PCI Performance on the RC32334/RC32332

Application Note AN-367

Notes

By Rakesh Bhatia and Pallathu Sadik

Revision History

April 19, 2002: Initial publication.

September 4, 2002: Updated for revision YC silicon.

Background

The RC32334/RC32332 devices are integrated processors that combine a 32-bit MIPS instruction set architecture (ISA) CPU core with a number of on-chip peripherals to enable direct connection to boot memory, main memory, IO, and PCI. The RC32334/RC32332 components also include system logic for DMA, reset, interrupts, timers, and UARTs. The RC32334/RC32332 components integrate many of the peripherals commonly associated with an embedded system to reduce board real estate, design time, and system cost.

Optimal vs. Actual PCI System Throughput

The RC32334/RC32332 devices are based around a standard bus architecture called the IDT internal peripheral bus architecture-generally referred to as the IP Bus. The SDRAM subsystem resides on this bus. Hence, any transaction that is targeted towards or from the external SDRAM memory must be transacted via this bus. Therefore, if both the PCI and the CPU core wish to access the SDRAM, they cannot simultaneously do so. These modules must take turns accessing the SDRAM. As a result, as the amount of CPU-initiated activity to external memory increases (for example, number of misses to primary cache and/or the need to manipulate data in memory), the PCI module is increasingly blocked from acquiring the IPBus and the PCI throughput begins to drop.

Therefore, the real-world PCI throughput on a RC32334-based system is highly operating system dependent. Resource intensive operating systems will significantly impact the PCI performance. Ideal test cases reveal that the current RC32334/RC32332 YC revision device offers a much higher bandwidth compared to the previous RC32334/RC32332 Z revision device. The nature of these tests is discussed more thoroughly in the relevant sections below.

Satellite vs. Host Mode

The PCI controller in the RC32334/RC32332 can be operated in either host or satellite mode. The terms satellite and host only refer to which device in the system is responsible for initially configuring the PCI subsystem. It has no bearing on subsequent transactions. All devices are treated equally once configuration is completed. Therefore, it makes no difference whether the RC32334/RC32332 are configured as host or satellite, the performance will be the same.

Supported PCI Transaction Types

The RC32334/RC32332 PCI controller supports several PCI transactions which can be broadly classified as Master and Target PCI transactions. When the CPU core or its internal DMA engine initiates a PCI read/write transaction to a PCI device on the bus, the transaction is called a “Master PCI transaction.” When a PCI device on the bus initiates a read/write transaction to access the RC32334/RC32332 internal registers or local memory, the transaction is called a “Target PCI transaction.” The integrated PCI controller on the RC32334/RC32332 is aided by a user transparent DMA engine to efficiently perform the Target PCI transactions. This DMA engine can’t be configured by the user. However, a set of configuration registers residing within the PCI controller can be programmed to select burst size and prefetching behavior during

Notes

Target PCI transactions. Configuration of these PCI controller registers is discussed in detail in IDT Application Note AN-366: Optimizing the PCI Interface on the RC32334/RC32332, which can be found on IDT's web site at (www.idt.com).

Master Read

In most PCI devices, the PCI master read transaction is the most inefficient. Because of this, nearly all real-world systems are developed in such a way as to make very sparing use of this mode. As a result, there is little incentive to highly optimize this interface. The RC32334/RC32332 is no exception. Once a read is initiated, the DMA or CPU waits on the local bus until the read is completed and the data becomes available. In the case of slow PCI devices, this can result in very long delays. Because the CPU core and the PCI module share the IP Bus, this effectively locks the CPU core off the IP Bus, and it can drastically impact overall system performance.

The master read interface will not queue transactions. Each transaction must complete prior to a new one being issued. Since the master read holds the IPBus during the entire transaction, there is no possibility of initiating another transaction before the first one completes. The master read logic is serviced by an eight-deep, 32-bit wide FIFO. However, since master reads are limited to a maximum of four words, the entire FIFO is never utilized.

Master Write

The master write transaction is limited to a maximum of four words. A four-word burst is accomplished by configuring the DMA engine to use quad-word burst mode, and initiating a DMA from local memory to an address which resides in PCI space. The RC32334/RC32332 PCI controller will sample the state of the PCI output FIFO to ensure there is adequate space to absorb the next quad-word burst prior to granting the bus to the DMA. This prevents the RC32334/RC32332 from waiting on the bus when DMA-ing data to slow PCI devices. Master writes can be queued in the output FIFO. The output FIFO is eight words deep. Therefore, at most, two quad-word bursts can be queued.

Target Read

The PCI target read logic has the most user-configurable settings. These must be configured properly for optimum performance. The RC32334/RC32332 contains several registers to assist users in optimizing target read transfers. The target read FIFO is 16 words in size. The maximum target read burst size is eight words. Only one read request can be queued at a time. However, if eager prefetching is enabled, the next eight words of data beyond the current eight words being fetched may be queued as well. Therefore, all 16 words of the FIFO are usable.

Target Write

The target write logic is very straightforward. There is only one configurable parameter which controls whether the target write DMA moves data in a four word burst or in an eight word burst. The target write FIFO is 16 words deep. Multiple transactions can be queued until the FIFO is full. The target address of the transaction is stored in the FIFO along with the data. So, a four-word burst would occupy five FIFO locations: four for the data and one for the address.

Testing Conditions

Maximum Performance Testing

These tests are intended to characterize the absolute maximum throughput of the PCI hardware under ideal conditions. Real-world performance will always be somewhat less.

