Development and analysis of a versatile, reusable, high speed, DMA controller for custom embedded applications using the PCI bus

Michael Eastman
Development and Analysis of a Versatile, Reusable, High Speed, DMA Controller for Custom Embedded Applications using the PCI Bus

by
Michael G. Eastman
Title of Thesis: Development and Analysis of a Versatile, Reusable, High Speed DMA Controller for Custom Embedded Applications

I Michael G. Eastman hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.
1. ABSTRACT

This thesis investigates the plausibility of designing and developing a versatile, reusable, high speed interface for custom computing applications, based on the Peripheral Component Interface (PCI) Bus. A PCI I/O board was developed, utilizing mainly Complex Programmable Logic Devices (CPLD's), which included a custom Direct Memory Access (DMA) Controller to take advantage of the unique feature set of the PCI bus. The arbitration mechanisms and performance characteristics of the PCI bus are taken advantage of in order to achieve a maximum burst throughput rate of 66 Megabytes per second. Performance characteristics of the I/O board are analyzed for two separate PCI host systems. In the faster of the two systems, a 166MHz Pentium PC, a maximum aggregate throughput rate of 54 Megabytes per second for PCI burst writes was achieved. In all cases throughput increased as a function of transfer size. Due to buffering implementations in the host systems write performance was always superior to read performance.

In addition to exceptional throughput capability, this implementation provides a design engineer with a versatile interface which can be mated to a number of high performance applications. The PCI I/O board's external interface is implemented with a CPLD which can be quickly and easily modified to meet the needs of practically any custom interface without decreasing PCI bus performance. Using the on-board latency timer and programmable FIFO's the board can be fine tuned to meet a variety of application requirements.

The two main design goals were to provide unlimited bursting capability and to transfer 32-bits of data on every clock. The first was achieved through the implementation of a 32-bit burst Transfer Count register. The second goal had to be reduced by 50% due to a timing margin violation discovered during board debug.
1. ABSTRACT ................................................................................................................... 3

2. INTRODUCTION AND BACKGROUND .................................................................. 5
   2.1. PLATFORM SELECTION ............................................................................... 5
   2.2. DESIGN GOALS .......................................................................................... 6
   2.3. TECHNICAL OVERVIEW ............................................................................. 8
       2.3.1. Custom Hardware .............................................................................. 8
       2.3.2. Validation Software .......................................................................... 9
   2.4. SYSTEM SPECIFICATION .......................................................................... 10
       2.4.1. System Architectural Block Diagram ................................................ 10
       2.4.2. PCI I/O Board Architectural Block Diagram ...................................... 12
       2.4.3. Equipment Configuration .................................................................. 13

3. FUNCTIONAL SPECIFICATION ............................................................................. 13
   3.1. HARDWARE SPECIFICATION .................................................................. 13
       3.1.1. Compatibility ....................................................................................... 13
       3.1.2. PCI Interface ....................................................................................... 14
       3.1.3. DMA Controller .................................................................................. 15
           3.1.3.1. DMA Source Address Register ................................................... 16
           3.1.3.2. DMA Transfer Count Register ..................................................... 16
           3.1.3.3. DMA Command/Status Register .................................................. 17
           3.1.3.4. DMA Next Chain Address Pointer Register ............................... 19
       3.1.4. On-board Data Flow Control .................................................................. 19
       3.1.5. PCI Configuration Region ................................................................... 20
       3.1.6. Parallel Data Interface ......................................................................... 20
       3.1.7. JTAG Interface ...................................................................................... 21
   3.2. SOFTWARE SPECIFICATION ...................................................................... 21
       3.2.1. Compatibility ....................................................................................... 21
       3.2.2. PCI Plug 'n Play Usage ....................................................................... 21
       3.2.3. DMA Controller Programming ........................................................... 21
           3.2.3.1. Programming the DMAC in Normal Mode .................................... 21
           3.2.3.2. Programming the DMAC in Chaining Mode ................................. 22

4. ANALYSIS .............................................................................................................. 23
   4.1. PERFORMANCE MEASUREMENT .................................................................. 23
       4.1.1. Logic Analyzer Measurement ............................................................... 24
       4.1.2. Programmable Interval Timer Measurement ....................................... 24
       4.1.3. Performance Data ............................................................................... 25
   4.2. ADHERENCE TO GOALS AND PROBLEMS ENCOUNTERED ..................... 34
       4.2.1. Goals Achieved .................................................................................... 35
       4.2.2. Transfer Rate Problem ....................................................................... 35
       4.2.3. Transfer Count Problem ..................................................................... 36
   4.3. POTENTIAL IMPROVEMENTS AND FOLLOW-UP WORK .............................. 37

5. APPENDIX .............................................................................................................. 39
   5.1. APPENDIX A - GLOSSARY ......................................................................... 40
   5.2. APPENDIX B - SCHEMATICS ..................................................................... 43
   5.3. APPENDIX C - PROGRAMMABLE LOGIC SOURCE CODE ......................... 51
       5.3.1. PCICTRL PLD ...................................................................................... 52
       5.3.2. DMAC PLD .......................................................................................... 64
       5.3.3. DECODE PLD ....................................................................................... 72
       5.3.4. PARIO PLD .......................................................................................... 94
   5.4. BIBLIOGRAPHY ........................................................................................... 101

6. END NOTES ........................................................................................................... 104
2. Introduction and Background

2.1. Platform Selection

Many of today’s computer based imaging applications, such as printing, scanning, and video rendering, require large amounts of data to be moved very quickly. The main requirement is to move data between a main memory resource (typically Dynamic Random Access Memory (DRAM)) and a specific, often custom, Input/Output (I/O) Interface. This thesis provides a Personal Computer (PC) based mechanism for high speed data transfer between system DRAM and a first-in-first-out (FIFO) Memory system. To perform this task a Peripheral Component Interconnect (PCI) add-in card was developed which may be easily interfaced to a variety of custom I/O subsystems using programmable logic. In fact a logic device has been provided which may be modified to meet a designers specific needs. It should be noted that although this design is targeted for a PC add-in card the PCI bus has spread to a variety of other systems with the same electrical characteristics but different mechanical requirements. There are numerous examples such as PCI Mezzanine Card (PMC) which is used for industrial bus (VME, Multibus) expansion, PowerPC platforms from Apple, Motorola, or IBM, etc. This design could be easily reused for these other system architectures without modifying any of the logic design. A modification of the mechanical form-factor would enable this design to function gracefully in these environments.

An embedded systems designer has a wide range of “platform” or Bus Choices today. Multibus II, Multibus, and VME are a few of the more prevalent “industrial” busses currently available. The problem with systems based on these busses is that they tend to be expensive and often have long development cycles. It becomes necessary to purchase multiple boards for various system requirements, or to design your entire system on a single board or set of boards. The PC, however, provides a wide range of commonly used features such as parallel ports, serial ports, and high quality video graphics. Recently Personal Computers have been migrating into a larger number of industrial or embedded applications. This is due to the fact that PC’s are becoming more and more ubiquitous causing their price/performance ratio to become smaller and smaller which in turn makes them much more attractive to designers of low volume applications. This trend has, over the last few years, enabled Original Equipment Manufacturers (OEMs) to leverage the PC volume manufacturing for a growing number of embedded systems. Their utilization covers the full spectrum, from using PC-type Chip-sets and microprocessors and designing them onto a custom board within a system, to actually embedding an entire PC (with or without chassis) into a larger system. In all cases the goal is to take advantage of the hardware price/performance and/or the vast amount of software that is currently available for PC-type machines. One of the compelling reasons for utilizing a PC in an embedded environment is the availability and ease of use of a Graphical User Interface (GUI). Historically when
designing an embedded system one of the most difficult tasks has been implementing an elegant operator interface. Development of graphical interfaces was typically custom leading to interface models which were difficult to develop, and often cumbersome to use. Additionally there was no standard look or feel between applications. With a PC Platform the Video interface is free, along with widely available development environments for graphical interfaces.

In the past, the main drawback of the PC for many embedded applications was limited performance. Today’s Industry Standard Architecture (ISA) Bus has remained essentially the same since IBM came out with the PC-AT and has a maximum bandwidth of just over five Megabytes Per Second\(^1\) Additionally, having multiple masters of an ISA Bus is cumbersome and inefficient. In order to perform DMA type operations the Central Processing Unit (CPU) was put into a hold state and no concurrent operations could take place between the CPU and alternate masters. The enhanced version of this bus (developed by a consortium led by COMPAQ) which increased the data bus width from 16 to 32 bits and was named EISA, for Enhanced ISA, has a theoretical bandwidth of 33 Megabytes Per Second, and provides a somewhat more robust method of handling multiple masters\(^2\) The advent of the PCI Bus, which is 32 bits wide and has a theoretical maximum bandwidth of 132 Megabytes Per Second, has enabled embedded system designers to utilize the Personal Computer in a much larger number of applications which require higher bandwidth. High speed burst transactions and multiple masters are gracefully provided for in the PCI Specification. The PCI Bus architecture was developed in such a way that it is decoupled from the CPU thus enabling concurrent operation between the CPU and alternate PCI masters.

One of the most common areas requiring high data throughput for embedded systems is imaging. Video applications must be clear and crisp and look pleasing to the eye. Documents and printing plates must be rendered rapidly, and often one document must be scanned in while another one is being printed out, and perhaps another one is being manipulated by a variety of image processing algorithms in preparation for print-out. Often there may be multiple events occurring which each require a throughput of greater than 20 Megabytes Per Second. This is clearly beyond the capability of either ISA or EISA. In the past the Personal Computer was not an option for this type of application. Today the PCI Bus provides a robust interface which enables multiple masters to utilize the high bandwidth bus.

2.2. Design Goals
The overall objective of this investigation was to design and implement a PCI Add-in-Board based upon a custom Direct Memory Access (DMA) Controller. This technique was chosen because a DMA Controller requires less programming overhead than a CPU solution, is less complex, and more efficient for data
movement operations. The most unique aspect of the DMA Controller designed for this thesis is the ability to perform very large numbers of data transactions per burst. Most typical DMA Controllers provide burst counts of four or less. The performance goal was to operate at a burst data transfer rate approaching the theoretical maximum throughput bandwidth of the PCI Bus. The aggregate rate would be somewhat slower depending upon bandwidth utilization by other potential PCI Bus initiators (e.g. Host CPU, other PCI add-in cards, etc.). Actual system throughput is ultimately determined by the performance of the PCI implementation and DRAM Subsystem within the host PC System as well as the efficiency of this design.

In addition to throughput performance, a goal of this development was to provide a solution which required very little CPU programming overhead. If the software necessary to control the DMAC was too complex and cumbersome the performance achieved may not be worth the overhead involved. To accomplish this goal, a small effective register set was implemented. Additionally both Normal and Chained DMA transfers are supported to further reduce software complexity.

A secondary goal of this thesis was to enable other design engineers to utilize the PCI portion of this design and customize the I/O interface to their own specific requirements. A functional interface was designed for evaluation purposes, but ultimately the PCI I/O board must be malleable enough to be incorporated into an actual application. More than twenty-two signals, including 8-data bits, are available within a single CPLD device to provide a design engineer with resources to implement a robust custom parallel interface.

Another unique aspect of this implementation is that it is suitable for applications which are considered low volume and high performance in the personal computer industry. In many instances an off-the-shelf solution is not suitable, and developing an ASIC is cost prohibitive. The solution proposed here is more expensive than an off-the-shelf solution, but much lower cost than an ASIC implementation for small volumes. For an application which would utilize hundreds of thousands of units, an ASIC may indeed be a less expensive overall solution. This implementation could be ported to an ASIC design if so desired. This thesis focuses on the DMA Controller and PCI Bus Interface Implementation including the data path from the PCI Bus to the on-board FIFO Subsystem. Details of some of the surrounding logic such as the PCI Configuration Region, which is a general PCI requirement and not inherently unique, are considered beyond the scope of this thesis.
2.3. **Technical Overview**

This section provides the details of the development portion of this thesis. It is broken down into two subsections Hardware Development, and Validation Software. The goals and reasons for the thesis are described.

### 2.3.1. Custom Hardware

Complex Programmable Logic Devices (CPLD’s) manufactured by Altera Corporation were selected as the logical interface to the PCI Bus and as the basis for a custom DMA Controller developed explicitly for this application. The PCI add-in card does not contain a microprocessor, but is programmed by the Personal Computer’s host processor. Once programmed, the DMA Controller acts as a PCI Initiator or Bus Master. It has the ability to initiate PCI Bus Cycles and perform high speed data transfers between main system memory and the on-board FIFO memory buffer data store. Once on the PCI Interface Board Data is moved between the FIFO and a custom bi-directional I/O interface designed specifically to demonstrate the functionality of this approach. The FIFO memory may be read from and written to at the same time. For example, when a DMA transfer is taking place such that data is being written from system memory into the on-board FIFO’s, data may also be moved from the on-board FIFO’s to an external device such as a laser marking engine. In a “real-life” application the I/O interface would be designed to match whatever device the PCI add-in card was delivering data to or receiving data from.

The intent here was to utilize the bandwidth available on the PCI bus in as efficient a manner as possible assuming that the device developed here was the highest priority device in the system. This requirement precluded the use of any currently available off-the-shelf DMA Controllers. This requirement meant that a device specifically designed to operate with the PCI bus was required. Some of the unique performance features of the DMA Controller developed for this thesis are:

- Seamless Logical Interface to the PCI Bus
- Virtually Unlimited Bursting Capability (4 Gig Burst Count)
- Fly-by mode optimized for DRAM-FIFO Interface
- 4 Gigabyte Addressing Capability

The items designed and/or developed for the Hardware Development portion of this thesis are as follows:

1) *Hierarchical Schematic Entry for board design*

2) *Logic Design & Synthesis of PCI DMA Controller Functionality*

3) *Logic Design & Synthesis of PCI Burst Bus Master & Slave Interfaces*
4) Logic Design & Synthesis of FIFO Input/Output Interface

5) *Layout, Routing, and Manufacture of Printed Circuit Board

6) Debug and Analysis of Printed Circuit Board

*This is a Purchased Service handled by a manufacturing facility.

2.3.2. Validation Software

The Validation software consists of a series of tests to show how the PCI I/O board performs from a data movement standpoint. These tests demonstrate functionality in addition to enabling performance measurement. There is a test for each one of the four possible modes of operation for the board as follows:

- Normal Mode Input
- Normal Mode Output
- Chaining Mode Input
- Chaining Mode Output

The direction of the each of the Modes refers to the direction of data movement with respect to an external device. In input mode data is transferred from the on-board FIFO memory subsystem to the host PC’s DRAM subsystem (Memory Write Operation with respect to the PCI Bus). In output mode data is transferred from DRAM to the on-board FIFO memories (Memory Read Operation with respect to the PCI Bus). The objective of the software is to provide a mechanism to evaluate functionality and overall board performance. The software is written in such a manner that the PCI I/O board will attempt to operate as fast as possible for user specified data sizes. Since the data transfer size capability of the add-in card is 4Gigabytes, the actual data transfer size is bounded only by the amount of memory in the PC. Actual throughput is measured with both a logic analyzer and with an internal software timer. This correlation demonstrates the accuracy of the performance analysis.

The software utilizes the on-board PCI Configuration Register information to determine the base address of the PCI I/O board. The on-board DMA Controller is then programmed to perform a specified number of 32-bit wide transfers between main memory and the PCI add-in board. The direction may be selected as input (as would be the case for a scanner type operation) or as output (as would be the case for a marking type operation). The DMA transfer process is initiated by programming the DMA Control register in either chaining or normal mode as described in the preceding sections. Upon completion of the DMA process an interrupt provides notification that the transfer has finished.
The goal of the evaluation process is not to investigate or analyze operating system latency issues, but to simply determine the performance of the hardware subsystem within a given PC environment. Since the PCI I/O board has no microprocessor, and therefore no software to execute, it is beyond the scope of this investigation to attempt to analyze operating system dependencies of the host system being tested. Additionally the portion of the design that is performance critical is the board’s ability to move data between system memory and the local on-board FIFO’s. The interface between the on-board FIFO’s and the external world (scanner, marking engine, etc.) is typically custom and/or specific to the particular interface device. The external interface for this demonstration vehicle is implemented as a simple 8-bit wide differential handshaken bus. For this design to be used in a real application it would be the designer’s responsibility to customize that portion of the design to interface requirements of the peripheral being accessed. This interface could be any number of bits wide depending upon the device chosen. Typically the PCI portion of the design will be considerably faster than the input or output device. Because of this it is beyond the scope of this investigation to analyze the data throughput between the on-board FIFO’s and an external device. It should be obvious that the maximum aggregate bandwidth to an external device would be something less than the maximum achieved across the PCI bus depending on the type of drivers and connection used.

In reality the test software is quite simple. It is responsible for programming the DMA Controller, and then waiting. It does nothing until an interrupt occurs to terminate the cycle.

2.4. System Specification

This section provides an architectural description of the development portion of this thesis. A detailed description of the hardware functionality is provided including block diagrams which show bus organization and data flow paths. A description of the development equipment is provided.

2.4.1. System Architectural Block Diagram

The following diagram describes the overall architecture of a standard PCI Based Personal Computer System. Two systems using this architecture were used to test the final finished prototype boards.
Diagram 2.1: PCI System Architectural Block Diagram
2.4.2. PCI I/O Board Architectural Block Diagram

The following diagram describes the architecture of the proposed PCI I/O board. This block diagram is the first step of the hierarchical top-down design.

Diagram 2.2: PCI I/O Controller Board Block Diagram
2.4.3. Equipment Configuration

Hardware Development and Analysis Tools:
Dataio Dash6 Schematic Entry.
Dataio ABEL6 Logic Development Software
TimingDesigner Professional from Chronology Corporation
Hewlett Packard 1654B Logic Analyzer
Tektronix Oscilloscope

Software Development Tools:
Borland C++ Version 4.2

Performance Analysis System:
Intel Pentium 100 (100MHz Pentium Processor Based PCI Desktop System using 82430NX PC1set Chip-set)
- Intel Pentium 166 (100MHz Pentium Processor Based PCI Desktop System using 82430FX PC1set Chip-set)

3. Functional Specification

An overall description and definition of the Development portion of this thesis is provided. Operational and implementation details are described to enable the reader to understand the system that has been developed. A top-down approach is used to describe the overall system with very specific information provided for the most important aspects of the development.

3.1. Hardware Specification

This section provides a detailed look at the operation of the development portion of this thesis. Operation of the PCI I/O board is explained and actual implementation details are provided where required for clarity.

3.1.1. Compatibility

The PCI I/O Board is PCI Bus Compatible as defined within the PCI Local Bus Specification Revision 2.1 dated June 1, 1995 as an Initiator/Target (Master/Slave) Device. Mechanically the Board follows the PCI Local Bus Specification for a 32-bit Standard length expansion card. The only known deviation from this document is that the length of the card will be considerably shorter than a standard full length PCI expansion card.
3.1.2. PCI Interface

The PCI Interface is implemented via multiple Altera EPX780 Field Programmable Gate Array (FPGA) Logic devices. The board is capable of being a PCI Initiator or Target, and is 32-bit PCI compatible. As a Target (Slave) the board is capable of responding to I/O Read, I/O Write, Configuration Read, and Configuration Write Cycles as defined in the PCI Bus Specification. In Initiator (Master) mode the PCI I/O board is capable of performing Memory Read and Memory Write Cycles. It is Initiator mode that is the focus of this board design. Every attempt has been made to make both the Memory Write and Memory Read operations take place as rapidly and efficiently as possible. The unique DMA Controller implementation, which takes full advantage of the PCI Burst Bus functionality, provides a very high bandwidth mechanism for data transfer across the PCI Bus. A state machine technique has been used within the programmable logic to control both Initiator mode cycle generation, and Target mode response. Diagram 3.1 below shows the design of the state machine in a graphical format:

![Diagram 3.1: PCI State Machine](image-url)
3.1.3. DMA Controller

The single channel Fly-by DMA (Direct Memory Access) Controller is implemented with an Altera EPX780 device and a series of discrete counter circuits. The performance goal for the DMA controller design was to be capable of performing 32-bit burst accesses across the PCI Bus in 0-wait states per data phase (maximum theoretical throughput is therefore 132 MBytes/Second). One of the most interesting performance features of this DMA Controller is the burst transfer mechanism. Most DMA Controllers are capable of bursting a fixed number of data elements per address phase. Typical High Speed DMA Controllers are capable of bursting four data elements. The PCI Bus however is not bound by a number of data phases per transaction. The theoretical maximum is infinite. This DMA controller takes full advantage of this bus feature and is capable of bursting up $2^{32}$ or 4G data elements. This feature is somewhat impractical in today’s computers, but the advantages of being able to burst 32 or 64 or even 256 Quad Word Data elements are quite apparent. Much of the overhead for typical bus transactions takes place during the addressing phase of the cycle. If 1 Megabyte of data must be transferred and only 4 bytes of data may be transferred for each address phase generated then 262,100 address phases must be generated in order to transfer the entire megabyte of data. If 128 bytes may be transferred for each address phase only 2048 address phases must be generated to transfer the entire block of data. It is obvious from this simple analysis that the fewer the number of addresses that must be generated the higher the performance if all other factors are equal.

The transaction length of the PCI Bus is bound not by a transfer count, but by a time limit. Each PCI Bus master must implement a Latency timer which is programmed by a systems programmer to provide maximum system performance without causing resource starvation. The system may then be fine tuned for optimum performance. It should be well within the limits of a typical system to be able to burst 32 Quad-Word data elements for a total of 128 bytes per address phase. The DMA controller designed here takes full advantage of that feature by enabling transfer requests to be the entire length of the programmed transfer. It is extremely unlikely that this DMA Controller will ever be the cause for a bottleneck due to burst count latency.

All registers within the DMA controller may be written to and read from by another PCI Master. All DMA transfers take place between system memory and a 32-bit wide on-board First-In-First-Out (FIFO) Memory Buffer. The transfers may be programmed as either Input (FIFO to System DRAM) or Output (System DRAM to FIFO). Memory to memory transfers are not supported in this design. All transfers take place on 32-bit aligned boundaries. Discrete counters are used to control Address generation and transfer counting operations.
The starting address for any DMA Process or Chain Descriptor must be a 32-bit aligned Memory Address; the transfer count is controlled with 32-bit granularity. All DMA Transfers move data between system memory and the on-board FIFO Memory Buffer, and operate in Demand Mode (DMA Requests take place based upon an input signal being driven to the active state by an external device). The PCI I/O board has four 32-bit Read/Write registers are available for controlling and determining status of the DMA Process.

- DMA Source Address
- DMA Transfer Count
- DMA Command/Status
- DMA Next Chain Pointer

The following four sections describe each of these registers in detail.

3.1.3.1. DMA Source Address Register
This 32-bit wide Read/Write register controls the beginning Memory Address of the current PCI Burst DMA Transfer. Prior to initiating a DMA Process this register is programmed with a 32-bit aligned main memory address. This address is the first system memory location accessed during a given DMA Process or Chaining Operation. The completion of a valid data transfer cycle on the PCI Bus is indicated by the signals IRDY# and TRDY# being sampled active at the same clock edge. When the PCI I/O Board is a PCI Master the value in the Source Address Register is incremented to the next 32-bit address for each valid data transfer cycle. The initial value in the DMA Source Address Register is either programmed explicitly by the application in Normal Mode, or it is loaded by the DMA Controller when operating in Chaining Mode.

3.1.3.2. DMA Transfer Count Register
This 32-bit wide Read/Write Register controls the number of 32-bit transfers that take place during a single DMA Process. When the PCI I/O Board is a PCI Master, the completion of a valid data transfer cycle is indicated by the PCI signals IRDY# and TRDY# being sampled active at the same clock edge. The DMA Transfer Count is decremented for each valid data transfer cycle. The DMA Process is completed when the programmed number of transfer cycles have successfully completed and the Transfer Count Register has reached zero. The DMA Transfer Count Register is either

16
programmed explicitly by the application in Normal Mode, or it is loaded by the DMA Controller in Chaining Mode.

3.1.3.3. **DMA Command/Status Register**

This 32-bit wide Read/Write register provides Status information from, and Command information to, the on-board DMA Controller. Table 3.1.3.3 on the following page provides a description of each of the valid bits in the DMA Control/Status Register. The DMA Command Register is either programmed explicitly by the application or it is loaded by the DMA Controller itself in Chaining Mode.
### Table 3.1.3.3 DMA Command/Status Register Bit Definition:

<table>
<thead>
<tr>
<th>Bit Location</th>
<th>Bit Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit 0</td>
<td><strong>DMA Done</strong> - This read-only status bit is active high upon completion of a Programmed DMA Transfer, and is reset upon system reset; by writing a &quot;0&quot; to the DMA Start Bit, or by writing a non-zero value to the DMA Transfer Count Register.</td>
</tr>
<tr>
<td>Bit 1</td>
<td><strong>INT Pending</strong> - This read-only status bit is active high upon generation of an Interrupt to the Host CPU. It is reset upon writing any data value to the Clear Interrupt Register at offset 6Ch from the Base I/O address.</td>
</tr>
<tr>
<td>Bits 7 - 2</td>
<td>Unused</td>
</tr>
<tr>
<td>Bit 8</td>
<td><strong>DMA Start</strong> - This read/write bit controls the initiation of a DMA Process. A Value of 0 indicates that a DMA Process has completed or has been stopped. A Value of 1 indicates that a DMA Process is in progress.</td>
</tr>
<tr>
<td>Bits 12, 11, 10, 9</td>
<td><strong>DMA Cycle Type S0, S1, S2, S3</strong> - These read/write bits control whether the DMA Process is an input (Read from I/O, Write to Memory) or an output (Read from Memory, Write to I/O) as well as the PCI Cycle type that is generated during the transfer: (Bit 12 is S3, Bit 9 is S0).</td>
</tr>
<tr>
<td>Bit 13</td>
<td>Unused</td>
</tr>
<tr>
<td>Bit 14</td>
<td><strong>DMA Mode</strong> - This read/write bit controls Chaining Mode and Memory direction.</td>
</tr>
<tr>
<td>Bit 15</td>
<td>Unused</td>
</tr>
<tr>
<td>Bits 17, 16</td>
<td><strong>Interrupt Control Mode 0 and 1</strong> These read/write bits control interrupt generation for DMA Processes either generate no interrupt, Interrupt on Terminal Count (T.C.) or Interrupt at the end of the Last Chain. (Bit 17 = Mode 1, Bit 16 = Mode 0)</td>
</tr>
<tr>
<td>Bits 18-31</td>
<td>Unused</td>
</tr>
</tbody>
</table>
3.1.3.4. **DMA Next Chain Address Pointer Register**

This 32-bit wide Read/Write register specifies the address of the next Chain Descriptor to be accessed when the DMA Controller is in Chaining Mode. This register provides the link between descriptors during scatter or gather type operations. When the DMA Controller is initially programmed in Chaining Mode this register is programmed with the value of the location of the first Chain in the programming link. Subsequent Chaining information is loaded without programmer intervention by the DMA Controller once the current chain has been completed. A value of 00000000h indicates the final Chain in the series. The format for a Chain Descriptor in Memory is shown below.

### Table 3.1.3.4 Chain Descriptor Format:

<table>
<thead>
<tr>
<th>Chaining Descriptor Information (32-bits Wide)</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control Word</td>
<td>n</td>
</tr>
<tr>
<td>Transfer Count</td>
<td>n + 4</td>
</tr>
<tr>
<td>Next Chain Address</td>
<td>n + 8</td>
</tr>
<tr>
<td>Start Address</td>
<td>n + 12</td>
</tr>
</tbody>
</table>

The memory locations for a single Descriptor must reside in four 32-bit aligned 32-bit wide contiguous memory locations. Individual Descriptors may be located anywhere within the 4 Gigabyte System Memory Space as long as they are 32-bit aligned. The 32-bit Next Chain Address values represent actual physical memory locations within the System Memory Space.

3.1.4. **On-board Data Flow Control**

The on-board DMA Controller operates in a Demand Mode fashion. When properly programmed, DMA Operation commences upon the receipt of a DMA Request signal from the Parallel Interface Control logic. This DMA Request is based upon a transfer being initiated, and data (or space depending upon direction of transfer) being available in the on-board FIFO Memory Buffer. There are two types of DMA requests that may be generated by the I/O Interface Logic. During normal operation a Burst Request is performed by the Parallel Interface Control Logic. This causes the DMA Controller to attempt to perform data transfers until all programmed DMA transactions have been completed. As long as there is a transfer request from the DMA Controller to the PCI Bus Interface Logic, the PCI Bus interface logic will request the PCI bus according to the rules described in the PCI Specification. Once a PCI transaction has been initiated, transfers will continue until one of the following events occurs:
a) The **GNT#** signal on the PCI Bus has been removed by the PC Host's PCI Arbiter and the local on-board latency timer has expired.

b) The **STOP#** signal has been asserted by the slave device being accessed on the PCI Bus indicating that it's transfer buffer's are full in input mode, or empty in output mode.

