A Novel Approach to Pci Simulation Using ScriptSim

Item Type	thesis
Authors	Andryc, Kevin R
DOI	10.7275/497979
Download date	2025-05-01 10:11:14
Link to Item	https://hdl.handle.net/20.500.14394/44740

A NOVEL APPROACH TO PCI SIMULATION USING SCRIPTSIM

A Thesis Presented

by

KEVIN R. ANDRYC

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

May 2008

Electrical and Computer Engineering

A NOVEL APPROACH TO PCI SIMULATION USING SCRIPTSIM

A Thesis Presented

by

KEVIN R. ANDRYC

Approved as to style and content by:

Russell G. Tessier, Chair

Patrick A. Kelly, Member

Eric Polizzi, Member

C. V. Hollot, Department Head
Electrical & Computer Engineering

ABSTRACT

A NOVEL APPROACH TO PCI SIMULATION USING SCRIPTSIM

MAY 2008

KEVIN ANDRYC, B.S., UNIVERISTY OF MASSACHUSETTS

M.S.E.C.E., UNIVERISTY OF MASSACHUSETTS

Directed by: Professor Russell G. Tessier

In recent years, the Peripheral Component Interconnect (PCI) has become one of the most widely used bus architectures in modern computers. Simulation of the PCI bus, however, has been limited in both research and development. Current commercial PCI simulation software is designed towards compliance and verification testing rather than accurately mimicking PCI bus systems. In addition, most PCI simulation software is inflexible and offers no graphical user interface, instead relying on text files for configuration.

This paper presents a novel approach to PCI simulation using ScriptSim, an open-source PCI simulation tool that supports all the features offered by the PCI Local Specification Version 2.2. In addition to extending ScriptSim to include PCI-X functionality and a web-based graphical interface, we introduce techniques that allow us to accurately simulate real-world systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Traditionally, PCI simulation software, like Synopsys' DesignWare PCI IP [18], is specifically designed for verification and compliance of the PCI/PCI-X standard. While these simulators provide accurate models to test complex user-developed hardware, they do not provide the means to accurately measure performance based on real-life systems. While some simulators take into account external factors that may affect performance, such as memory latency, they are either statically defined or targeted towards a specific architecture. In addition, most simulators allow for configuration using text files or by manually editing the source program which requires timely recompilation for each change. Furthermore, simulating bus activity requires the user to individually add commands to each device. While this allows the user greater detail and control over bus activity, it is also timely and can become unwieldy. The goal of this thesis is to propose a novel approach to PCI simulation by taking into account external system factors in addition to providing an easy to use interface.

1.1 Motivation

The main characteristic of simulators is the ability to model real-life situations. Having accurate models will help designers make informed decisions and help reduce costly mistakes. When designing hardware to be used on the PCI bus, knowing what type of performance to expect is essential. The design of a PCI simulator can be a complicated proposition. While ensuring the simulator meets the specifications set by the PCI Special Interest Group (PCISIG), it is important to model the effects of devices and system components in obtaining accurate performance measures. However, the design of the user interface is just as important. It should provide all the functionality necessary to easily perform simulations while also being flexible enough to allow the designer to customize it to adjust various parameters.

Bus performance is usually measured by its theoretical *peak bandwidth*. While the peak bandwidth is easily calculated, it is also the least relevant number when determining bus performance. In a shared bus system, such as the PCI bus, this number is rarely achieved. To more accurately represent performance, bus *latency* must be taken into account. Devices attached to the bus can incur several types of latency such as read, write, and arbitration latency. The PCI bus helps reduce the affects of latency by allowing devices to transfer data in *burst mode*. While increasing the length of the burst increases device bandwidth, minimizing the effect of the initial read or write latency, it also increases the latency for other devices waiting to access the bus.

To estimate a device's maximum available bandwidth, it was previously shown in [15] that each device incurs a certain amount of overhead, $ov(d)$, which is proportional to the transaction's burst size d. Therefore, the *sustained bandwidth* of one device irrespective of other devices can be calculated by:

$$bw_{sus} = \frac{d}{d + ov(d)} * bw_{pci} \tag{1.1}$$

If other devices wish to access the bus while it is currently in use, then the device incurs a certain amount of arbitration latency. If there are n devices attached to the bus, all needing access, then while one device is granted access to the bus, there are $n - 1$ devices that must wait. If we assume a round-robin arbitration scheme, then the worst case latency τ_{lat} for n devices, as shown in [15], can be given by:

$$\tau_{lat} = (n-1)(d + ov(d)) \tag{1.2}$$

In Figure 1.1 we show the relation between the burst size and the maximum available bandwidth which is shared amongst all devices. It also shows the linear relation between the burst size and the worst-case latency to access the bus when 5 devices are attached.