The maximum PCI throughput measurement tests used blocks of contiguous data. The RC32334/RC32332 CPU core was made to execute an indefinite small loop program completely residing within the CPU cache. This ensured 100% availability of the internal IP Bus for performing the Target PCI transactions. The CPU core was notified of completion of the PCI transfers via an interrupt. During these tests, data was sent in one direction using only one type of PCI transaction throughout the specific test.

Notes

Test Set-up

Figure 1 below shows the lab test set-up. The PCI analyzer card HPE2928 was used along with its GUI to control the types of transactions between the RC32334/RC32332 device and the HPE2928. IDT's evaluation board 79S334 contained the BGA packaged RC32334. Logic analyzer 16702B was used throughout the test to measure timings of the transactions. IDT SIM software version 9.2 was used to boot the evaluation board. The byte ordering configuration was Big-endian. The system clocks used in the tests were 50MHz, 66.5MHz, and 75MHz. Tests were done with both 33MHz and 66MHz PCI clocks for these system clock frequencies.

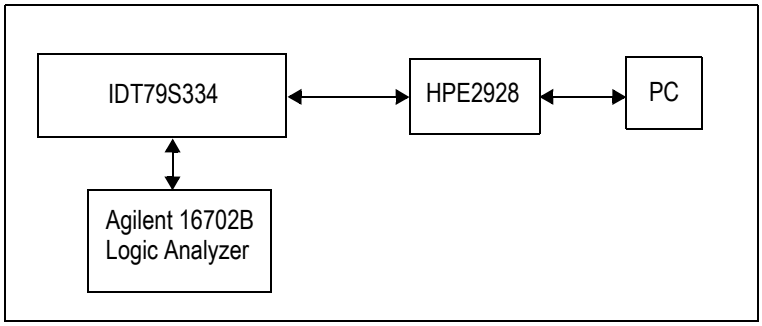

Figure 1 Test Setup for PCI Performance Tests

Test Procedure

Using the HPE2928's GUI, target transactions can be ordered between the analyzer card and the 79S334 board. Since the HPE2928 card retains the settings, after each test / configuration, the system need to be reset. Also, once the CPU is locked for the RC32334/RC32332 through the IDT SIM software, no further commands can be given unless the 79S334 board is reset.

For master transactions, the IDT SIM software was used. The boot-up EPROM on the 79S334 provides for this facility.

Test Results

Maximum Performance Master Transactions

These tests were conducted using the DMA to move data from the SDRAM to the PCI and from the PCI to the SDRAM. Traffic was not mixed, meaning that each test consisted of moving data in only one direction using only one type of master PCI transaction. The write test consisted of moving different size blocks of data from the SDRAM of the RC32334/RC32332 to the memory of an Agilent HPE2928 PCI analyzer card. The read test consisted of moving the same blocks of data from the Agilent PCI analyzer back to the memory of the RC32334/RC32332.

Signal[1]	50MHz Sys. 100MHz CPU 33MHz PCI	50MHz Sys. 100MHz CPU 66MHz PCI	66MHz Sys. 133MHz CPU 33MHz PCI	66MHz Sys. 133MHz CPU 66MHz PCI	75MHz Sys. 150MHz CPU 33MHz PCI	75MHz Sys. 150MHz CPU 66MHz PCI
SDRAM to PCI						
Mem_wr (1)	8.3 MB/sec	8.31 MB/sec	11.05 MB/sec	11.04 MB/sec	12.49 MB/sec	12.5 MB/sec
Wr_inval (4)	26.6 MB/sec	26.68 MB/sec	35.18 MB/sec	35.26 MB/sec	39.92 MB/sec	39.93 MB/sec
PCI to SDRAM						
Mem_rd (1)	5.66 MB/sec	7.08 MB/sec	6.55 MB/sec	8.58 MB/sec	7.19 MB/sec	9.3 MB/sec
Rd_multi (4)	18.21 MB/sec	22.5 MB/sec	20.96 MB/sec	27.6 MB/sec	22.56 MB/sec	30.37 MB/sec

Table 1 RC32334/RC32332 as Master

1. Mem_wr = memory write, Wr_inval = write invalidate, Mem_rd = memory read, Rd_multi = read multiple. Number in parentheses refers to number of words involved in the transfer.

Notes

Maximum Performance Target Transactions

These tests were conducted by doing target reads and writes to the RC32334/RC32332 using an Agilent HPE2928 PCI analyzer. For the target read case, various size blocks of contiguous data were read from the SDRAM of theRC32334/RC32332 into the memory of the Agilent PCI analyzer card. For the write case, the same blocks of data were written from the Agilent PCI analyzer card back to the SDRAM of the RC32334/RC32332. Test results are shown in Table 2.

Signal[1]	50MHz Sys. 100MHz CPU 33MHz PCI	50MHz Sys. 100MHz CPU 66MHz PCI	66MHz Sys. 133MHz CPU 33MHz PCI	66MHz Sys. 133MHz CPU 66MHz PCI	75MHz Sys. 150MHz CPU 33MHz PCI	75MHz Sys. 150MHz CPU 66MHz PCI
PCI to SDRAM						
Mem_wr (1)	15.24 MB/sec	15.4 MB/sec	20.44 MB/sec	20.38 MB/sec	21.91 MB/sec	22.96 MB/sec
Wr_inval (4)	49.77 MB/sec	50.53 MB/sec	57.83 MB/sec	65.86 MB/sec	58.16 MB/sec	74.42 MB/sec
Wr_inval (8)	48.98 MB/sec	70.47 MB/sec	79.19 MB/sec	72.68 MB/sec	79.42 MB/sec	72.18 MB/sec
SDRAM to PCI						
Mem_rd (1)	5.33 MB/sec	7.15 MB/sec	6.24 MB/sec	7.33 MB/sec	6.57 MB/sec	7.32 MB/sec
Rd_line (4)	19.41 MB/sec	26.0 MB/sec	21.89 MB/sec	26.4 MB/sec	22.89 MB/sec	26.52 MB/sec
Rd_multi (8)	34.92 MB/sec	46.98 MB/sec	37.53 MB/sec	47.84 MB/sec	38.68 MB/sec	48.7 MB/sec

Table 2 RC32334/RC32332 as Target

1. Mem_wr = memory write, Wr_inval = write invalidate, Mem_rd = memory read, Rd_line = read line, Rd_multi = read multiple. Number in parentheses refers to number of words involved in the transfer.