The PCI Interface logic will remove it's bus request signal (**BREQ#**) and re-initiate the arbitration process by re-asserting **BREQ#** upon the occurrence of one of these two events.

In Input Mode, when **PAGE_SYNC** is active and the FIFO status is "Almost Full", DMA Burst requests occur. The same is true in Output Mode when the FIFO status is "Almost Empty". During Input Mode, once **PAGE_SYNC** is removed by the external Input logic, the DMA requests will be non-burst. This causes the PCI Transfers to contain a single data phase until the FIFO Memory Buffer has been completely emptied. During this non-burst mode there will be a single data phase for each PCI Transaction. This logic is in place to prevent an overrun or underrun condition from occurring in the FIFO's.

### 3.1.5. PCI Configuration Region

The PCI Configuration Region is implemented via an Altera EPX780 FPGA Device. A 256 byte register space organized as sixty-four 32-bit words is implemented to perform the configuration functionality. The PCI Configuration Region is required in all PCI compliant devices. The register set associated with this region is configured at system initialization for proper operation in the host system. Since this region is standard PCI fare it will not be discussed in detail during this thesis, and is considered beyond the scope of this investigation.

### 3.1.6. Parallel Data Interface

The Parallel Data Interface is an 8-bit Data, Control, and Status interface for communication between the on-board FIFO memory buffer and the off-board I/O Device. This is a general purpose interface used for demonstration purposes only. In an actual application this logic, which is contained in a CPLD, would be designed to communicate with a specific device or set of devices. The on-board control registers are used to program the interface in either the input or output mode. In input mode data is transferred from an input device to the FIFO based upon the status of Input device and the FIFO (i.e. Does the input device have data to send, and does the FIFO have room in which to store the data being sent.).
the output mode data is transferred from the FIFO to an output device. Again, the status signals are used to condition this transfer.

In the CPLD Source file provided with this thesis there are fourteen pins available for implementing a custom bi-directional interface. This flexibility enables the designer to connect the PCI I/O board to a wide range of devices without any modification to the PCI interface logic.

3.1.7. JTAG Interface
A JTAG (Joint Test Action Group, IEEE 1149.1) interface is implemented in order to enable on-board programming of the SRAM Logic Array's in the Altera Flexlogic devices during system debug. A standard 20-pin Insulation Displacement Connector (IDC) header is used to interface the PCI I/O Board to a PC parallel port based JTAG programming interface for the Flexlogic.

3.2. Software Specification

3.2.1. Compatibility
The software developed for this thesis executes in a DOS 5.0 or greater environment. Protected mode drivers are utilized in order to read and write memory above the 1 Megabyte DOS boundary in order to test operation of the DMA process. The software is not intended to be a robust end-user application, but rather, to demonstrate the functionality and performance of the PCI DMA subsystem.

3.2.2. PCI Plug’n Play Usage
The PCI I/O Board is fully PCI compliant and supports the entire required Configuration Region. Because of this, the diagnostic software is able to determine where the board is located within the system’s I/O Map and perform accesses to it accordingly. Unlike ISA or EISA type add-in cards PCI Boards require no I/O location, Memory location, or interrupt jumpers.

3.2.3. DMA Controller Programming

3.2.3.1. Programming the DMAC in Normal Mode
When initializing the DMA Controller for Normal (non-chaining) Mode the first programming operation is to write bit 8 of the DMA Control Register (DMA Start) with a "0". This ensures that the DMA Controller is in a halted state. The Source Address and Transfer Count Registers may then be programmed in any order.
Finally the DMA Controller should be written with the appropriate values for the desired mode, as described in Table 3.1.3.3, with the DMA Start Bit set to a "1" to initiate the DMA Process.

Writing a "1" to the Start Bit of the DMA Controller causes the DMA Controller to initiate a DMA Transfer by requesting the PCI Bus. Once the bus has been granted the DMA Controller will drive the current address onto the PCI Bus along with the appropriate PCI Command signals to indicate the type of transfer as specified in Table 3.1.3.3. Once initiated the Normal Mode DMA Process will continue until one of the following occurs:

- The Transfer Count Register has reached a value of zero.

- The DMA Start bit in the DMA Control Register is written with a Zero (Process will be suspended at the completion of the current transaction, and may be restarted by rewriting the DMA Start bit with a One).

### 3.2.3.2. Programming the DMAC in Chaining Mode

When initializing the DMA Controller for Chaining Mode, the first programming operation is to write bit 8 of the DMA Control Register (DMA Start) with a "0". This ensures that the DMA Controller is in a halted state. Prior to initiating the DMA Process by programming the DMA Control Register for the desired chaining operation, the Chaining Descriptors should be programmed in System Memory. Each chaining descriptor must be located at a 32-bit aligned address. This enables the DMA Controller to properly fetch the chaining information. The chaining descriptors do not need to be located in a contiguous block of memory. Each descriptor may exist anywhere in memory as long as the address is 32-bit aligned. The Next Chain Address Pointer Register within the DMAC should be programmed after the chaining descriptors have been loaded into system memory. The value of this register should be the base address of the first chaining descriptor in System Memory. The Source Address and Transfer Count Registers are don't care values for the Chaining Mode initialization process. They will be loaded by the DMA controller when the first chaining descriptor is accessed. Programming the Next Address Pointer Register enables the DMA Controller to access the first Chain Descriptor in System Memory. Once the Next Address Pointer Register and Chaining Descriptors have been established the DMA Control Register may now be programmed to
operate in Chaining Mode according to the description in Table 3.1.3.3, and the DMA Start bit may be set to a "1" to initiate the DMA Process.

Once initiated, the Chaining Mode Process will continue until the Transfer Count has reached zero and the Next Address Pointer has been loaded with null descriptor reference (00000000h). As presented in the Control Register Bit Definition, an interrupt may be generated at the end of each chain (each time Terminal Count Reaches Zero), at the end of the last chain (Terminal Count is Zero and Next Address Pointer is Zero), or not at all. This may be changed on a chain by chain basis as the Control Word is the first value loaded during the chaining process.

4. **Analysis**

Performance data acquired during testing is presented and analyzed. Two measurement techniques were invoked in order ensure the data obtained was accurate. Two popular PCI systems were used to host the PCI I/O board for these performance measurements.

**4.1. Performance Measurement**

One of the main goals during the development of this thesis was data movement performance. This system architecture was defined to enable the DMA Controller to operate with minimal CPU overhead. This accomplishes two goals:

1) The CPU is freed up to perform other tasks

2) The overall DMA task becomes more efficient

Data throughput was performed in two different types of Personal Computer Systems. This was done to demonstrate that overall performance is greatly dependent upon the performance of the PCI system in which the PCI I/O board resides. Additionally, two independent measurement techniques were used in order to validate DMA transfer performance. One method invoked a logic analyzer to time the duration of a given transfer by sampling signals specific to the beginning and end of the process. The second technique involved writing software to utilize the programmable interval timer on the PC and calculate the bandwidth of a transfer. This approach provided two separate and independent measurements for a series of complete DMA transactions.

In addition to the diagnostic equipment used to measure performance, the board design was slightly modified to more accurately determine the maximum aggregate bandwidth of the system. The obvious bottleneck in the overall system is the ability of an external device to either fill or empty the on-board FIFO memory. The FIFO memory is meant to be used as a buffer to alleviate bandwidth discrepancies between the PCI Bus and the external resource. The goal of the
performance measurement is to determine what type of sustained throughput capability this board could have in a lightly loaded system. To achieve this end the DMA request input to the DMA Controller was modified for testing purposes. During normal operation the DMA request input is used to signal the DMA Controller that the FIFO is able to accept (in output mode) or provide more data. Even for a relatively fast external system the DMA Controller would not typically be able to attempt to burst more than 4 to 6 K Bytes without having to wait for the external device to move more data. In order to simply test the throughput of this PCI board the DMA request input was modified to be always requesting a DMA Transfer. In Output Mode this would mean that the FIFO would always appear to have space available for more data, and in input mode this would mean that the FIFO would always appear to have more data to provide. None of the cycle timing was in any way modified to accomplish this. All read and write cycles to the FIFO were performed to ensure that no performance enhancement occurred as a result of this modification. This enabled much larger blocks of data to be transferred without having to wait for an external device to process data. This enabled testing to be performed with data transfer sizes limited only by the amount of memory in the host PC System.

4.1.1. Logic Analyzer Measurement

A Hewlett Packard 1654B Logic Analyzer configured in state mode was used to capture signals pertinent to the beginning and end of a DMA Transfer Operation. The goal of this type of measurement is to accurately select an event that occurs immediately prior to the DMA process initiating, and then another event that occurs immediately after the transfer has been completed.

For measuring throughput the signal DMA_REG2_RD# was selected as a trigger mechanism to enable the logic analyzer to generate time stamps for both the beginning and end of the DMA transfer. This is the write strobe for the DMA Control Register which is written once to initiate the DMA operation, and then again, immediately subsequent to the DMA operation completing, to clear the DMA_START bit within the register. These two write operations serve to envelope the entire programmed DMA operation providing an analyzer reading of total elapsed time for the DMA transfer. Throughput, in bytes per second, is then calculated by dividing the total number of bytes transferred by this elapsed time in seconds. The logic analyzer has a 4ns granularity which is more than sufficient to provide reasonable accuracy for these throughput measurements. This technique was used to determine all measurements listed under the heading of Logic Analyzer.

4.1.2. Programmable Interval Timer Measurement

In order to provide correlation with the hardware measurements described above, the test routine included a mechanism to provide automatic
performance measurement information. The System Timer on the PC Host was used to enable the software to acquire accurate time-stamp information at the beginning and end of each block of data transferred.

The PC's Timer 0 is a programmable counter with an input frequency of 1.191318 MHz. The timer is typically configured to operate at an output frequency of 18.20 Hz when the PC is powered up. Immediately prior to setting the START bit to initiate the programmed DMA operation the on-board 16-bit timer, Timer 0, is interrogated for use as the initial time stamp. Once the programmed DMA transfer is finished, a completion notification interrupt is generated to the host CPU. Immediately subsequent to this interrupt being serviced, Timer 0 is once again interrogated to determine the end time of the transfer. For those transactions which caused an overlap in the counter (i.e. the counter decrements to zero and automatically starts counting at 0xFFFF again) an Interrupt Service Routine (ISR) was installed to count each of these situations. Therefore the algorithm used for determining the actual number of timer ticks was implemented as follows:

Assuming the count parameters of:

- Initial count = i
- Final Count = f
- Interrupt Count = n
- Timer Ticks = t;

\[\text{if}(n = 0) \text{ then} \]
\[t = i - f\]
\[\text{else} \]
\[t = (n-1) \times (0xFFFF) + (i + (0xFFFF - f))\]

Once found, t is multiplied by the period of 1 timer tick or 838.0965 x 10^9 to acquire the total elapsed time. Throughput, in bytes per second, is then acquired by dividing the total number of bytes transferred by this elapsed time in seconds. This technique was used to determine all measurements listed under the heading of Software Timer.

4.1.3. Performance Data

Two separate off-the-shelf PC systems were used to benchmark the transfer performance of the PCI I/O Board. Both systems included a 32-bit wide 33Mhz PCI bus. The first system, referred to as the Pentium 100, was a 100Mhz Pentium based platform using the Intel 82430NX PCI Chipset. The System Controller in this Chipset provides two 4-DWORD PCI-to-Memory Posted Write Buffers and one 4 QWORD PCI-to-Memory Read Pre-fetch Buffer. The buffering is used to enhance PCI Burst performance by not requiring the PCI Initiator to wait for accesses to
memory to take place. The second System, referred to as the Pentium 166, was a 166Mhz Pentium based machine using the Intel 82430FX PCIset. This Chip-set provides 12-DWORDs of PCI-to-Memory write posting and a snoop-ahead feature so that PCI-to-memory bursts in either direction can be sustained at up to 120MBytes/Second. The performance differences between platforms are not indicative of the processor speed, but are the result of an improved buffering and DRAM interface scheme within the PCI Chip-set used in the 166Mhz platform. It is important to understand that the CPU is not involved in the data transfer process, other than initially programming the registers of the DMA Controller. Additionally, performance measurements are not initiated until the CPU has finished programming the DMA operation. As the PCI bus has become more prolific the designers have continually improved the performance of the Chip-sets. It is also interesting to note the discrepancy between PCI read and write performance especially in the Pentium 100 System. During a write operation the data is present at the Initiator and is therefore ready and valid very early in the cycle. The initiator does not have to wait for the actual write to DRAM to take place; only for the data to be latched into the Chip-set’s buffer. A read operation, however, requires the data to be extracted from the DRAM subsystem prior to completing the data transfer across the PCI bus. This requires the initiator to wait for the complete DRAM access prior to receiving data.

The following two pages include tables which provide data size and throughput information for PCI reads and writes for each of the host systems utilized for this analysis:
Table 4.1 PCI Read Performance Data

<table>
<thead>
<tr>
<th>Transfer Size (Bytes)</th>
<th>Pentium100 Software Timer Rate (MB/S)</th>
<th>Pentium100 Logic Analyzer Rate (MB/S)</th>
<th>Pentium166 Software Timer Rate (MB/S)</th>
<th>Pentium166 Logic Analyzer Rate (MB/S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.92</td>
<td>1.93</td>
<td>2.6</td>
<td>2.63</td>
</tr>
<tr>
<td>64</td>
<td>3.31</td>
<td>3.02</td>
<td>4.85</td>
<td>5.04</td>
</tr>
<tr>
<td>128</td>
<td>4.41</td>
<td>4.55</td>
<td>8.57</td>
<td>8.98</td>
</tr>
<tr>
<td>256</td>
<td>6.20</td>
<td>5.95</td>
<td>15.33</td>
<td>15.45</td>
</tr>
<tr>
<td>512</td>
<td>7.57</td>
<td>7.32</td>
<td>22.40</td>
<td>22.44</td>
</tr>
<tr>
<td>1,024</td>
<td>8.03</td>
<td>8.12</td>
<td>30.66</td>
<td>30.33</td>
</tr>
<tr>
<td>2,048</td>
<td>8.60</td>
<td>8.62</td>
<td>36.41</td>
<td>36.63</td>
</tr>
<tr>
<td>4,096</td>
<td>8.93</td>
<td>8.90</td>
<td>40.17</td>
<td>40.45</td>
</tr>
<tr>
<td>8,192</td>
<td>9.05</td>
<td>9.04</td>
<td>43.15</td>
<td>43.07</td>
</tr>
<tr>
<td>16,384</td>
<td>9.10</td>
<td>9.11</td>
<td>44.49</td>
<td>44.52</td>
</tr>
<tr>
<td>32,768</td>
<td>9.14</td>
<td>9.14</td>
<td>46.31</td>
<td>46.37</td>
</tr>
<tr>
<td>65,536</td>
<td>9.15</td>
<td>9.16</td>
<td>47.52</td>
<td>47.53</td>
</tr>
<tr>
<td>131,072</td>
<td>9.18</td>
<td>9.18</td>
<td>48.12</td>
<td>48.15</td>
</tr>
<tr>
<td>262,144</td>
<td>9.18</td>
<td>9.18</td>
<td>48.41</td>
<td>48.44</td>
</tr>
<tr>
<td>524,288</td>
<td>9.15</td>
<td>9.16</td>
<td>48.57</td>
<td>48.59</td>
</tr>
<tr>
<td>1,048,576</td>
<td>9.18</td>
<td>9.18</td>
<td>48.63</td>
<td>48.66</td>
</tr>
<tr>
<td>2,097,152</td>
<td>9.17</td>
<td>9.17</td>
<td>48.67</td>
<td>48.69</td>
</tr>
<tr>
<td>4,194,304</td>
<td>9.17</td>
<td>9.18</td>
<td>48.69</td>
<td>48.71</td>
</tr>
<tr>
<td>8,388,608</td>
<td>9.17</td>
<td>9.18</td>
<td>48.69</td>
<td>48.72</td>
</tr>
<tr>
<td>12,000,000</td>
<td>9.17</td>
<td>9.18</td>
<td>48.69</td>
<td>48.72</td>
</tr>
</tbody>
</table>

Table 4.1 clearly shows the increased read performance of the 82430FX Chip-set demonstrated by the Pentium166 system. It is also very interesting to note the relationship between transfer size and aggregate throughput. This trend is present for all transfers analyzed. It is important to realize that the CPU programming overhead is not present in any of these numbers. The time required for the CPU to program the DMA registers is not measured, and therefore does not skew the results in the favor of the longer transfer sizes.
Table 4.2 PCI Write Performance Data

<table>
<thead>
<tr>
<th>Transfer Size (Bytes)</th>
<th>Pent100 Software Timer Rate (MB/S)</th>
<th>Pent100 Logic Analyzer Rate (MB/S)</th>
<th>Pent166 Software Timer Rate (MB/S)</th>
<th>Pent166 Logic Analyzer Rate (MB/S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>2.14</td>
<td>2.20</td>
<td>3.10</td>
<td>3.26</td>
</tr>
<tr>
<td>64</td>
<td>4.55</td>
<td>4.03</td>
<td>6.05</td>
<td>6.20</td>
</tr>
<tr>
<td>128</td>
<td>7.28</td>
<td>7.02</td>
<td>11.27</td>
<td>11.30</td>
</tr>
<tr>
<td>256</td>
<td>10.79</td>
<td>11.14</td>
<td>18.20</td>
<td>18.17</td>
</tr>
<tr>
<td>512</td>
<td>16.64</td>
<td>15.67</td>
<td>26.48</td>
<td>26.08</td>
</tr>
<tr>
<td>1,024</td>
<td>19.75</td>
<td>19.78</td>
<td>35.30</td>
<td>35.59</td>
</tr>
<tr>
<td>2,048</td>
<td>23.07</td>
<td>22.44</td>
<td>42.37</td>
<td>42.79</td>
</tr>
<tr>
<td>4,096</td>
<td>24.66</td>
<td>24.55</td>
<td>48.04</td>
<td>48.01</td>
</tr>
<tr>
<td>8,192</td>
<td>25.61</td>
<td>25.54</td>
<td>50.93</td>
<td>51.06</td>
</tr>
<tr>
<td>16,384</td>
<td>26.14</td>
<td>26.09</td>
<td>52.66</td>
<td>52.73</td>
</tr>
<tr>
<td>32,768</td>
<td>26.40</td>
<td>26.37</td>
<td>53.49</td>
<td>53.57</td>
</tr>
<tr>
<td>65,536</td>
<td>26.50</td>
<td>26.49</td>
<td>53.99</td>
<td>54.02</td>
</tr>
<tr>
<td>131,072</td>
<td>26.56</td>
<td>26.58</td>
<td>54.21</td>
<td>54.23</td>
</tr>
<tr>
<td>262,144</td>
<td>26.61</td>
<td>26.62</td>
<td>54.31</td>
<td>54.35</td>
</tr>
<tr>
<td>524,288</td>
<td>26.44</td>
<td>26.46</td>
<td>54.37</td>
<td>54.37</td>
</tr>
<tr>
<td>1,048,576</td>
<td>26.63</td>
<td>26.65</td>
<td>54.40</td>
<td>54.44</td>
</tr>
<tr>
<td>2,097,152</td>
<td>26.64</td>
<td>26.65</td>
<td>54.41</td>
<td>54.44</td>
</tr>
<tr>
<td>4,194,304</td>
<td>26.64</td>
<td>26.65</td>
<td>54.42</td>
<td>54.44</td>
</tr>
<tr>
<td>8,388,608</td>
<td>26.64</td>
<td>26.65</td>
<td>54.41</td>
<td>54.42</td>
</tr>
<tr>
<td>12,000,000</td>
<td>26.64</td>
<td>26.66</td>
<td>54.41</td>
<td>54.44</td>
</tr>
</tbody>
</table>

Again the Pentium166’s 82430FX Chip-set provides a significant performance benefit over the 82430NX Chip-set used in the Pentium100 system. Additionally there is very good correlation between the two measurement techniques implemented for both systems at all performance levels. It is also interesting to see the significant performance improvement realized by the Pentium 166 system. The write performance is greater than twice that of the Pentium 100 system and the read performance is greater than five times better. The following four traces captured on the Hewlett Packard Logic Analyzer demonstrate why this performance discrepancy was realized:
Diagram 4.1 shows that the Pentium 100 system used does not handle PCI reads very effectively. There is a delay of approximately 300ns before the NX Chip-set returns TRDY/active indicating that it is able to provide the requested data. Additionally a maximum of three data phases (and sometimes only one) take place for each assertion of the signal FRAME/ on the Pentium 100 System. The reason for this is that every transaction ends with the active low assertion of the STOP/ signal by the Host Chip-set. This indicates that the motherboard is no longer able to provide data for the current request in a timely fashion. The STOP/ signal is used to notify the Initiator (in this case the PCI I/O Board) that it should re-attempt the request later. It is essentially a negative acknowledge. Because of STOP/ being driven the PCI Board only has control of the PCI bus for about 560ns each time it is granted the bus. This trend continues for the duration of the transfer and is the main reason the transfer rate for PCI Reads in the Pentium 100 System are relatively low performance compared to the Pentium 166.
Diagram 4.2 shows much better performance characteristics than the previous example. In this case the Host system returns TRDY/ active low indicating it is ready to accept the data very early in the cycle. Additionally this signal is held low for the duration of the cycle until STOP/ is asserted. This illustrates the improved performance that is acquired through write-posting. The internal buffering of the NX Chip-set is able to accept four 32-bit values before it is filled and must wait for the data to actually be transferred to the DRAM subsystem. The FIFO can be filled by the high-speed PCI accesses thus efficiently using the bus. The data can then be written to DRAM at the speed of the Memory subsystem without impacting other devices which may want to utilize the PCI bus. In this case, for each sequence of four data phases, the Host is not slowing the transaction. The PCI Board, which must negate IRDY/ each time a data transfer occurs, is causing the FIFO filling process to be approximately twice as long as would be required by the Host. The current request, however, is stopped each time, by the Host Target, and must be re-attempted by the Initiator. The PCI Board only has control of the bus for approximately 420ns, but is able to transfer 16 bytes of data in that time.
Diagram 4.3 shows a marked improvement in performance over either of the traces acquired from the Pentium 100 system. The initial delay to TRDY/ being driven is approximately half of that seen in Diagram 4.1. Even more significant is the fact that STOP/ is not driven at all during these transfers. This transfer is terminated because the PCI I/O Board's Latency Timer has expired indicating that it should relinquish control of the bus for overall system performance reasons. In this example 33 data phases take place in a single transaction. The PCI Board is clearly the performance limiting device here. This indicates that in FX Chip-set and the associated DRAM subsystem are able to keep up with the PCI Board over relatively long spans of time. The PCI Board maintains control of the bus for approximately 2.28μS and transfers over 10 times as much data as it was able to in the Pentium 100 system. This is due to the fact that the FX Chip-set has the ability to pre-fetch read data if it anticipates that it may be needed for subsequent accesses by a PCI initiator. Therefore data is actually read from DRAM before it is needed in order to satisfy the timing requirements of high performance systems.
Diagram 4.4 shows the fastest transfer investigated. PCI write throughput was measured at a maximum of more than 54 Megabytes per Second. In this example 36 PCI write transfers occur during a single bus transaction. The reason that this performance is slightly higher than in the Read direction is the delay to the initial TRDY/ being asserted. Because this is a write, and no pre-fetching is required, the PCI Board does not have to wait for data from the Host. The write-posting buffers are filled immediately. Once again it is the PCI Board's Latency Timer that expires causing the cycle to be terminated.
The following two graphs demonstrate not only the close correlation between the two performance measurement techniques, but also magnify the relationship between transfer size and data throughput for each of the two systems analyzed:

Graph 4.1: Software Timer DMA Performance
Because of the close correlation between the two performance measurement techniques it is somewhat difficult to see any difference in the data if plotted according to system. Therefore, the data is provided in graphical format according to method of measurement as shown in the two graphs above. It is quite clear that as transfer size increases, performances increases until an asymptote is reached. This demonstrates the advantages of the increased burst count capability of the DMA Controller designed for this thesis. Many standard off-the-shelf DMA controllers are capable of transferring only 16 bytes per burst transaction. At that rate less than 10% of the capability of the 82430NX PCIset used in the Pentium 166 system is realized. Even without the ability to transfer data at the maximum theoretical bandwidth of the bus a significant performance increase was realized by the increased burst size capability of this DMA Controller.

4.2. Adherence to Goals and Problems Encountered

In general this design was extremely successful, and met the vast majority of design goals. The design is versatile, reliable, has a straightforward programming interface, and is very high in performance.
4.2.1. Goals Achieved

The two most significant goals were the ability to burst in an essentially unlimited fashion and the ability to transfer one 32-bit wide piece of data on every clock. The first of these two goals was met and the second was not. A 15.2ns timing margin error resulted in a transfer data reduction from the maximum PCI bus throughput rate of 132 Megabytes per second to 66 Megabytes per second. Even with this reduction in throughput the PCI I/O board performed commendably in both systems analyzed. Based upon the results acquired it would be preferable to sacrifice the transfer rate over the burst transfer size. The data indicates that there is a 5x increase in performance for larger size transfers. Doubling the transfer rate to the design goal would likely not have produced twice the performance for small transfers in the Pentium 100 system due to the fact that buffers would have filled faster. In the Pentium 166 system a doubling effect may have been achieved, but that would have only provided a performance of approximately 5 MBytes/Second for 32 byte transfers. An acceptable performance compromise might seem to be to provide a 16-bit counter, which would enable the transfer of 256Kbytes of data, if the ability to transfer at 132MBytes/Second were present. This however would provide limitations to the image size that could be transferred with a single programmed transfer, and may render the PCI I/O board unacceptable for certain applications.

The secondary goals of reliability, reusability and a robust programming interface were all accomplished. The design of the PCI I/O board was successfully partitioned to enable a designer to easily transition this design into a custom interface solution. The PARIO PLD shown in the appendix includes fourteen control signals which may be used to implement a robust solution. The programming interface for the PCI I/O board is extremely straightforward. By simply programming the source and destination registers, the transfer count register, and the command register a DMA transfer is initiated. Completion can be detected by either an interrupt or polling mechanism. Scatter-gather or chained DMA operations are also easily understood.

4.2.2. Transfer Rate Problem

Reducing the data throughput from one data transfer on every clock, to one data transfer every other clock, was necessary due to a timing problem found when the PCI I/O board was operating in input mode (Performing PCI Write Operations). This was due to the read data delay time of the FIFO. The fastest FIFO device which met the overall functional requirements that could be found at design time was a Texas Instruments SN74ACT2235-20. This 1024 x 9 x 2 device has a read timing delay of
25ns. This, coupled with a propagation delay of 7ns through the IDT74FCT16646CT Registered Transceivers which are connected to the PCI Bus created an overall delay in the data pipe of 38.5ns as shown in the following timing diagram:

A PCI Bus operating at 33MHz requires the worst-case data delay be limited to 23.3 ns from the rising edge of the clock. This conflict was only present when performing PCI Writes, and was not noticed until the board was manufactured and being tested. PCI Read operations could still be performed at the desired 132Mbytes per second. Because it was intended that this board be production-worthy, it was decided that it made more sense from a marketing standpoint to operate the board at the same rate in both directions. This meant operating at a maximum sustainable rate of 66Mbytes per second in both directions.

4.2.3. Transfer Count Problem
The on-board Transfer Counter/Register is 32-bits wide and can be used to count transfer’s up to 4Gigabytes in length. Whenever a transfer has been initiated and the Transfer Count Register has not expired, the PCI I/O Board is ready to perform a transfer, and will continue to do so until it is removed from the bus. There was a significant problem which occurred in
this area of the design. A series of four 8-bit high-speed counters are
daisy-chained to provide a full 32-bit count. In the original design only the
output of the final counter stage was sampled by the logic to determine if
the transfer count had expired. This configuration was intended to operate
as a synchronous counter using the 33Mhz system clock. This created a
problem due to the propagation delay of the counter devices themselves.
The maximum propagation delay from the Count Enable input to the
Terminal Count Output for the 74F779 device is 8ns. Four of these delay’s
is longer than the minimum period of the system clock. The problem that
resulted was that one extra transfer would take place depending upon the
number programmed into this Transfer Count Register. If the upper
register was utilized in the count size an extra data phase would occur.
The solution to this problem was to re-synchronize the output of the
second stage of the counter with the 33Mhz system clock. This guarantees
adequate setup and hold time for sampling of both the second-stage re-
synchronization, and the final Terminal Count output signal DMA DONE#. The one wrinkle that this hardware modification
introduced was that the DMA Controller operation was altered to perform
one additional transfer after the transfer count register expires. This can be
worked around in software by simply programming the transfer count
register with one less than the actual number of transfers desired. This is a
common technique used in counting type devices such as DMA Controllers
or synchronous counters.

