Cycle



Bit 31 of the 36-bit physical address is a one and causes the PCI controller to emit “01” on PCI_AD[1:0]. Bits [30:2] of the 36-bit physical address are copied directly to bits AD[30:2]. The following table illustrates the 36-bit physical addresses for Type 1 configuration space addresses.

Table 5. Au1500 Type 1 Configuration Cycle Base Addresses

Bus	36-Bit Physical Address	Bus	36-Bit Physical Address
0	0x6 8000df00	4	0x6 8004df00
1	0x6 8001df00	5	0x6 8005df00
2	0x6 8002df00	255	0x6 80FFdf00
3	0x6 8003df00	bb (<= 255)	0x6 80bbdf00

In the table above, “df” represents the device number and function number which form PCI_AD[15:8], and “bb” represents the bus number which forms PCI_AD[23:16].

6.3 Auto-Configuration Considerations

At boot and/or run-time, PCI auto-configuration software scans the PCI configuration space in search of PCI devices. When a device is detected, the device is allocated PCI memory and/or I/O space and its MBAR(s) are programmed with a unique base address. When completed, the PCI bus scan creates a conflict-free address map for the PCI bus.

In general, standard PCI auto-configuration software works as intended with the Au1500 PCI bus, but there are a few additional items to take into consideration (these items are handled outside the standard PCI bus scan):

1. The Au1500 PCI controller does not respond to a configuration cycle from itself. As such, the Au1500 memory window is **not** detected during a PCI bus scan.
2. Devices on the PCI bus which are fast-back-to-back capable should be allocated space in the area defined by the Au1500 PCI controller back-to-back windows.

The Au1500 PCI controller does not respond to configuration cycles initiated by itself. Consequently, the PCI auto-configuration software is not aware of the Au1500 memory window. This behavior has two inter-related side-effects which must be accomodated by software:

- The location and size of the Au1500 memory window must be determined in advance, so that...
- The PCI auto-configuration software must allocate memory regions that **do not conflict** with the Au1500 memory window.

PCI auto-configuration software is usually instructed as to the address ranges from which PCI device memory and I/O addresses can be assigned; therefore, once the Au1500 memory window location (in PCI space) and size are known, the PCI auto-configuration software must be instructed to avoid the Au1500 memory window.

The Au1500 PCI controller supports fast back-to-back transactions. The pci_b2bmask, pci_b2bbase0 and pci_b2bbase1 registers form two windows for utilizing fast back-to-back PCI signalling on outgoing memory transactions. To take advantage of this performance feature, the following must be accomodated by software:

- The location and size of the fast back-to-back windows must be determined in advance, so that...
- The PCI auto-configuration software should allocate memory regions from the fast back-to-back windows for devices that advertise the fast back-to-back capability.

During the PCI bus-scan, for those PCI devices that are fast back-to-back capable (Command register bit 9 is set), the PCI auto-configuration software assigns, if possible, a memory region from the Au1500 PCI controller fast back-to-back windows. PCI auto-configuration software is usually instructed as to the address ranges to which fast back-to-back transactions are possible; therefore, once the Au1500 fast back-to-back windows location (in PCI space) and size are known, the PCI auto-configuration software is provided the fast back-to-back address range so that fast back-to-back devices can be assigned base addresses within this range.

Note: *NOTE: When allocating PCI memory addresses, the range 0xC0000000 to 0xDFFFFFFF may be desirable since a 1:1 processor-to-PCI mapping can be created. More specifically, an Au1 core access to MIPS KSEG2 virtual address 0xC0000000 can be mapped via TLBs to a PCI access to 0xC0000000, a 1:1 mapping. In certain software environments this may greatly simplify accessing the PCI devices (e.g. the PCI device MBAR value can be used as a pointer). This 512MB address range should easily accomodate most PCI memory space needs (Au1500 memory window, fast back-to-back window and general device memory space).*

