Introduction

This application note offer some suggestions for proper device interfacing to PCI "protocol chips", despite certain limitations in Marvell devices.

This application note applies to the following families of devices:

- GT6426x/6xA
- MV64360
- MV9823x

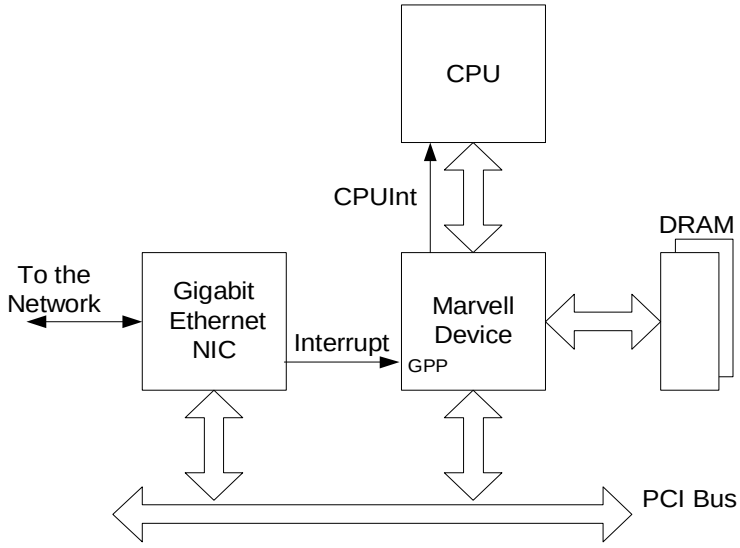
For the purposes of this document, these devices are collectively referred to as "Marvell devices" or simply "the device(s)".

Background

In many applications for these devices, the local PowerPC processor interfaces with a "protocol chip" over the PCI bus. This "protocol chip" PCI device can be a user specific FPGA, connected to user proprietary fabric, or an off the shelf PCI Application Specific Integrated Circuit (ASIC), such as an Ethernet NIC or Fiber Channel (FC) adapter.

Figure 1 shows a typical "protocol chip" interface.

Figure 1: Typical "Protocol Chip" Interface



Typically, the handshake between the local CPU and the PCI device is based on some producer-consumer model, in which buffers and descriptors are placed in the local DRAM and change ownership.

An example of such handshake is:

- The CPU prepares a chain of Rx descriptors in DRAM, to be used by the NIC, and sets a pointer to the first descriptor in the NIC.
- When the NIC receives a new Rx packet, it fetches a descriptor from DRAM. Then, it writes the received packet to the DRAM location pointed by the descriptor.
- After storing the packet in DRAM, the NIC writes back the descriptor to DRAM (with some status indication). Then it interrupts the CPU (via one of the device's GPP inputs). This interrupt signals the CPU that there is a pending Rx packet in DRAM.
- The CPU interrupt handler, after reading the system controller Interrupt Cause register and identifying the source of the interrupt, reads the NIC status register. Then, the CPU reads from the DRAM the descriptor and the received packet.
- After processing the packet, the CPU changes the descriptor status, so it can be re-used by the NIC.

Note

In many cases, both descriptors and buffers are cacheable. For the producer-consumer model to work properly, cache coherency must be maintained, whether by software means (for example, the CPU performs a cache flush after preparing descriptors to be used by the NIC) or by hardware means.

This CPU-to-PCI device handshake is based on the assumption that when the interrupt handler reads the descriptors/buffers, they are already stored in the DRAM. For this assumption to be correct, the following conditions must apply:

- The NIC only asserts an interrupt after the descriptors/packet data write over the PCI is completed. All standard PCI devices meet this requirement.
- After the interrupt handler reads NIC status, it is guaranteed that the data is stored in DRAM. In other words, the CPU read over the PCI (read of the NIC status register) must initiate the flushing of all posted write data in the system controller buffers towards the DRAM.

The Marvell® device implementation does not meet the second condition. This application note offers some suggestions for proper device interfacing to PCI "protocol chips", despite this limitation.