Most PCI transactions are reads from main memory or writes to main memory, thus making up the majority of overhead. However, simulators usually do not explicitly account for latency due to memory reads or writes. In our research, only one functional simulator handled latency in a dynamic fashion [14]. Depending on the type of latency, such as a read or write, the simulator would assign a range of typical latency values. When calculating the latency, a probability would be assigned to each value within the range. For example, a range of 8 to 12 latency cycles would be assigned when reading from memory, each with equal probability.

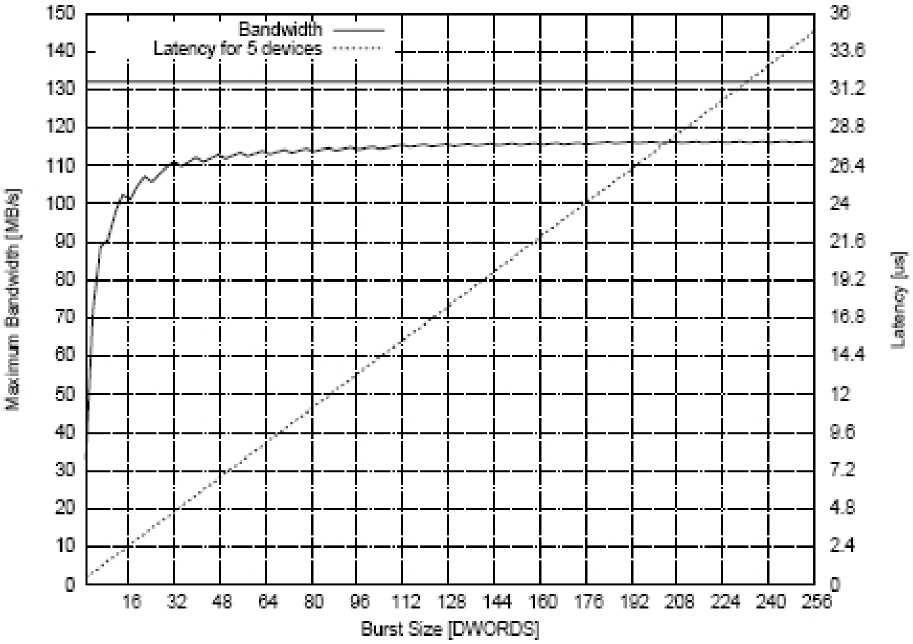


Figure 1.1: Relation between burst size and both latency and maximum bandwidth

In modern PC systems, *memory bus* bandwidth is shared amongst all processors and devices that access main memory. The system's *Northbridge*, typically known as the Memory Controller Hub (MCH), helps control access to system memory while also controlling access to the various other buses attached to it. The system's processors attach to the Northbridge via the *frontside bus*. To prevent performance bottlenecks, the frontside and memory bus should, and usually does, provide the highest bandwidth in the system. The system's Southbridge, also known as the I/O Controller Hub (ICH), acts as a hub to route traffic from typically slower peripheral I/O devices. A typical PC architecture is shown in Figure 1.2.

This thesis will attempt to define the affects of bandwidth and latency on the PCI bus with respect to the behavior of other devices. We use statistics gathered from a PCI analyzer tool, VMETRO's Vanguard PCI Bus Analyzer [19], to provide real-world results which are used to support our conclusions. From this we can build an accurate software PCI simulation tool based on ScriptSim. In addition, we support a fully-functional web interface which allows the designer to quickly set-up and run simulations. The interface will allow full control over various parameters as well as various analysis tools.