4.3. Potential Improvements and follow-up work

Much was learned during the development and analysis of the PCI I/O board and
DMA controller. As with any project of this magnitude and complexity, upon
completion, areas of improvement can be clearly defined. It is my opinion that this
work could be improved upon given similar development constraints from a
performance standpoint. When this project was initiated there were no CPLD’s
which were fully compliant with the PCI Specification from both a drive capability,
and a capacitive loading standpoint. Rather than violate that specification the
decision was made to utilize buffers between the PCI bus and the on-board logic.
This decision ultimately lead to the requirement to turn IRDY# off for one clock
after every successful data transaction which reduced the maximum potential
throughput to 66Mbytes per second. Devices are now available which adhere to
the PCI specification and could be used to make this a simpler and more elegant
design from a bus interface standpoint. In addition, the usage of these compliant
components along with faster FIFO’s should lead to the ability to transfer data on
every clock instead of on every other clock, thus doubling the performance.

This analysis was performed by focusing on the PCI bus bandwidth. Every effort
was made to successfully transfer data across the PCI bus in an efficient manner.
It would also be very interesting to analyze the impact of peripheral device speed
on overall system performance. Rather than remove the peripheral device from the problem, it could be used to gather data on transfer size and FIFO size versus peripheral transfer rate and PCI Chip-set capabilities. This would be an involved and detailed investigation that should provide some interesting results.

Another interesting project would be to develop a device driver for this board for Windows NT™ or Windows 95™. Microsoft should be introducing their new Windows Driver Model or WDM in the very near future. Developing a driver for the PCI I/O Board using this new Model would be a worthwhile and rewarding effort.
5. Appendix

5.1 Appendix A - Glossary

5.2 Appendix B - Schematics

5.3 Appendix C - Programmable Logic Source
  5.3.1 PCICTRL PLD
  5.3.2 DMAC PLD
  5.3.3 DECODE PLD
  5.3.4 PARIO PLD

5.4 Bibliography
5.1. Appendix A - Glossary


ASIC - Application Specific Integrated Circuit.

Bandwidth - the amount of data that can be delivered over a specified medium in a specified time.

Burst - A data transfer mechanism that permits multiple contiguous pieces of data to be transferred without specifying the address information for each piece of data.

Bus - A collection of related signals which are logically described as a single unit.

Chaining Mode - A Mode of operation in which a DMA controller requires reduced CPU interaction. Once initiated the DMA controller can perform multiple DMA events using a linked-list or “chain” of control information.

CPLD - Complex Programmable Logic Device. A single chip which contains the logic of multiple common Programmable Logic Devices. These devices are typically very deterministic in propagation delay timings between input and output.

CPU - Central Processing Unit. Typically used interchangeably with the term Microprocessor.

Chipset - A group of standard chips which are typically used as a group to perform a function or set of functions. The core group of chips which are used in a PC-type system are referred to as a chipset.

DMA - Direct Memory Access. Direct access to Memory (and/or I/O) resources without CPU intervention. Typically a special device known as a DMA Controller is used to move large blocks of data between memory and/or I/O resources to reduce the workload of the microprocessor.

DRAM - Dynamic Random Access Memory. A relatively low cost volatile memory resource with a multiplexed address bus commonly used in
computers. This memory may be read from or written to in a random fashion.


**FIFO Memory** - First-In-First-Out Memory. This is a typically high speed memory resource which does not use unique addressing for its data elements. This memory is accessed in a sequential, not random, fashion. The first item written to the memory is the first item read.

**FPGA** - Field Programmable Gate Array. A device similar to a CPLD that is typically more flexible, but also less deterministic in its timing results.

**GUI** - Graphical User Interface. A user interface typically operated with a touch-screen or some sort of pointing device such as a mouse or pen. This contrasts with a console interface which requires the operator to enter commands with a keypad or keyboard.

**IEEE** Institute of Electrical and Electronics Engineers.

**Initiator** - In the PCI specification this term is used interchangeably with Master. An initiator is a device which gains control of the bus and begins or initiates a bus cycle.

**I/O** - Input/Output.

**ISA** - Industry Standard Architecture. The architecture developed by International Business Machines (IBM) and used in the first Personal Computers.

**master** - A device which resides on a bus and may gain control of the bus.

**Megabyte** - $2^{20}$ or 1,048,576 bytes.

**Multibus** - A bus specification developed by Intel Corporation in the 1970's commonly used for industrial applications.

**Multibus II** A bus specification developed by Intel Corporation in the early 1980's as a 32-bit more robust upgrade to Multibus.

**OEM** - Original Equipment Manufacturer. A company that manufactures any type of equipment under their own brand name and sells to an end customer in the retail marketplace.
PCI - Peripheral Component Interconnect. A high speed bus specification developed by a consortium of companies, spearheaded by Intel Corporation in an effort to increase the performance of the Personal Computer.

platform - The architectural basis of a computer system. For example an ISA platform is a computer based on the ISA specification.

S-Bus - A high speed synchronous bus developed by Sun Microsystems Inc.

Scatter-Gather - A technique often used in imaging applications which causes different data sets of potentially different sizes to be grouped separately both logically, and physically in memory. The different data sets are stored in a “scattered” arrangement, but then “gathered” together to render the final image. A DMA Controller must support chaining in order to effectively implement this feature.

Target - In a PCI system the target is the device being addressed by the Initiator or Master. Synonymous with Slave in a typical master-slave system.

VMEbus - A 32-bit bus standard spearheaded by Motorola in the late 70’s early 80’s meant to dethrone Multibus as the popular bus standard.
5.2. Appendix B - Schematics
5.3. Appendix C - Programmable Logic Source Code
5.3.1. PCICTRL PLD

module PCICTRL flag 'r3'
title 'PCI Interface Master/Slave Controller'
Sheet 1 of 1, Schematic Diagrams

Description:
This FPGA implements the following functionality on the PCI I/O Board:
- Generation of the PCI Bus Control and error Signals when in Master Mode.
- This is performed by the PCI Master State Machine.
- Generation of Slave Mode PCI Bus signals.
- Control of Arbitration PCI Master Mode Transfers.
- Control of PCI Bus Transceivers for Master and Slave accesses.
- Provides PCI Status Information to Configuration Region.
- Provides Parity Error Detection and Notification based upon Parity Register input values.

Output and Bi-directional Signal Descriptions:
- STATUS_STROBE_SYN(0): Synchronized PCI Status Strobe. This signal is a synchronized version of the STATUS_STROBE signal which is generated by the SDECODE PLD during a valid PCI Configuration Space Read decoded at the Status Register Address. This registered output is synchronized to the Buffered PCI Bus Clock BCLK.
- MPARER_RPT(0): Master Parity Error Reporting. This signal is driven to an active high state when the PCI I/O Interface Board is a PCI Master and the PCI Parity signal FERR# is asserted. This output is connected to the SDECODE PLD and remains active until a Status Register Read is performed to the PCI Configuration Region on this board by another PCI Master. This registered output is synchronized to the Buffered PCI Bus Clock BCLK.
- MTRG_ABORT(0): PCI Target Abort Received. This signal is asserted when the Interface Board is a PCI Master and a Target Abort is issued by the PCI Target being accessed. This output is connected to the SDECODE PLD and remains active until a Status Register Read is performed to the PCI Configuration Region on this board by another PCI Master. This registered output is synchronized to the Buffered PCI Bus Clock BCLK.
- STRG_ABORT(0): PCI Target Abort Issued. This signal is asserted when the Interface Board is a PCI Target, and this board issues a Target Abort. This output is connected to the SDECODE PLD and remains active until a Status Register Read is performed to the PCI Configuration Region on this board by another PCI Master. This registered output is synchronized to the Buffered PCI Bus Clock BCLK.
- MASTER_ABORT(0): PCI Master Abort Issued. This signal is generated when the Parallel Interface Board is a PCI Master, and the addressed Target does not respond by asserting DEVSEL# by Clock #6 of the PCI Cycle. This output is connected to the SDECODE PLD and remains active until a Status Register Read is performed to the PCI Configuration Region on this board by another PCI Master. This registered output is synchronized to the Buffered PCI Bus Clock BCLK.
- MAS5, MAS4, MAS3, MAS2, MAS1, MAS0(0): PCI Control State Machine State Control Pins. This State Machine Controls PCI Interfacing on the Parallel Interface Board. Refer to the PCI Controller State Machine State Diagram in this Document. This State Machine is clocked by the Buffered PCI Bus Clock BCLK.
- LDATA1, IDLE(0): Last Data Phase Detection State Machine Control Pins. This State Machine Performs Detection of the final Data Phase in a Burst Transaction. This Data Phase takes place after the PCI FRAME# signal has been deasserted. The signal LDATA1 is only active during the last DATA Phase of a given PCI Bus Transaction (i.e. FRAME# is high and IRDY# and TRDY# have not yet been sampled active). The signal IDLE_ is Low when the PCI Bus is Idle. Refer to the PCI Last Data Phase Detection State Machine in this document. This State Machine is clocked by the Buffered PCI Bus Clock BCLK.
- BFRAME, IBFRAME (I/O): Buffered PCI FRAME# Output Pin. This signal is asserted to initiate a Master (initiator) cycle on the PCI Bus by the
Interface Board. It is an input when the board is not a Master, and is driven as an output when the board is a Master. IFRAME is used to enable both pin and register feedback into the global macrocell. The same pin is indicated for both signal names.

- BIRDY_(I/O): Buffered PCI Initiator Ready#. This signal is asserted to indicate that the PCI bus transaction Initiator is Ready for either a Read or Write Cycle. This signal is driven as an output when the Interface Board is a PCI and is an input when a PCI Slave.

- BSTRDY_(I/O): Buffered PCI Target Ready#. This signal is asserted to indicate that the PCI bus transaction Target is Ready during either a Read or Write Cycle. This signal is driven as an output when the Interface Board is a PCI Target, and is an input when a PCI Master.

- BSTOP_(I/O): Buffered PCI Stop#. This signal is asserted by a PCI Target to indicate a request for the Initiator to stop the current transaction. This pin is driven high as an output when the Interface Board is a slave (a stop is never requested by the Interface Board). When received as a Master, the current transaction is stopped by the Interface Board.

- BPAR, BPARIN(I/O): Buffered PCI Parity. This signal indicates even parity across AD[31..0] and C/BE[3..0] for address and data phases of all PCI Transactions. This signal is driven valid one clock after the address phase by the Master, and one clock after IDRY# is asserted on a write transaction (by Initiator) or one clock after TRDY# is asserted on a read transaction (by Target). This signal is used in conjunction with the BAD and BC/BE signals to determine if a Parity Error has occurred. BPARIN is used to enable both registered and pin feedback.

- BREQ_(I/O): Buffered PCI Request#. This signal indicates to the PCI central resources arbiter that the Interface Board desires to use the bus. This is an output only to the Interface Board.

- BC_BE3_.. BC_BE0_(I/O): Buffered PCI Command/Byte Enable [3..0]. These multiplexed signals are driven by the PCI Initiator. During the address phase they contain cycle type information, during the data phase they are used as byte enables. The Interface board always drives these signals active low during the data phase, as all transactions are 32-bits. During the address phase they are driven based upon the DMA Cycle definition bits as programmed in the DMA Control Register.

- BDEVSEL(I/O): Buffered PCI Device Select#. This signal is driven active by the Interface Board when it is accessed by a PCI Master to indicate that it will be responding to the access. When a Master, this signal is received to indicate that a slave is responding to the access initiated by the Interface Board.

- LDEVSEL(O): Local Device Select#. This signal is generated on board to indicate to the on-board logic that the current PCI Bus cycle generated by another is meant for the Interface Board.

- BAD[7..0] (I): Buffered Address/Data Bit 7. The Buffered Address/Data bus bits are inputs to this device, used to load the Cache Line Size into the Cache Burst Counter. This counter is used on the Interface Board for Memory Write and Invalidate and Memory Read Line PCI Transaction Types.

- CLS_WRITE_ (I): CLEAR STROBE WRITE#. This signal is used to Clear the Interrupt Signal (INTA#) when it has been latched onto the PCI Bus.

- CBC[7..0] (N): Cache Burst Counter 7..0: These internal nodes are used to store the Cache Line Size Count for Memory Write and Invalidate and Memory Read Line PCI Transaction Types. The stored value is used as a starting point for the down count. When the down count reaches zero, the PCI transaction will be terminated.

- BC3U..BC0U & BC3L..BC0L(N): 8-bit Cache Burst Counter. This 8-bit counter is implemented as two 4-bit counters with underflow. When the low order counter (BC3L..BC0L) reaches zero the higher order counter will decrement. When the entire 8-bit count expires, the PCI transaction will terminate.

- LOWUFLOW (O): Low Count Underflow. This signal is driven active high when the low order nibble of the Cache Burst Counter reaches zero.

- BURST_DONE(O): Burst Done. This signal is driven active high when the 8-bit Cache Burst Counter expires indicating that the current PCI transaction should be completed with the current data phase.

- CMD_ENABLE_(O): Command Enable. This signal is used to control the direction of the PCI Bus Command Signal Transceiver. The PCI Command Signal Set is always driven by the current PCI transaction Initiator, so this signal is used to envelope the Initiator Transaction cycle when the PCI I/O board is a Master.

- ADDR_PHASE(O): Address Phase. This signal is driven active high for one BCLK Period during the PCI Bus address phase when the PCI I/O board is a Master.

- PCI_WRITE_(O): PCI Write Detect. This signal is driven active high to indicate that the
current PCI transaction is a PCI Write. When the PCI I/O board is a Master,
this signal is driven at the start of the Address Phase and held until the bus is
relinquished. When the PCI I/O board is a slave, this signal is driven one
BCLK Period following the Address Phase, and deasserted once the PCI Bus
returns to Idle.

DMA_ACK(O): DMA Cycle Acknowledge. This signal is driven active high to indicate
to the DMA Controller Logic that a successful Data transfer has taken place on the PCI Bus.
A successful DMA transfer is indicated by IRDY# and TRDY# being asserted at the same
time.

MASTER_(O): Master#. This signal is driven active low to indicate that the Parallel
Interface Board is currently the PCI Bus Master and that a PCI transaction is in Progress.
Master# is asserted at the beginning of the Address Phase and held active until the end of
the PCI transaction.

DMA_MEM_RD_(I): Memory Read Command. This signal indicates that the On-board DMA
Controller is generating a Memory-to-Memory transaction across the PCI Bus, and that
the direction is from System Memory to the PCI I/O board.

DMA_MEM_WR_(I): Memory Write Command. This signal indicates that the On-Board
DMA Controller is generating a Memory-to-Memory transaction across the PCI Bus, and that
the direction is from the PCI I/O board to System Memory.

INTA_(O): PCI Interrupt A#: This open drain output is used to generate an interrupt across
the PCI Bus. The two interrupt sources are the DMA Controller (end of chain, or terminal count)
or the Parallel Interface Interrupt Request.

PCI_ADDR_(O): PCI Address/Data Direction Control. This signal controls the direction of the
PCI Address/Data transceivers. When this signal is driven high the PCI I/O board is
driving the PCI Address/Data Bus. When this signal is driven low the PCI Bus is being driven onto
the PCI I/O board Bus.

PCI_AEN_(O): PCI Address/Data Bus Enable. This signal controls the output enable pins of the
PCI Address/Data Transceivers. When this signal is driven low the transceivers are in a high-
impedance state. This signal is part of the PCI Bus control State Machine.

ADDR_OE_(O): Address Output Enable. This signal Controls the Address Output Enable of the on-
board DMA Controller. During the

Declarations

*---------------------------------------------------------------------*
PCICTRL device 'lx780fp' ;" Intel FX780 PQ6Q Device

*---------------------------------------------------------------------*
Input Pins

BCLK pin 118 ; "33Mhz PCI Bus Clock
PCI_CLKIN pin 1 ; "33Mhz PCI Bus Clock for Equation use
LRESET pin 2 ; "Buffered Active Low PCI Reset
DMA_S3 pin 3 ; "DMA Cycle Definition Bit 3
DMA_S2 pin 4 ; "DMA Cycle Definition Bit 2
DMA_S1 pin 5 ; "DMA Cycle Definition Bit 1
DMA_S0 pin 33 ; "DMA Cycle Definition Bit 0
CHAIN pin 34 ; "DMA Controller Chaining Status
LTOVF pin 35 ; "Latency Timer Overflow Input
DMA_INTRQ pin 36 ; "DMA Controller Interrupt Request
PAR_INTRQ pin 37 ; "Parallel Interface Interrupt Request

BERR pin 54 ; "Buffered PCI Parity Error#
IBERR pin 54 ; "Buffered PCI Parity Error Input#
DMA_REQ pin 80 ; "Local DMA Start Request
DMA_SINGLE pin 45 ; "PCI Single Data Phase Request
DMA_DONE pin 30 ; "Local DMA Transfer Completed#
BGNT pin 43 ; "Buffered PCI Bus Grant#
DMA_DIR pin 30 ; "DMA Cycle Write/Read#
DMA_STAT_RD_ pin 67 ;"DMA Control/Status Register Read#
SLAVE_FIFO_RD_ pin 68 ; "FIFO Read Cycle from Another Master#
FIFO_GBA pin 69 ; "FIFO Read Enable Input
SYSERR_ENABLE pin 70 ;" SEVR_ Generation Enable

*---------------------------------------------------------------------*
DMA Request & Mode Register Inputs

*---------------------------------------------------------------------*
PCl Valid Command Register Enable Inputs from SDECODE

STATUS_STROBE pin 103 ; Status Read Strobe from Slave Decoder
STATUS_STROBE_SYN node 42 ; Status Read Strobe from Slave Decoder Synch'd to BCLK
MASTER_ENABLE pin 104 ; PCl Command Register Master Enable
MMCMV_ENABLE pin 64 ; PCl Command Register Memory Write and Invalidate En.
PARERR_ENABLE pin 40 ; PCl Command Register Parity Error Enable

PCI Error Reporting Logic

MPAR_RPT pin 57 ; PCl Bus Master Parity Error Report
MPAR_ABORT pin 89 ; PCl TI target abort received as master
MPAR_ABORT pin 90 ; PCl Target Abort Issues as Target
MPAR_ABORT pin 92 ; PCl Master Abort Received as Target
SEFF pin 55 ; PCl System Error (Open Drain)
LEFF pin 58 ; PCl System Error Issued as Master

Bi-Directional Pins

MASS pin 115 ; "DMA Master State Machine S5
MAST pin 123 ; "DMA Master State Machine S4
MASS pin 122 ; "DMA Master State Machine S3
MASS pin 124 ; "DMA Master State Machine S2
MAST pin 12 ; "DMA Master State Machine S1
MASS pin 120 ; "DMA Master State Machine S0
LDATA1 pin 111 ; "Last Data Detection State Machine S1
LDATA pin 114 ; "Last Data Detection State Machine S0

Outputs

COM_ADDR_ENABLE pin 108 ; "Control for Command Transceiver"
COM_ADDR pin 95 ; "P CI Address Phase Detected
COM_ADDR pin 127 ; "P CI Address/OMM Latch Control
COM_ADDR pin 75 ; "P CI Write Cycle Detects
DMA_TAK pin 109 ; "DMA Transfer Acknowledged
DMA_TAK pin 26 ; "P CI Master Status
DMA_TAK pin 112 ; "Local FRAME (Envelopes PCl Cycle)
DMA_TAK pin 63 ; "P CI Interrupt Pin A# (Open Drain)

P Cl_ADDR pin 88 ; "P CI Address/Data Direction Control
P Cl_ADDR pin 46 ; "P CI Address/Data Direction Control Delayed
P Cl_ADDR pin 121 ; "P CI Address/Data Bus Output Enable
P Cl_ADDR pin 126 ; "P CI Address/Data Bus Output Enable Delayed
STORF_WRITE pin 23 ; "P CI Address/Data Transceiver Register Input Storage Ctrl.
STORF_WRITE pin 94 ; "P CI Address/Data Transceiver Register Output Storage Ctrl.
STORF_WRITE pin 39 ; "P CI Data Output Storage Clock Strobe

ADDR pin 113 ; "Address Output Enable for DMA Controller

FIFO_RD_ENA node 29 ; "On-board FIFO Read Enable Control
IFIFO_RD pin 47 ; "On-board FIFO Read Strobe
DMA_DONE pin 30 ; "DMA DONE Internally Sync'd
CYCLE_START pin 105; "Master Cycle State Machine Start"
BRST_ pin 98; "Local Open Collector Buffered PCI RESET"

Pins Associated With Parity Generation and Error Detection

IPAR_ node 48, "Internal Parity Generation"
PANO_3 pin 29, "Parity Logic Output 3"
PANO_2 pin 48, "Parity Logic Output 2"
PANO_1 pin 78, "Parity Logic Output 1"
PANO_0 pin 41, "Parity Logic Output 0"
PARITY_CHECK pin 107, "Parity Checking Clock Cycle"

Set Definitions and Constants

H, L, X, C, Z, K 1, 0, X, .., Z, .., K;
SIMULATE 0;

PCI_ADDIR_D istype 'buffer, reg, pos';
PCI_ADDIR istype 'com';
STORE_DATA1 istype 'buffer, reg, pos';
CMD_ENABLE_ istype 'reg';
BRST_ istype 'reg';
MASTER istype 'reg';
ADDR OE istype 'buffer, reg, pos';
PCI ADEN istype 'reg';
PCI ADEN_D istype 'reg';
MAS_5 istype 'buffer, reg, pos';
MAS_4 istype 'buffer, reg, pos';
MAS_3 istype 'buffer, reg, pos';
MAS_2 istype 'buffer, reg, pos';
MAS_1 istype 'buffer, reg, pos';
MAS_0 istype 'buffer, reg, pos';
LDATA1 istype 'buffer, reg, pos';
IDLE_ istype 'buffer, reg, pos';
FIFO_RD_ENA istype 'buffer, reg, pos';

Intel Property Statements for Open Drain Outputs

Intel PROPERTY '@PIN_ATTRIB INTA_OPEN_DRAIN';
Intel PROPERTY '@PIN_ATTRIB SERR_OPEN_DRAIN';
Intel PROPERTY '@PIN_ATTRIB BRST_OPEN_DRAIN';

Definition of Burst Counter Register

CurBurstData [BAD3, BAD2, BAD1, BAD0];
CurCount [BC5L, BC2L, BC1L, BC0L];

PCI Transfer Command Definitions

IO_Read [BC_BE3 & BC_BE2 & BC_BE1 & BC_BE0];
IO_Write [BC_BE3 & BC_BE2 & BC_BE1 & BC_BE0];

PCI_Cmd [BC_BE3, BC_BE2, BC_BE1, BC_BE0];

DMA_Cycle [DMA_MODE2, DMA_MODE1, DMA_MODE0];

MemRD [0, 1, 1, 0];
MemRDMult [1, 1, 0, 0];
MemRDLine [1, 1, 1, 0];
IO_RD [0, 0, 1, 0];
IO_WR [0, 0, 1, 1];
Config_Read [1, 0, 1, 0];
Config_Write [1, 0, 1, 1];
State Machine State Definitions