Optimizing the RC32334/RC32332 for Maximum PCI Bandwidth

RC32334/RC32332 as Master

In the master mode, the RC32334/RC32332 does not have too many parameters that can be modified to improve performance. However, performance was seen to improve significantly when using transfer sizes of 4 words (see Table 2 above). Transaction sizes of 1024 bytes and 4096 bytes were done between the RC32334/RC32332 and a PCI analyzer card (HPE2925). The bandwidth did not change between different transaction sizes.

RC32334/RC32332 as Target

When the RC32334/RC32332 devices are configured for target mode, various bits can be set to improve performance. Some of the bits have an effect with certain types of PCI commands only. For example, the Eager Prefetch bit in the Target control register would have no effect on a mem_read command. Therefore, this bit was not considered as a parameter when doing target read transactions. To increase the PCI throughput, the following bits were used most frequently in the tests referenced above:

Eager Prefetch bits [30:27]

Bits [30:27] allow for eager prefetching of data on certain types of commands, such as mem_read_line and mem_read_multiple.

Memory Write / Memory Write Invalidate bit [26]

The MW / MWI bit [26] of the Target Control Register can be used to burst up to 8 words on the local bus.

Threshold for Target Write FIFO bits [25:24]

The intention of Threshold bits [25:24] is to wait until at least a certain number of data words are free in the FIFO before accepting any new write command.

Notes

MRML bits [23:20]

The MRML bits [23:20] are used to prefetch 8 words for memory_read and memory_read_line target reads. These commands behave like the memory_read_multiple command.

Dtimer bits [15:8]

The Disconnect Timer bits [15:8] assist in delaying the issuance of a “disconnect” in a PCI transaction. This can be useful if slower PCI devices are being used.

Rtimer bits [7:0]

The Retry Timer default value is 16, per PCI Specification 2.2. However, many devices respond in 20 PCI clock cycles. Instead of issuing a retry after 16 clock cycles, the limit can be increased to 20 clock cycles which will reduce, in most cases, the number of times the retry command is issued.

A detailed explanation of each bit is provided in the RC32334/RC32332 User Reference Manual. Also refer to Application Note AN-366, cited earlier, which provides detailed guidelines for increasing PCI throughput on these devices. Both documents are located on IDT’s web site at www.IDT.com.

Table 3 below shows all the tests done for Target reads and Table 4 shows the test results for target writes.

> **Note:** These readings are ideal case readings in that the HPE2928 PCI card waits forever. In addition, the RC32334/RC32332 CPU core was locked, meaning that no other transactions were taking place. Real world test environments will vary from user-to-user, so the objective here is to provide a baseline and show the impact of various settings used in the PCITC register. The following section, RC32334/RC32332 Ethernet Bridge Throughput Analysis, describes a real-world scenario in some detail using the Linux operating system. Although it is not intended to imply any kind of recommendation, this section provides an explanation of the time used by various sub-routines in the kernel and offers some suggestions.

Notes

PCITC	Throughput (MB/s) with 50MHz system, 100MHz CPU, 33MHz PCI	Throughput (MB/s) with 50MHz system, 100MHz CPU, 66MHz PCI	Throughput (MB/s) with 66MHz system, 133MHz CPU, 33MHz PCI	Throughput (MB/s) with 66MHz system, 133MHz CPU, 66MHz PCI	Throughput (MB/s) with 75MHz system, 150MHz CPU, 33MHz PCI	Throughput (MB/s) with 75MHz system, 150MHz CPU, 66MHz PCI
Mem_Rd						
01080810	3.67	7.13	3.66	7.29	3.67	7.29
01080818	5.33	6.00	6.21	5.98	6.54	5.99
01180810	3.66	7.13	3.67	7.32	3.66	7.32
01180818	5.3	6.00	6.21	5.99	6.56	5.99
01081010	3.67	7.15	3.67	7.33	3.67	7.30
01081018	5.29	6.00	6.19	6.00	6.56	5.98
01181010	3.66	7.13	3.67	7.32	3.66	7.32
01181018	5.33	6.00	6.24	5.99	6.56	5.99
09080810	3.66	5.31	3.66	7.32	3.67	7.32
09081010	3.65	5.32	3.66	7.29	3.67	7.32
09180810	3.65	6.03	3.67	6.03	3.67	6.81
09181010	3.66	6.03	3.66	6.03	3.67	6.83
09080818	5.31	5.56	6.19	5.98	6.56	5.99
09081018	5.33	5.55	6.19	5.99	6.56	5.99
09180818	3.87	5.41	6.04	5.99	6.57	5.99
09181018	3.87	5.39	6.00	5.99	6.56	6.03
Rd_Line						
01080810	13.13	25.75	13.21	26.40	13.21	26.28
01080818	19.34	21.95	21.88	21.95	22.76	21.95
09080810	13.22	22.32	13.21	26.26	13.22	26.26
09080818	19.25	22.14	21.46	21.96	22.23	21.95
01180810	13.12	25.75	13.20	26.26	13.19	26.39
01180818	19.09	21.95	21.89	22.14	22.89	21.95
09180818	18.22	18.22	21.80	21.96	22.89	22.14
01081010	13.18	25.75	13.20	26.40	13.19	26.39
01081018	19.39	21.95	21.89	22.14	22.89	21.95
09081010	13.12	22.51	13.20	26.26	13.21	26.26
09081018	19.41	21.95	21.54	21.96	22.24	21.95
01181010	13.22	26.00	13.20	26.26	13.19	26.52
01181018	19.38	21.95	21.82	21.96	22.89	21.95
09181018	18.22	18.34	21.82	21.93	22.89	22.00
09180810	13.26	18.22	13.20	24.13	13.19	26.26
09181010	13.26	18.34	13.18	24.13	13.16	26.26
RD_multiple						
01080810	23.91	45.40	24.05	47.66	24.02	48.21
01080818	33.91	41.20	37.53	40.70	38.62	40.54
09080810	23.98	36.69	23.83	47.83	24.02	47.81
09080818	33.83	36.69	35.54	41.20	36.57	40.58
01180810	23.87	46.18	23.94	47.83	23.91	47.81
01180818	34.26	41.20	37.27	41.20	38.60	41.20
09180818	33.48	37.19	36.50	41.20	36.44	39.98
01081010	23.98	46.98	24.05	47.81	24.02	47.39
01081018	33.83	40.58	37.53	40.58	38.54	40.58
09081010	23.98	36.69	24.05	47.83	24.02	47.81
09081018	34.92	36.69	36.50	40.58	36.44	40.58
01181010	23.98	46.18	24.05	47.83	24.02	48.70
01181018	33.83	40.58	37.53	41.20	38.68	41.20
09181018	33.91	36.69	35.78	40.51	36.48	40.74
09180810	23.91	36.20	23.95	47.84	24.02	47.81
09181010	23.91	36.20	23.94	47.84	24.02	47.81