6.4 Software Techniques

Most operating systems have a PCI configuration space support application programming interface, API, similar to the following: [4]

```
uint8  pciCfgRd8 (bus, device, func, reg);
uint16 pciCfgRd16 (bus, device, func, reg);
uint32 pciCfgRd32 (bus, device, func, reg);
void   pciCfgWr8 (bus, device, func, reg, data);
void   pciCfgWr16 (bus, device, func, reg, data);
void   pciCfgWr32 (bus, device, func, reg, data);
```

For Type 0 configuration cycles, the lower 32-bits of the 36-bit physical address are computed in this fashion:

```
addr = ((1 << device) << 11) | (func << 8)| reg;
```

For Type 1 configuration cycles, the lower 32-bits of the 36-bit physical address are computed in this fashion:

```
addr = (1 << 31) | (bus << 16) | (device << 11) | (func << 8) | reg;
```

With the lower 32-bits of the physical address computed, it is necessary to create a valid TLB entry in order to obtain a mapping onto the PCI configuration space.

There are three primary approaches for mapping PCI configuration space and performing PCI configuration cycles:

Technique #1) Allocate a fixed TLB entry, and dynamically map/unmap the TLB to the computed 36-bit PCI configuration cycle address, or

Technique #2) Allocate a fixed TLB entry which covers most of the configuration space, or

Technique #3) Dynamically map/unmap the 36-bit PCI configuration address from the current process address space.

6.4.1 Configuration Space Access Technique #1

Technique #1 is for operating systems (e.g. Linux, RTOSes, or other operating environments), in which device drivers do NOT have their own unique address space, and therefore can not dynamically map/unmap PCI configuration space on demand.

A TLB entry (e.g. TLB index 0) must be allocated for the specific purpose of PCI configuration cycles. The MIPS CP0 register Wired (i.e. CP0 register 6) must be adjusted accordingly to prevent random TLB updates from over-writing the entry allocated to the PCI configuration space. The PageMask should be set to 4KB.

Furthermore, a TLB-translated address range (KUSEG, KSEG2 or KSEG3) must be reserved for the purpose of providing a 4KB window into PCI configuration space. This 4KB window is sufficient to cover a PCI Configuration header and is the single window for **all** PCI configuration accesses.

Note: *NOTE: The least significant bit of the device number field is masked off during address translation with a 4KB PageMask (a 4KB PageMask utilizes bits [31:12] of the virtual address). As a result, least significant bit of the device number field (bit 11) must be included in the computation of the register offset from the virtual base of the PCI configuration space.*

The CCA encoding for the TLB entry must be the value 2 for non-cached accesses.

After a PCI configuration space access, examine the pci_config[ERD,ET,EF,EP] bits for access errors.

See the sample PCI bus scan code which demonstrates this technique.

6.4.2 Configuration Space Access Technique #2

Technique #2 is quite similar to technique #1, but it simplifies the management of the PCI configuration space. If the hardware design allows, more specifically all device IDSELs are connected to one of PCI_AD[24:11] and no Type 1 configuration cycles are needed, then a single fixed (i.e. wired) TLB entry can be established to cover the useful PCI configuration space. Thus the PCI configuration space is not continually mapped and unmapped during run-time.

A TLB entry (e.g. TLB index 0) must be allocated for the specific purpose of PCI configuration cycles. The MIPS CP0 register Wired (i.e. CP0 register 6) must be adjusted accordingly to prevent random TLB updates from over-writing the entry allocated to the PCI configuration space. The TLB PageMask is set to 16MB and utilizes both EntryLo0 and EntryLo1 to map a continguous 32MB of PCI configuration space.

A TLB-translated address range (KUSEG, KSEG2 or KSEG3) must be reserved for the purpose of providing a 32MB window into PCI configuration space. This is not a general purpose solution, but may suit many embedded hardware and software designs. The PCI configuration headers for devices with IDSEL connected to one of PCI_AD[24:11] are then directly visible in this KUSEG/KSEG2/ KSEG3 address range.