PCI_Control

```
Idle

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
state MDATA:
  if IBIRDY_ & !DMA_DONE_
    then Last_Burst_Xfer
  else if !IBIRDY_ & IBRDY_ & !DMA_DONE_
    then Last_Burst_Xfer
  else if !DMA_REQ & IBIRDY_
    then Last_Burst_Xfer
  else if !BDEVSEL_ & LDATA1
    then Last_Burst_Xfer
  else if LTOVF & BGNT_ & IBIRDY_
    then Last_Burst_Xfer
  else if !BDEVSEL_ & !IBIRDY_
    then Last_Burst_Xfer
  else if !IBSTOP_ & !IBIRDY_
    then Last_Burst_Xfer
  else
    then Last_Burst_Xfer

state Last_Burst_Xfer:
  if !IBRDY_ & !IBIRDY_
    then Idle
  else if !IBSTOP_
    then Idle
  else if CMD_ENABLE_
    then Last_Burst_Xfer
  else
    then Last_Burst_Xfer

state WaitDevice0:
  if !BDEVSEL_ & !DMA_SINGLE
    then MDATA
  else if !BDEVSEL_ & !DMA_SINGLE
    then Last_Burst_Xfer
    WaitDevice1;
  else
    then Last_Burst_Xfer

state WaitDevice1:
  if !BDEVSEL_ & !DMA_SINGLE
    then MDATA
  else if !BDEVSEL_ & !DMA_SINGLE
    then Last_Burst_Xfer
    WaitDevice2;
  else
    then Last_Burst_Xfer

state WaitDevice2:
  if !BDEVSEL_ & !DMA_SINGLE
    then MDATA
  else if !BDEVSEL_ & !DMA_SINGLE
    then Last_Burst_Xfer
    WaitDevice3;
  else
    then Last_Burst_Xfer

state MABORT:
  if DMA_REQ
    then MABORT;
  else
    then MABORT;
    Idle;

state Slave_Cycle:
  if !IBIRDY_ & !IBRDY_ & !LDEVSEL_
    then Idle*Valid Data Phase
  else if !IBIRDY_ & !IBRDY_ & !BFRAME_
    then Idle*Master Abort
  else if !IBSTOP_ & !IBIRDY_
    then Idle*Target Disconnect
    Slave_Cycle;
  else
    then Idle

"----------------------------------------------------------------------
" Last Data Phase Detection State Machine
----------------------------------------------------------------------
state diagram LDATA_Detect
" NOTE: LDATA0 State and Signal Added to account for the situation
" when !IBIRDY_ and !IBRDY_ are asserted in the same clock
" that BFRAME_ is deasserted.
state DATA_Idle:
  if !BFRAME_ & !BRST_ & !IBSTOP_
    then Start_Trans
    DATA_Idle;
  else
    then DATA_Idle
state Start_Trans:
  if !IBRST_
    then Idle
  else if !IBFRAME_ & !IBSTOP_ & !IBIRDY_ & !IBRDY_
    then DATA_Idle
else if
  : IBFRAME_ & IBSTOP_
else if
  IBFRAME_ & !IBIRDY_ & IBTRDY_
else if
  !IBSTOP_
else if
  !ADDR_CLK
else
  start_TRANS

state Last_DATAO:
if !BRST_
else if
  !IBIRDY_ & !IBTRDY_
else if
  !IBSTOP_
else if "Cycle for other slave
  LDESEL_ & CMD_ENABLE_ & ADDR_CLK
else if
  !CMD_ENABLE_ & (PCI_Control==MABORT)
else
  DATA_Idle

Equations

BRST_.OE 1;
BRST_.CLF BCLK;

Parity Generation and Error Detection Logic

PARITY_CHECK.OE 1;
PARITY_CHECK.AR = !BRST_;;
PARITY_CHECK.CLK = BCLK;
PARITY_CHECKitize = !IBIRDY_ & !IBTRDY_;
IPAR_.OE   0;
IPAR_.CLK  := BCLK;
IPAR_ := PARO_3 $ PARO_2 $ PARO_1 $ PARO_0;
BPAR_.OE  := PCI_ADDR_D;
BPAR_.CLK  := BCLK;
BPAR_ := PARO_3 $ PARO_2 $ PARO_1 $ PARO_0;
BPERR_.OE  := 1;
BPERR_.CLK  := BCLK;
BPERR_.ASET := !BRST;
BPERR_ := (IPAR_ $ PARO_) & CMD_ENABLE & PARITY_CHECK & PCI_WRITE
            # (BPAR_ $ PARO_) & CMD_ENABLE & PARITY_CHECK & PCI_WRITE;
SERR_.OE  := 1;
SERR_.CLK  := BCLK;
SERR_.ASET := !BRST;j
SERR_ := SERR;
LSERR_.OE := !SERR_ & SYSERR_ENABLE;
LSERR_.CLK := BCLK;
LSERR_.ASET := !BRST;j
LSERR_ := LSERR & !STATUS_STROBE_SYN;
BC3L_.OE := 1;
BC3L_.CLK := BCLK;
BC3L_  := "CBC3.FB & ADDR_PHASE & (DMACycle $ MemRDLine)
            # CBC3.FB & ADDR_PHASE & (DMACycle == MemWRInv)
            # BC3L & BC2L & ADDR_PHASE & !CMD_ENABLE
            # BC3L & BC1L & ADDR_PHASE & !CMD_ENABLE
            # BC3L & !BC2L & !IBRDY & !CMD_ENABLE
            # BC3L & !BC1L & !IBRDY & !CMD_ENABLE
            # BC3L & !IBRDY & !ADDR_PHASE & !CMD_ENABLE
            # BC3L & CMD_ENABLE;
BC2L_.OE := 1;
BC2L_.CLK := BCLK;
BC2L_  := "CBC2.FB & ADDR_PHASE & (DMACycle == MemRDLine)
            # CBC2.FB & ADDR_PHASE & (DMACycle == MemWRInv)
            # BC2L & BC1L & ADDR_PHASE & !CMD_ENABLE
            # BC2L & BCOL & !ADDR_PHASE & !CMD_ENABLE
            # BC2L & BCTRL & IBRDY & !CMD_ENABLE
            # BC2L & !BC1L & !IBRDY & !CMD_ENABLE
            # BC2L & CMD_ENABLE;
BC1L_.OE := 1;
BC1L_.CLK := BCLK;
BC1L_  := "CBC1.FB & ADDR_PHASE & (DMACycle == MemRDLine)
            # CBC1.FB & ADDR_PHASE & (DMACycle == MemWRInv)
            # BC1L & BCOL & ADDR_PHASE & !CMD_ENABLE
            # BC1L & BCTRL & IBRDY & !CMD_ENABLE
            # BC1L & !BCOL & !IBRDY & !CMD_ENABLE
            # BC1L & CMD_ENABLE;
BCOL_.OE := 1;
BCOL_.CLK := BCLK;
BCOL_  := "CBC0.FB & ADDR_PHASE & (DMACycle == MemRDLine)
            # CBC0.FB & ADDR_PHASE & (DMACycle == MemWRInv)
            # BC0L & !IBRDY & !CMD_ENABLE
            # BC0L & !IBRDY & !ADDR_PHASE & !CMD_ENABLE
            # BCOL & CMD_ENABLE;
BURST_DONE_.OE := 1;
BURST_DONE_.CLK := BCLK;
BURST_DONE_ := !BC3L.FB & !BC2L.FB & !BC1L.FB & !BCOL.FB & DMA_S3 & DMA_S1
                          # !BC3L.FB & !BC2L.FB & !BC1L.FB & !BCOL.FB & DMA_S3 & DMA_S1
                          # !IBRDY & !IBRDY;
                          # !BURST_DONE & !CMD_ENABLE;
PCI_WRITE_.OE := 1;
PCI_WRITE_.CLK := BCLK;
PCI_WRITE_ := !CMD_ENABLE & (DMACycle == MemWR) & !CHAIN & Master Write
              # CMD_ENABLE & !IBRDY & IO_Write "Slave IO Write
              # CMD_ENABLE & !IBRDY & (PCI_Cmd==Config_Write)
              # CMD_ENABLE & IDLE & PCI_WRITE; "Hold until Idle

-------------------------------
PCI Bus Signal Control Logic
-------------------------------
IBSTOP_ OE = CMD_ENABLE_; 
IBSTOP_ 1;

DMA_SINGLE  = DMA_BREQUEST_ & DMA_REQUEST_
# DMA_MODE2 & DMA_MODE1 & DMA_MODE0  "Fly By Count Down
# DMA_MODE2 & DMA_MODE1 & DMA_MODE0;  "Chain Count Down

!IBTRDY_IN  = IFRAME_ & IDADEVSEL_ & IDLE_ & LDATA1 & IBSTOP_ & !IBTRDY_D & !IBTRDY_DD &

!IBTRDY_D = LFRAME_ & IDLE_ & LDATA1 & IBSTOP_; 
IBTRDY_DD = 0;
IBTRDY_D.CLK = BCLK;
IBTRDY_D.ASET = 1BRS;

!BREQ_BOOT = DMA_REQ & IBSTOP_ & ILTOVF_ & DMA_MODE;  "BREQ Must be deasserted for 2 clocks
# DMA_REQ & IBSTOP_ & ILTOVF_ & DMA_DONE_ & DMA_MODE;
# DMA_REQ & IBSTOP_ & IBGNT_ & DMA_MODE;
# DMA_REQ & IBSTOP_ & IBGNT_ & DMA_DONE_ & DMA_MODE;

IBIRDY_CL = BCLK;
IBIRDY_OE = CMD_ENABLE_; 

!IBIRDY_ = IBIRDY_ & !CMD_ENABLE_ & !IBFREQ_  "Master Writes Wait for TPDY for FIFOCS Strobe
# IBIRDY_ & !CMD_ENABLE_ & IDADEVSEL_ & IBFRAME_ & IBSTOP_; 
# !IBIRDY_ & !CMD_ENABLE_ & IBFREQ_ & !IBFREQ_ & IBSTOP_; 
# IBIRDY_ & !CMD_ENABLE_ & !IBFREQ_ & !IBFREQ_ & IBSTOP_; 
# BREQ_ & LDATA Detect = Last DATA0 & IBTRDY_ & !CMD_ENABLE_; 

!MASTER_ = 1;
!MASTER_CL = BCLK;
!MASTER_ASE = 1BRS;

!MASTER_ = !CMD_ENABLE_ & IBIRDY_ & LFRAME_ & MASTER_  "Turn on 1 clock after CMD
# !MASTER_ & IDLE_ "Maintain Until End of Transaction
# !MASTER_ & !IBIRDY_ "Maintain Until End of Transaction
# !MASTER_ & !CMD_ENABLE_ & !IBFREQ_ & "Turn off after FRAME

!BC_BE3_ = !CMD_ENABLE_;
!BC_BE3_ = !CMD_ENABLE_ & IDLE_ "Active for Data Transfers
# !CMD_ENABLE_ & IDLE_ & (DMACycle == CRmmRD); "Active for Mem Read
# !CMD_ENABLE_ & IDLE_ & (DMACycle == CRmmWR) "Active for Mem Write
# !CMD_ENABLE_ & IDLE_ & (DMACycle == CRmmWRInv); 
# "Active for Mem Write and Invalidate if Enabled

!BC_BE2_ = !CMD_ENABLE_;
!BC_BE2_ = !CMD_ENABLE_ & IDLE_ "Only Active for Data Transfers

!BC_BE1_ = !CMD_ENABLE_;
!BC_BE1_ = !CMD_ENABLE_ & IDLE_ & (DMACycle == MemRDMult); "Active for Mem Read Mult
# !CMD_ENABLE_ & IDLE_;  "Active for Data Transfers

!BC_BE0_ = !CMD_ENABLE_;
!BC_BE0_ = !CMD_ENABLE_ & IDLE_ & (DMACycle == MemRD);
# !CMD_ENABLE_ & IDLE_ & (DMACycle == MemRDMult);
# !CMD_ENABLE_ & IDLE_ & (DMACycle == MemRDLine);  "Active for All Mem Read CMDs
# CHAIN  "Active in Chaining Mode
# !CMD_ENABLE_ & IDLE_;  "Active for Data Transfers

!LFRAME_ = 1;
!LFRAME_ASE = 1BRS;
!LFRAME_CL = BCLK;
!LFRAME_ = !IBFRAME;
!LFRAME_ F & IDLE_  "Hold until PCI Bus Idle
"The CMD_ENABLE signal signifies that we have been granted the
"PCI Bus and will begin an initiator cycle. This signal
"envelopes the entire initiator cycle.

CMD_ENABLE_OE = 1;
CMD_ENABLE_CLK = BCLK;
CMD_ENABLE_ASET = IBRST;

"CMD_ENABLE_ := CMD_ENABLE_ & CYCLE_START_ & !BGNT
" # CMD_ENABLE_ & DMA_REQ_ & (PCI_Control==Drive_Address)
" # CMD_ENABLE_ & IBFRAME_
" # CMD_ENABLE_ & IBFRAME_ & (LDATA_Detect != DATA_Idle)
" # CMD_ENABLE_ & IBFRAME_ & (LDATA_Detect == Last_DATA0)
" & IBFDIR_ & IBFDIR_;

CYCLE_START_OE = 1;
CYCLE_START_CLK = BCLK;
CYCLE_START := (PCI_Control==Idle) & !BGNT_ & DMA_REQ_ & MASTER_ENABLE_ & IBFRAME_ & !IDLE_ &
LFRAME_ & !LDEVSEL_;

!PCI_ADDR_ := 1;
!PCI_ADDR_CLK = BCLK;

STORE_DATAI_ := DMA_REQ_ & MASTER_ & !IDLE_ & !PCI_WRITE_ & "Slave Reads
# !MASTER_ & STORE_DATA_ & !PCI_WRITE & "Hold till last cycle
" # MASTER_ & ADDR_CLK
# !MASTER_ & STORE_DATA_ & !PCI_WRITE & "Slave Cycles
" # MASTER_ & !LFRAME_ & PCI_WRITE;

STORE_DATAO_ := 1;
STORE_DATAO_CLK = BCLK;
STORE_DATAO := !CMD_ENABLE_ & ADDR_OE;

STORE_CLKO_ := 1;
STORE_CLKO_CLK = BCLK;
STORE_CLKO := ADDR_OE;

DMA_ACK_OE = 1;
DMA_ACK_CLK = BCLK;
DMA_ACK := !CMD_ENABLE_ & !IBFDIR_ & !IBFDIR_;

ADDR_PHASE_OE = 1;
ADDR_PHASE_CLK = BCLK;
ADDR_PHASE := !CMD_ENABLE_ & MASTER_
" # CMD_ENABLE_ & MASTER_ & !IBFRAME_ & LFRAME_;

ADDR_CLKO_ := 1;
ADDR_CLKO_CLK = BCLK;
ADDR_CLKO := !LFRAME_;

"---------------------------------------------------------------------------------------------------------------------
" "---------------------------------------------------------------------------------------------------------------------

STATUS_STROBE_SYN_OE = 1;
STATUS_STROBE_SYN_CLK = BCLK;
STATUS_STROBE_SYN := STATUS_STROBE;

MPERR_RPT_OE = 1;
MPERR_RPT_CLK = BCLK;
MPERR_RPT := !MASTER_ & !BPERR_ & PARERR_ENABLE

62
\# MTRER RET \& .STATUS_STROBE_SYN;
MTRER RET = RST;

MTRG ABCR: DE = 1;
MTRG ABCR: LD = CLKN;
MTRG ABCR: = (MTRG ENABLE \& ._IRQ \& G1E\_SSEL1)
\# MTRG ABCR \& .STATUS_STROBE_SYN;
MTRG ABCR:AR = RST;

STRG ABCR:GE = 1;
STRG ABCR:GN = GTK;
STRG ABCR: = (MTRG ENABLE \& ._IRQ \& .STATUS)
\# STRG ABCR \& .STATUS_STROBE_SYN;
STRG ABCR:AR = RST;

MASTER ABCR:GE = 1;
MASTER ABCR:LN = CLKN;
MASTER ABCR: = E2\_SSEL1 \& (PCI Controller \& RealDevices)
\# MASTER ABCR \& .STATUS_STROBE_SYN;
MASTER ABCR:AR = RST;

ASPER:WE = 1;
ASPER:LN = CLKN;
ASPER: = (MTRG ENABLE \& RTK)
\# ASPER \& .STATUS_STROBE_SYN;
ASPER:AR = RST;

and PCI2278.
5.3.2. DMAC PLD

module DMAC

title 'DMAC PCI Master Controller Logic for the PCI Interface Board'

Sheet 2 of 10, Schematic Diagrams

This FPGA implements the following functionality on the PCI
PCI I/O board:

Initiates Direct Memory Access Cycles as a PCI Master when required by the
Parallel Interface.

Acts as the Address Generation Control Logic for all Master PCI Cycles.
This includes the register functionality for receiving the initial DMA
address from another PCI Master, properly incrementing address with
each successful DMA Cycle, and Driving Address information during the
Address Phase of each PCI Master Cycle.

Performs Address Generation and I/O Selection for 32-bit aligned
fly-by or single cycle DMA Transfers between system memory and the
on-board FIFO memory buffer via the DMA State Machine.

Performs Address Generation for dual cycle memory-to-memory DMA
Transfers between two system memory locations.

DMA Control Register Inputs

The least significant 16-bits of the DMA Control Register are inputs
to this FPGA. They are READ/WRITE accessible by a PCI Master.
Currently Bits 15-10 are undefined and therefore not described here.

Bits 9 and 8 determine the Interrupt Mode Selected as follows:

<table>
<thead>
<tr>
<th>Mode 1(9)</th>
<th>Mode 0(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>No Interrupt Generated</td>
</tr>
<tr>
<td>0 1</td>
<td>Interrupt on Terminal Count</td>
</tr>
<tr>
<td>1 0</td>
<td>Interrupt Only on END OF Last Chain</td>
</tr>
<tr>
<td>1 1</td>
<td>No Interrupt Generated.</td>
</tr>
</tbody>
</table>

Bits 7, 6, and 5 determine the DMA Transfer Mode Selected as follows:

<table>
<thead>
<tr>
<th>Mode 2(7)</th>
<th>Mode 1(6)</th>
<th>Mode 0(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>Fly-by Count Up in Normal Mode</td>
<td></td>
</tr>
<tr>
<td>0 0 1</td>
<td>Fly-by Count Down in Normal Mode</td>
<td></td>
</tr>
<tr>
<td>0 1 0</td>
<td>Fly-by Count Up in Chaining Mode</td>
<td></td>
</tr>
<tr>
<td>0 1 1</td>
<td>Fly-by Count Down in Chaining Mode</td>
<td></td>
</tr>
<tr>
<td>1 0 0</td>
<td>Mem-mem Count Up in Normal Mode</td>
<td></td>
</tr>
<tr>
<td>1 0 1</td>
<td>Mem-mem Count Down in Normal Mode</td>
<td></td>
</tr>
<tr>
<td>1 1 0</td>
<td>Invalid Combination</td>
<td></td>
</tr>
<tr>
<td>1 1 1</td>
<td>Invalid Combination</td>
<td></td>
</tr>
</tbody>
</table>

Bits 4, 3, 2, and 1 Describe the PCI Cycle Type as follows:

<table>
<thead>
<tr>
<th>S3 S2 S1 S0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 1 0</td>
</tr>
<tr>
<td>0 1 1 1</td>
</tr>
<tr>
<td>1 1 0 0</td>
</tr>
<tr>
<td>1 1 1 0</td>
</tr>
<tr>
<td>1 1 1 1</td>
</tr>
</tbody>
</table>

Note that these 4 Mode bits not only determine the Cycle type, but they
determine the direction of the transfer. All transfer directions are
with respect to the PCI Bus. For example a Memory Read indicates that
a READ will take place across the PCI Bus, and a write will take place
to the local resource. Likewise, a Memory Write & Invalidate indicates
that the data transfer direction will be from the board to a system
memory resource on the PCI Bus.
Rochester Institute of Technology Computer Science Department

---

**Declarations**

**DMA device 'ix780fp';**

---

**Inputs Pins**

- **BCLK** pin 118; *33MHz PCI Bus Clock*
- **BRST** pin 1; *Buffered Active Low PCI Reset*
- **ADDR_PHASE** pin 2; *PCI Address Phase Detect*
- **DATA_PHASE** pin 3; *PCI Data Phase Detect*
- **MASTER** pin 4; *PCI Bus Master Status*
- **DMA_SOURCE_WR** pin 5; *DMA Source Address Write Strobe*
- **DMA_SOURCE_RD** pin 33; *DMA Source Address Read Strobe*
- **DMA_MASTER_ADDR** pin 42; *DMA Status Read Strobe*
- **ADDR OE** pin 34; *PCI Master Address Output Enable*
- **DMA_GRANT** pin 35; *DMA Transfer Grant*
- **DMA_ACK** pin 36; *DMA Transfer Acknowledging*
- **DMA_STS** pin 37; *DMA Start Command Signal*
- **DMA_REQ** pin 38; *DMA Burst Request*
- **DMA_REQUEST** pin 67; *DMA Single Request*
- **DMA_DONE** pin 68; *DMA Done From Longword Transfer Counter*
- **CHAIN_OVF0** pin 111; *Chaining Register Overflow*
- **CHAIN_OVF1** node 28; *Chaining Register Overflow 0*
- **CHAIN_OVF2** node 48; *Chaining Register Overflow 1*
- **CHAIN_OVF3** node 68; *Chaining Register Overflow 2*
- **CHAIN_OVF4** node 88; *Chaining Register Overflow 3*
- **DMA_COUNT_WR** pin 70; *DMA Count Write Input*
- **DMA_CHAIN_RD** pin 71; *DMA Chaining Register Read Strobe*
- **DMA_CHAIN_WR** pin 109; *DMA Chaining Register Write Strobe*
- **TC_STAGE1** pin 114; *Terminal Count Stage 1 Out*
- **CLS_WRITE** pin 123; *T/T Clear Strobe*

---

**DMA Control Register Inputs**

- **INT_MODE1** pin 99; *DMA Control Register Bit 9 (Interrupt Mode 1)*
- **INT_MODE0** pin 100; *DMA Control Register Bit 8 (Interrupt Mode 0)*
- **DMA_MODE2** pin 101; *DMA Control Register Bit 7 (DMA Mode 2)*
- **DMA_MODE1** pin 102; *DMA Control Register Bit 6 (DMA Mode 1)*
- **DMA_MODE0** pin 103; *DMA Control Register Bit 5 (DMA Mode 0)*
- **DMA_S0** pin 62; *T/H Control Register Bit 1 (Cycle Type S0)*

---

**Bi-Directional Pins**

- **LODXCTRL** pin 51; *DMA Master State Machine Load Control Word*
- **CHAIN** pin 122; *DMA Master State Machine 0*
- **S2** node 128; *DMA Master State Machine S2*
- **S1** node 121; *DMA Master State Machine S1*
- **S0** node 127; *DMA Master State Machine S0*
- **DMA_INTRQ** pin 129; *DMA Controller Interrupt Request*
- **PCICLKIN_D** node 65; *PCI Input Clock used for Equations*
- **PCICLKIN** pin 57; *PCI Input Clock used for Equations*
- **BADDR31** pin 22; *Buffered Address Value 31*
- **BADDR30** pin 23; *Buffered Address Value 30*
- **BADDR29** pin 24; *Buffered Address Value 29*
- **BADDR28** pin 25; *Buffered Address Value 28*
- **BADDR27** pin 6; *Buffered Address Value 27*
- **BADDR26** pin 98; *Buffered Address Value 26*
- **BADDR25** pin 29; *Buffered Address Value 25*
- **BADDR24** pin 30; *Buffered Address Value 24*
- **BADDR23** pin 39; *Buffered Address Value 23*
- **BADDR22** pin 40; *Buffered Address Value 22*
- **BADDR21** pin 41; *Buffered Address Value 21*
- **BADDR20** pin 47; *Buffered Address Value 20*
- **BADDR19** pin 43; *Buffered Address Value 19*
- **BADDR18** pin 45; *Buffered Address Value 18*
- **BADDR17** pin 7; *Buffered Address Value 17*
- **BADDR16** pin 47; *Buffered Address Value 16*
- **BADDR15** pin 54; *Buffered Address Value 15*
- **BADDR14** pin 55; *Buffered Address Value 14*
- **BADDR13** pin 56; *Buffered Address Value 13*
- **BADDR12** pin 8; *Buffered Address Value 12*
- **BADDR11** pin 58; *Buffered Address Value 11*
- **BADDR10** pin 60; *Buffered Address Value 10*
BADDR9  pin  61 ; "Buffered Address Value 9"
BADDR8  pin  96 ; "Buffered Address Value 8"
BADDR7  pin  72 ; "Buffered Address Value 7"
BADDR6  pin  73 ; "Buffered Address Value 6"
BADDR5  pin  74 ; "Buffered Address Value 5"
BADDR4  pin  75 ; "Buffered Address Value 4"
BADDR3  pin  76 ; "Buffered Address Value 3"
BADDR2  pin  9 ; "Buffered Address Value 2"
BADDR1  pin  79 ; "Buffered Address Value 1"
BADDR0  pin  80 ; "Buffered Address Value 0"
BAD31  pin  22 ; "Buffered Address/Data Bit 31"
BAD30  pin  23 ; "Buffered Address/Data Bit 30"
BAD29  pin  24 ; "Buffered Address/Data Bit 29"
BAD28  pin  25 ; "Buffered Address/Data Bit 28"
BAD27  pin  6 ; "Buffered Address/Data Bit 27"
BAD26  pin  98 ; "Buffered Address/Data Bit 26"
BAD25  pin  29 ; "Buffered Address/Data Bit 25"
BAD24  pin  30 ; "Buffered Address/Data Bit 24"
BAD23  pin  39 ; "Buffered Address/Data Bit 23"
BAD22  pin  40 ; "Buffered Address/Data Bit 22"
BAD21  pin  41 ; "Buffered Address/Data Bit 21"
BAD20  pin  97 ; "Buffered Address/Data Bit 20"
BAD19  pin  43 ; "Buffered Address/Data Bit 19"
BAD18  pin  45 ; "Buffered Address/Data Bit 18"
BAD17  pin  7 ; "Buffered Address/Data Bit 17"
BAD16  pin  47 ; "Buffered Address/Data Bit 16"
BAD15  pin  54 ; "Buffered Address/Data Bit 15"
BAD14  pin  55 ; "Buffered Address/Data Bit 14"
BAD13  pin  56 ; "Buffered Address/Data Bit 13"
BAD12  pin  8 ; "Buffered Address/Data Bit 12"
BAD11  pin  58 ; "Buffered Address/Data Bit 11"
BAD10  pin  60 ; "Buffered Address/Data Bit 10"
BAD9   pin  61 ; "Buffered Address/Data Bit 9"
BAD8   pin  96 ; "Buffered Address/Data Bit 8"
BAD7   pin  72 ; "Buffered Address/Data Bit 7"
BAD6   pin  73 ; "Buffered Address/Data Bit 6"
BAD5   pin  74 ; "Buffered Address/Data Bit 5"
BAD4   pin  75 ; "Buffered Address/Data Bit 4"
BAD3   pin  76 ; "Buffered Address/Data Bit 3"
BAD2   pin  9 ; "Buffered Address/Data Bit 2"
CH_AD7  node  31; "Chain Address Register Bit 7"
CH_AD6  node  49; "Chain Address Register Bit 6"
CH_AD5  node  26; "Chain Address Register Bit 5"
CH_AD4  node  46; "Chain Address Register Bit 4"
CH_AD3  node  42; "Chain Address Register Bit 3"
CH_AD2  node  32; "Chain Address Register Bit 2"
CARRY_ pin  78; "Address Counter Overflow#"
ADR7   node  82; "Address Counter Bit 7"
ADR6   node  135; "Address Counter Bit 6"
ADR5   node  114; "Address Counter Bit 5"
ADR4   node  107; "Address Counter Bit 4"
ADR3   node  16; "Address Counter Bit 3"
ADR2   node  62; "Address Counter Bit 2"

# Outputs
#---------------------------------------------------------------------
DMA_REQ  pin  120; "DMA Cycle Request#"
DMA_COUNT_LOAD_ pin  130; "DMA Transfer Count Load#"
DMA_COUNT_ENA_ pin  106; "DMA Transfer Count Enable#"
DMA_DONE_S_ pin  108; "DMA Transfer Count Done Sync#"
DMA_CHAIN_OE_ pin  121; "DMA Chain Output Enable#"
DMA_ADDR_ENA_ pin  94; "DMA Address Counter Enable#"
DMA_ADDR_CLK_ pin  95; "DMA Address Counter Clock"
DMA_ADDR_LOAD_ pin  126; "DMA Address Load Strobe#"
DMA_ADDR_OE_ pin  92; "DMA Address Output Enable#"
DMA_DIR  pin  105; "DMA Direction Status Output"
DONE_RST_ node ; "Async Reset Term For DMA State Machine"
TC_STAGE1_S_ pin  13; "Terminal Count Stage 1 Synchronized"

# Set Definitions and Constants
#---------------------------------------------------------------------
H,L,X,C,K,Z,SIMULATE = 1,0,X,...,C,...,Z,0;
LOADCTR1 istype 'buffer,reg,pcs';
CHAIN istype 'buffer,reg,pcs';
SI istype 'buffer,reg,pcs';
S1 istype 'buffer,reg,pcs';
S0 istype 'buffer,reg,pos';
DMA_REQ istype 'buffer,reg,pcs';

BADDR2 istype 'buffer,reg,pcs';
BADDR3 istype 'buffer,reg,pcs';
BADDR15 istype 'buffer,reg,pcs';
BADDR25 istype 'buffer,reg,pcs';


CHAIN6.Q,CHAIN5.Q,CHAIN4.2,CHAIN3.2,CHAIN2.2];

Cur_Addr_B6_B3 = [CHAIN7.Q,CHAIN6.Q,CHAIN5.Q,CHAIN4.2,CHAIN3.2,CHAIN2.2];

Addr_B0 = [BAD7,BAD6,BAD5,BAD4,BAD3,BAD2];
Rochester Institute of Technology Computer Science Department

Addr_In  [BAD31, BAD30, BAD29, BAD28, BAD27, BAD26, BAD25, BAD24, BAD23, BAD22, BAD21, BAD20, BAD19, BAD18, BAD17, BAD16, BAD15, BAD14, BAD13, BAD12, BAD11, BAD10, BAD9, BAD8, BAD7, BAD6, BAD5, BAD4, BAD3, BAD2];

Addr_In_All [BAD31, BAD30, BAD29, BAD28, BAD27, BAD26, BAD25, BAD24, BAD23, BAD22, BAD21, BAD20, BAD19, BAD18, BAD17, BAD16, BAD15, BAD14, BAD13, BAD12, BAD11, BAD10, BAD9, BAD8, BAD7, BAD6, BAD5, BAD4, BAD3, BAD2, BAD1, BAD0];

Addr_Out [BADR31, BADR30, BADR29, BADR28, BADR27, BADR26, BADR25, BADR24, BADR23, BADR22, BADR21, BADR20, BADR19, BADR18, BADR17, BADR16, BADR15, BADR14, BADR13, BADR12, BADR11, BADR10, BADR9, BADR8, BADR7, BADR6, BADR5, BADR4, BADR3, BADR2, BADR1, BADR0];

Addr_Out_All [BADR31, BADR30, BADR29, BADR28, BADR27, BADR26, BADR25, BADR24, BADR23, BADR22, BADR21, BADR20, BADR19, BADR18, BADR17, BADR16, BADR15, BADR14, BADR13, BADR12, BADR11, BADR10, BADR9, BADR8, BADR7, BADR6, BADR5, BADR4, BADR3, BADR2, BADR1, BADR0];

Addr_Out_B0 [BADR7, BADR6, BADR5, BADR4, BADR3, BADR2, BADR1, BADR0];

DMA_Mode [DMA_MODE1];
FlyBy_Up_Normal [0];
FlyBy_Up_Chain [1];

Chain Mode DMA_MODE1;
Normal_Mode !DMA_MODE1;
Count_Up 1;

"---------------------------------------------------
" DMA State Machine State Definition
"---------------------------------------------------

DMA_States [LOADCTRL,CHAIN,S2,S1,S0,DMA_REQ];
Idle "b000000;
NormalTransfer "b000001;

DMA_Idle "b000010;
ControlWord "b110001;
TransferCount "b010001;
NextChain "b011001;
StartAddress "b010101;
ChainComplete = "b010100;

."---------------------------------------------------
" DMA State Machine State Diagram
"---------------------------------------------------
state_diagram DMA_States

state Idle: "fly-by, Normal Mode, Count Up
if DMA_START & DMA_REQUEST & DMA_DONE_S & (DMA_Mode == FlyBy_Up_Normal)
then NormalTransfer
else if "fly-by Chaining Mode, Count Up
DMA_START & CHAIN_OVF & (DMA_Mode == FlyBy_Up_Chain) & MASTER_
then ControlWord
else if DMA_START & DMA_DONE_S & (DMA_Mode == FlyBy_Up_Chain) & MASTER_
then ControlWord
else if "non-Burst Request
DMA_START & DMA_REQUEST & DMA_BREQUEST & DMA_DONE_S & MASTER_ "Wait Until Bus is Released
then NormalTransfer
else
Idle;

state NormalTransfer: if DMA_START "if User aborts
then Idle
else if DMA_START & DMA_DONE_S_
then Idle
else if DMA_BREQUEST & DMA_ACK "if No Request
then DMA_Idle
else NormalTransfer:

state DMA_Idle:
  if DMA_START = DMA_DONE then idle
  else if DMA_REQUEST & MASTER = DMA_BREQUEST & MASTER then NormalTransfer
  else DMA_Idle;

state ControlWord:
  if DMA_ACK then TransferCount
  else ControlWord:

state TransferCount:
  if DMA_ACK then NextChain
  else TransferCount;

state StartAddress:
  if DMA_ACK then ChainComplete
  else StartAddress;

state NextChain:
  if DMA_ACK then StartAddress
  else NextChain;

state ChainComplete:
  if MASTER & DMA_BREQUEST then NormalTransfer "Stay until MASTER so don't
  else if MASTER & DMA_REQUEST then NormalTransfer
  else ChainComplete;

"------------------------------------------------------------------------
* Equations
"------------------------------------------------------------------------
equations

"----------------------------------------------------------------------
* DMA State Machine Clock and Reset Definitions
"----------------------------------------------------------------------
  LOADCTRL.CLk = BCLK;
  LOADCTRL.AE = 1;
  DMA_REQ.CLk = BCLK;
  DMA_REQ.AE = 1;
  S0.CLk = BCLK;
  S0.AE = 1;
  S1.CLk = BCLK;
  S1.AE = 1;
  S2.CLk = BCLK;
  S2.AE = 1;
  CHAIN.CLk = BCLK;
  CHAIN.AP = 1;

"----------------------------------------------------------------------
* Address Generation Counter Logic
"----------------------------------------------------------------------
[BADDR31..BADDR8].CLk = BCLK;
[CH_ADDR7..CH_ADDR2].CLk = BCLK;
[ADDR7..ADDR2].AP = SIMULATE;
[BADDR7..BADDR2].AE = SIMULATE;
BADDR5 = DMAM_WRITE8 & DMA_CHAIN_PD & MASTER;
BADDR1 = DMAM_WRITE8 & DMA_CHAIN_PD;

[BADDR31..BADDR8].AE = 1;
[BADDR7..BADDR0].OE = ADDR.OE = 1;
[BADDR7..BADDR2] = ChainAddr8 & LOADCTRL & DMA_CHAIN_PD
  & ChainAddr8 & LOADCTRL & DMA_CHAIN_PD;
CARRY_CLK = BCLK;
CARRY_OE = 1;
!CARRY_ := CARRY_ & (Cur_Addr_BO == ^h3F) & !DMA_ADDR_ENA;

Cur_Addr_Next := Cur_Addr & MASTER & !S2
# Cur_Addr & MASTER & !DMA_REQ
# Addr_In & MASTER & !DMA_CHAIN_WR
# Addr_In & MASTER & S2 & DMA_REQ "Next Chain State
# Cur_Addr & MASTER & DMA_CHAIN_WR;

Cur_Addr_Next_BO := (Cur_Addr_BO +1) & DMA_ADDR_LOAD & !DMA_ADDR_ENA & !DMA_ACK
# Cur_Addr_BO & DMA_ADDR_LOAD & !DMA_ADDR_ENA & !DMA_ACK
# Addr_In_BO & MASTER & S2 & DMA_REQ "Loading Next Chain Pointer (Slave)
# Addr_In_BO & !DMA_ADDR_LOAD & !DMA_ADDR_ENA; "Loading Address During Chaining
# Cur_Addr_BO & !DMA_ADDR_LOAD & !DMA_ADDR_ENA;

Chain_Addr_Next_BO := Chain_Addr_BO & MASTER & !S2
# Chain_Addr_BO & MASTER & !DMA_REQ
# Addr_In_BO & MASTER & !DMA_CHAIN_WR
# Addr_In_BO & MASTER & S2 & DMA_REQ "Next Chain State
# Chain_Addr_BO & MASTER & DMA_CHAIN_WR;

"------------------------------------------------------------------
" DMA Control/Status Register Logic
"------------------------------------------------------------------

1:DONE_BRST_ := !BRST
# !DMA_DONE_S & (DMA_Mode != FlyBy_Up_Chain)
# !DMA_DONE_S & (DMA_Mode == FlyBy_Up_Chain) & !CHAIN_OVF;

1:DMA_ADDR_OE_ := !DMA_SOURCE_RD & MASTER
# ADDR_OE & !LOADCTRL;

1:DMA_ADDR_LOAD_OE = 1;
1:DMA_ADDR_LOAD_ := !DMA_SOURCE_WR
# LOADCTRL & ADDR_PHASE "Load Chain Address
# DMA_REQ & S1 & DMA_ACK; "Start Address State

DMA_ADDR_CLK = PCICLKR;
DMA_ADDR_ENA_ := DMA_ACK & (DMA_States != ChainComplete) & (DMA_States != StartAddress);

DMA_INTRQ_CLK = BCLK;
DMA_INTRQ_AR := !BRST;
DMA_INTRQ.OE = 1;

DMA_INTRQ := !INT_MODE1 & !INT_MODE0 & !DMA_DONE_S & !DMA_MODE1 "Int on T.C. Non-Chaining
# !INT_MODE1 & !INT_MODE0 & !DMA_DONE_S & !DMA_MODE1 "Int on T.C. Chaining
# !INT_MODE1 & !INT_MODE0 & !DMA_DONE_S & !CHAIN_OVF & !DMA_MODE1 "Int on T.C.

Chaining
# DMA_INTRQ & CLS_WRITE; "Hold until INT Clear Strobe

DMA_COUNT_LOAD_CLK := BCLK;
DMA_COUNT_LOAD_OE = 1;
1:DMA_COUNT_LOAD_ := !DMA_COUNT_WR
# Chain_Mode & !DMA_REQ & (DMA_States == TransferCount);

DMA DONE S := DMA_ACK & !CHAIN & !DMA_DONE_S;

DMA DONE S_CLK := BCLK;
DMA DONE S_OE = 1;
1:DMA DONE S := !DMA DONE S & !DMA_START & !DMA_COUNT_LOAD;

DMA DIR := DMA_S0;
1:CHAIN_OVF0 := !BADDR0 & !BADDR0 & !BADDR2 & !BADDR2 & !BADDR2 & !BADDR2 & !BADDR2 & !BADDR2 & !BADDR2 & !BADDR2;
1:CHAIN_OVF1 := !BADDR1 & !BADDR1 & !BADDR0 & !BADDR0 & !BADDR0 & !BADDR0 & !BADDR0 & !BADDR0 & !BADDR0 & !BADDR0;
1:CHAIN_OVF2 := !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1 & !BADDR1;
1:CHAIN_OVF4 := !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3 & !BADDR3;

1:CHAIN_OVF_ := 1:CHAIN_OVF1 & 1:CHAIN_OVF2 & 1:CHAIN_OVF3 & 1:CHAIN_OVF4;
CHAIN_OVF_CLK := BCLK;
CHAIN_OVF_OE = 1;
TC_STAGE1_S_.CLK = BCLK;
TC_STAGE1_S_.OE = 1;
!TC_STAGE1_S_ := !TC_STAGE1_.

end DMAC
5.3.3. **DECODE PLD**

module DECODE flag '-r3'  
title 'Slave Decoder, PCI Status/Command Register and PCI Latency Timer'  
Sheet 1 of 1, Schematic Diagrams

This FLD implements the following functionality on the PCI Parallel Interface Board:

Decodes the PCI I/O board IO Address Map:

The PCI I/O board utilizes an IO Address Space of 128 bytes. The DECODE FLD decodes these 128 IO addresses into 32 longword Read/Write IO Registers. It does this by monitoring the least significant seven address bits of the Local Address Bus (LA7-LA0) and the LDEVSEL signal from the CONFIG PLD. The LDEVSEL signal is asserted active Low when the 24 most significant bits of the incoming IO Address are equal to the value stored in the 24 most significant bits of Base Address Register 0 in the Primary PCI Configuration Region. A valid address decode results in the generation of a read or write control strobe used on the board to control the timing of the data transfer operation. The layout of the Parallel Interface Board IO Address Map is illustrated as follows:

<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
<th>LA7</th>
<th>LA0 (Byte Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>0</td>
<td>0x00</td>
<td>Parallel Interface Control/Status Register</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>0x04</td>
<td>Parallel Interface Interrupt Vector Register (Not Implemented)</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>0x08</td>
<td>Not Decoded</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>0x10</td>
<td>Parallel Interface Data Loopback Output Reg.</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>0x14</td>
<td>Byte 0 Output FIFO Read In Diagnostic Mode</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
<td>0x18</td>
<td>Parallel Interface Data Loopback Output Reg.</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td>0x20</td>
<td>Byte 1 Output FIFO Read In Diagnostic Mode</td>
</tr>
<tr>
<td>6</td>
<td>24</td>
<td>0x24</td>
<td>Parallel Interface Data Loopback Input Reg.</td>
</tr>
<tr>
<td>6</td>
<td>28</td>
<td>0x28</td>
<td>Byte 1 Input FIFO Write In Diagnostic Mode</td>
</tr>
<tr>
<td>6</td>
<td>2C</td>
<td>0x2C</td>
<td>Parallel Interface Data Loopback Input Reg.</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>0x30</td>
<td>Byte 2 Input FIFO Write In Diagnostic Mode</td>
</tr>
<tr>
<td>6</td>
<td>34</td>
<td>0x34</td>
<td>Parallel Interface Data Loopback Input Reg.</td>
</tr>
<tr>
<td>6</td>
<td>38</td>
<td>0x38</td>
<td>Byte 3 Input FIFO Write In Diagnostic Mode</td>
</tr>
<tr>
<td>6</td>
<td>3C</td>
<td>0x3C</td>
<td>Parallel Interface Data Loopback Input Reg.</td>
</tr>
<tr>
<td>6</td>
<td>40</td>
<td>0x40</td>
<td>DMA Controller Source Address Register</td>
</tr>
<tr>
<td>6</td>
<td>44</td>
<td>0x44</td>
<td>DMA Controller Transfer Count Register</td>
</tr>
<tr>
<td>6</td>
<td>48</td>
<td>0x48</td>
<td>DMA Controller Control/Status Register</td>
</tr>