Table 3 PCI Target Read Transactions

Notes

PCITC	Throughput (MB/s) with 50MHz system, 100MHz CPU, 33MHz PCI	Throughput (MB/s) with 50MHz system, 100MHz CPU, 66MHz PCI	Throughput (MB/s) with 66MHz system, 133MHz CPU, 33MHz PCI	Throughput (MB/s) with 66MHz system, 133MHz CPU, 66MHz PCI	Throughput (MB/s) with 75MHz system, 150MHz CPU, 33MHz PCI	Throughput (MB/s) with 75MHz system, 150MHz CPU, 66MHz PCI
Mem_wr						
01080810	15.22	15.40	20.44	20.37	21.91	22.86
01080818	15.22	15.31	20.43	20.28	21.91	22.86
01081010	15.22	15.31	20.44	20.36	21.90	22.86
01081018	15.24	15.40	20.43	20.10	21.90	22.96
02080810	15.24	15.21	20.20	20.34	21.91	22.96
02080818	15.24	15.21	20.20	20.30	21.85	22.96
02081010	15.24	15.21	20.20	20.37	21.90	22.96
02081018	15.20	15.40	20.20	20.30	21.87	22.94
05080810	15.20	15.21	20.44	20.37	21.87	22.94
05080818	15.24	15.31	20.41	20.38	21.89	22.94
05081010	15.24	15.31	20.38	20.37	21.89	22.94
05081018	15.24	15.21	20.36	20.35	21.89	23.05
06080810	15.24	15.31	20.37	20.36	21.89	22.94
06080818	15.24	15.21	20.35	20.36	21.87	22.84
06081010	15.24	15.21	20.37	20.38	21.89	22.84
06081018	15.20	15.21	20.37	20.36	21.89	22.94
Wr_inv (4)						
01080810	48.43	49.60	57.79	65.73	57.73	73.41
01080818	48.95	49.60	57.79	65.71	57.91	74.42
01081010	48.87	49.60	57.79	65.71	57.91	73.41
01081018	48.98	50.53	57.79	66.11	57.91	74.42
02080810	42.39	49.60	57.79	64.69	58.04	74.42
02080818	42.56	49.60	57.79	65.56	57.81	72.79
02081010	42.39	49.60	57.79	64.96	57.88	72.61
02081018	42.39	49.60	57.79	64.28	57.77	72.79
05080810	49.77	49.60	57.83	65.45	57.85	73.16
05080818	49.77	50.53	57.83	65.86	57.85	73.16
05081010	48.98	49.60	57.79	65.70	58.16	73.16
05081018	48.54	50.53	57.79	65.31	57.89	73.16
06080810	42.73	50.53	57.79	65.17	57.71	72.18
06080818	42.39	48.70	57.75	64.47	57.90	71.23
06081010	42.06	49.60	57.83	65.56	57.72	72.18
06081018	42.39	49.60	57.75	64.81	57.85	72.18
Wr_inv (8)						
01080810	43.33	31.88	61.45	42.77	71.11	52.02
01080818	47.26	31.88	61.45	42.61	71.11	51.46
01081010	47.32	32.27	61.41	42.61	69.84	53.04
01081018	47.26	33.48	61.45	43.98	69.84	51.52
02080810	36.35	49.60	50.00	66.30	58.05	72.00
02080818	36.35	49.60	50.00	66.30	56.46	72.00
02081010	36.35	49.60	50.00	65.63	58.12	72.00
02081018	36.35	49.60	50.00	65.63	58.12	72.00
05080810	48.90	33.07	79.12	52.09	79.24	59.38
05080818	48.98	33.07	79.19	53.65	79.24	65.16
05081010	41.74	33.91	79.19	53.55	79.42	59.38
05081018	43.07	33.07	79.12	53.29	79.42	60.04
06080810	36.59	70.47	36.72	72.42	36.86	72.18
06080818	36.59	70.47	36.72	72.42	36.80	72.18
06081010	36.59	70.47	36.72	72.68	36.77	72.18
06081018	36.59	70.47	36.73	72.68	36.79	72.18

Table 4 PCI Target Write Transactions

RC32334/RC32332 Ethernet Bridge Throughput Analysis

The 79S334 evaluation board hosts the 79RC32334 integrated communications processor (ICP), which incorporates a PCI module. An Ethernet bridge can be implemented on the evaluation board using two widely available PCI-based Ethernet network interface cards (NIC) and bridging software. A test was conducted to gain insight into the execution of this bridging software to better utilize the CPU core's resources.