The CCA encoding for the TLB entry must be the value 2 for non-cached accesses.

After a PCI configuration space access, examine the pci_config[ERD,ET,EF,EP] bits for access errors.

See the sample PCI bus scan code which demonstrates this technique.

6.4.3 Configuration Space Access Technique #3

Technique #3, on the other hand, permits device drivers in operating systems that do support a separate virtual address space for device drivers (e.g. Windows CE) to dynamically map/unmap PCI configuration address space. The driver utilizes the established mapping/unmapping routines and the resulting virtual pointers to access PCI configuration space, and therefore the need to dedicate a fixed TLB entry and reserve a KUSEG address is un-necessary.

In all techniques, the CCA encoding for the TLB entry must be the value 2 for non-cached accesses.

7. PCI Memory Space

The Au1 core is able to generate non-cache-able accesses, cache-able accesses and fast back-to-back accesses to PCI memory. All PCI accesses first travel through the Au1500 TLB to yield a 36-bit physical address and a CCA encoding to determine cache-ability. The Au1500 PCI controller also features different types of windows into PCI space to improve performance.

7.1 Non-cache-able Accesses

Non-cache-able accesses are designated by the TLB producing the 36-bit physical address 0x4 xxxxxxxx with a CCA encoding of 2 or 7. On a read to non-cache-able PCI memory space, the Au1 core stalls waiting for data, and on a write, the data flows through the write-buffer, stalling only if the write-buffer is full.

A CCA encoding of 2 prevents gathering in the write buffer, which in turns causes single-beat accesses to PCI memory. CCA encoding 7 permits gathering in the write buffer, which in turns allows for burst transfers on the PCI bus. See Au1500 data book “2.3 Write Buffer” for more information.

In general, non-cache-able PCI memory space accesses occur when referencing PCI device registers and/or memory and tend to be the most frequent type of PCI memory space access.

7.2 Cache-able Accesses

Generally speaking, PCI memory space on the Au1500 is non-cache-able. However, the Au1500 PCI controller features the pci_cmem register which creates a cache-able window into PCI memory space. To utilize this feature, the pci_cmem must be enabled and the TLB must produce a physical address that hits in pci_cmem address range and a CCA encoding of 4.

The pci_cmem[CM_BASE] value is inherently prepended with 0x0 to compare against a 36-bit physical address (the Au1 core cache tags are for a 32-bit physical address). If the physical address hits in pci_cmem, then a cache-able PCI memory access takes place.

Note: *NOTE: It is NOT possible to use the TLB (with CCA 4) by itself to acheive cache-able PCI memory space accesses. The Au1 core cache tags are 32-bit physical address tags, and so a 36-bit physical address, like PCI memory space 0x4 xxxxxxxx, is inherently non-cache-able.*

The cache-able PCI memory window can be located anywhere, and in particular, within the first 512MB of memory; the MIPS KSEG0 region. However, to use the cache-able PCI memory window in KSEG0, the Config[K0] field, the CCA encoding for the KSEG0 region, must be set to 4. Do note, though, that changing KSEG0 to CCA 4 may not be suitable for all/most applications since it may introduce more processor stalls (since the critical word is no longer accessed first).

A very important consideration in determining to use the cache-able PCI memory window is that cache-able PCI memory window is **NOT** coherent with the PCI memory space. As such, the cache-able PCI memory window may not be suitable for all applications.

The address and mask values for pci_cmem are application-specific; there is no general-purpose setting for pci_cmem. As such, the need for this performance feature must be determined by the application designer.

A graphics video frame buffer would benefit from the cache-able PCI memory window. Note that that the PCI auto-configuration software must allocate the frame buffer MBAR in an address covered by pci_cmem. This may require more careful planning of the PCI memory space address map.