<tr>
<td>6</td>
<td>4C</td>
<td>0x4C</td>
<td>DMA Controller Next Chain Pointer Register</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>0x50</td>
<td>Input FIFO Data Read Register (Read Only)</td>
</tr>
<tr>
<td>6</td>
<td>54</td>
<td>0x54</td>
<td>Input FIFO Clear Command (Write Only)</td>
</tr>
<tr>
<td>6</td>
<td>58</td>
<td>0x58</td>
<td>Input FIFO AE/AF Flag Register (Write Only)</td>
</tr>
<tr>
<td>6</td>
<td>5C</td>
<td>0x5C</td>
<td>Not Decoded (Will Return All Ones On A Read)</td>
</tr>
<tr>
<td>6</td>
<td>60</td>
<td>0x60</td>
<td>Output FIFO Data Read Register (Read Only)</td>
</tr>
<tr>
<td>Command Register Offset 0x04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output FIFO Clear Command (Write Only)</td>
<td>0x64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output FIFO AE/AF Flag Register (Write Only)</td>
<td>0x68</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implements a 16-Bit Shadow version of the PCI Configuration Region's Command Register or Configuration Region Offset 0x04. This shadowed version of the PCI Command Register is read/write and is used to store the active PCI Configuration Region bits for real time presentation to logic on the PCI I/O Board. This 16-Bit write only register is internal only with no external connections except to local data bus bits BAH & BAF. External Command Register connections which are supported by this board are implemented by the following output signals:

**IOAces_ENABLE**
This signal is the active high PCI Command Register's IO Access Enable Bit. This signal is connected to the CONF SPEC and is used to enable PCI IO Access Access.

**MASTER_ENABLE**
This signal is the active high PCI Command Register's Master Access Enable Bit. This signal is connected to the PCI CFGX SPEC and is used to enable PCI Master Accesses by the DMA Controller. When this signal is low, the PCI Parallel Interface Board is prohibited from initiating Master Accesses on the PCI Interface.

**MEMORY_ENABLE**
This signal is the active high PCI Command Register's Memory Write And Invalidate Access Enable Bit. This signal is connected to the PCI CFGX SPEC and is used to enable PCI Memory Write And Invalidate Master Accesses by the DMA Controller. When this signal is low, the PCI Parallel Interface Board is prohibited from initiating Memory Write And Invalidate Master Accesses on the PCI Interface. In the DMA Controller is programmed to generate Memory Write And Invalidate Master Accesses and this bit is low, then the CPU Corrext will initiate Memory Write type PCI Master Accesses instead. The PCI CFGX SPEC will still use its shadow version of the Cache Line Size Register as the Burst Count size for any PCI Master transactions which have been programmed as Memory Write And Invalidate type in the DMA Control/Status Register.

**PARITY_ENABLE**
This signal is the active high PCI Command Register's Parity Checking and Reporting Enable Bit. This signal is connected to the PCI CFGX SPEC and is used to enable PCI Parity Error Response via the SEER# signal. When this signal is low, the PCI I/O Board is prohibited from generating a System Error when a Parity Error is detected. The PCI CFGX SPEC is still responsible to check and generate parity regardless of the setting of this bit. The System Error Signal Enable bit must also be set to a 1 in the PCI Command before the PCI CFGX SPEC is allowed to assert or drive the PCI SEER# signal.

**SYSERR_ENABLE**
This signal is the active high PCI Command Register's System Error Signal Enable Bit. This signal is connected to the PCI CFGX SPEC and is used to enable the System Error signal driver. When this signal is low, then the PCI I/O Board is prohibited from driving the System Error signal, SYSERR#.

This bit (and bit 6) must be set to a 1 in order to enable the PCI Parallel Interface Board to report address or data parity errors via the PCI SEER# signal via the PCI CFGX SPEC.

Implements a 6-Bit Shadow version of the PCI Configuration Region's Status Register at Configuration Region Offset 0x04. This shadowed version of the PCI Command Register is read/write and is used to store the active PCI Configuration Region Status bits for real time presentation of status signals from the PCI CFGX SPEC. This 6-Bit Status Register loads real time status information into the shadowed version of this register during PCI Configuration Space Read Cycles performed at address offset 0x04.
The PCI CFGX SPEC is prohibited from allowing any changes in the real time status inputs during these read operations which are indicated by an active high.

| **MPARER**  | PCI Bus Master Parity Error Reported |
| **STRT_ABORT**  | PCI Target Abort Issued |
| **MPA_ABORT**  | PCI Target Abort Received |
| **APARER**  | PCI Master Abort Issued |

Any Kind of PCI Parity Error Detected.
LSERR  PCI System Error Issued by This Master

Once latched into the shadowed version of the PCI Configuration Region Status Register, an active bit will remain active until the host writes a one into the corresponding data bit location during a PCI Configuration Space Write operation.

Generates the IO_CTL_WR_signal active low in response to an IO Write operation at address offset 0x00-0x1F that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0.

Generates the IO_CTL_RD_signal active low in response to an IO Read operation at address offset 0x00-0x1F that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0.

Generates the DMA_REG0_RD_signal active low in response to an IO Read operation at address offset 0x20 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a read operation of the DMA Controller's Source Address Register.

Generates the DMA_REG0_WR_signal active low in response to an IO Write operation at address offset 0x20 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a write operation of the DMA Controller's Source Address Register.

Generates the DMA_REG1_RD_signal active low in response to an IO Read operation at address offset 0x24 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a read operation of the DMA Controller's Transfer Count Register.

Generates the DMA_REG1_WR_signal active low in response to an IO Write operation at address offset 0x24 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a write operation of the DMA Controller's Transfer Count Register.

Generates the DMA_REG2_RD_signal active low in response to an IO Read operation at address offset 0x28 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a read operation of the DMA Controller's Control/Status Register.

Generates the DMA_REG2_WR_signal active low in response to an IO Write operation at address offset 0x28 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a write operation of the DMA Controller's Control/Status Register.

Generates the DMA_REG3_RD_signal active low in response to an IO Read operation at address offset 0x2C that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a read operation of the DMA Controller's Next Pointer (Chaining Address) Register.

Generates the DMA_REG3_WR_signal active low in response to an IO Write operation at address offset 0x2C that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a write operation of the DMA Controller's Next Pointer (Chaining Address) Register.

Generates the SLAVE_FIFO_RD_signal active low in response to any 32-Bit IO Read operation at address offset 0x30 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is also generated when the DMA Controller performs a DMA Fly-By IO Read/Write Memory Operation over the PCI Bus. This signal is used to read data from Byte 0 of the Interface 4K byte Input FIFO Buffer.

Generates the FIFO_CLEAR_signal active low in response to an IO Write operation at address offset 0x34 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to perform a data clear operation on the 1/0 Interface 4K byte Input FIFO Buffer with the data value written a don't care.

Generates the IFIFO_DBF_signal active high in response to an IO Write Operation at address offset 0x38 that is decoded equivalent to the address stored in the PCI Configuration Region's Base Address Register 0. This signal is used to program the 8-bit Almost Full/Almost Empty Flag in the 1/0 Interface 4K byte Input FIFO Buffer.

Generates the OFFSET_WR_signal active low in response to any 32-Bit IO Write operation at address offset 0x40 that is decoded equivalent to the
Generates the OPTF0_CLEAR signal active low in response to an IO Write.
Operation at address offset 0x44 that is decoded equivalent to the
address stored in the PCI Configuration Region's Base Address Register 0.
This signal is used to perform a data clear operation on the I/O Interface
4 byte Output FIFO Buffer with the data value written a don't care.

Generates the OPTF0_MFR signal active high in response to an IO Write.
Operation at address offset 0x48 that is decoded equivalent to the
address stored in the PCI Configuration Region's Base Address Register 0.
This signal is used to program the 8-bit Almost Full/Almost Empty Flag
in the I/O Interface 4K byte Output FIFO Buffer.

Generates the CLS_WRITE signal active low in response to a PCI Configuration
Space Write Operation decoded at the Cache Line Size Register. The Cache Line
Size Register is shadowed in the PCICTRL PFD so used as the Basic Counter Size
for Memory Write and Invalidate Indicator PCI Cycles. The CLS_WRITE signal is
generated to the PCICTRL PFD so that Cache Line Size Configuration Region Writes
operation can be shadowed (stored) in a 5-bit Register in the PCICTRL PFD.

Generates the signal STATUS_STORE signal active high during the valid PCI data phase of
a PCI Configuration Space Read decoded at the Status Register address. This signal
is used as an Internal feedback term for the equations which clock and control
the valid and invalid terms of the Status Register data bits implemented in the PCICTRL PFD.
The STATUS_STORE signal is also used by the PCI_CTRL PFD to clear latched
PCI Status Register Bits.

Generates the signal PEB01_PD active low to the CONFIG PFD for data byte 0 read
accesses to either the PCI Configuration Region of the DMA Temporary Data Register
Storage Area. Both of these memory spaces are implemented in the SRAM contained
within the CONFIG PFD. The active low state of the PEB01_PD signal allows the
CONFIG PFD SRAM outputs to drive local data bits BAD0 BAD3. This will occur for
valid decoded PCI Configuration Space Read Operations, PCI IO Read Operations to the
DMA Temporary Data Register Storage Area and during Memory-To-Memory DMA Transfers
under the control of the DMA_PD_read strobe from the DMAC PFD.

Generates the signal PEB02_PD active low to the CONFIG PFD for data byte 1 read
accesses to either the PCI Configuration Region of the DMA Temporary Data Register
Storage Area. Both of these memory spaces are implemented in the SRAM contained
within the CONFIG PFD. The active low state of the PEB02_PD signal allows the
CONFIG PFD SRAM outputs to drive local data bits BAD4 BAD7. This will occur for
valid decoded PCI Configuration Space Read Operations, PCI IO Read Operations to the
DMA Temporary Data Register Storage Area and during Memory-To-Memory DMA Transfers
under the control of the DMA_PD_read strobe from the DMAC PFD.

Generates the signal PEB03_PD active low to the CONFIG PFD for data byte 2 read
accesses to either the PCI Configuration Region of the DMA Temporary Data Register
Storage Area. Both of these memory spaces are implemented in the SRAM contained
within the CONFIG PFD. The active low state of the PEB03_PD signal allows the
CONFIG PFD SRAM outputs to drive local data bits BAD8 BADB. This will occur for
valid decoded PCI Configuration Space Read Operations, PCI IO Read Operations to the
DMA Temporary Data Register Storage Area and during Memory-To-Memory DMA Transfers
under the control of the DMA_PD_read strobe from the DMAC PFD.

Generates the signal PEB04_WR active low to the CONFIG PFD for data byte 0 write
accesses to either the PCI Configuration Region of the DMA Temporary Data Register
Storage Area. Both of these memory spaces are implemented in the SRAM contained
within the CONFIG PFD. The active low state of the PEB04_WR signal allows the
CONFIG PFD SRAM data inputs to be driven by local data bits BADD BAD0. This will
occur for valid decoded PCI Configuration Space Write Operations, PCI IO Write
Operations to the DMA Temporary Data Register Storage Area and during Memory-To-Memory
DMA Transfers under the control of the DMA_WR write strobe from the DMAC PFD.

Generates the signal PEB01_WR active low to the CONFIG PFD for data byte 1 write
accesses to either the PCI Configuration Region of the DMA Temporary Data Register
Storage Area. Both of these memory spaces are implemented in the SRAM contained
within the CONFIG PFD. The active low state of the PEB01_WR signal allows the
CONFIG PFD SRAM data inputs to be driven by local data bits BAD8 BAD3. This will
occur for valid decoded PCI Configuration Space Write Operations, PCI IO Write
Operations to the DMA Temporary Data Register Storage Area and during Memory-To-Memory
DMA Transfers under the control of the DMA_WR_write strobe from the DMAC PLD.

Generates the signal REGB2_WR active low to the CONFIG PLD for data byte 2 write accesses to either the PCI Configuration Region of the DMA Temporary Data Register Storage Area. Both of these memory spaces are implemented in the SRAM contained within the CONFIG PLD. The active low state of the REGB2_WR signal allows the CONFIG PLD SRAM data inputs to drive by local data bits BAD16 BAD23. This will occur for valid decoded PCI Configuration Space Write Operations, PCI I/O Write Operations to the DMA Temporary Data Register Storage Area and during Memory-To-Memory DMA Transfers under the control of the DMA_WR_write strobe from the DMAC PLD.

Generates the signal REGB3_WR active low to the CONFIG PLD for data byte 3 write accesses to either the PCI Configuration Region of the DMA Temporary Data Register Storage Area. Both of these memory spaces are implemented in the SRAM contained within the CONFIG PLD. The active low state of the REGB3_WR signal allows the CONFIG PLD SRAM data inputs to be driven by local data bits BAD25 BAD31. This will occur for valid decoded PCI Configuration Space Write Operations, PCI I/O Write Operations to the DMA Temporary Data Register Storage Area and during Memory-To-Memory DMA Transfers under the control of the DMA_WR_write strobe from the DMAC PLD.

Generates the LT_OVF_active low in response to the Latency Timer counting down to zero. LT_OVF is synchronized to LCCLK and presented to the PCICTRL PLD to instruct it to disconnect from the PCI Bus if the PCI Arbiter has removed its Grant. This signal is designated as the Latency Timer Overflow Output.

Implements an 8-Bit PCI Latency Timer which is activated when the board arbitrates for and becomes a PCI Initiator. The latency timer prevents the board from holding the PCI Bus as a Initiator for more than the programmed number of PCI Clocks in the Configuration Regions Latency Timer Field. When this counter overflows by counting down to zero, the signal, LT_OVF, is asserted active low to the PCICTRL PLD to instruct it to release the PCI bus upon completion of the current data phase provided that its PCI Bus Grant has been relinquished by the Host PCI Bus Arbitrator.

Generates demultiplexed versions of the PCI Command Signals CMD0, CMD3, CMD2 and CMD3. These are used by the DECODE and CONFIG PLDs to decode the type of PCI transaction being executed during a PCI Frame.

------------------------------------------------------------------------------------------------------------------

DECODE device 'ifx780fp',

------------------------------------------------------------------------------------------------------------------

Inputs Pins

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK6</td>
<td>52: Local (up to 66MHz) x2 PCI Bus Clock</td>
</tr>
<tr>
<td>BCLK</td>
<td>1: Local PCI Bus Clock for combinational logic</td>
</tr>
<tr>
<td>ADDR_CLOCK</td>
<td>115: Address Demultiplexing Clock</td>
</tr>
<tr>
<td>BRST</td>
<td>2: Local Active Low PCI Reset</td>
</tr>
<tr>
<td>BIDSEL</td>
<td>3: Buffered Configuration Region Select</td>
</tr>
<tr>
<td>BTRDY</td>
<td>4: Local PCI Target Ready</td>
</tr>
<tr>
<td>BIRDY</td>
<td>5: Local PCI Initiator Ready</td>
</tr>
<tr>
<td>BTRMEM</td>
<td>29: Buffered PCI FRAME signal</td>
</tr>
<tr>
<td>LTRMEM</td>
<td>33: Local PCI FRAME# signal</td>
</tr>
<tr>
<td>MASTER</td>
<td>34: Local PCI Device is PCI Bus Master</td>
</tr>
<tr>
<td>DMA_DONE</td>
<td>35: DMA Transfer Count Complete</td>
</tr>
<tr>
<td>DMA_DONE_S</td>
<td>22: DMA Transfer Count Complete Latched</td>
</tr>
<tr>
<td>DMA_ACK</td>
<td>36: DMA Cycle Acknowledge</td>
</tr>
<tr>
<td>DMA_DIRECTION</td>
<td>37: DMA Cycle Direction</td>
</tr>
<tr>
<td>CHAIN</td>
<td>38: DMA Chaining Parameter Fetching Operation</td>
</tr>
<tr>
<td>LOADCTRL</td>
<td>67: DMA State Machine Chain Load DMA Control Register</td>
</tr>
<tr>
<td>DATA_PHASE</td>
<td>68: Valid PCI Data Phase Detected</td>
</tr>
<tr>
<td>CMD_ENABLE</td>
<td>25: Valid PCI Command Enable</td>
</tr>
<tr>
<td>BC_BE0</td>
<td>69: Local PCI Bus Enable 0#</td>
</tr>
<tr>
<td>BC_BE1</td>
<td>70: Local PCI Bus Enable 1#</td>
</tr>
<tr>
<td>BC_BE2</td>
<td>71: Local PCI Bus Enable 2#</td>
</tr>
<tr>
<td>BC_BE3</td>
<td>99: Local PCI Bus Enable 3#</td>
</tr>
<tr>
<td>CMD0</td>
<td>55: Local PCI Command Bit 0</td>
</tr>
<tr>
<td>CMD1</td>
<td>10: Local PCI Command Bit 1</td>
</tr>
<tr>
<td>CMD2</td>
<td>12: Local PCI Command Bit 2</td>
</tr>
<tr>
<td>CMD3</td>
<td>80: Local PCI Command Bit 3</td>
</tr>
<tr>
<td>LDEVSEL</td>
<td>100: Local PCI Device Select#</td>
</tr>
</tbody>
</table>
**PCI Local Address Bits To Decode**

<table>
<thead>
<tr>
<th>LA7</th>
<th>pin 48;</th>
<th>&quot;Local Address Bit 7&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA6</td>
<td>pin 47;</td>
<td>&quot;Local Address Bit 6&quot;</td>
</tr>
<tr>
<td>LA5</td>
<td>pin 46;</td>
<td>&quot;Local Address Bit 5&quot;</td>
</tr>
<tr>
<td>LA4</td>
<td>pin 45;</td>
<td>&quot;Local Address Bit 4&quot;</td>
</tr>
<tr>
<td>LA3</td>
<td>pin 44;</td>
<td>&quot;Local Address Bit 3&quot;</td>
</tr>
<tr>
<td>LA2</td>
<td>pin 43;</td>
<td>&quot;Local Address Bit 2&quot;</td>
</tr>
</tbody>
</table>

**Latched Latency Timer Values**

- **LT7**: node 40; "Latched Latency Timer Value 7"
- **LT6**: node 39; "Latched Latency Timer Value 6"
- **LT5**: node 38; "Latched Latency Timer Value 5"
- **LT4**: node 37; "Latched Latency Timer Value 4"
- **LT3**: node 36; "Latched Latency Timer Value 3"
- **LT2**: node 35; "Latched Latency Timer Value 2"
- **LT1**: node 34; "Latched Latency Timer Value 1"
- **L_TO**: node 33; "Latched Latency Timer Value 0"
- **LT_WRITE**: node 32; "Latency Timer Write Strobe"

**PCI Status Register Inputs From PCI Master/Slave State Machines**

- **MPARER_RPT**: pin 103; "PCI Bus Master Parity Error Reported"
- **STRG_ABORT**: pin 104; "PCI Target Abort Issued"
- **MTFG_ABORT**: pin 105; "PCI Target Abort Received"
- **MASTER_ABORT**: pin 40; "PCI Master Abort Received"
- **APPEP**: pin 41; "Any Kind of PCI Parity Error Detected"
- **LSBF**: pin 32; "PCI System Error Issued by This Master"

**PCI Valid Command Register Outputs To PCI Master/Slave State Machines**

- **IORRCE_ENABLE**: pin 89; "PCI Command Register IO Access Enable"
- **MASTER_ENABLE**: pin 45; "PCI Command Register Master Enable"
- **MEMIV_ENABLE**: pin 46; "PCI Command Register Memory Write and Invalidate"
- **APPEWP_ENABLE**: pin 71; "PCI Command Register Parity Error Enable"
- **SYPEP_ENABLE**: pin 8; "PCI Command Register System Error Enable"

**PCI Status Register Bi-Directional Pins**

| STATUS_B8 | pin 58; | "Status Register Data Bit 8" |
| STATUS_B9 | pin 57; | "Status Register Data Bit 9" |
| STATUS_B10| pin 56; | "Status Register Data Bit 10" |
| STATUS_B11| pin 55; | "Status Register Data Bit 11" |
| STATUS_B12| pin 54; | "Status Register Data Bit 12" |
| STATUS_B13| pin 53; | "Status Register Data Bit 13" |
| STATUS_B14| pin 52; | "Status Register Data Bit 14" |
| STATUS_B15| pin 51; | "Status Register Data Bit 15" |
| BAD24     | pin 50; | "Buffered Data/Address Bit 24" |
| BAD25     | pin 49; | "Buffered Data/Address Bit 25" |
| BAD26     | pin 48; | "Buffered Data/Address Bit 26" |
| BAD27     | pin 47; | "Buffered Data/Address Bit 27" |
| BAD28     | pin 46; | "Buffered Data/Address Bit 28" |
| BAD29     | pin 45; | "Buffered Data/Address Bit 29" |
| BAD30     | pin 44; | "Buffered Data/Address Bit 30" |
| BAD31     | pin 43; | "Buffered Data/Address Bit 31" |
| COMMAND_B0| pin 82; | "Command Register Data Bit 0" |
| COMMAND_B1| pin 81; | "Command Register Data Bit 1" |
| COMMAND_B2| pin 80; | "Command Register Data Bit 2" |
| COMMAND_B3| pin 79; | "Command Register Data Bit 3" |
| COMMAND_B4| pin 78; | "Command Register Data Bit 4" |
| COMMAND_B5| pin 77; | "Command Register Data Bit 5" |
| COMMAND_B6| pin 76; | "Command Register Data Bit 6" |
| COMMAND_B7| pin 75; | "Command Register Data Bit 7" |
| COMMAND_B8| pin 74; | "Command Register Data Bit 8" |
| COMMAND_B9| pin 73; | "Command Register Data Bit 9" |
| COMMAND_B10| pin 72; | "Command Register Data Bit 10" |
| COMMAND_B11| pin 71; | "Command Register Data Bit 11" |
| COMMAND_B12| pin 70; | "Command Register Data Bit 12" |
| COMMAND_B13| pin 69; | "Command Register Data Bit 13" |
| COMMAND_B14| pin 68; | "Command Register Data Bit 14" |
Rochester Institute of Technology Computer Science Department

COMMAND_B15  pin 56; "Command Register Data Bit 15"
BAD0  pin 82; "Status Register Data Bit 0"
BAD1  pin 81; "Status Register Data Bit 1"
BAD2  pin 80; "Status Register Data Bit 2"
BAD3  pin 79; "Status Register Data Bit 3"
BAD4  pin 78; "Status Register Data Bit 4"
BAD5  pin 76; "Status Register Data Bit 5"
BAD6  pin 75; "Status Register Data Bit 6"
BAD7  pin 74; "Status Register Data Bit 7"
BAD8  pin 64; "Status Register Data Bit 8"
BAD9  pin 63; "Status Register Data Bit 9"
BAD10 pin 62; "Status Register Data Bit 10"
BAD11 pin 61; "Status Register Data Bit 11"
BAD12 pin 60; "Status Register Data Bit 12"
BAD13 pin 58; "Status Register Data Bit 13"
BAD14 pin 57; "Status Register Data Bit 14"
BAD15 pin 56; "Status Register Data Bit 15"

*----------------------------------------*
<table>
<thead>
<tr>
<th>Outputs and Feedback Terms</th>
</tr>
</thead>
</table>

IO_CTL_WR_  pin 106; "I/O Control Register Write#"
IO_STA_RD_  pin 105; "I/O Status Register Read#"
DRPM_FIFO_RD_  pin 72; "Input FIFO 32-Bit Read Strobe in Slave Mode#"
IFIFO_CLEAR_  pin 107; "Input FIFO Clear Strobe#"
IFIFO_DBF_  pin 109; "Input FIFO Define Flag Strobe"
OFIFO_WR_  pin 122; "Output FIFO 32-Bit Write Strobe#"
OFIFO_CLEAR_  pin 113; "Output FIFO Clear Strobe#"
OFIFO_DBF_  pin 126; "Output FIFO Define Flag Strobe"
FIFO_GBA_  pin 9; "FIFO PCI Side Transceiver Control"

DMA_REG8_RD_  pin 108; "DMA Controller Source Address Register Read#"
DMA_REG1_RD_  pin 112; "DMA Controller Transfer Count Register Read#"
DMA_REG2_RD_  pin 111; "DMA Controller Control/Status Register Read#"
DMA_REG3_RD_  pin 127; "DMA Controller Next Pointer Register Read#"
DMA_REG0_WR_  pin 128; "DMA Controller Source Address Register Write#"
DMA_REG1_WR_  pin 129; "DMA Controller Transfer Count Register Write#"
DMA_REG2_WR_  pin 123; "DMA Controller Control/Status Register Write#"
DMA_REG3_WR_  pin 130; "DMA Controller Next Pointer Register Write#"

CLS_WRITE_  pin 73; "Cache Line Size Configuration Region Write Strobe#"
STATUS_STROBE_  pin 54; "PCI Configuration Region Status Register Strobe#"

REG00_RD_  pin 120; "Config Region and DMA Temp Data Storage Byte 0 Read Strobe#"
REG01_RD_  pin 121; "Config Region and DMA Temp Data Storage Byte 1 Read Strobe#"
REG02_RD_  pin 115; "Config Region and DMA Temp Data Storage Byte 2 Read Strobe#"
REG03_RD_  pin 114; "Config Region and DMA Temp Data Storage Byte 3 Read Strobe#"

REG00_WR_  pin 16; "Config Region and DMA Temp Data Storage Byte 0 Write Strobe#"
REG01_WR_  pin 15; "Config Region and DMA Temp Data Storage Byte 1 Write Strobe#"
REG02_WR_  pin 14; "Config Region and DMA Temp Data Storage Byte 2 Write Strobe#"
REG03_WR_  pin 13; "Config Region and DMA Temp Data Storage Byte 3 Write Strobe#"

*----------------------------------------*
<table>
<thead>
<tr>
<th>PCI Latency Timer Pin Definitions</th>
</tr>
</thead>
</table>

LT_OVF_  pin 28; "Latency Timer Overflow Output"

LTCD7  node 24; "Latency Timer Data Bit 7"
LTCD6  node 22; "Latency Timer Data Bit 6"
LTCD5  node 23; "Latency Timer Data Bit 5"
LTCD4  node 32; "Latency Timer Data Bit 4"
LTCD3  node 29; "Latency Timer Data Bit 3"
LTCD2  node 31; "Latency Timer Data Bit 2"
LTCD1  node 26; "Latency Timer Data Bit 1"
LTCD0  node 25; "Latency Timer Data Bit 0"

LT timer { LTCD7, LTCD6, LTCD5, LTCD4, LTCD3, LTCD2, LTCD1, LTCD0 };
LT timer_hi { LTCD7, LTCD6, LTCD5, LTCD4 };
LT timer_lo { LTCD3, LTCD2, LTCD1, LTCD0 }; 
LT timer_tpa { LTCD7.fb, LTCD6.fb, LTCD5.fb, LTCD4.fb, LTCD3.fb, LTCD2.fb, LTCD1.fb, LTCD0.fb }; 
LT timer_In { BAD15, BAD14, BAD13, BAD12, BAD11, LTCD2, LTCD1, LTCD0 }; 
LT timer_In1 { BAD15, BAD14, BAD13, BAD12 }; 
LT timer_Out { LTCD7.fb, LTCD6.fb, LTCD5.fb, LTCD4.fb, LTCD3.fb, LTCD2.fb, LTCD1.fb, LTCD0.fb }; 
LT timer_out_sim { LTCD7, LTCD6, LTCD5, LTCD4 }; 
LT timer_out_hi { LTCD7, LTCD6, LTCD5, LTCD4 }; 
LT timer_out_lo { LTCD3, LTCD2, LTCD1, LTCD0 }; 
LT timer_out_tpa { LTCD3, LTCD2, LTCD1, LTCD0 };
Set Definitions and Constants

\[ H, L, X, C, Z, \text{SIMULATE} \]

PCI Command Definitions

\[ \text{PCI_CMD_BE} = [BC\ BE3, BC\ BE2, BC\ BE1, BC\ BE0]; \]
\[ \text{PCI\ Command} = \{\text{CMO3, CMOD2, CMOD1, CMOD0}\}; \]
\[ \text{Interrupt\ Acknowledge} = [0, 0, 0, 0]; \]
\[ \text{Special\ Cycle} = [0, 0, 0, 0]; \]
\[ \text{IO\ Read} = [0, 0, 0, 0]; \]
\[ \text{IO\ Write} = [0, 0, 0, 0]; \]
\[ \text{Reserved1} = [0, 0, 0, 0]; \]
\[ \text{Reserved2} = [0, 0, 0, 0]; \]
\[ \text{Memory\ Read} = [0, 0, 0, 0]; \]
\[ \text{Memory\ Write} = [0, 0, 0, 0]; \]
\[ \text{Reserved3} = [0, 0, 0, 0]; \]
\[ \text{Configuration\ Read} = [0, 0, 0, 0]; \]
\[ \text{Configuration\ Write} = [0, 0, 0, 0]; \]
\[ \text{Memory\ Read\ Multiple} = [0, 0, 0, 0]; \]
\[ \text{Memory\ Write\ Invalidate} = [0, 0, 0, 0]; \]
\[ \text{Local\ Data} = \{BAD7, BAD6, BAD5, BAD4, BAD3, BAD2, BAD1, BAD0\}; \]
\[ \text{Timer\ Data} = \{BAD15, BAD14, BAD13, BAD12, BAD11, BAD10, BAD9, BAD8\}; \]
\[ \text{Local\ Address} = \{LA6, LA5, LA4, LA3, LA2\}; \]
\[ \text{Local\ Space} = \{LA7, LA6\}; \]
\[ \text{IO\ Control\ Status} = \{00\}; \]
\[ \text{DMA\ Source\ Address\ Reg} = \{00\}; \]
\[ \text{DMA\ Transfer\ Count\ Reg} = \{00\}; \]
\[ \text{DMA\ Control\ Status\ Reg} = \{00\}; \]
\[ \text{DMA\ Write\ Pointer\ Reg} = \{00\}; \]
\[ \text{Input\ FIFO} = \{00\}; \]
\[ \text{Clear\ Input\ FIFO} = \{00\}; \]
\[ \text{Write\ Input\ FIFO\ Flag} = \{00\}; \]
\[ \text{Output\ FIFO} = \{00\}; \]
\[ \text{Clear\ Output\ FIFO} = \{00\}; \]
\[ \text{Write\ Output\ FIFO\ Flag} = \{00\}; \]
\[ \text{Interrupt\ Clear} = \{00\}; \]
\[ \text{DMA\ Temp\ Space} = \{00\}; \]
\[ \text{Config\ Region\ Space} = \{00\}; \]
\[ \text{Config\ Address} = \{LA6, LA5, LA4, LA3, LA2, BAD1, BAD0\}; \]
\[ \text{Latency\ Timer} = \{00\}; \]
\[ \text{Status\ Register} = \{00\}; \]
\[ \text{Command\ Register} = \{00\}; \]
\[ \text{Cache\ Line\ Size} = \{00\}; \]
\[ \text{Address\ Zero} = \{00\}; \]
\[ \text{Base\ Address\ 1} = \{00\}; \]
\[ \text{Base\ Address\ 2} = \{00\}; \]
\[ \text{Base\ Address\ 3} = \{00\}; \]
\[ \text{Base\ Address\ 4} = \{00\}; \]
\[ \text{Base\ Address\ 5} = \{00\}; \]
\[ \text{Reserved\ 0} = \{00\}; \]
\[ \text{Reserved\ 1} = \{00\}; \]
\[ \text{Reserved\ 2} = \{00\}; \]
\[ \text{Reserved\ 3} = \{00\}; \]
\[ \text{Reserved\ 4} = \{00\}; \]
\[ \text{Reserved\ 5} = \{00\}; \]
\[ \text{Reserved\ 6} = \{00\}; \]
\[ \text{Reserved\ 7} = \{00\}; \]
\[ \text{Reserved\ 8} = \{00\}; \]
\[ \text{Reserved\ 9} = \{00\}; \]
\[ \text{Reserved\ 10} = \{00\}; \]
\[ \text{Reserved\ 11} = \{00\}; \]
\[ \text{Reserved\ 12} = \{00\}; \]
\[ \text{Reserved\ 13} = \{00\}; \]
\[ \text{Reserved\ 14} = \{00\}; \]
\[ \text{Reserved\ 15} = \{00\}; \]

Equations

\[ \begin{align*}
\text{IO\_CTL\_WR} & \rightarrow \text{IFRAME} & (\text{Local\ Space} \equiv \text{IO\ Control\ Status}) \\
& \& \text{DATA\ PHASE} & \& \text{IDEVSEL} \\
& \& (\text{PCI\ Command} \equiv \text{IO\ Write}) & \& \text{BRST} \\
& \& \text{IFRAME} & (\text{Local\ Address} \equiv \text{Write\ Input\ FIFO\ Flag}) \\
& \& \text{IDEVSEL} \\
& \& (\text{PCI\ Command} \equiv \text{IO\ Write}) & \& \text{BRST}; \\
\end{align*} \]

\[ \begin{align*}
\text{IO\_STA\_RD} & \rightarrow \text{IFRAME} & (\text{Local\ Space} \equiv \text{IO\ Control\ Status}) \\
& \& \text{IDEVSEL} \\
& \& (\text{PCI\ Command} \equiv \text{IO\ Read}) & \& \text{BRST}; \\
\end{align*} \]

*Equations for Decoding and Generating The DMA Source Address Register

*Read And Write Strobes:
| DMA_REG0_RD_ | LFRAME_ & (LocalAddress == DMASourceAddressReg) & !LDEVSEL & (PCI_Command == IO_Read) & BRST_; |
| DMA_REG0_WR_ | LFRAME_ & (LocalAddress == DMASourceAddressReg) & !LDEVSEL & (PCI_Command == IO_Write) & BRST_; |

*Equations for Decoding and Generating The DMA Transfer Count Register*

*Read And Write Strokes:*

| DMA_REG1_RD_ | LFRAME_ & (LocalAddress == DMATransferCountReg) & !LDEVSEL & (PCI_Command == IO_Read) & BRST_; |
| DMA_REG1_WR_ | LFRAME_ & (LocalAddress == DMATransferCountReg) & !LDEVSEL & !BCLK & (PCI_Command == IO_Write) & BRST_; |

*Equations for Decoding and Generating The DMA Control/Status Register*

*Read And Write Strokes:*

| DMA_REG2_RD_ | LFRAME_ & (LocalAddress == DMAControlStatusReg) & !LDEVSEL & (PCI_Command == IO_Read) & BRST_; |
| DMA_REG2_WR_ | LFRAME_ & (LocalAddress == DMAControlStatusReg) & !LDEVSEL & !BCLK & LDEVSEL & CHAIN & DATA_PHASE_; |

*Equations for Decoding and Generating The DMA Next Pointer Address Pointer*

*Read And Write Strokes:*

| DMA_REG3_RD_ | LFRAME_ & (LocalAddress == DMANextPointerReg) & !LDEVSEL & (PCI_Command == IO_Read) & BRST_; LDEVSEL & LOADCTRL & CHAIN & DATA_PHASE_; |
| DMA_REG3_WR_ | LFRAME_ & (LocalAddress == DMANextPointerReg) & DATA_PHASE_ & !LDEVSEL & !BCLK & (PCI_Command == IO_Write) & BRST_; |

*Equation For Decoding and Generating The Input FIFO Clear Strobe:*