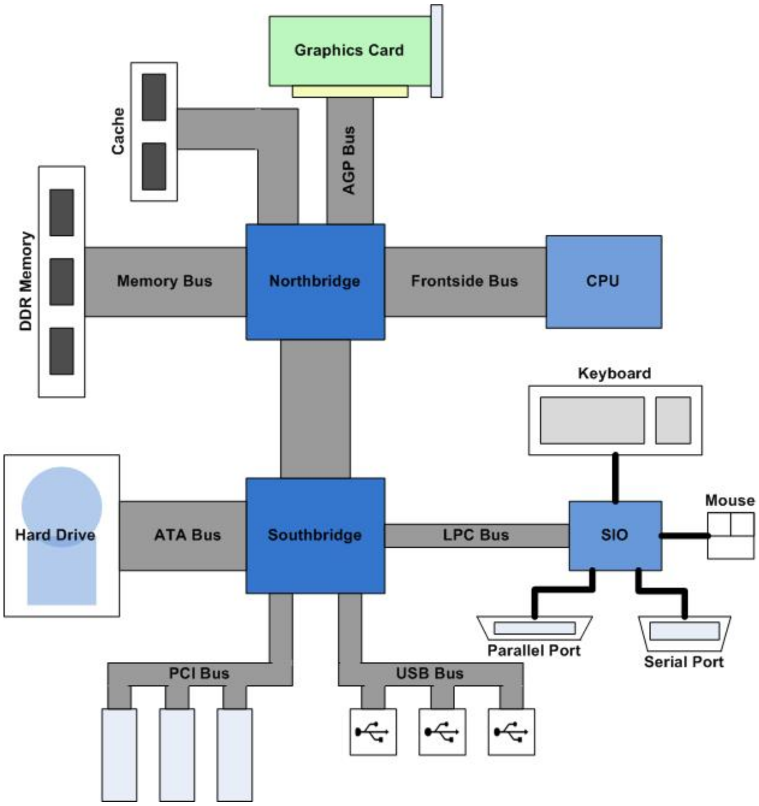


Figure 1.2: Modern PC architecture

1.2 Contributions

Our main contributions in this thesis include:

- The development of a behavioral PCI simulator. While ScriptSim accurately models functionality as defined by the PCI local specification, it does not model how specific devices affect bus performance. We develop an approach which modifies ScriptSim to allow for behavioral simulation.

- The integration of PCI-X functionality into a PCI simulator. Currently, we know of no behavioral simulators that include PCI-X. Most commercial simulators do include PCI-X functionality, but are used for verification. The inclusion of PCI-X functionality is an important feature since most PCI-X offers significant benefits and has been widely implemented.
- The development of a web-based graphical user interface, PCI Web Sim, for ScriptSim. PCI Web Sim is built on the popular Java Enterprise Edition architecture which features a robust and scalable infrastructure. This interface allows users to perform many important functions relating to PCI simulation including the ability to adjust key performance parameters. Simulations may also be stored in a relational database and various tools can be used to analyze the resulting data.

1.3 Synopsys

This thesis is structured as follows. The next chapter describes a historical abstract as well as a current look at bus architectures. Chapter 3 presents related work. In Chapter 4 we describe the PCI bus providing a synopsis of the protocol and important characteristics. In Chapter 5 we extend the thesis to describe the PCI-X standard which includes additions and changes from PCI. Chapter 6 analyzes ScriptSim, the basis of our PCI software simulator. In Chapter 7 we discuss our novel approach to PCI simulation. Chapter 8 provides a look at future uses and contributions. Finally, Chapter 9 provides a conclusion and future work involved.

CHAPTER 2

CURRENT BUS SYSTEMS

Computer busses are widely used in systems to connect several peripheral devices together. While some devices are connected using a daisy chain topology (e.g.: SCSI) or via a switched hub as in the case of USB, most modern systems employ a multidrop bus (MDB) topology. A MDB allows devices, which are usually perpendicularly plugged into a connector on the circuit board, to share the same set of electrical wires. Since multiple devices may need to communicate over the bus, an arbiter must be employed to determine which device can use the bus and for how long. Also, a priority scheme may be employed allowing devices with greater need to access the bus before other devices.

Bus designs can be classified by two different control mechanisms – synchronous and asynchronous. Synchronous busses are used where maximum performance is needed, such as the local (memory) bus but at a cost on inflexibility since increasing the bus speed may cause devices to fail. Asynchronous bus designs are favored because of their flexibility allowing older devices to be used on newer and faster bus systems. However, this price comes at a cost of increased complexity and generally lower performance. Thus, modern computer systems employ both types of buses as illustrated in Figure 2.1. In a computer system, the synchronous bus is noted as the *local bus*, and the asynchronous bus is noted as the *external bus*. This design provides optimum performance for the path between the CPU and memory since this is what mainly determines the overall performance.

Isolating the CPU allows increases in speed to become unhindered. However, the rate at which CPUs and memory increase their speed is faster than the buses that interconnect them. As a result, slow buses left many systems starved for data. In addition, high-bandwidth devices such as video cards were quickly outrunning newer bus systems.