Test set-up

The following hardware and test set-up was used to perform these tests.

Target Board IDT79S334

Processor: RC32334 YC @150MHz, 75 MHz memory bus, 33 MHz PCI clock.

Notes

Memory: 32 MB

NIC: Intel 82559 PCI cards inserted in the PCI slots of S334 board

The Target board was modified to bring out four GPIO pins and the INTA line on the PCI bus to the Logic Analyzer.

Logic Analyzer

Model: Agilent 16702B

Linux Host

This machine was used for compiling the Linux 2.4.18 kernel. It also provided the console for the target.

Abatron Debugger

Ethernet based EJTAG probe from Abatron was used for downloading the kernel from the Linux Host to the IDT79S334 target board. It could be used to debug the kernel if needed.

SmartBits Traffic Generator/analyzer

Model: SMB-2000

This unit had two Ethernet interface modules (7710) installed at Port 17 and Port 18.

A Windows2000 computer system was used to control the SmartBits traffic generator/analyzer. There were two applications, SmartWindows and SmartApplications, installed on this machine. SmartWindows provides an interface where one or more packets can be transmitted from one of the ethernet ports in SmartBits and the arrival of these packets can be monitored at the other ethernet port in SmartBits. The timing measurements explained in this application note were performed using SmartWindows. SmartApplication provided an interface that is similar to a real world scenario. This application can transmit a stream of packets for a fixed duration of time and measure the throughput.

The test setup used for the exercise is shown in Figure 2 below.

Notes

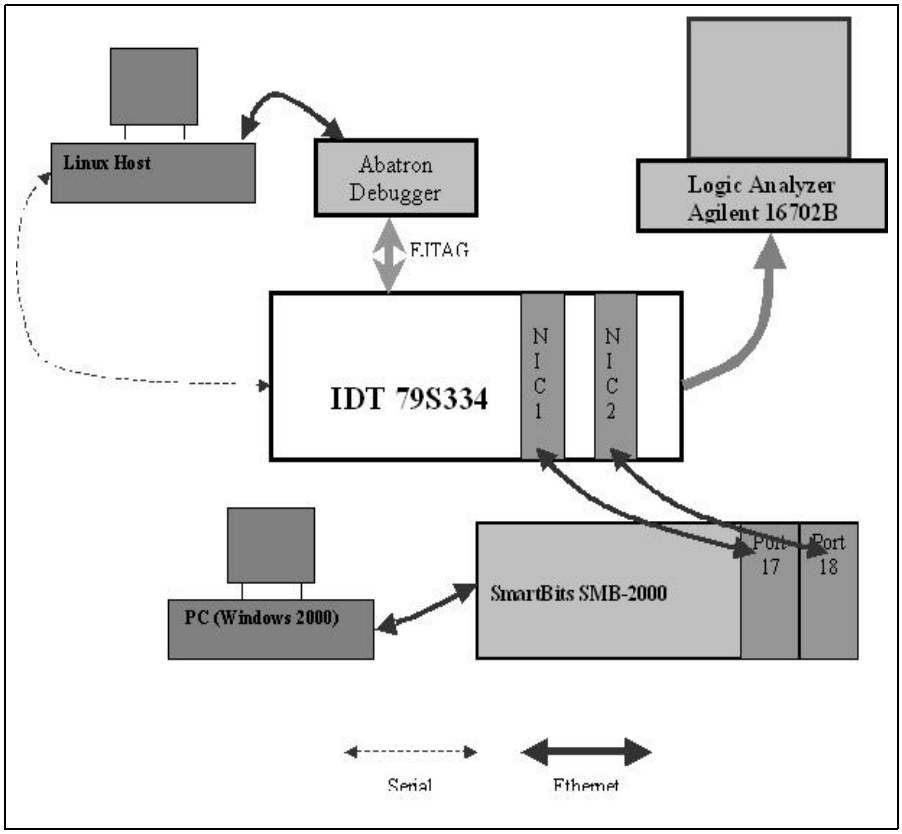

Figure 2 Performance Analysis Measurement Setup

Software Set-up

Operating System built to run on the target board: Linux 2.4.18.

Compiler tool chain used to build the above operating system: gcc-2.96 and binutils-2.12.

Kernel Configuration

The Linux kernel was configured to support Ethernet bridging. The utilities to setup the bridge were compiled and included as part of the ramdisk image. The bridge functioned as follows: When a packet arrived at one of the NICs, the bridge checked for the destination IP then forwarded the packet to the other NIC, provided the destination IP was not that of the bridge itself.

Bridge Setup

After Linux booted on the target board, the Ethernet bridge was set up by issuing the following commands at the target console:

brctl addbr br0 # Creates a logical bridge instance with name 'br0'

brctl addif br0 eth0 # Adds Ethernet interface 0 to the bridge

brctl addif br0 eth1 # Adds Ethernet interface 1 to the bridge

ifconfig eth0 0.0.0.0 promisc # Put Ethernet interface 0 to promiscuous mode

ifconfig eth1 0.0.0.0 promisc # Put Ethernet interface 1 to promiscuous mode

ifconfig br0 192.168.1.3 promisc up # Assign IP address 192.168.1.3 to bridge

#and put the bridge in promiscous mode.