Note

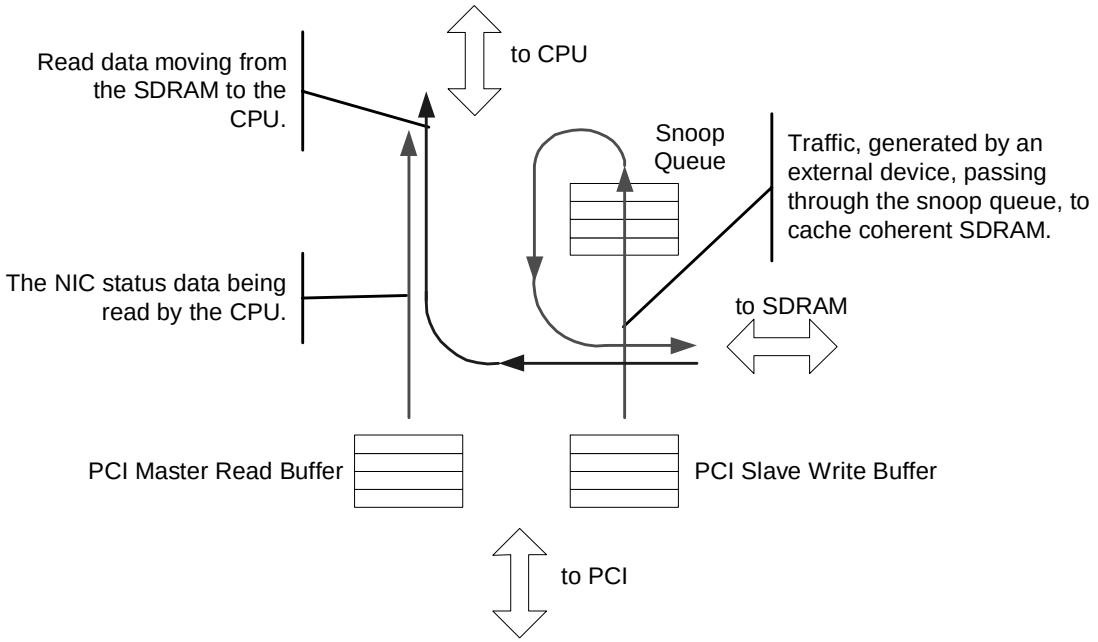
For GT6426x/6xA users, a further description of this limitation is available in the GT-6426x FEr-MISC-5 erratum.

This app note is relevant to GT6426x rev0 and revA devices. The issue described above will be fixed in GT6426x revB.

Device Buffering Implementation

Figure 2 shows the internal buffering scheme for the Marvell devices.

Figure 2: GT-6426x Internal Buffers



Note

Figure 2 only details the buffering implementation that is relevant to this application note. The figure does not detail the whole buffering and queuing implementation.

When the NIC writes data (descriptors or packet) to DRAM, the data is first posted to the device's PCI slave write buffer. If the device's cache coherency is used, the transaction goes through the snoop queue and, only after snoop is resolved, goes to DRAM (red).

When the CPU reads the NIC status over the PCI (green), the read data is stored in the device's PCI master read buffer and then goes to the CPU.

With this implementation, the CPU might receive the read data while the write data from the NIC is still "stuck" in the PCI slave write buffer or in the snoop queue. And once the CPU reads from DRAM (blue), the relevant data might still not be there.

Note

The "race" between the read data and the write data might never happen in real life applications. Typically, it takes many cycles from the interrupt assertion until the interrupt handler code starts executing. By the time the interrupt handler reads the NIC status register, all posted write data is flushed to DRAM.

Sync Barrier

The Marvell devices support a sync barrier mechanism. This feature can be used by software, to flush PCI slave posted write buffers, before going to read the data from DRAM.

Note

Sync barrier is not supported when using the Marvell device's cache coherency (see GT-6426x Res #CPU-2). When using the cache coherency feature, use one of the methods described in the following sections.

If using the Marvell devices without the cache coherency capabilities, before reading from DRAM, the interrupt handler must activate the sync barrier mechanism by reading CPU Sync Barrier Virtual register. When the read is completed, it is guaranteed that the data stored in the PCI slave write buffer is flushed to DRAM and that the CPU can safely read from DRAM.

Interfacing User Specific FPGAs

When the user has control on the PCI device behavior (whether it is a user specific FPGA/ASIC or an off the shelf device), it is easy to flush the Marvell device buffers.

To flush the buffers, the PCI device:

1. Generates a dummy read transaction to a cache coherent DRAM region.
2. Receives the read response data.
3. Asserts the interrupt after receiving the read response data from the dummy read transaction.

Note

Since the Marvell device's PCI slave queue and snoop queue work in order, it is guaranteed that this dummy read will flush all write data that was posted.

Interfacing Off the Shelf ASICs

This solution depends on the exact CPU-to-PCI device handshake model being used. For example, the descriptor pointers are maintained in DRAM with each descriptor containing an ownership bit. The ownership bit indicates whether the descriptor is owned by the CPU (a new Rx packet that has not yet been processed) or by the NIC.

When the NIC receives a new packet, it writes the packet to DRAM. Then, it write backs the descriptor with the ownership bit set to the CPU. Only then, the interrupt is asserted. If, by the time the interrupt handler reads the descriptor, the descriptor is still "stuck" in the device's write buffer, the CPU reads an old descriptor from DRAM (or from its internal cache) that is marked as owned by the NIC. In that case, the CPU realizes that this is a "false" interrupt and returns to the main program.

Eventually, the new descriptor arrives to DRAM but the CPU is not aware of it. If packets are constantly received, and if the system is not sensitive to packets processing latency, this is not a problem. The packet is simply processed with the next interrupt assertion – the next packet to come.