```
IFIFO_CLEAR_.CLK  BCLK;
IFIFO_CLEAR_.OE  1;

!IFIFO_CLEAR_ := LFRAME_ & (LocalAddress == ClearInputFIFO) & !LDEVSEL & (PCI_Command == IO_Write) & BRST_ & !IFIFO_DBF.CLK & BCLK;
```

*Equation To Decode and Generate The Input FIFO*

*Almost Full/Almost Empty Flag Write Strobe:*

```
IFIFO_DBF.CLK  =  BCLK;
IFIFO_DBF.OE  =  1;

```

*Equations For Decoding and Generating The Input FIFO Read Strokes in Slave Mode:*

| SLAVE_FIFO_RD_ := LFRAME_ & (LocalAddress == InputFIFO) & !LDEVSEL & !DATA_PHASE & (PCI_Command == IO_Read) & BRST_; |
| SLAVE_FIFO_RD_.CLK  := ICLK66;
| SLAVE_FIFO_RD_.OE  := 1;

*Equation To Decode and Generate The Output FIFO Clear Strobe:*

```
```
"Equations To Decode and Generate the Output FIFO Write Strobes:

\[
\begin{align*}
\text{OFIFO\_DAF\_CLK} &= \text{BCLK}; \\
\text{OFIFO\_DAF\_OE} &= 1; \\
\text{!OFIFO\_DAF} &= \text{!LFRAME\_ & (LocalAddress == WriteOutputFIFOFlag)} \\
&\quad \& \text{!DATA\_PHASE\_ & !LDEVSEL} \\
&\quad \& \text{(PCI\_Command == IO\_Write) && BRST} \\
&\quad \& \text{!OFIFO\_CLEAR\_ && BRST};
\end{align*}
\]

"Equations To Decode and Generate the Output FIFO Write Strobes:

\[
\begin{align*}
\text{OFIFO\_WR\_} &= \text{!LFRAME\_ & (LocalAddress == OutputFIFO)} \\
&\quad \& \text{!DATA\_PHASE\_ & !LDEVSEL} \\
&\quad \& \text{(PCI\_Command == IO\_Write) && BRST} \\
&\quad \& \text{DMA\_ACK\_ && !CHAIN \& !DMA\_DIRECTION \& !BCLK \& BRST \& DMA\_DONE\_ \& DMA\_DONE\_S}; \\
\text{OFIFO\_WR\_D} &= \text{!DMA\_DONE}; \\
\text{OFIFO\_WR\_CLK} &= \text{CLK66}; \\
\text{OFIFO\_WR\_OE} &= 1;
\end{align*}
\]

"Equation For enabling the 32-Bit PCI transceiver side of the FIFO Buffers

\[
\begin{align*}
\text{FIFO\_GBA\_CLK} &= \text{BCLK}; \\
\text{FIFO\_GBA\_AR} &= \text{CHAIN}; \\
\text{FIFO\_GBA\_OE} &= \text{1};
\end{align*}
\]

"Equations For Controlling Latency Timer Operation

\[
\begin{align*}
\text{LTimer\_CLK} &= \text{BCLK}; \\
\text{LTimer\_ASET} &= \text{1BRST}; \\
\text{LTREG\_OUT\_HI\_ASET} &= \text{!BRST}; \\
\text{LTCD7} &= \text{LTC7\_fb \& MASTER}; \\
\text{LTCD6} &= \text{LTC6\_fb \& MASTER}; \\
\text{LTCD5} &= \text{LTC5\_fb \& MASTER}; \\
\text{LTCD4} &= \text{LTC4\_fb \& MASTER}; \\
\text{LTCD3} &= \text{LTC3 \& MASTER}; \\
\text{LTCD2} &= \text{LTC2 \& MASTER}; \\
\text{LTCD1} &= \text{LTC1 \& MASTER}; \\
\text{LTCD0} &= \text{LTC0 \& MASTER}; \\
\text{LT\_OVT\_CLK} &= \text{BCLK}; \\
\text{LT\_OVT} &= \text{!LTCD7 \& !LTCD6 \& !LTCD5 \& !LTCD4} \\
&\quad \& \text{!LTCD3 \& !LTCD2 \& !LTCD1 \& !LTCD0}; \\
\text{LT\_OVT\_OE} &= \text{1}; \\
\text{!LT\_WRITE} &= \text{!LFRAME\_ & (ConfigAddress == LatencyTimer)} \\
&\quad \& \text{!DATA\_PHASE\_ \& BRST} \\
&\quad \& \text{(PCI\_Command == ConfigurationWrite) \& !BC\_BE1};
\end{align*}
\]

"Equation to Latch Latency Timer Value Into Internal Register:

\[
\begin{align*}
\text{LTC7} &= \text{BAD\_5 \& !LT\_WRITE} \\
&\quad \& \text{!LTC7\_fb \& !LT\_WRITE \& !BRST}; \\
\text{LTC7\_OE} &= \text{1}; \\
\text{LTC7\_CLK} &= \text{BCLK}; \\
\text{LTC6} &= \text{BAD\_4 \& !LT\_WRITE} \\
&\quad \& \text{!LTC6\_fb \& !LT\_WRITE \& !BRST}; \\
\text{LTC6\_OE} &= \text{1}; \\
\text{LTC6\_CLK} &= \text{BCLK}; \\
\text{LTC5} &= \text{BAD\_3 \& !LT\_WRITE} \\
&\quad \& \text{!LTC5\_fb \& !LT\_WRITE \& !BRST}; \\
\text{LTC5\_OE} &= \text{1}; \\
\text{LTC5\_CLK} &= \text{BCLK}; \\
\text{LTC4} &= \text{BAD\_2 \& !LT\_WRITE} \\
&\quad \& \text{!LTC4\_fb \& !LT\_WRITE \& !BRST}; \\
\text{LTC4\_OE} &= \text{1}; \\
\text{LTC4\_CLK} &= \text{BCLK};
\end{align*}
\]

"Equations For Accessing Configuration Region and DMA Temporary Data Storage SRAM:

\[
\begin{align*}
\text{!REG83\_WR} &= \text{!LDEVSEL\_ && (PCI\_Command == ConfigurationWrite)} \\
&\quad \& \text{!BRST \& !DATA\_PHASE \& !BC\_BE3 \& CMD\_ENABLE} \\
&\quad \& \text{(ConfigAddress != BaseAddress1\_3)} \\
&\quad \& \text{(ConfigAddress != BaseAddress2\_5)} \\
&\quad \& \text{(ConfigAddress != AddressZero)};
\end{align*}
\]
# !LFRAME_ & !BC_BE1_ & BRST_ & !DATA_PHASE_ & !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Write) & CMD_ENABLE_;

!REGB2_WR_
| !LDEVSEL_ & (PCI_Command == ConfigurationWrite) & BRST_ & !DATA_PHASE_ & !BC_BE2_ & CMD_ENABLE_ & (ConfigAddress != BaseAddress1_3) & (ConfigAddress != BaseAddress2) & (ConfigAddress != StatusRegister) & (ConfigAddress != AddressZero) |
# !LFRAME_ & !BC_BE2_ & BRST_ & !DATA_PHASE_ & !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Write) & CMD_ENABLE_; |

!REGB1_WR_
| !LDEVSEL_ & (PCI_Command == ConfigurationWrite) & BRST_ & !DATA_PHASE_ & !BC_BE1_ & CMD_ENABLE_ & (ConfigAddress != BaseAddress1_3) & (ConfigAddress != BaseAddress2) & (ConfigAddress != BaseAddress4_5) & (ConfigAddress != AddressZero) |
# !LFRAME_ & !BC_BE1_ & BRST_ & !DATA_PHASE_ & !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Write) & CMD_ENABLE_; |

!REGB0_WR_
| !LDEVSEL_ & (PCI_Command == ConfigurationWrite) & BRST_ & !DATA_PHASE_ & !BC_BE0_ & CMD_ENABLE_ & (ConfigAddress != BaseAddress1_3) & (ConfigAddress != BaseAddress2) & (ConfigAddress != BaseAddress4_5) & (ConfigAddress != AddressZero) |
# !LFRAME_ & !BC_BE0_ & BRST_ & !DATA_PHASE_ & !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Write) & CMD_ENABLE_; |

!REGB0_RD_
| !LDEVSEL_ & (PCI_Command == ConfigurationRead) & CMD_ENABLE_ & (ConfigAddress != StatusRegister) & BRST_ & !BC_BE0_; |

!REGB1_RD_
| !LDEVSEL_ & (PCI_Command == ConfigurationRead) & CMD_ENABLE_ & (ConfigAddress != StatusRegister) & BRST_ & !BC_BE1_ & !LFRAME_ & BRST_ |
| !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Read) & CMD_ENABLE_ & !BC_BE1_; |

!REGB2_RD_
| !LDEVSEL_ & (PCI_Command == ConfigurationRead) & CMD_ENABLE_ & BRST_ & !BC_BE2_ & !LFRAME_ & BRST_ |
| !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Read) & CMD_ENABLE_ & !BC_BE2_; |

!REGB3_RD_
| !LDEVSEL_ & (PCI_Command == ConfigurationRead) & CMD_ENABLE_ & (ConfigAddress != StatusRegister) & BRST_ & !BC_BE3_ & !LFRAME_ & BRST_ |
| !LDEVSEL_ & (LocalAddress == DMAControlStatusReg) & (PCI_Command == IO_Read) & CMD_ENABLE_ & !BC_BE3_; |

"Equation to Generate Cache Line Size Shadow Register Write Strobe:"

!CLS_WRITE_ & !LFRAME_ & (LocalAddress == InterruptClear) & !LDEVSEL_ & BRST_ & (PCI_Command == IO_Write) & CMD_ENABLE_ & !BC_BE0_; |

"Equations For Setting, Clearing And Enabling Command Register Bits:"

COMMAND_B15_ := 0;
COMMAND_B15.CLK_ (ConfigAddress == CommandRegister) & !DATA_PHASE_ & !LDEVSEL_ & (PCI_Command == ConfigurationWrite) & !BC_BE1_;
COMMAND_B15.DO_ = (ConfigAddress == CommandRegister) & !LDEVSEL_ & (PCI_Command == ConfigurationRead) & !BC_BE1_;
COMMAND_B15.AR_ := 0;
COMMAND_B14_ := 0;
COMMAND_B14.CLK_ (ConfigAddress == CommandRegister) & !DATA_PHASE_ & !LDEVSEL_
*Equations For Generating Valid Command Register Latched Outputs:*

\[
\text{IOACC_ENABLE} := \text{COMMAND_B0.FB} \& \text{BRST}_\text{;} \]
\[
\text{IOACC_ENABLE.CLK} = \text{BCLK} ;
\]
\[
\text{IOACC_ENABLE.OE} = 1 ;
\]
\[
\text{IOACC_ENABLE.AR} = \text{BRST}_\text{;} 
\]
\[
\text{MASTER_ENABLE} := \text{COMMAND_B2.FB} \& \text{BRST}_\text{;} 
\]
\[
\text{MASTER_ENABLE.CLK} = \text{BCLK} ;
\]
\[
\text{MASTER_ENABLE.OE} = 1 ;
\]
\[
\text{MASTER_ENABLE.AR} = \text{BRST}_\text{;} 
\]
\[
\text{MEMWIV_ENABLE} := 0 ;
\]
\[
\text{MEMWIV_ENABLE.CLK} = \text{BCLK} ;
\]
\[
\text{MEMWIV_ENABLE.OE} = 1 ;
\]
\[
\text{MEMWIV_ENABLE.AR} = \text{BRST}_\text{;} 
\]
\[
\text{PARERR_ENABLE} := \text{COMMAND_B6.FB} \& \text{BRST}_\text{;} 
\]
\[
\text{PARERR_ENABLE.CLK} = \text{BCLK} ;
\]
\[
\text{PARERR_ENABLE.OE} = 1 ;
\]
\[
\text{PARERR_ENABLE.AR} = \text{BRST}_\text{;} 
\]

*Equations For Setting, Clearing And Enabling Status Register Bits:*

\[
\text{STATUS_B8} := \text{MPARER_RPT} \& \text{PCICommand == ConfigurationRead}
\]
\[
\# \text{BAD2} \& \text{PCICommand == ConfigurationWrite};
\]
\[
\text{STATUS_B8.CLK} = \text{PCICommand == ConfigurationRead} \& \text{!BC_BE3}_\text{;} 
\]
\[
\text{STATUS_B8.OE} = \text{PCICommand == ConfigurationWrite} \& \text{!BC_BE3}_\text{;} 
\]
\[
\text{STATUS_B8.AR} = \text{BRST}_\text{;} 
\]
STATUS_B8.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B8.AR  = BRST.

STATUS_B9  0;

STATUS_B9.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B10  = 1;

STATUS_B10.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B11 := STG_ABORT & (PCI_Command == ConfigurationRead) \& BAD27 & (PCI_Command == ConfigurationWrite);

STATUS_B11.CLK  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3
& STATUS_STROBE
& (PCI_Command == ConfigurationWrite) \& BC_BE3;

STATUS_B11.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B11.AR  = BRST.

STATUS_B12 := STG_ABORT & (PCI_Command == ConfigurationRead) \& BAD28 & (PCI_Command == ConfigurationWrite);

STATUS_B12.CLK  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3
& STATUS_STROBE
& (PCI_Command == ConfigurationWrite) \& BC_BE3.

STATUS_B12.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B12.AR  = BRST.

STATUS_B13 := MAXST_ABORT & (PCI_Command == ConfigurationRead) \& BAD19 & (PCI_Command == ConfigurationWrite);

STATUS_B13.CLK  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3
& STATUS_STROBE
& (PCI_Command == ConfigurationWrite) \& BC_BE3;

STATUS_B13.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B13.AR  = BRST.

STATUS_B14 := MSER & (PCI_Command == ConfigurationRead) \& BAD10 & (PCI_Command == ConfigurationWrite);

STATUS_B14.CLK  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3
& STATUS_STROBE
& (PCI_Command == ConfigurationWrite) \& BC_BE3;

STATUS_B14.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B14.AR  = BRST.

STATUS_B15 := ARER & (PCI_Command == ConfigurationRead) \& BAD11 & (PCI_Command == ConfigurationWrite);

STATUS_B15.CLK  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3
& STATUS_STROBE
& (PCI_Command == ConfigurationWrite) \& BC_BE3;

STATUS_B15.OE  STATUS_STROBE
& (PCI_Command == ConfigurationRead) \& BC_BE3.

STATUS_B15.AR  = BRST.

STATUS_STROBE (PCI_Command == ConfigurationRead) \& ICE/SE-
& (ConfigAddress == StatusRegister);
CMDO.CLK  :CLK66;
CMDO.OE  1;

/*
Equations for Demultiplexing some Local Address Bits:
*/
LA7 := BAD7;
LA7.CLK ADDR_CLOCK;
LA7.OE  1;

LA6 := BAD6;
LA6.CLK ADDR_CLOCK;
LA6.OE  1;

LA5 := BAD5;
LA5.CLK ADDR_CLOCK;
LA5.OE  1;

LA4 := BAD4;
LA4.CLK ADDR_CLOCK;
LA4.OE  1;

LA3 := BAD3;
LA3.CLK ADDR_CLOCK;
LA3.OE  1;

LA2 := BAD2;
LA2.CLK ADDR_CLOCK;
LA2.OE  1;

/* Latency Timer Truth Table */
/* Output Enable Equations added for Simulation Purposes */
LTCD7.OE = SIMULATE;
LTCD6.OE = SIMULATE;
LTCD5.OE = SIMULATE;
LTCD4.OE = SIMULATE;
LTCD3.OE = SIMULATE;
LTCD2.OE = SIMULATE;
LTCD1.OE = SIMULATE;
LTCD0.OE = SIMULATE;

state_diagram LTimerLo

State 15:
  if !MASTER then 14;
    else if MASTER & (LTREG_OUT_LO=0) then 0
    else if MASTER & (LTREG_OUT_LO=1) then 1
    else if MASTER & (LTREG_OUT_LO=2) then 2
    else if MASTER & (LTREG_OUT_LO=3) then 3
    else if MASTER & (LTREG_OUT_LO=4) then 4
    else if MASTER & (LTREG_OUT_LO=5) then 5
    else if MASTER & (LTREG_OUT_LO=6) then 6
    else if MASTER & (LTREG_OUT_LO=7) then 7
    else if MASTER & (LTREG_OUT_LO=8) then 8
    else if MASTER & (LTREG_OUT_LO=9) then 9
    else if MASTER & (LTREG_OUT_LO=10) then 10
    else if MASTER & (LTREG_OUT_LO=11) then 11
    else if MASTER & (LTREG_OUT_LO=12) then 12
    else if MASTER & (LTREG_OUT_LO=13) then 13
    else if MASTER & (LTREG_OUT_LO=14) then 14
    else 15;

State 14:
  if !MASTER then 13;
    else if MASTER & (LTREG_OUT_LO=0) then 0
    else if MASTER & (LTREG_OUT_LO=1) then 1
    else if MASTER & (LTREG_OUT_LO=2) then 2
    else if MASTER & (LTREG_OUT_LO=3) then 3
    else if MASTER & (LTREG_OUT_LO=4) then 4
    else if MASTER & (LTREG_OUT_LO=5) then 5
    else if MASTER & (LTREG_OUT_LO=6) then 6
    else if MASTER & (LTREG_OUT_LO=7) then 7
    else if MASTER & (LTREG_OUT_LO=8) then 8
    else if MASTER & (LTREG_OUT_LO=9) then 9
    else if MASTER & (LTREG_OUT_LO=10) then 10
    else if MASTER & (LTREG_OUT_LO=11) then 11
    else if MASTER & (LTREG_OUT_LO=12) then 12
    else if MASTER & (LTREG_OUT_LO=13) then 13
    else if MASTER & (LTREG_OUT_LO=15) then 15
    else 14;

State 13:
  if !MASTER then 12;
    else if MASTER & (LTREG_OUT_LO=0) then 0
else if MASTER_6 (LITREG_OUT_LO==1) then 1
else if MASTER_6 (LITREG_OUT_LO==2) then 2
else if MASTER_6 (LITREG_OUT_LO==3) then 3
else if MASTER_6 (LITREG_OUT_LO==4) then 4
else if MASTER_6 (LITREG_OUT_LO==5) then 5
else if MASTER_6 (LITREG_OUT_LO==6) then 6
else if MASTER_6 (LITREG_OUT_LO==7) then 7
else if MASTER_6 (LITREG_OUT_LO==8) then 8
else if MASTER_6 (LITREG_OUT_LO==9) then 9
else if MASTER_6 (LITREG_OUT_LO==10) then 10
else if MASTER_6 (LITREG_OUT_LO==11) then 11
else if MASTER_6 (LITREG_OUT_LO==12) then 12
else if MASTER_6 (LITREG_OUT_LO==14) then 14
else if MASTER_6 (LITREG_OUT_LO==15) then 15
else 13;

State 12:
if !MASTER_ then 11:
else if MASTER_6 (LITREG_OUT_LO==0) then 0
else if MASTER_6 (LITREG_OUT_LO==1) then 1
else if MASTER_6 (LITREG_OUT_LO==2) then 2
else if MASTER_6 (LITREG_OUT_LO==3) then 3
else if MASTER_6 (LITREG_OUT_LO==4) then 4
else if MASTER_6 (LITREG_OUT_LO==5) then 5
else if MASTER_6 (LITREG_OUT_LO==6) then 6
else if MASTER_6 (LITREG_OUT_LO==7) then 7
else if MASTER_6 (LITREG_OUT_LO==8) then 8
else if MASTER_6 (LITREG_OUT_LO==9) then 9
else if MASTER_6 (LITREG_OUT_LO==10) then 10
else if MASTER_6 (LITREG_OUT_LO==11) then 11
else if MASTER_6 (LITREG_OUT_LO==13) then 13
else if MASTER_6 (LITREG_OUT_LO==14) then 14
else if MASTER_6 (LITREG_OUT_LO==15) then 15
else 12;

State 11:
if !MASTER_ then 10:
else if MASTER_6 (LITREG_OUT_LO==0) then 0
else if MASTER_6 (LITREG_OUT_LO==1) then 1
else if MASTER_6 (LITREG_OUT_LO==2) then 2
else if MASTER_6 (LITREG_OUT_LO==3) then 3
else if MASTER_6 (LITREG_OUT_LO==4) then 4
else if MASTER_6 (LITREG_OUT_LO==5) then 5
else if MASTER_6 (LITREG_OUT_LO==6) then 6
else if MASTER_6 (LITREG_OUT_LO==7) then 7
else if MASTER_6 (LITREG_OUT_LO==8) then 8
else if MASTER_6 (LITREG_OUT_LO==9) then 9
else if MASTER_6 (LITREG_OUT_LO==10) then 10
else if MASTER_6 (LITREG_OUT_LO==12) then 12
else if MASTER_6 (LITREG_OUT_LO==13) then 13
else if MASTER_6 (LITREG_OUT_LO==14) then 14
else if MASTER_6 (LITREG_OUT_LO==15) then 15
else 11;

State 10:
if !MASTER_ then 9:
else if MASTER_6 (LITREG_OUT_LO==0) then 0
else if MASTER_6 (LITREG_OUT_LO==1) then 1
else if MASTER_6 (LITREG_OUT_LO==2) then 2
else if MASTER_6 (LITREG_OUT_LO==3) then 3
else if MASTER_6 (LITREG_OUT_LO==4) then 4
else if MASTER_6 (LITREG_OUT_LO==5) then 5
else if MASTER_6 (LITREG_OUT_LO==6) then 6
else if MASTER_6 (LITREG_OUT_LO==7) then 7
else if MASTER_6 (LITREG_OUT_LO==8) then 8
else if MASTER_6 (LITREG_OUT_LO==9) then 9
else if MASTER_6 (LITREG_OUT_LO==11) then 11
else if MASTER_6 (LITREG_OUT_LO==12) then 12
else if MASTER_6 (LITREG_OUT_LO==13) then 13
else if MASTER_6 (LITREG_OUT_LO==14) then 14
else if MASTER_6 (LITREG_OUT_LO==15) then 15
else 10;

State 9:
if !MASTER_ then 8:
else if MASTER_6 (LITREG_OUT_LO==0) then 0
else if MASTER_6 (LITREG_OUT_LO==1) then 1
else if MASTER_6 (LITREG_OUT_LO==2) then 2
else if MASTER_6 (LITREG_OUT_LO==3) then 3
else if MASTER_6 (LITREG_OUT_LO==4) then 4
else if MASTER_6 (LITREG_OUT_LO==5) then 5
else if MASTER_6 (LITREG_OUT_LO==6) then 6
else if MASTER_6 (LITREG_OUT_LO==7) then 7
else if MASTER_6 (LITREG_OUT_LO==8) then 8
else if MASTER_6 (LITREG_OUT_LO==10) then 10
else if MASTER_6 (LITREG_OUT_LO==11) then 11
else if MASTER_6 (LITREG_OUT_LO==12) then 12
else if MASTER_6 (LITREG_OUT_LO==13) then 13
else 10;

87
else if MASTER_ < (LREG_OUT_LO=14) then 14
else if MASTER_ < (LREG_OUT_LO=15) then 15
else 9;

State 8:
if MASTER_ then 7:
else if MASTER_ < (LREG_OUT_LO=0) then 0
else if MASTER_ < (LREG_OUT_LO=1) then 1
else if MASTER_ < (LREG_OUT_LO=2) then 2
else if MASTER_ < (LREG_OUT_LO=3) then 3
else if MASTER_ < (LREG_OUT_LO=4) then 4
else if MASTER_ < (LREG_OUT_LO=5) then 5
else if MASTER_ < (LREG_OUT_LO=6) then 6
else if MASTER_ < (LREG_OUT_LO=7) then 7
else if MASTER_ < (LREG_OUT_LO=9) then 9
else if MASTER_ < (LREG_OUT_LO=10) then 10
else if MASTER_ < (LREG_OUT_LO=11) then 11
else if MASTER_ < (LREG_OUT_LO=12) then 12
else if MASTER_ < (LREG_OUT_LO=13) then 13
else if MASTER_ < (LREG_OUT_LO=14) then 14
else if MASTER_ < (LREG_OUT_LO=15) then 15
else 8;

State 7:
if MASTER_ then 6:
else if MASTER_ < (LREG_OUT_LO=0) then 0
else if MASTER_ < (LREG_OUT_LO=1) then 1
else if MASTER_ < (LREG_OUT_LO=2) then 2
else if MASTER_ < (LREG_OUT_LO=3) then 3
else if MASTER_ < (LREG_OUT_LO=4) then 4
else if MASTER_ < (LREG_OUT_LO=5) then 5
else if MASTER_ < (LREG_OUT_LO=6) then 6
else if MASTER_ < (LREG_OUT_LO=8) then 8
else if MASTER_ < (LREG_OUT_LO=9) then 9
else if MASTER_ < (LREG_OUT_LO=10) then 10
else if MASTER_ < (LREG_OUT_LO=11) then 11
else if MASTER_ < (LREG_OUT_LO=12) then 12
else if MASTER_ < (LREG_OUT_LO=13) then 13
else if MASTER_ < (LREG_OUT_LO=14) then 14
else if MASTER_ < (LREG_OUT_LO=15) then 15
else 7;

State 6:
if MASTER_ then 5:
else if MASTER_ < (LREG_OUT_LO=0) then 0
else if MASTER_ < (LREG_OUT_LO=1) then 1
else if MASTER_ < (LREG_OUT_LO=2) then 2
else if MASTER_ < (LREG_OUT_LO=3) then 3
else if MASTER_ < (LREG_OUT_LO=4) then 4
else if MASTER_ < (LREG_OUT_LO=5) then 5
else if MASTER_ < (LREG_OUT_LO=8) then 8
else if MASTER_ < (LREG_OUT_LO=9) then 9
else if MASTER_ < (LREG_OUT_LO=10) then 10
else if MASTER_ < (LREG_OUT_LO=11) then 11
else if MASTER_ < (LREG_OUT_LO=12) then 12
else if MASTER_ < (LREG_OUT_LO=13) then 13
else if MASTER_ < (LREG_OUT_LO=14) then 14
else if MASTER_ < (LREG_OUT_LO=15) then 15
else 6;

State 5:
if MASTER_ then 4:
else if MASTER_ < (LREG_OUT_LO=0) then 0
else if MASTER_ < (LREG_OUT_LO=1) then 1
else if MASTER_ < (LREG_OUT_LO=2) then 2
else if MASTER_ < (LREG_OUT_LO=3) then 3
else if MASTER_ < (LREG_OUT_LO=4) then 4
else if MASTER_ < (LREG_OUT_LO=6) then 6
else if MASTER_ < (LREG_OUT_LO=7) then 7
else if MASTER_ < (LREG_OUT_LO=8) then 8
else if MASTER_ < (LREG_OUT_LO=9) then 9
else if MASTER_ < (LREG_OUT_LO=10) then 10
else if MASTER_ < (LREG_OUT_LO=11) then 11
else if MASTER_ < (LREG_OUT_LO=12) then 12
else if MASTER_ < (LREG_OUT_LO=13) then 13
else if MASTER_ < (LREG_OUT_LO=14) then 14
else if MASTER_ < (LREG_OUT_LO=15) then 15
else 5;

State 4:
if MASTER_ then 3:
else if MASTER_ < (LREG_OUT_LO=0) then 0
else if MASTER_ < (LREG_OUT_LO=1) then 1
else if MASTER_ < (LREG_OUT_LO=2) then 2
else if MASTER_ < (LREG_OUT_LO=3) then 3
else if MASTER_ < (LREG_OUT_LO=5) then 5
else if MASTER_ < (LREG_OUT_LO=6) then 6
else if MASTER_ < (LREG_OUT_LO=7) then 7
else if MASTER_ 4 (LTREG_OUT_LOW=8) then 8
else if MASTER_ 6 (LTREG_OUT_LOW=9) then 9
else if MASTER_ 4 (LTREG_OUT_LOW=10) then 10
else if MASTER_ 6 (LTREG_OUT_LOW=11) then 11
else if MASTER_ 4 (LTREG_OUT_LOW=12) then 12
else if MASTER_ 6 (LTREG_OUT_LOW=13) then 13
else if MASTER_ 4 (LTREG_OUT_LOW=14) then 14
else if MASTER_ 6 (LTREG_OUT_LOW=15) then 15
else 4;

State 3:
if MASTER then 2;
else if MASTER_ 4 (LTREG_OUT_LOW=0) then 0
else if MASTER_ 6 (LTREG_OUT_LOW=1) then 1
else if MASTER_ 4 (LTREG_OUT_LOW=2) then 2
else if MASTER_ 6 (LTREG_OUT_LOW=4) then 4
else if MASTER_ 4 (LTREG_OUT_LOW=5) then 5
else if MASTER_ 6 (LTREG_OUT_LOW=7) then 7
else if MASTER_ 4 (LTREG_OUT_LOW=8) then 8
else if MASTER_ 6 (LTREG_OUT_LOW=9) then 9
else if MASTER_ 4 (LTREG_OUT_LOW=10) then 10
else if MASTER_ 6 (LTREG_OUT_LOW=11) then 11
else if MASTER_ 4 (LTREG_OUT_LOW=12) then 12
else if MASTER_ 6 (LTREG_OUT_LOW=13) then 13
else if MASTER_ 4 (LTREG_OUT_LOW=14) then 14
else if MASTER_ 6 (LTREG_OUT_LOW=15) then 15
else 3;

State 2:
if MASTER then 1;
else if MASTER_ 4 (LTREG_OUT_LOW=0) then 0
else if MASTER_ 6 (LTREG_OUT_LOW=1) then 1
else if MASTER_ 4 (LTREG_OUT_LOW=3) then 3
else if MASTER_ 6 (LTREG_OUT_LOW=4) then 4
else if MASTER_ 4 (LTREG_OUT_LOW=5) then 5
else if MASTER_ 6 (LTREG_OUT_LOW=6) then 6
else if MASTER_ 4 (LTREG_OUT_LOW=7) then 7
else if MASTER_ 6 (LTREG_OUT_LOW=8) then 8
else if MASTER_ 4 (LTREG_OUT_LOW=9) then 9
else if MASTER_ 6 (LTREG_OUT_LOW=10) then 10
else if MASTER_ 4 (LTREG_OUT_LOW=11) then 11
else if MASTER_ 6 (LTREG_OUT_LOW=12) then 12
else if MASTER_ 6 (LTREG_OUT_LOW=13) then 13
else if MASTER_ 4 (LTREG_OUT_LOW=14) then 14
else if MASTER_ 6 (LTREG_OUT_LOW=15) then 15
else 2;

State 1:
if MASTER then 0;
else if MASTER_ 4 (LTREG_OUT_LOW=0) then 0
else if MASTER_ 6 (LTREG_OUT_LOW=2) then 2
else if MASTER_ 4 (LTREG_OUT_LOW=3) then 3
else if MASTER_ 6 (LTREG_OUT_LOW=4) then 4
else if MASTER_ 4 (LTREG_OUT_LOW=5) then 5
else if MASTER_ 6 (LTREG_OUT_LOW=6) then 6
else if MASTER_ 4 (LTREG_OUT_LOW=7) then 7
else if MASTER_ 6 (LTREG_OUT_LOW=8) then 8
else if MASTER_ 4 (LTREG_OUT_LOW=9) then 9
else if MASTER_ 6 (LTREG_OUT_LOW=10) then 10
else if MASTER_ 6 (LTREG_OUT_LOW=11) then 11
else if MASTER_ 4 (LTREG_OUT_LOW=12) then 12
else if MASTER_ 6 (LTREG_OUT_LOW=13) then 13
else if MASTER_ 4 (LTREG_OUT_LOW=14) then 14
else if MASTER_ 6 (LTREG_OUT_LOW=15) then 15
else 1;

State 0:
if MASTER then 15
else if MASTER_ 4 (LTREG_OUT_LOW=1) then 1
else if MASTER_ 6 (LTREG_OUT_LOW=2) then 2
else if MASTER_ 4 (LTREG_OUT_LOW=3) then 3
else if MASTER_ 6 (LTREG_OUT_LOW=4) then 4
else if MASTER_ 4 (LTREG_OUT_LOW=5) then 5
else if MASTER_ 6 (LTREG_OUT_LOW=6) then 6
else if MASTER_ 4 (LTREG_OUT_LOW=7) then 7
else if MASTER_ 6 (LTREG_OUT_LOW=8) then 8
else if MASTER_ 4 (LTREG_OUT_LOW=9) then 9
else if MASTER_ 6 (LTREG_OUT_LOW=10) then 10
else if MASTER_ 6 (LTREG_OUT_LOW=11) then 11
else if MASTER_ 4 (LTREG_OUT_LOW=12) then 12
else if MASTER_ 6 (LTREG_OUT_LOW=13) then 13
else if MASTER_ 4 (LTREG_OUT_LOW=14) then 14
else if MASTER_ 6 (LTREG_OUT_LOW=15) then 15
else 0;
State diagram LTimerHi

State 15:
- If (LTimerLo==0) & MASTER then 14
  - else if MASTER & (LTREG_OUT_HI==0) then 0
  - else if MASTER & (LTREG_OUT_HI==1) then 1
  - else if MASTER & (LTREG_OUT_HI==2) then 2
  - else if MASTER & (LTREG_OUT_HI==3) then 3
  - else if MASTER & (LTREG_OUT_HI==4) then 4
  - else if MASTER & (LTREG_OUT_HI==5) then 5
  - else if MASTER & (LTREG_OUT_HI==6) then 6
  - else if MASTER & (LTREG_OUT_HI==7) then 7
  - else if MASTER & (LTREG_OUT_HI==8) then 8
  - else if MASTER & (LTREG_OUT_HI==9) then 9
  - else if MASTER & (LTREG_OUT_HI==10) then 10
  - else if MASTER & (LTREG_OUT_HI==11) then 11
  - else if MASTER & (LTREG_OUT_HI==12) then 12
  - else if MASTER & (LTREG_OUT_HI==13) then 13
  - else if MASTER & (LTREG_OUT_HI==14) then 14
  - else 15;

State 14:
- If (LTimerLo==0) & MASTER then 13
  - else if MASTER & (LTREG_OUT_HI==0) then 0
  - else if MASTER & (LTREG_OUT_HI==1) then 1
  - else if MASTER & (LTREG_OUT_HI==2) then 2
  - else if MASTER & (LTREG_OUT_HI==3) then 3
  - else if MASTER & (LTREG_OUT_HI==4) then 4
  - else if MASTER & (LTREG_OUT_HI==5) then 5
  - else if MASTER & (LTREG_OUT_HI==6) then 6
  - else if MASTER & (LTREG_OUT_HI==7) then 7
  - else if MASTER & (LTREG_OUT_HI==8) then 8
  - else if MASTER & (LTREG_OUT_HI==9) then 9
  - else if MASTER & (LTREG_OUT_HI==10) then 10
  - else if MASTER & (LTREG_OUT_HI==11) then 11
  - else if MASTER & (LTREG_OUT_HI==12) then 12
  - else if MASTER & (LTREG_OUT_HI==13) then 13
  - else if MASTER & (LTREG_OUT_HI==14) then 14
  - else 14;

State 13:
- If (LTimerLo==0) & MASTER then 12
  - else if MASTER & (LTREG_OUT_HI==0) then 0
  - else if MASTER & (LTREG_OUT_HI==1) then 1
  - else if MASTER & (LTREG_OUT_HI==2) then 2
  - else if MASTER & (LTREG_OUT_HI==3) then 3
  - else if MASTER & (LTREG_OUT_HI==4) then 4
  - else if MASTER & (LTREG_OUT_HI==5) then 5