For more information on Ethernet Bridging, refer to Linux BRIDGE-STP-HOWTO at http://www.tldp.org.

Notes

Test Procedure

Using the SmartWindow application, the SmartBits Ethernet port at Slot 17 was configured to generate an IP packet with source address 192.168.1.1 and destination address 192.168.1.2. The packet would arrive at NIC-1 of the target and would be forwarded to NIC-2 by the linux-based bridge running on the target board. Using a Logic Analyzer, the time taken for the packet to travel from NIC-1 to NIC-2 at various levels of the protocol was measured. The measured time intervals are presented in Table 5. A brief description explaining the journey of a packet through the kernel is provided below to give a better understanding of the timings.

Journey of a Packet

NIC to Memory

When the NIC card receives a packet, it requests for a PCI transaction from the RC32334/RC32332 which then grants the PCI bus to the NIC. Following the grant, the PCI Frame signal is pulled low and the NIC, using DMA, transfers the packet from its FIFO to the SDRAM memory on the target board. At the end of the DMA transfer, the NIC is programmed to generate a PCI Interrupt (INTA).

Interrupt handler

The kernel executing on the target board, upon receiving the interrupt, calls the Interrupt Service Subroutine located in:

~linux/arch/mips/rc32300/common/int-handler.S.

Linux groups interrupts into two categories, timer interrupts and “other” interrupts. In the assembly code mentioned above, the kernel checks the source of the interrupt. Once it determines that the interrupt is not from the timer, it calls the interrupt handler that handles "other" interrupts:

(function: rc32300_irqdispatch; file linux/arch/mips/rc32300/79S334/irq.c).

In the RC32334/RC32332 under Linux, the major sources of interrupts are PCI and onboard devices, such as the serial I/O port. In function *rc32300_irqdispatch*, the interrupt source information is further decoded to distinguish between the above two. Once the kernel determines that the interrupt is not from an onboard device, it calls *‘do_IRQ’ (~linux/arch/mips/kernel/irq.c)*.

During device initialization, each device registers its own interrupt handler with the kernel and they form a linked list of interrupt handlers. The *‘do_IRQ’* function calls *handle_IRQ_event (~linux/arch/mips/kernel/irq.c)*. This function calls each of the interrupt handlers from the linked list. In the case of Intel NIC, the interrupt handler function is ‘*speedo_interrupt’(~linux/drivers/net/eepro100.c)*.

Memory to Memory

Once the interrupt handler for NIC determines that the interrupt was generated due to the arrival of a packet, it calls the *"receive handler" (speedo_rx)*. The *‘speedo_rx’* function creates *'skbuff'* (a data structure to hold the packet and header) and copies the packet to it.

Driver to Protocol Independent Device Support Routine

The *"receive handler"*, after copying the packet to an skbuff, calls a function *‘netif_rx' (~linux/net/core/dev.c)* which queues the packet for further processing. This function first checks the queue. If it is full, the packet is dropped. Otherwise, the packet is placed in the queue, raises a 'soft interrupt', and returns. The *‘speedo_rx’* then continues to perform some housekeeping tasks. The interrupts that were disabled up to this point get enabled again, and the interrupt handler returns.

Soft Interrupts (softirq)

Soft interrupts (softirq) are similar to hardware interrupts, except that they can run with the interrupts enabled. Like hardware interrupts, software interrupts also need to be registered with the kernel. The function *net_dev_init* (~linux/net/core/dev.c) registers two network soft interrupt handlers, net_rx_action and net_tx_action, during kernel boot.

Notes

The soft interrupts are called one by one by the *do_softirq (~linux/kernel/softirq.c)* function. The function *do_softirq* is called from various places in the kernel. One of them is *do_IRQ (~linux/arch/mips/kernel/irq.c)*, described above.

net_rx_action

The softirq *'net_rx_action' (~linux/net/core/dev.c)* is responsible for handling the received packets. Here, the packet is removed from the queue and appropriate packet handler is called. In the case of the current exercise, the packet handler is '*handle_bridge' (~linux/net/core/dev.c)* which in turn calls '*br_handle_frame_hook' (~linux/net/core/dev.c)* for further packet processing.

br_handle_frame

This function is implemented in *br_forward.c (~linux/net/bridge)*. Here, the *'skbuff'* is attached to the bridge with its destination and source addresses properly set. It then calls *'br_handle_frame_finish'* where the *'skbuff'* is retrieved from the bridge and *'br_flood'* gets called. *'br_flood'* calls *'__br_forward'*, which in turn calls *__br_forward_finish*. This function calls *__dev_queue_push_xmit* that calls *dev_queue_xmit (~net/core/dev.c)*.

dev_queue_xmit

Here, the *'skbuff'* gets queued into the transmit queue associated with a network device. Then the routine *'qdisk_run'* is called. This routine picks up the packet from the queue and calls *'hard_start_xmit'* associated with the device driver. In the case of Intel NIC, the function *speedo_start_xmit* (mapped to *'hard_start_xmit')* gets called *(linux/drivers/net/eepro100.c)*.

speedo_start_xmit

This function updates certain data structures and initiates a data transfer from memory to the NIC and data is transferred to the output FIFO of the NIC through DMA. The SmartBits Ethernet module on Port 18 reports arrival of packet.

With this, the journey of the packet ends.