In case there is some latency limit, it is possible to activate one of the device timers. This will generate a periodic interrupt. With each timer interrupt assertion, the CPU will access DRAM to determine if there is no pending descriptor waiting for processing.

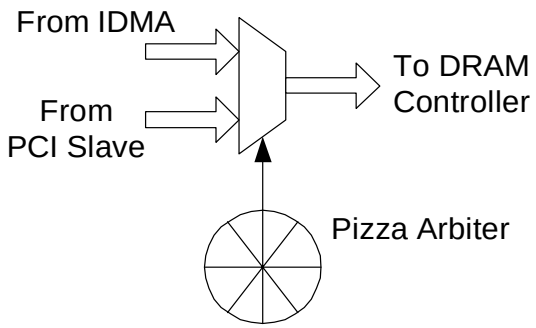
An alternative solution is to use a mechanism that guarantees that posted write data is flushed to DRAM, before the CPU read from DRAM.

The basic problem with this solution is that the interrupt handler cannot determine when the posted write data is flushed to DRAM. It is possible to use a counter that, upon expiration, guarantees the posted write data is flushed to DRAM. This solution's problem is that it's very hard to calculate a precise counter preset value – a value guaranteeing that all posted write data is flushed.

So, instead of a counter, a better solution is to use one of the Marvell device's IDMAs to compete with the PCI slave on access to the DRAM controller. Since the DRAM controller arbitration logic guarantees fair arbitration, it is assured that once the IDMA transfer is completed all PCI posted write data is flushed to DRAM.

Figure 3 shows the DRAM controller arbitration.

Figure 3: DRAM Controller Arbitration



In the worst case, there are four write transactions "stuck" in the PCI slave write buffer (the transaction size limit is 32 bytes when using the device's cache coherency). If giving the PCI slave four more time slots than the IDMA (one IDMA access to the DRAM controller for every four PCI accesses), it is guaranteed that after one IDMA transaction the whole PCI slave write buffer is flushed.

However, PCI transactions might still be stuck in the snoop queue. So, the IDMA transfer must also target a cache coherent region. This targeting guarantees that it also goes through the snoop queue. When the IDMA transfer completes, it is guaranteed that both the PCI slave write buffer and snoop queue are flushed and it is safe for CPU to access the DRAM.

The following is a detailed algorithm to activate the IDMA to assure that the PCI slave write buffer is completely flushed. It includes an initialization step (performed once) and a synchronization routine (performed by the interrupt handler, whenever synchronization is required).

Note

For the GT6426x Rev0 device, see the note following this initialization and synchronization procedure.

Initialization:

1. Allocate aligned 64 bytes in cacheable DRAM space (32 bytes IDMA source region, 32 bytes IDMA destination region).
2. Set the Snoop Control registers to enable cache coherency during IDMA access to DRAM.
3. Modify the DRAM controller pizza arbiter to enable one IDMA slice for every four PCI slices.

Synchronization:

1. Set the IDMA source to point to the source region.
2. Set the IDMA destination to point to the destination region.
3. Set the IDMA byte count to 16 bytes.
4. Activate the IDMA channel with a BurstLimit of 8 bytes.
5. Wait until the DMA transfer ends (read polling on IDMA channel active bit).

If using the Marvell device's PCI_0 and PCI_1 interfaces, set the DRAM controller pizza arbiter to enable one IDMA slice for every four PCI_0 slices + four PCI_1 slices. Also, PCI_0 and PCI_1 slices must be interleaved (meaning, one slice per PCI0, one per PCI1 and so on).

Note

In the GT6426x Rev0 device, the pizza arbiter cannot guarantee four PCI transactions for every single IDMA transaction (see GT6426x rev0 errata FEr #50). For this device, the algorithm should be slightly modified:

Initialization:

1. Allocate aligned 256 bytes in cacheable DRAM space (128 bytes IDMA source region, 128 bytes IDMA destination region).
2. Set the Snoop Control registers to enable cache coherency during IDMA access to DRAM.
3. Modify the DRAM controller pizza arbiter to enable one IDMA slice per every four PCI slices.

Synchronization:

1. Set the IDMA source to point to the source region.
2. Set the IDMA destination to point to the destination region.
3. Set the IDMA byte count to 128 bytes.
4. Activate the IDMA channel with BurstLimit of 32 bytes.
5. Wait until the DMA transfer ends (read polling on IDMA channel active bit).

Summary

Where the local PowerPC processor interfaces with a "protocol chip" over the PCI, it is important to be aware of the potential ordering issues described in this application note.

In most cases, the race between the read data and the write data might never occur.

Yet, in applications where it is possible, it is advisable to use one of the methods described in this application note to avoid this limitation.

Marvell Semiconductor, Inc.
2350 Zanker Road, San Jose, CA 95131
Phone: (408) 367-1400, Fax: (408) 367-1401