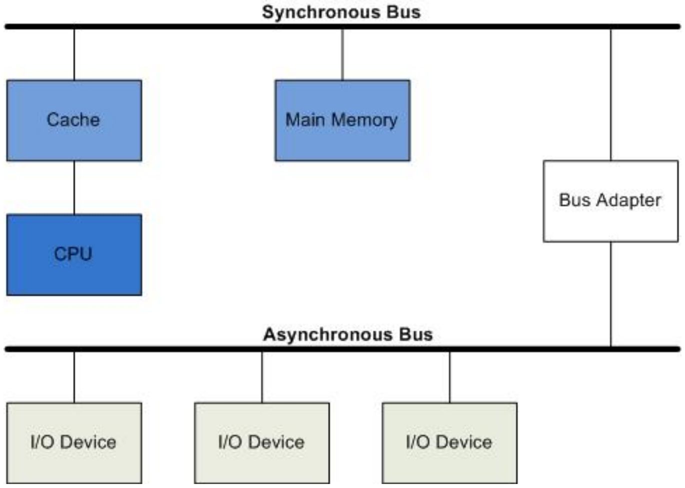


Figure 2.1: Modern computer system which uses both synchronous and asynchronous buses

The remainder of this chapter is structured as follows. In Section 2.1, we describe a variety of alternative bus systems such as the Industry Standard Architecture, VersaModule Eurocard Bus, FutureBus and FutureBus+, InfiniBand, Universal Serial Bus, IEEE-1394 or Firewire, Accelerated Graphics Port, and Peripheral Component Interconnect Express.

2.1 Bus Systems

In this section we present a brief overview of a variety of bus systems along with their architecture and design principles. These buses provide a historical perspective which may have influenced the design of the PCI bus. We also look at some buses that proceeded PCI for completeness. In Table 2.1, we summarize the various bus systems discussed in the following sections.

2.1.1 Industry Standard Architecture Bus

The *Industry Standard Architecture* [1] (ISA) bus originally started out as an 8-bit bus system in 1981 for the IBM PC. Dubbed the XT architecture, it was later extended to a 16-bit data path in 1983 and was renamed to the AT bus architecture after the IBM AT. The maximum transfer rate of 8.3 MB/s at 8.3 MHz in the original version was reasonable enough for low bandwidth requirements. Every transaction is actively controlled by either the CPU or a single direct memory access (DMA) controller which synchronize their bus accesses using a direct wire. While the ISA protocol allows for bus masters, the devices are “passive” in that they cannot actively request the bus and transfer data to main memory autonomously. This allows the bus to operate without an arbiter. In order to transfer data, devices could be programmed to announce the availability of data to the DMA controller which then would perform the transfer without the interaction of the CPU. The proliferation of high bandwidth applications has rendered ISA virtually extinct in modern computer systems.

Table 2.1: Comparison of various common bus systems (continued on pg. 10)

Bus Name	**Width (bits)**	**Frequency (MHz)**	**Max. Bandwidth (MB/s)**
ISA	8 / 16	8.3 / 8.3	8.3 / 16
EISA	32	8.3	33.2
VLB	32	33	132
PCI	32	33	132
PCI 2.1	64	66	528
PCI-X 1.0	64	133	1,064
PCI-X 2.0 (mode 1)	64	133x2	2,128
PCI-X 2.0 (mode 2)	64	133x4	4,256
AGP	32	66	264

AGP (x2 mode)	32	66x2	528
AGP (x4 mode)	32	66x4	1,056
USB 1.0	1 (serial)	N/A	0.185 (sync) 1.5 (async)
USB 2.0	1 (serial)	N/A	60
Firewire/IEEE-1394	1 (serial)	N/A	400
FutureBus	32	100	400
FutureBus+	256	100	3,200
VMEbus	64	10 (100ns cycle time)	80
PCI Express (x1)	1 (serial)	2,500	400
PCI Express (x2)	2 (serial)	2,500	800
PCI Express (x4)	4 (serial)	2,500	1,600
PCI Express (x8)	8 (serial)	2,500	3,200
PCI Express (x16)	16 (serial)	2,500	6,400

Table 2.2: Comparison of various common bus systems (continued from pg. 9)

2.1.2 VersaModule Eurocard Bus

The *VersaModule Eurocard* [2] (VME) is an asynchronous, master-slave, parallel bus which was developed in the late 1970’s and based off the architecture of the VERSAbus. The VERSAbus architecture featured a 16-bit data path and 24-bit address and was later renamed VME which expanded the data path to 32 bits and supported transfer rates of up to 40 MB/s. An enhancement to VME, named VME64, extended support to 64-bits and provided transfer rates of 80 MB/s. Later versions of the VMEbus (2eSST) could achieve sustained throughput of 1GB/s.