  - else if MASTER & (LTREG_OUT_HI==6) then 6
  - else if MASTER & (LTREG_OUT_HI==7) then 7
  - else if MASTER & (LTREG_OUT_HI==8) then 8
  - else if MASTER & (LTREG_OUT_HI==9) then 9
  - else if MASTER & (LTREG_OUT_HI==10) then 10
  - else if MASTER & (LTREG_OUT_HI==11) then 11
  - else if MASTER & (LTREG_OUT_HI==12) then 12
  - else if MASTER & (LTREG_OUT_HI==13) then 13
  - else if MASTER & (LTREG_OUT_HI==14) then 14
  - else 13;

State 12:
- If (LTimerLo==0) & MASTER then 11
  - else if MASTER & (LTREG_OUT_HI==0) then 0
  - else if MASTER & (LTREG_OUT_HI==1) then 1
  - else if MASTER & (LTREG_OUT_HI==2) then 2
  - else if MASTER & (LTREG_OUT_HI==3) then 3
  - else if MASTER & (LTREG_OUT_HI==4) then 4
  - else if MASTER & (LTREG_OUT_HI==5) then 5
  - else if MASTER & (LTREG_OUT_HI==6) then 6
  - else if MASTER & (LTREG_OUT_HI==7) then 7
  - else if MASTER & (LTREG_OUT_HI==8) then 8
  - else if MASTER & (LTREG_OUT_HI==9) then 9
  - else if MASTER & (LTREG_OUT_HI==10) then 10
  - else if MASTER & (LTREG_OUT_HI==11) then 11
  - else if MASTER & (LTREG_OUT_HI==12) then 12
  - else if MASTER & (LTREG_OUT_HI==13) then 13
  - else if MASTER & (LTREG_OUT_HI==14) then 14
  - else 12;

State 11:
- If (LTimerLo==0) & MASTER then 10
  - else if MASTER & (LTREG_OUT_HI==0) then 0
  - else if MASTER & (LTREG_OUT_HI==1) then 1
  - else if MASTER & (LTREG_OUT_HI==2) then 2
  - else if MASTER & (LTREG_OUT_HI==3) then 3
  - else if MASTER & (LTREG_OUT_HI==4) then 4
  - else if MASTER & (LTREG_OUT_HI==5) then 5
  - else if MASTER & (LTREG_OUT_HI==6) then 6
  - else if MASTER & (LTREG_OUT_HI==7) then 7
  - else if MASTER & (LTREG_OUT_HI==8) then 8
  - else if MASTER & (LTREG_OUT_HI==9) then 9
  - else if MASTER & (LTREG_OUT_HI==10) then 10
  - else if MASTER & (LTREG_OUT_HI==11) then 11
  - else if MASTER & (LTREG_OUT_HI==12) then 12
  - else if MASTER & (LTREG_OUT_HI==13) then 13
  - else if MASTER & (LTREG_OUT_HI==14) then 14
  - else 11;
else if MASTER_6 (LTREG_OUT_HI==9) then 9
else if MASTER_6 (LTREG_OUT_HI==10) then 10
else if MASTER_6 (LTREG_OUT_HI==12) then 12
else if MASTER_6 (LTREG_OUT_HI==13) then 13
else if MASTER_6 (LTREG_OUT_HI==14) then 14
else if MASTER_6 (LTREG_OUT_HI==15) then 15
else 11;

State 10:
If (LTimerLo==0) & MASTER then 9
else if MASTER_6 (LTREG_OUT_HI==0) then 0
else if MASTER_6 (LTREG_OUT_HI==1) then 1
else if MASTER_6 (LTREG_OUT_HI==2) then 2
else if MASTER_6 (LTREG_OUT_HI==3) then 3
else if MASTER_6 (LTREG_OUT_HI==4) then 4
else if MASTER_6 (LTREG_OUT_HI==5) then 5
else if MASTER_6 (LTREG_OUT_HI==6) then 6
else if MASTER_6 (LTREG_OUT_HI==7) then 7
else if MASTER_6 (LTREG_OUT_HI==8) then 8
else if MASTER_6 (LTREG_OUT_HI==9) then 9
else if MASTER_6 (LTREG_OUT_HI==11) then 11
else if MASTER_6 (LTREG_OUT_HI==12) then 12
else if MASTER_6 (LTREG_OUT_HI==13) then 13
else if MASTER_6 (LTREG_OUT_HI==14) then 14
else if MASTER_6 (LTREG_OUT_HI==15) then 15
else 10;

State 9:
If (LTimerLo==0) & MASTER then 8
else if MASTER_6 (LTREG_OUT_HI==0) then 0
else if MASTER_6 (LTREG_OUT_HI==1) then 1
else if MASTER_6 (LTREG_OUT_HI==2) then 2
else if MASTER_6 (LTREG_OUT_HI==3) then 3
else if MASTER_6 (LTREG_OUT_HI==4) then 4
else if MASTER_6 (LTREG_OUT_HI==5) then 5
else if MASTER_6 (LTREG_OUT_HI==6) then 6
else if MASTER_6 (LTREG_OUT_HI==7) then 7
else if MASTER_6 (LTREG_OUT_HI==8) then 8
else if MASTER_6 (LTREG_OUT_HI==10) then 10
else if MASTER_6 (LTREG_OUT_HI==11) then 11
else if MASTER_6 (LTREG_OUT_HI==12) then 12
else if MASTER_6 (LTREG_OUT_HI==13) then 13
else if MASTER_6 (LTREG_OUT_HI==14) then 14
else if MASTER_6 (LTREG_OUT_HI==15) then 15
else 9;

State 8:
If (LTimerLo==0) & MASTER then 7
else if MASTER_6 (LTREG_OUT_HI==0) then 0
else if MASTER_6 (LTREG_OUT_HI==1) then 1
else if MASTER_6 (LTREG_OUT_HI==2) then 2
else if MASTER_6 (LTREG_OUT_HI==3) then 3
else if MASTER_6 (LTREG_OUT_HI==4) then 4
else if MASTER_6 (LTREG_OUT_HI==5) then 5
else if MASTER_6 (LTREG_OUT_HI==6) then 6
else if MASTER_6 (LTREG_OUT_HI==7) then 7
else if MASTER_6 (LTREG_OUT_HI==9) then 9
else if MASTER_6 (LTREG_OUT_HI==10) then 10
else if MASTER_6 (LTREG_OUT_HI==11) then 11
else if MASTER_6 (LTREG_OUT_HI==12) then 12
else if MASTER_6 (LTREG_OUT_HI==13) then 13
else if MASTER_6 (LTREG_OUT_HI==14) then 14
else if MASTER_6 (LTREG_OUT_HI==15) then 15
else 8;

State 7:
If (LTimerLo==0) & MASTER then 6
else if MASTER_6 (LTREG_OUT_HI==0) then 0
else if MASTER_6 (LTREG_OUT_HI==1) then 1
else if MASTER_6 (LTREG_OUT_HI==2) then 2
else if MASTER_6 (LTREG_OUT_HI==3) then 3
else if MASTER_6 (LTREG_OUT_HI==4) then 4
else if MASTER_6 (LTREG_OUT_HI==5) then 5
else if MASTER_6 (LTREG_OUT_HI==6) then 6
else if MASTER_6 (LTREG_OUT_HI==8) then 8
else if MASTER_6 (LTREG_OUT_HI==9) then 9
else if MASTER_6 (LTREG_OUT_HI==10) then 10
else if MASTER_6 (LTREG_OUT_HI==11) then 11
else if MASTER_6 (LTREG_OUT_HI==12) then 12
else if MASTER_6 (LTREG_OUT_HI==13) then 13
else if MASTER_6 (LTREG_OUT_HI==14) then 14
else if MASTER_6 (LTREG_OUT_HI==15) then 15
else 7;

State 6:
If (LTimerLo==0) & MASTER then 5
else if MASTER_6 (LTREG_OUT_HI==0) then 0
else if MASTER_6 (LTREG_OUT_HI==1) then 1
else if MASTER_6 (LTREG_OUT_HI==2) then 2
else if MASTER_ & (LTREG_OUT_HI==3) then 3
else if MASTER_ & (LTREG_OUT_HI==4) then 4
else if MASTER_ & (LTREG_OUT_HI==5) then 5
else if MASTER_ & (LTREG_OUT_HI==7) then 7
else if MASTER_ & (LTREG_OUT_HI==8) then 8
else if MASTER_ & (LTREG_OUT_HI==9) then 9
else if MASTER_ & (LTREG_OUT_HI==10) then 10
else if MASTER_ & (LTREG_OUT_HI==11) then 11
else if MASTER_ & (LTREG_OUT_HI==12) then 12
else if MASTER_ & (LTREG_OUT_HI==13) then 13
else if MASTER_ & (LTREG_OUT_HI==14) then 14
else if MASTER_ & (LTREG_OUT_HI==15) then 15
else 6;

State 5:
If (LTimerLo==0) & !MASTER_ then 4
else if MASTER_ & (LTREG_OUT_HI==0) then 0
else if MASTER_ & (LTREG_OUT_HI==1) then 1
else if MASTER_ & (LTREG_OUT_HI==2) then 2
else if MASTER_ & (LTREG_OUT_HI==3) then 3
else if MASTER_ & (LTREG_OUT_HI==5) then 5
else if MASTER_ & (LTREG_OUT_HI==6) then 6
else if MASTER_ & (LTREG_OUT_HI==7) then 7
else if MASTER_ & (LTREG_OUT_HI==8) then 8
else if MASTER_ & (LTREG_OUT_HI==9) then 9
else if MASTER_ & (LTREG_OUT_HI==10) then 10
else if MASTER_ & (LTREG_OUT_HI==11) then 11
else if MASTER_ & (LTREG_OUT_HI==12) then 12
else if MASTER_ & (LTREG_OUT_HI==13) then 13
else if MASTER_ & (LTREG_OUT_HI==14) then 14
else if MASTER_ & (LTREG_OUT_HI==15) then 15
else 5;

State 4:
If (LTimerLo==0) & !MASTER_ then 3
else if MASTER_ & (LTREG_OUT_HI==0) then 0
else if MASTER_ & (LTREG_OUT_HI==1) then 1
else if MASTER_ & (LTREG_OUT_HI==2) then 2
else if MASTER_ & (LTREG_OUT_HI==3) then 3
else if MASTER_ & (LTREG_OUT_HI==5) then 5
else if MASTER_ & (LTREG_OUT_HI==6) then 6
else if MASTER_ & (LTREG_OUT_HI==7) then 7
else if MASTER_ & (LTREG_OUT_HI==8) then 8
else if MASTER_ & (LTREG_OUT_HI==9) then 9
else if MASTER_ & (LTREG_OUT_HI==10) then 10
else if MASTER_ & (LTREG_OUT_HI==11) then 11
else if MASTER_ & (LTREG_OUT_HI==12) then 12
else if MASTER_ & (LTREG_OUT_HI==13) then 13
else if MASTER_ & (LTREG_OUT_HI==14) then 14
else if MASTER_ & (LTREG_OUT_HI==15) then 15
else 4;

State 3:
If (LTimerLo==0) & !MASTER_ then 2
else if MASTER_ & (LTREG_OUT_HI==0) then 0
else if MASTER_ & (LTREG_OUT_HI==1) then 1
else if MASTER_ & (LTREG_OUT_HI==2) then 2
else if MASTER_ & (LTREG_OUT_HI==4) then 4
else if MASTER_ & (LTREG_OUT_HI==5) then 5
else if MASTER_ & (LTREG_OUT_HI==6) then 6
else if MASTER_ & (LTREG_OUT_HI==7) then 7
else if MASTER_ & (LTREG_OUT_HI==8) then 8
else if MASTER_ & (LTREG_OUT_HI==9) then 9
else if MASTER_ & (LTREG_OUT_HI==10) then 10
else if MASTER_ & (LTREG_OUT_HI==11) then 11
else if MASTER_ & (LTREG_OUT_HI==12) then 12
else if MASTER_ & (LTREG_OUT_HI==13) then 13
else if MASTER_ & (LTREG_OUT_HI==14) then 14
else if MASTER_ & (LTREG_OUT_HI==15) then 15
else 3;

State 2:
If (LTimerLo==0) & !MASTER_ then 1
else if MASTER_ & (LTREG_OUT_HI==0) then 0
else if MASTER_ & (LTREG_OUT_HI==1) then 1
else if MASTER_ & (LTREG_OUT_HI=3) then 3
else if MASTER_ & (LTREG_OUT_HI=4) then 4
else if MASTER_ & (LTREG_OUT_HI=5) then 5
else if MASTER_ & (LTREG_OUT_HI=6) then 6
else if MASTER_ & (LTREG_OUT_HI=7) then 7
else if MASTER_ & (LTREG_OUT_HI=8) then 8
else if MASTER_ & (LTREG_OUT_HI=9) then 9
else if MASTER_ & (LTREG_OUT_HI=10) then 10
else if MASTER_ & (LTREG_OUT_HI=11) then 11
else if MASTER_ & (LTREG_OUT_HI=12) then 12
else if MASTER_ & (LTREG_OUT_HI=13) then 13
else if MASTER_ & (LTREG_OUT_HI=14) then 14
else if MASTER_ & (LTREG_OUT_HI=15) then 15
else 2:

State 1:

If (LTimerLo==0) & MASTER then 0
else if MASTER & (LTPEG_OUT_HI==0) then 0
else if MASTER & (LTREG_OUT_HI==2) then 2
else if MASTER & (LTREG_OUT_HI==3) then 3
else if MASTER & (LTREG_OUT_HI==4) then 4
else if MASTER & (LTREG_OUT_HI==5) then 5
else if MASTER & (LTREG_OUT_HI==6) then 6
else if MASTER & (LTREG_OUT_HI==7) then 7
else if MASTER & (LTPEG_OUT_HI=8) then 8
else if MASTER & (LTPEG_OUT_HI=9) then 9
else if MASTER & (LTREG_OUT_HI==10) then 10
else if MASTER & (LTREG_OUT_HI==11) then 11
else if MASTER & (LTREG_OUT_HI==12) then 12
else if MASTER & (LTPEG_OUT_HI==13) then 13
else if MASTER & (LTPEG_OUT_HI==14) then 14
else if MASTER & (LTPEG_OUT_HI==15) then 15
else 1;

State 0:

If :MASTER then 0
else if MASTER & (LTREG_OUT_HI==1) then 1
else if MASTER & (LTREG_OUT_HI==2) then 2
else if MASTER & (LTREG_OUT_HI==3) then 3
else if MASTER & (LTREG_OUT_HI==4) then 4
else if MASTER & (LTREG_OUT_HI==5) then 5
else if MASTER & (LTREG_OUT_HI==6) then 6
else if MASTER & (LTPEG_OUT_HI=7) then 7
else if MASTER & (LTPEG_OUT_HI=8) then 8
else if MASTER & (LTREG_OUT_HI==9) then 9
else if MASTER & (LTREG_OUT_HI==10) then 10
else if MASTER & (LTPEG_OUT_HI==11) then 11
else if MASTER & (LTPEG_OUT_HI==12) then 12
else if MASTER & (LTPEG_OUT_HI==13) then 13
else if MASTER & (LTPEG_OUT_HI=14) then 14
else if MASTER & (LTPEG_OUT_HI=15) then 15
else 0;

end DECODE
5.3.4. PARIO PLD

module PARIO flag-'r3'
title 'Parallel I/O Interface Controller for PCI Based PCI I/O board'
Sheet 6 of Schematic Diagrams

* This PLD implements the following functionality on the PCI Parallel
* Interface Add-In Board:

* Implements the Parallel Interface State Machine which controls data access to the
* 4k byte Input/Output FIFO Buffers. The Input/Output FIFO Buffers are 32-Bits
* wide on the PCI Interface Side and 8-Bits wide on the I/O Side.
* This PLD provides fourteen (14) Control pins which may be defined by the user. This
* enables the designer to implement a complete custom interface
* without adding any additional devices other than the required drivers for the
* interface selected. In some cases the signal CONTROLx is inserted into equations to
* indicate that one or more of the interface control strobes should be used to control
* the particular equation. This design file may be used as a template, but it is up to the
* designer to make sure that the interface can be properly implemented in this device.

* Declarations
*-------------------------------------------------------------
* PARIO device 'iux780fp';
*-------------------------------------------------------------
* Local PCI Interface Input Pins
*-------------------------------------------------------------
* BCLK_  pin 118;  "Buffered (up to 33MHz) PCI Bus Clock"
* BRST_  pin 1;  "Local Active Low PCI Reset"
* LA2_   pin 2;  "Local Demultiplexed Address Bit 2"
* LA3_   pin 3;  "Local Demultiplexed Address Bit 3"
* LA4_   pin 4;  "Local Demultiplexed Address Bit 4"
* LA5_   pin 5;  "Local Demultiplexed Address Bit 5"
* LA6_   pin 33;  "Local Demultiplexed Address Bit 6"
* LA7_   pin 34;  "Local Demultiplexed Address Bit 7"
* I_CTL_WR_ pin 35;  "IO Control Address Space Write"
* I_STA_RD_ pin 36;  "IO Status Address Space Read"
* DMA_ACK pin 37;  "DMA Cycle Acknowledge"
*-------------------------------------------------------------
* Local PCI Interface Output Pins
*-------------------------------------------------------------
* PAR_INTRQ_ pin 106;  "IO Interface PCI Interrupt Request"
* DMA_REQUEST_ pin 78;  "FIFO DMA Request"
* DMA_REQEST_ pin 47;  "FIFO DMA Burst Request"
*-------------------------------------------------------------
* Parallel Interface Input Pins
*-------------------------------------------------------------
* I_WR_STROBE_ pin 96;  "Parallel Interface Byte Write Strobe"
* I_RD_STROBE_ pin 36;  "Parallel Interface Byte Read Strobe"
* I_RD_STROBE2_ pin 67;  "Parallel Interface Byte Read Clock (i_rd_strobe)"
* I_WR_STROBE2_ pin 63;  "Parallel Byte Write Strobe Synced to BCLK"
* I_RD_STROBE2_ pin 64;  "Parallel Byte Read Strobe Synced to BCLK"
*-------------------------------------------------------------
* Parallel Interface Output Pins
*-------------------------------------------------------------
* I_EMPTY_ pin 79;  "Parallel Interface Buffer Empty Status"
* I_FULL_ pin 95;  "Parallel Interface Buffer Half Status"
* I_HALF_FULL_ pin 80;  "Parallel Interface Buffer Half Full Status"
* I_DEBUG pin 32;  "Parallel Interface Buried Debug Control"
*-------------------------------------------------------------
* Parallel Interface User Defined Pins
*-------------------------------------------------------------
* I_CONTROL1 pin 7;  "Parallel Interface User Defined Control 1"
* I_CONTROL2 pin 14;  "Parallel Interface User Defined Control 2"
* I_CONTROL3 pin 66;  "Parallel Interface User Defined Control 3"
* I_CONTROL4 pin 69;  "Parallel Interface User Defined Control 4"
* I_CONTROL5 pin 109;  "Parallel Interface User Defined Control 5"
* I_CONTROL6 pin 24;  "Parallel Interface User Defined Control 6"
* I_CONTROL7 pin 91;  "Parallel Interface User Defined Control 7"
* I_CONTROL8 pin 94;  "Parallel Interface User Defined Control 8"
* I_CONTROL9 pin 25;  "Input Status Connection from Scanner or Marker"
* I_CONTROL10 pin 91;  "Parallel Interface User Defined Control 9"
* I_CONTROL11 pin 106;  "Parallel Interface User Defined Control 11"
* I_CONTROL12 pin 107;  "Parallel Interface User Defined Control 12"
* I_CONTROL13 pin 8;  "Parallel Interface User Defined Control 13"
* I_CONTROL14 pin 114;  "Parallel Interface User Defined Control 14"
*FIFO Interface Input Pins*

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPD0</td>
<td>Input FIFO Byte 0 Full Status</td>
</tr>
<tr>
<td>IPD1</td>
<td>Input FIFO Byte 1 Full Status</td>
</tr>
<tr>
<td>IPD2</td>
<td>Input FIFO Byte 2 Full Status</td>
</tr>
<tr>
<td>IPD3</td>
<td>Input FIFO Byte 3 Full Status</td>
</tr>
<tr>
<td>IPD4</td>
<td>Input FIFO Byte 4 Full Status</td>
</tr>
<tr>
<td>IPD5</td>
<td>Input FIFO Byte 5 Full Status</td>
</tr>
<tr>
<td>IPD6</td>
<td>Input FIFO Byte 6 Full Status</td>
</tr>
<tr>
<td>IPD7</td>
<td>Input FIFO Byte 7 Full Status</td>
</tr>
</tbody>
</table>

*FIFO Interface Output Pins*

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFIFO0</td>
<td>Output FIFO Byte 0 Read Strobe</td>
</tr>
<tr>
<td>OFIFO1</td>
<td>Output FIFO Byte 1 Read Strobe</td>
</tr>
<tr>
<td>OFIFO2</td>
<td>Output FIFO Byte 2 Read Strobe</td>
</tr>
<tr>
<td>OFIFO3</td>
<td>Output FIFO Byte 3 Read Strobe</td>
</tr>
<tr>
<td>OFIFO4</td>
<td>Output FIFO Byte 4 Read Strobe</td>
</tr>
<tr>
<td>OFIFO5</td>
<td>Output FIFO Byte 5 Read Strobe</td>
</tr>
<tr>
<td>OFIFO6</td>
<td>Output FIFO Byte 6 Read Strobe</td>
</tr>
<tr>
<td>OFIFO7</td>
<td>Output FIFO Byte 7 Read Strobe</td>
</tr>
</tbody>
</table>

*PCI Bidirectional Local Data Bits LSB*

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BAD0</td>
<td>Local Data Bit 0</td>
</tr>
<tr>
<td>BAD1</td>
<td>Local Data Bit 1</td>
</tr>
<tr>
<td>BAD2</td>
<td>Local Data Bit 2</td>
</tr>
<tr>
<td>BAD3</td>
<td>Local Data Bit 3</td>
</tr>
<tr>
<td>BAD4</td>
<td>Local Data Bit 4</td>
</tr>
</tbody>
</table>

*Parallel Interface Bidirectional Parallel Data Bus (Data Connection To Transceivers)*

<table>
<thead>
<tr>
<th>Pin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPD0</td>
<td>Parallel Data Bit 0</td>
</tr>
<tr>
<td>IPD1</td>
<td>Parallel Data Bit 1</td>
</tr>
<tr>
<td>IPD2</td>
<td>Parallel Data Bit 2</td>
</tr>
<tr>
<td>IPD3</td>
<td>Parallel Data Bit 3</td>
</tr>
<tr>
<td>IPD4</td>
<td>Parallel Data Bit 4</td>
</tr>
<tr>
<td>IPD5</td>
<td>Parallel Data Bit 5</td>
</tr>
</tbody>
</table>
Rochester Institute of Technology Computer Science Department

IFD2_IN  pin 76; *Parallel Data Bit 2
IFD1    pin 66; *Parallel Data Bit 1
IFD1_IN pin 56; *Parallel Data Bit 1
IFD0    pin 46; *Parallel Data Bit 0
IFD0_IN pin 46; *Parallel Data Bit 0

" Parallel Interface Bidirectional Parallel Data Bus (Data Connection To FIFOs)

FPD7 pin 54; *Parallel Data Bit 7
FPD6 pin 58; *Parallel Data Bit 6
FPD5 pin 55; *Parallel Data Bit 5
FPD4 pin 60; *Parallel Data Bit 4
FPD3 pin 56; *Parallel Data Bit 3
FPD2 pin 61; *Parallel Data Bit 2
FPD1 pin 57; *Parallel Data Bit 1
FPD0 pin 62; *Parallel Data Bit 0

" Parallel Interface State Machine Internal (Buried) State Registers

FIFO_OP node 16; *Parallel Interface FIFO Operation Flag
PAR_S3 node 13; *Parallel Interface Internal State Register 3
PAR_S2 node 6; *Parallel Interface Internal State Register 2
PAR_S1 node 105; *Parallel Interface Internal State Register 1
PAR_S0 node 115; *Parallel Interface Internal State Register 0

" FIFO Buffer Mode Control Outputs

FIFO_SAB pin 130; *FIFO A to B Transceiver Mode Control
FIFO_SBA pin 112; *FIFO B to A Transceiver Mode Control
FIFO_B3GAB pin 120; *FIFO Byte 3 Data Output Enable Control
FIFO_B2GAB pin 113; *FIFO Byte 2 Data Output Enable Control
FIFO_B1GAB pin 15; *FIFO Byte 1 Data Output Enable Control
FIFO_B0GAB pin 111; *FIFO Byte 0 Data Output Enable Control

" Outputs and Feedback Terms

IFIFO_DBF pin 75; *Input FIFO Define B Flag Strobe
PDBUS_ENABLE pin 23; *P-Term to Enable driving PD Bus
P_DATACLKO pin 12; *Combinatorial Clock Eq to Drive Data onto IPD Bus
P_DATACLKI pin 52; *Clock Input to Drive Data onto IPD Bus

" Set Definitions and Constants

H,L,X,C,K,Z,SIMULATE 1,0,,X,,C,,K,,Z,,0:
I_DEBUG istype 'buffer, pos, reg';
I_REQUEST istype 'buffer, pos, reg';
FIFO_OP istype 'buffer, pos, reg';
PAR_S3 istype 'buffer, pos, reg';
PAR_S2 istype 'buffer, pos, reg';
PAR_S1 istype 'buffer, pos, reg';
PAR_S0 istype 'buffer, pos, reg';
IO_AddrIn [ LA5, LA4, LA3, LA2 ];
IO_Address [ LA7, LA6, LA5, LA4, LA3, LA2 ];
Status [ X, 0, 0, 0, 0, 0, 0 ];
Vector [ X, 0, 0, 0, 0, 1 ];
LoopBackOut [ X, 0, 0, 1, X, X ];
LoopBackOut0 [ X, 0, 0, 1, 0, 0 ];
LoopBackOut1 [ X, 0, 0, 1, 0, 1 ];
LoopBackOut2 [ X, 0, 0, 1, 1, 0 ];
LoopBackOut3 [ X, 0, 0, 1, 1, 1 ];
LoopBackIn [ X, 0, 1, X, X ];
LoopBackIn0 [ X, 0, 1, 0, 0, 0 ];
LoopBackIn1 [ X, 0, 1, 0, 0, 1 ];
LoopBackIn2 [ X, 0, 1, 0, 1, 0 ];
LoopBackIn3 [ X, 0, 1, 1, 1, 0 ];
FIFO_Status0 [ X, 0, 1, 1, 0, 0 ];
FIFO_Status1 [ X, 0, 1, 1, 0, 1 ];
FIFO_Status2 [ X, 0, 1, 1, 1, 0 ];
FIFO_Status3 [ X, 0, 1, 1, 1, 1 ];

96
IFIFO_FLAG [ X, 1, 0, 1, 0, 1 ];
IO_Mode [ I_HS_MODE ];
IFIFO_Input [ 0, 0 ];
IFIFO_Output [ 1, 0 ];
HS_Input [ 0, 1 ];
HS_Output [ 1, 1 ];

IFIFO_Status [ IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_FULL, IFIFO0_F
"Equations for accessing internal registers

BAD7  I_REQUEST & (IO_Address == Status)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status0)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status1)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status2)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status3)
  # PD7 & (IO_Address == LoopBackOut);
BAD7.OE = I_O_STA_RD;

BAD6  = I_CONTROLx & (IO_Address == Status)
  # IFIFO0_HALFF & (IO_Address == FIFO_Status0)
  # IFIFO1_HALFF & (IO_Address == FIFO_Status1)
  # IFIFO2_HALFF & (IO_Address == FIFO_Status2)
  # IFIFO3_HALFF & (IO_Address == FIFO_Status3)
  # PD6 & (IO_Address == LoopBackOut);
BAD6.OE = I_O_STA_RD;

BAD5  I_CONTROLx & (IO_Address == Status)
  # IFIFO0_FULL & (IO_Address == FIFO_Status0)
  # IFIFO1_FULL & (IO_Address == FIFO_Status1)
  # IFIFO2_FULL & (IO_Address == FIFO_Status2)
  # IFIFO3_FULL & (IO_Address == FIFO_Status3)
  # PD5 & (IO_Address == LoopBackOut);
BAD5.OE = I_O_STA_RD;

BAD4  = I_CONTROLx & (IO_Address == Status)
  # IFIFO0_EMPTY & (IO_Address == FIFO_Status0)
  # IFIFO1_EMPTY & (IO_Address == FIFO_Status1)
  # IFIFO2_EMPTY & (IO_Address == FIFO_Status2)
  # IFIFO3_EMPTY & (IO_Address == FIFO_Status3)
  # PD4 & (IO_Address == LoopBackOut);
BAD4.OE = I_O_STA_RD;

BAD3  I_CONTROLx & (IO_Address == Status)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status0)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status1)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status2)
  # IFIFO3_AF_AEB & (IO_Address == FIFO_Status3)
  # PD3 & (IO_Address == LoopBackOut);
BAD3.OE = I_O_STA_RD;

BAD2  I_CONTROLx & (IO_Address == Status)
  # IFIFO0_HALFF & (IO_Address == FIFO_Status0)
  # IFIFO1_HALFF & (IO_Address == FIFO_Status1)
  # IFIFO2_HALFF & (IO_Address == FIFO_Status2)
  # IFIFO3_HALFF & (IO_Address == FIFO_Status3)
  # PD2 & (IO_Address == LoopBackOut);
BAD2.OE = I_O_STA_RD;

BAD1  = I_CONTROLx & (IO_Address == Status)
  # IFIFO0_FULL & (IO_Address == FIFO_Status0)
  # IFIFO1_FULL & (IO_Address == FIFO_Status1)
  # IFIFO2_FULL & (IO_Address == FIFO_Status2)
  # IFIFO3_FULL & (IO_Address == FIFO_Status3)
  # PD1 & (IO_Address == LoopBackOut);
BAD1.OE = I_O_STA_RD;

BAD0  = I_CONTROLx & (IO_Address == Status)
  # IFIFO0_EMPTY & (IO_Address == FIFO_Status0)
  # IFIFO1_EMPTY & (IO_Address == FIFO_Status1)
  # IFIFO2_EMPTY & (IO_Address == FIFO_Status2)
  # IFIFO3_EMPTY & (IO_Address == FIFO_Status3)
  # PD0 & (IO_Address == LoopBackOut);
BAD0.OE = I_O_STA_RD;

"Equations to allow programming of the Input FIFO Almost Empty/Almost Full Flag

PDBUS_ENABLE = IFIFO_DBF & !CONTROLx & I_CTL_WR & (IO_Address == IFIFO_FLAG)
  # !I_CTL_WR & (IO_Address == LoopBackIn)
  # CONTROLx & 1,
PD7  = BAD7 & !CONTROLx
  # IFD7_IN & CONTROLx;
PD7.OE = PDBUS_ENABLE;

PD6  = BAD6 & !CONTROLx
  # IFD6_IN & CONTROLx;
PD6.OE = PDBUS_ENABLE;
PD5 = BAD5 & !CONTROLX
    # IPD5_IN & CONTROLX;
PD5.OE = PDBUS_ENABLE;
PD4 = BAD4 & !CONTROLX
    # IPD4_IN & CONTROLX;
PD4.OE = PDBUS_ENABLE;
PD3 = BAD3 & !CONTROLX
    # IPD3_IN & CONTROLX;
PD3.OE = PDBUS_ENABLE;
PD2 = BAD2 & !CONTROLX
    # IPD2_IN & CONTROLX;
PD2.OE = PDBUS_ENABLE;
PD1 = BAD1 & !CONTROLX
    # IPD1_IN & CONTROLX;
PD1.OE = PDBUS_ENABLE;
PD0 = BAD0 & !CONTROLX
    # IPD0_IN & CONTROLX;
PD0.OE = PDBUS_ENABLE;

IPD7 := PD7;
IPD7.CLK = P_DATACLKI;
IPD7.OE = CONTROLX.fb;

IPD6 := PD6;
IPD6.CLK = P_DATACLKI;
IPD6.OE = CONTROLX.fb;

IPD5 := PD5;
IPD5.CLK = P_DATACLKI;
IPD5.OE = CONTROLX.fc;

IPD4 := PD4;
IPD4.CLK = P_DATACLKI;
IPD4.OE = CONTROLX.fb;

IPD3 := PD3;
IPD3.CLK = P_DATACLKI;
IPD3.OE = CONTROLX.fc;

IPD2 := PD2;
IPD2.CLK = P_DATACLKI;
IPD2.OE = CONTROLX.fb;

IPD1 := PD1;
IPD1.CLK = P_DATACLKI;
IPD1.OE = CONTROLX.fb;

IPD0 := PD0;
IPD0.CLK = P_DATACLKI;
IPD0.OE = CONTROLX.fb;

"Clock, Output Enable and Reset Equations for Parallel Interface State Machine"

FIFO_OP.CLK = BCLK;
FIFO_OP.AR = !BRST_1;
FIFO_OP.OE = SIMULATE;
PAR_S3.CLK = BCLK;
PAR_S3.AR = !BRST_1;
PAR_S3.OE = SIMULATE;
PAR_S2.CLK = BCLK;
PAR_S2.AR = !BRST_1;
PAR_S2.OE = SIMULATE;
PAR_S1.CLK = BCLK;
PAR_S1.AF = !BRST_1;
PAR_S1.OE = SIMULATE;
PAR_S0.CLK = BCLK;
PAR_S0.AR = !BRST_1;
PAR_S0.OE = SIMULATE;

"Equations for FIFO Input/Output on Parallel Interface Side"
OFIFO_R signals are based upon control signals and State Machine Transitions.

IFIFO_W signals are based upon control signals and State Machine Transitions.

Equations for generating DMA Requests to DMA Controller on PCI Side:

```
DMA_BREQUEST_.CLK = BCLK;
DMA_BREQUEST_.OE  = 1;
DMA_BREQUEST_.ASET = !BRST ;
```

```
!!DMA_BREQUEST_ := (OFIFO_Status != AlmostFull) \& \& DMA_BREQUEST_  
# (OFIFO_Status == AlmostEmpty) \& \& DMA_BREQUEST_  
# (IFIFO_Status != AlmostEmpty) \& \& DMA_BREQUEST_  
# (IFIFO_Status == AlmostFull) \& \& DMA_BREQUEST_ ;
```

```
DMA_REQUEST_.CLK = BCLK;
DMA_REQUEST_.OE  = 1;
DMA_REQUEST_.ASET = !BRST ;
```

```
!!DMA_REQUEST_ := (IFIFO_Status == NotEmpty) \& \& I_CONTROLx  
# (IFIFO_Status == NotEmpty) \& \& I_CONTROLx ;
```

Equations for generating Parallel Interface Interrupt Requests:

```
PAR_INTRQ := !I_CONTROLx;
PAR_INTRQ.CLK = BCLK;
PAR_INTRQ.OE  = 1;
```

Equations for synchronizing Parallel Interface FIFO Read and Write Strobes to BCLK:

```
!!I_WR_STROBES_ := !I_WR_STROBE_;  
!I_RD_STROBES_ := !I_RD_STROBE_;  
```

```
I_WR_STROBES_.CLK = BCLK;
I_WR_STROBES_.OE  = 1;
```

```
I_RD_STROBES_.CLK = BCLK;
I_RD_STROBES_.OE  = 1;
```

Equations for controlling FIFO transceivers mode and direction:

```
!!FIFO_BOGAB signals would be controlled based upon state machine design;
```

```
FIFO_SAB  = 1;
FIFO_SBA  = 1;
```

deck PARI
5.4. **BIBLIOGRAPHY**


6. **End Notes**

---

3. PCI Local Bus Specification Revision 2.1, p. 25
5. Intel Pentium and Pentium Pro Processors and Related Products 1996, p. 2-585