Timing Analysis

The time taken for the packet to arrive at a particular function is shown in the table below. The time measurements are relative to the moment (time=0) when the interrupt is asserted by the NIC that receives the Ethernet packet from the SmartBits traffic generator. This assertion indicates that the NIC finished moving the packet across to the SDRAM on the 79S334 board. In the absence of a profiling tool for Linux kernel, the measurement had to be done manually, using Logic Analyzer.

The measurement was performed as follows:

Four GPIO pins available on 79S334 were programmed to be output pins. At the beginning and end of each function explained above, the state of the GPIO pin changed (high, low or toggle). Using the logic analyzer various elapsed time measurements were made from the traces indicating changes of states of the GPIO pins. Since only four GPIO pins were available, in most cases only one function could be analyzed at a time. The kernel was re-compiled each time for analyzing each function and loaded to the target using Abatron debugger.

As can be seen, the measured throughput for a single 64-byte packet is 2.51 Mbps and that for a single 1500-byte packet is 25.6 Mbps. In real-life applications, the throughput will be higher due to packet queuing. When the packets are sent continuously, they get queued and the queue handler processes all available packets in one shot. Using SmartApplications, the throughput for a real-life scenario was measured to be 3.12 Mbps for 64-byte packet and 40 Mbps for 1500-byte packet.

Notes

Function	File	Time from INTA (in µSec)	
		Packet size: 64	Packet size: 1500
PCI_FRM Assertion		-3.8	-117.46
DMA begin		-3.5	-117.17
INTA: Interrupt from NIC1 (Trigger for Analyzer)		0	0
rc32300_irqdispatch	~linux/arch/mips/rc32300/79S334/irq.c	4.32	4.32

{
do_IRQ
}
	~linux/arch/mips/kernel/irq.c	5.96	5.95

do_IRQ	~linux/arch/mips/kernel/irq.c	5.96	5.95

{
handle_IRQ_event
}
	~linux/arch/mips/kernel/irq.c	11.36	10.65

handle_IRQ_event			

{
action->handler (= speedo_interrupt)
}
	~linux/arch/mips/kernel/irq.c	14.13	13.37

speedo_interrupt	~linux/drivers/net/eepro100.c	14.4	14.58

{
speedo_rx
}
	~linux/drivers/net/eepro100.c	19.72	19.27

speedo_rx			

{
eth_copy_and_csum
netif_rx
}
	~linux/drivers/net/eepro100.c
	~linux/net/core/dev.c
		48.4
		58.27
			55.35
			118.3

netif_rx	~linux/net/core/dev.c	58.27	118.3

{
dev_hold
__skb_queue_tail
cpu_raise_soft_irq
}
	~linux/include/linux/netdevice.h
	~linux/include/linux/skbuff.h
	~linux/kernel/softirq.c
		67.6
		69.8
		70.81
			128.67
			130.75
			131.98

do_softirq	~linux/kernel/softirq.c	97.05	161.68

{
h->action(h) (=net_rx_action)
}
	~linux/net/core/dev.c	102.15	162.42

net_rx_action	~linux/net/core/dev.c	102.15	162.42

{
handle_bridge
}
	~linux/net/core/dev.c	107.52	167.61

handle_bridge			

Table 5 Measured Time Intervals for Sub-routines (Page 1 of 3)

Notes

Function	File	Time from INTA (in µSec)	
		Packet size: 64	Packet size: 1500

{
br_handle_frame_hook
}
	~linux/net/bridge/br_input.c	107.52	170.96

br_handle_frame_hook (=br_handle_frame)	~linux/net/bridge/br_input.c	107.52	170.96

{
br_fdb_insert
br_handle_frame_finish
}
	~linux/net/bridge/br_fdb.c
	~linux/net/bridge/br_input.c
		109.81
		119.67
			173.95
			183.45

br_handle_frame_finish	~linux/net/bridge/br_input.c	119.67	183.45

{
br_fdb_get
br_flood_forward
}
	~linux/net/bridge/br_fdb.c
	~linux/net/bridge/br_forward.c
		139.92
		144.995
			202.46
			207.53

br_flood_forward			

{
br_flood
}
	~linux/net/bridge/br_forward.c	144.995	207.53

br_flood	~linux/net/bridge/br_forward.c		

{
__packet_hook
(=__br_forward)
}
	~linux/net/bridge/br_forward.c	145.12	207.89

__br_forward	~linux/net/bridge/br_forward.c	145.12	207.89

{
br_forward_finish
}
	~linux/net/bridge/br_forward.c	146.87	209.59

__br_forward_finish	~linux/net/bridge/br_forward.c	146.87	209.59

{
__dev_queue_push_xmit
}
	~linux/net/bridge/br_forward.c	148.02	210.73

__dev_queue_push_xmit	~linux/net/bridge/br_forward.c	148.02	210.73

{
dev_queue_xmit
}
	~linux/net/core/dev.c	148.1	209.96

dev_queue_xmit	~linux/net/core/dev.c	148.1	209.96

{
q->enqeue
(=pfifo_fast_enqueue)
{
__skb_queue_tail
}
qdisc_run (=qdisc_restart)
}
	~linux/net/sched/
	sch_generic.c
	
	~linux/include/linux/skbuff.h
	
	~linux/net/sched/
	sch_generic.c
		150.88
		
		155.38
			212.68
			
			218.32

Table 5 Measured Time Intervals for Sub-routines (Page 2 of 3)

Notes

Function	File	Time from INTA (in µSec)	
		Packet size: 64	Packet size: 1500
qdisc_restart	~linux/net/sched/ sch_generic.c	155.38	218.32

{
q->dequeue (=pfifo_fast_dequeue)
dev->hard_start_xmit (=speedo_start_xmit)
}
	~linux/net/sched/ sch_generic.c
	