Every VMEbus must contain a system controller to perform arbitration via the arbitration bus. The system controller resides in slot one of Eurocard chassis and the functionality provided is independent of the physical card which resides in the slot. In other words, a master or slave may exist along with the system controller in the same slot.

The VMEbus allows for three arbitration modes: *fixed-priority mode* (PRI), *round-robin* (RR), and *single level* (SGL). The arbitration mode is selected during initialization of the system and cannot by changed while the system is running. Each arbitration mode offers four bus request levels (BRL) giving highest priority to level three and lowest to level zero. Devices that are at the same level will be granted bus access dependent upon the proximity to slot one. Fixed-priority arbitration arbitrates between all four levels granting access to the highest priority device first. Round-robin grants bus access sequentially giving all four levels equal access. In single level mode, arbitration only occurs in BRL 3 while all other levels are ignored. VMEbus also allows for a mix of PRI and RR modes.

The VME system supports a wide range of microprocessors including the Motorola 680X0, SPARC, ALPHA, and the x86. It is most widely used in high performance multiprocessor servers and embedded systems such as factory automation, in-flight video servers, avionics, and cellular-phone base stations, as well as many others.

2.1.3 FutureBus / FutureBus+

FutureBus+ [3, 4] is described as a high-performance asynchronous bus which improved on the original *FutureBus* design. Work on FutureBus first started in 1979 and was seen as a replacement to the VMEBus. In the following years, complexity of FutureBus grew, slowing down the standardization process. It wasn’t until 1987 that an agreement was made on the FutureBus standard which was published as IEEE 896.1-1987.

Shortly after the announcement of the FutureBus standard, the U.S. Navy selected FutureBus+ for its Next Generation Computing Resources. The architecture of

FutureBus+ is processor independent and can support bus widths of up to 256 bits. Arbitration of the bus can either be *centralized* or *decentralized*. Each FutureBus+ module carries with it a unique 8-bit arbitration number. When requesting bus access, the module with the highest number is granted access as determined by the arbitration logic. Data transfers are burst oriented – the address is first sent followed by the entire data burst.

Unfortunately, in the end, very little use was made of FutureBus. However, the development strongly influenced the development of other bus systems and provided some of the original work performed on cache coherency.

2.1.4 InfiniBand Architecture

The *InfiniBand Architecture* [12] (IBA) specifies a “first order interconnect technology” which describes a system for connecting processor and I/O nodes forming a system area network. The InfiniBand architecture is independent of both the host operating system and processor platform.

Originally named *System I/O*, InfiniBand was the result of merging two competing designs: *Future I/O* and *Next Generation I/O*. InfiniBand was seen as a comprehensive replacement for datacenter I/O including PCI, Fibre Channel, and various other networks. Technically, InfiniBand is based on a switched fabric, point-to-point bidirectional serial communications link intended to connect processors with high-speed peripherals. The serial connection allows for a signaling rate of 2.5 Gbps in each direction. Support for double and quad data speeds are also available offering 5 Gbps and 10 Gbps of data throughput respectively. InfiniBand is similar to ATM networks in that a certain amount of bandwidth is reserved per connection. It is the responsibility of the switch fabrics to guarantee and enforce bandwidth reservations.

As of today, InfiniBand is used mostly in computer clusters or supercomputers such as the low-cost System X and Cray XD1.

2.1.5 Universal Serial Bus

The *Universal Serial Bus* [5] (USB) is a serial bus standard designed to allow plug-and-play capability while overcoming other known issues such as inflexibility, limited scalability, and extensibility. USB is built upon a well-defined interface for both hardware and software interconnections. This allows new devices to be easily integrated into current systems. With USB-1, bandwidth of 12Mbps was provided for asynchronous transmission, while a rate of 1.5Mbps could be achieved using synchronous data transmissions. USB-2 extends bandwidth capability to 480Mbps to support higher performance systems able to process large amounts data. Devices are classified based on their bandwidth: *low-speed* (10-100kbps), *full-speed* (500kbps-10Mbps), and *high-speed* (25-480Mbps).

The USB design is asymmetric consisting of multiple daisy-chained peripheral devices connected point to point to a hub device. Adding additional hubs to the chain allows for building a hierarchy as shown in Figure 2.2.