7.3 Fast Back-To-Back Accesses

The Au1500 PCI controller supports fast back-to-back accesses. If the 36-bit physical from the TLB hits in the Au1500 PCI controller fast back-to-back range (as determined by pci_b2bmask, pci_b2bbase0 and pci_b2bbase1), then the Au1500 controller will use fast back-to-back signalling.

7.4 Direct Memory Access, DMA

PCI devices can perform direct memory accesses, DMA, into Au1500 memory. For PCI device DMA into Au1500 memory, the Au1500 memory window must be enabled, see “Au1500 Memory Window” on page 8.

PCI devices that perform DMA are PCI bus masters. The Au1500 internal PCI bus arbiter can be used to grant different PCI bus masters (up to 4) the bus, or an external arbiter can be used. The decision is application specific, and the choice of arbiter must be reflected in the pci_config[AEN] bit.

The device drivers for PCI devices should utilize the bus address translation routines previously described in “Memory Window Considerations” on page 8. DMA engines need PCI space address pointers, and software needs virtual address pointers. The bus address translation routines are used to convert pointers from PCI address space to MIPS virtual address space.

7.5 Miscellaneous

All PCI memory accesses are first translated by the TLB into a PCI physical address. Since the TLB is of finite size (32 entries), it is possible that TLB faults will occur while accessing PCI memory space. Performance to PCI memory space improve by utilizing static, or wired, TLB entries. With static TLB entries, TLB faults are eliminated so as to remove the overhead involved in a TLB fault. Of course, the number of static TLB entries to use and which PCI spaces to cover are application specific.

8. PCI I/O Space

All PCI accesses first travel through the Au1500 TLB to yield a 36-bit physical address. PCI I/O space accesses are designated by the TLB producing the 36-bit physical address 0x5 xxxxxxxx. On a read to PCI I/O space, the Au1 core stalls waiting for data, and on a write, the data flows through the write-buffer, stalling only if the write-buffer is full.

PCI I/O space accesses may only be non-cache-able, and therefore must utilize CCA encoding 2.

A static, or wired, TLB entry may be desirable to reduce the possibility of TLB faults and the overhead associated with handling a TLB fault. The number of static TLB entries to use is application specific.

9. PCI Interrupts

TBD

10. Big-Endian Considerations

TBD

11. Satellite Mode

The Au1500 PCI controller is configured for satellite mode when the pin PCI_CFG is zero. The use of a satellite-mode Au1500 is very much application-specific; however, there are a few guidelines to follow.

- The PCI configuration registers at 0xB40051xx are **not** visible to the processor.

When in satellite mode, the registers at KSEG1 address 0xB40051XX are not visible to the Au1 core. As such, these registers must not be accessed during initialization of the PCI controller.

- The PCI configuration registers must be configured before clearing the pci_config[PD] bit.

The pci_config[PD] prevents the Au1500 PCI controller from responding to accesses. The Au1500 PCI configuration registers must be programmed with the appropriate values, then pci_config[PD] bit cleared. By default, this bit is set so that the controller does not respond to PCI accesses until the PCI controller is configured.

- The Au1500 responds to configuration cycles.

When in satellite mode, the Au1500 PCI controller does respond to configuration cycles. As such, it does appear during a normal PCI bus scan. (In host mode, the Au1500 PCI controller does not respond to configuration cycles.)

- The Au1500 memory window size (pci_mwmask) is reflected in the MBAR during a configuration cycle access.

During a PCI bus scan, the auto-configuration software manipulates the MBAR to determine the size of the memory window. The size programmed via pci_mwmask is reflected in the MBAR during the PCI bus scan so that the proper size is reported to the auto-configuration software.

12. References

[1] “The Alchemy™ Au1500™ Processor from AMD Data Book”, AMD, 2002.

[2] “PCI Local Bus Specification”, PCI Special Interest Group, 1998.

[3] “MIPS32™ Architecture for Programmers”, MIPS Technologies, Inc., 2001.

[4] “PCI BIOS Specification”, PCI Special Interest Group, 1994.