	~linux/drivers/net/eepro100.c
		155.38
		
		160.0
			218.32
			
			222.48

speedo_start_xmit			

{
wait_for_cmd_done
}
	~linux/drivers/net/eepro100.c	171.85	242.11

DMA from memory to NIC2		178.26	248.61
Interrupt from NIC2		190.95	330.34
Throughput		$(64*8*10^6)$/ (1024*1024*(191+3.8)) **=2.51 Mbps**	$(1500*8*10^6)$/ (1024*1024*(330+117)) **=25.6 Mbps**
Throughput from SmartApplications		**3.12 Mbps**	**40 Mbps**

Table 5 Measured Time Intervals for Sub-routines (Page 3 of 3)

Suggestions to Optimize Software

1. Avoiding memcpy from DMA'ed memory to skbuff memory.

As can be seen from Table 5, the function eth_copy_and_csum takes 7 microseconds in the case of 64-byte packet and 57.4 microseconds in the case of 1500 byte packet. This can be avoided. The driver does provide a function that avoids memcpy. A variable rx_copybreak is declared towards the beginning of the driver code that is set to 1518 by default. This number (variable) is defined as the threshold above which the operating system will not perform memcpy. This value can be made extremely small, or even zero or 1. The trade-off is memory wastage. To avoid memcpy, the system assumes that every arriving packet will be the maximum size possible and allocates maximum number of buffers of the maximum packet size ahead of time. Depending on the application, this threshold value may be reduced to a level that represents a reasonable balance between memory wastage and higher performance. In a focused experiment, with this value set to 200, the throughput when measured with SmartApplications jumped from 40 Mbps to 52 Mbps.

2. Improving the Interrupt Handler.

Once the CPU is interrupted, 14 microseconds elapse before the "driver's interrupt handler (speedo_interrupt)" begins to execute. (Note: This is different from the conventional measure of interrupt latency which in the case of Linux relates to "kernel's interrupt handler" and is found to be approximately 4 microseconds.) Since the interrupt lines of both the NIC cards are tied together in the 79S334 evaluation

Notes

board, the CPU sees them as one interrupt, and the interrupt handler in the driver has to poll for the source of the interrupt. If they were provided in separate interrupt lines, the interrupt handler can directly call the appropriate routine soon after an interrupt arrives. This will also save few clock cycles.

3. SMP (Symmetric Multi Processing).

Linux supports SMP (Symmetric Multi Processing), and in a single CPU system, like the 79S334, calls that are related to SMP are not relevant. Removing them can save few CPU cycles. These calls are spread all across the kernel so that removing them will require careful review of the kernel. It is recommended that this step be performed only when the code base is deemed to be relatively stable. An update in the kernel revision, for example, will not be easy once the SMP calls are removed from the existing revision.

4. CPU clock.

As shown in Table 5, the time CPU took to process the packet once it was made available to the upper layer to the point where the packet is handed over to the device driver (from netif_rx to dev->hard_start_xmit) is (160 - 58 =) 102 microseconds. This time can be safely assumed to be a function of the CPU clock, since no peripheral access is performed during this time. By increasing the CPU clock rate, this time could be reduced.

Conclusion

From the above results, a valid co-relation between the lab tests and the design simulations was established. The RC32334/RC32332 YC revision devices are seen to have improved performance over the previous Z revision, especially for target write transactions. The results from the bridge experiment using Linux are shown to provide a basic platform for any real world application. As seen from these results, certain sub-routines require a fixed amount of time, while other sub-routines can be modified and optimized to achieve the maximum possible throughput at a system level.

For a detailed software code that was used in the above software tests, please email rischelp@idt.com.

For hints on optimizing hardware, refer to IDT Application Note AN-366 which can be found on IDT's web site at www.idt.com.

Acknowledgements

The authors would like to thank the following contributors from IDT who provided relevant technical information for this document: Upendra Kulkarni, Kasi Chopperla, Paul Snell, and Dan Steinberg.

References

1. IDT Application Note AN-350: RC32334/RC32332 Differences Between Z and Y Revisions.
1. IDT Application Note AN-366: Optimizing the PCI Interface on the RC32334/RC32332.
2. IDT79RC32334 and IDT79RC32332 Integrated Communications Processor User Reference Manual.
3. PCI Local Bus Specification. Rev. 2.2 - PCI special interest group.
4. PCI System Architecture. 4th edition - Mindshare, Inc. Tom Shanley and Don Anderson.
5. http://www/tldp.org - The Linux Modular Bridge And STP - Uwe Böhme, 2000.

IMPORTANT NOTICE AND DISCLAIMER

RENESAS ELECTRONICS CORPORATION AND ITS SUBSIDIARIES (“RENESAS”) PROVIDES TECHNICAL SPECIFICATIONS AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT OF THIRD-PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for developers who are designing with Renesas products. You are solely responsible for (1) selecting the appropriate products for your application, (2) designing, validating, and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. Renesas grants you permission to use these resources only to develop an application that uses Renesas products. Other reproduction or use of these resources is strictly prohibited. No license is granted to any other Renesas intellectual property or to any third-party intellectual property. Renesas disclaims responsibility for, and you will fully indemnify Renesas and its representatives against, any claims, damages, costs, losses, or liabilities arising from your use of these resources. Renesas' products are provided only subject to Renesas' Terms and Conditions of Sale or other applicable terms agreed to in writing. No use of any Renesas resources expands or otherwise alters any applicable warranties or warranty disclaimers for these products.

(Disclaimer Rev.1.01)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Contact Information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit www.renesas.com/contact-us/.

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.