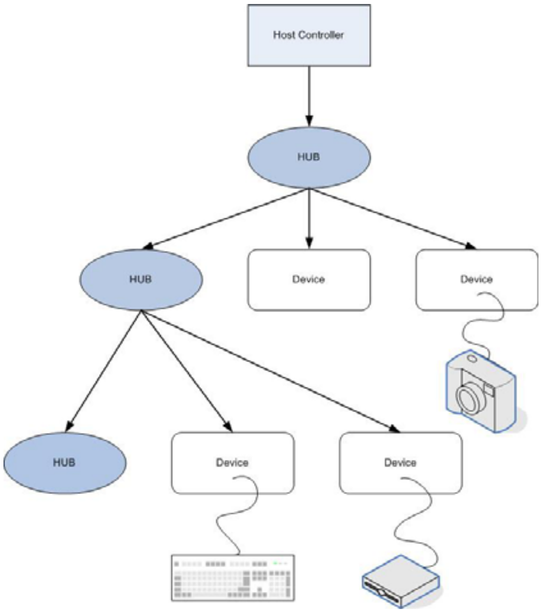


Figure 2.2: USB device hierarchy

Individual devices are referred to as *functions* in USB terminology since each device may host one or more functions, such as webcam with microphone. Transfers of data are based on *pipes* (logical channels) and connect from the USB host controller to an endpoint. Each pipe is unidirectional and holds information about the transfer type and stream requirements. USB provides three types of data transfers: control transfer, interrupt-data transfer, bulk-data transfer, and isochronous-data transfer. Control transfers are used to support configuration information when the device is attached to the bus as well as command and status information. Interrupt transfer is intended for low latency devices that require reliable delivery of data, such as mice and keyboards. Bulk transfers are used for devices that generate of consume large and bursty data, such as printers and scanners. Isochronous transfers are mostly used for time-dependent or real-time information, such as telephony and multimedia streams. USB guarantees the bandwidth for the transfer and provides a bounded latency; both are negotiated prior to the isochronous transfer.

2.1.6 IEEE-1394 Serial Bus / FireWire

FireWire [6, 7, 8], also known as *iLink*, is the standard as described in *IEEE-1394* – a composite of documents including *IEEE-1394*, *IEEE-1394a*, and *IEEE-1394b*. Originally created by Apple in 1990, FireWire provides high-speed communication and asynchronous real-time data transfers targeting devices such as video cameras, audio devices, and other peripherals. FireWire can connect up to 63 devices in an acyclic topology allowing peer-to-peer device communication.

There are two current standards of FireWire: FireWire 400 and FireWire 800 (IEEE-1394b). FireWire 400 supports transfer rates of 100, 200, and 400Mbps. FireWire 800, introduced by Apple in 2003, supports a transfer rate of 786.432Mbps and backward compatibility with FireWire 400.

Addresses in FireWire are 64-bits in length and consists of a 10-bit network ID, 6-bit node ID, and 48 bits for memory addressing within each node. Thus, 1023 networks or buses, each with 63 devices, can be supported. Since memory is addressable from every point in the acyclic topology, device connectivity is truly peer-to-peer.

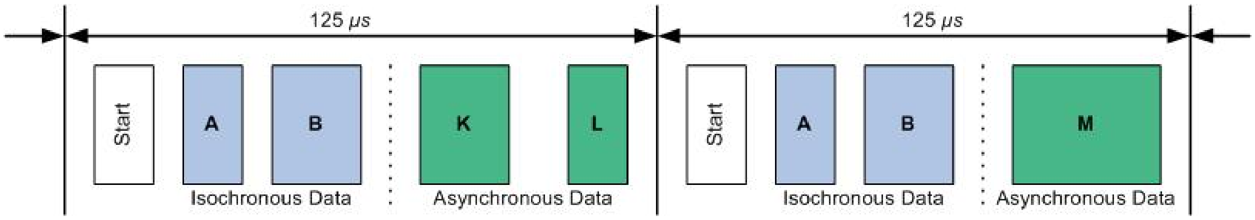


Figure 2.3: IEEE-1394 bus cycle with isochronous and asynchronous data

FireWire defines both the mechanical and timing constraints for arbitration and, unlike PCI, also defines the arbitration scheme. Technically, both FireWire and USB are similar in their technical design construction. The bus is divided into separate time frames each of 125 μs in length as shown in Figure 2.3. Up to 80% of the bandwidth may be reserved for isochronous data streams, ensuring bandwidth guarantees, while the rest is available for asynchronous data. In between each data packet is a variable-length gap essential for arbitration and maximum propagation delay. These gaps are required to allow bus participants to recognize a packet end since each packet is of variable length.

2.1.7 Accelerated Graphics Port

The *Accelerated Graphics Port* [9, 10] (AGP) is a high-speed point-to-point connection between a graphics card controller and the systems main memory. AGP is targeted at 3D graphical display applications; primarily to improve performance. As of this writing, AGP is currently being phased out in favor of PCI Express.

The AGP specification, in general, is based on the 66MHz PCI specification yet provides significant performance enhancements such as deeply pipelined memory, and demultiplexing of address and data on the bus. By allowing data transfers on both the rising and falling edge of a clock cycle, transfer rates are twice that of PCI. Also, a new low voltage electrical specification allows four data transfers per 66-MHz clock cycle.

By providing significant bandwidth improvement between the graphics controller and system memory, some of the data structures used for 3D rendering may be effectively moved into main memory. Texture data are well suited for a shift to main memory since they are typically read-only and thus do not have special access ordering or coherency problems. The point-to-point connection of the AGP bus allows guaranteed bandwidth and elimination from bus contention.

However, contention may arise in the host bridge which connects AGP, PCI, and other outside connection to the system memory bus.

Intel's first version of AGP in 1997, titled “AGP specification 1.0,” included both the 1x and 2x speeds. Since then, two additional AGP specifications were detailed: specification 2.0 documented AGP 4X and 3.0 documented 8X. The various AGP versions as well as some of the features are outlined below:

AGP 1x

- 32-bit channel operating at 66 MHz
- Maximum data rate of 266 megabytes per second (MB/s), doubled from the 133 MB/s transfer rate of PCI bus 33 MHz / 32-bit
- 3.3 V signaling.

AGP 2x

- 32-bit channel operating at 66 MHz double pumped to an effective 133 MHz
- Maximum data rate of 533 MB/s
- Signaling voltages the same as AGP 1x

AGP 4x

- 32-bit channel operating at 66 MHz quad pumped to an effective 266 MHz
- Maximum data rate of 1066 MB/s (1 GB/s)
- 1.5 V signaling

AGP 8x

- 32-bit channel operating at 66 MHz, strobing eight times per clock, delivering an effective 533 MHz
- Maximum data rate of 2133 MB/s (2 GB/s)
- 0.8 V signaling.

2.1.8 Peripheral Component Interface Express

The *Peripheral Component Interface Express* [11] (PCIe) bus is a full duplex point-to-point connection between device card and motherboard. In order to meet higher bandwidth requirements, Intel first expanded PCI to increase the bus width to 64 bits and increase the clock rate to 133MHz. However, increasing the clock speed led to stricter timing requirements and reduced bus slots. In realizing PCI was reaching its upper-bounds, in late 1990, Intel began work on its third generation I/O (3GIO) which later became known as PCIe. PCIe is designed as a replacement for PCI, PCI-X, and AGP interfaces.

In contrast to PCI, PCIe uses a shared switch instead of a shared bus where each device in the system has direct and exclusive access to the switch. Thus, each device has its own dedicated bus, which is called a *link*. Each link is built around dedicated unidirectional, point-to-point connections known as *lanes*. An example of which is shown in Figure 2.4. A lane consists of a transmit (T) and receive (R) *low voltage differential signal* (LVDS) pair each at 2.5 Gbps. This is in contrast to PCI, where all devices share the same bidirectional, 32-bit (or 64-bit), parallel bus.

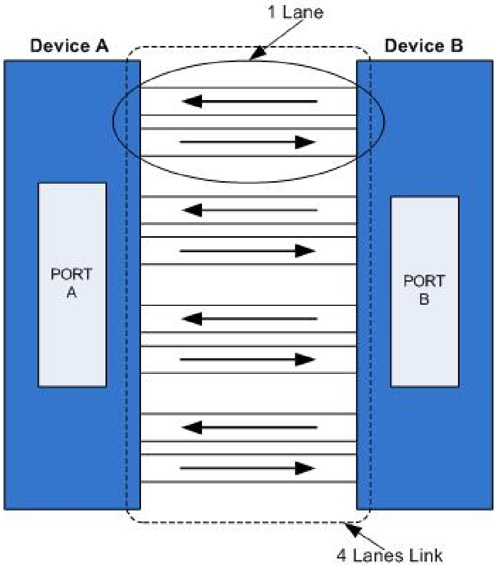


Figure 2.4: Two PCIe devices connected by a link consisting of 4 lanes

PCIe is built using a layered protocol similar to that of the IEEE 802 model of computer networking. The protocol consists of:

- *Transaction Layer Protocol* (TLP) – This layer transports read and write requests in the form of packets. The transaction layer supports both 32-bit and 64-bit addressing as well as PCI memory, I/O, and configuration address space. Similar to PCI-X, PCIe implements split transactions which allow the link to carry other traffic while the target device gathers data for the response. PCIe also utilizes credit-based flow control allowing for quality of service (QoS).

- *Data Link Layer* – Adds sequencing of TLP packets generated by the Transaction Layer, 32-bit error detection cyclic redundancy codes (CRC) to the data packets, and an acknowledgement protocol (ACK and NAK signaling) to create a reliable data transfer mechanism.
- *Physical Layer* (PHY) – Implements the PCIe link which is built using a collection of 1 or more lanes. All devices minimally support a single-lane (x1) link and may optionally support wider links of 2, 4, 8, 12, 16, or 32 lanes. PCIe data transmitted on multiple links is interleaved, which is known as data striping, and significantly increases the throughput.

As of today, PCI Express is starting to become the de-facto backplane standards on personal computers. The main reason is its design transparency transparent to software developers; operating systems designed for PCI can boot in a PCI Express system without any modification to the code.

CHAPTER 3

RELATED WORK

Work related to PCI simulation is very rare and in most cases not the focal point of previous research studies. To our knowledge, only two papers focus on PCI simulation [14] [15], and only [14] provides implementation details. However, for comparison purposes, we describe two papers that discuss JSIM, a java based simulator.

The remainder of this chapter is structured as follows. In Section 3.1, we discuss PCI Bus modeling, summarized from Chapter 6 of [15]. This provides a basis for calculating worst-case calculations of bandwidth and latency. Section 3.2 describes simulation software, including a PCI Simulator written in C++.

3.1 Using PCI-Bus Systems in Real-Time Environments

Schönberg's thesis [15] analyses and tries to predict what affect the load of the PCI-bus has on execution times of applications. While the thesis focuses primarily on real-time capabilities of the PCI bus, it provides us with an in-depth analysis of the PCI bus and details a unique method of calculating bandwidth and latency. In addition, we borrow ideas from his mathematical approach to model devices and implement them into our simulator.

3.1.1 Simplified PCI-Bus Model

While PCI simulation aims to provide approximations of real-world results, in some cases worst-case performance may also be useful. In Section 4.2 in [15], Schönberg presents a unique way to model the behavior of the PCI bus allowing us to calculate worst-case bandwidth and latency. Schönberg does reference a PCI simulation tool, “PCItrace”, which was developed to test the performance of various arbitration schemes and arbiters. However no implementation details were provided, and thus, we decided to focus on the modeling aspect described in this section.

To characterize bus transactions, the author presents a simple state model that describes transitions from one bus state to a subsequent state given possible states of the bus. There are four possible states defined regardless of the type of transaction:

Idle: The idle state defines the bus to be unused by any device. During this state, no data is being transferred and device requests to access the bus are immediately granted. At least two cycles of latency are needed before the address is transmitted since arbitration is not hidden in the idle state.

Busy: The busy state indicates that the bus granted access to a device and is performing an address or data transfer. Wait states are added if either the initiator or target cannot accept or deliver data. This state does not define how the bus is being used, only that is in use.

Busy/Data: This state represents the bus is currently busy and it is known that data is transferred.

Busy/Non-Data: This state represents the bus is currently busy, however, it is known that no data is transferred (i.e.: address or stalling cycle).

Based on the states described, device behavior can be represented by descriptor D:

$$D = (s, d, r) \tag{3.1}$$

Where s defines a *non-data phase*, d defines a *data phase*, and r defines the *recovery phase*. The recovery phase is not a physical bus phase and only guarantees that devices do not access the bus during this time. Using the device descriptor provides a basis to determine upper bounds for bandwidth and latency when arbitrary devices interact.

Our simulator also defines the notion of a descriptor; however, we decompose it into a master and target descriptor, each having different parameters. It should also be noted that we provide a strict definition of recovery period, which is defined in Section 7.1.1.1.

3.1.2 Model for Identical Devices

Before describing the behavior of arbitrary devices, the author first presents a model for n identical devices. The term "Identical Devices" defines all devices which can be described by the same device descriptor D.

The worst-case latency, τ_{lat}, occurs when one device must wait all other devices are granted bus access and is given by:

$$\tau_{lat} = (n-1)(s+d) \tag{3.2}$$

Thus, the worst-case bandwidth occurs when every access to the bus is delayed by this amount and is calculated by:

$$bw = bw_{pci}\frac{d}{(s+d+r)+((n-1)(s+d)-1)} \tag{3.3}$$

There are two cases which must be considered depending on the length of the recovery phase. It is noted that each device is assumed to operate at its maximum bandwidth (i.e.: there are no gaps between two requests).

1. In the first case, $r \geq (n-1)(s+d)$ and thus a device can access the bus immediately following the end of its recovery phase. Figure 3.1 illustrates an example of for two devices.

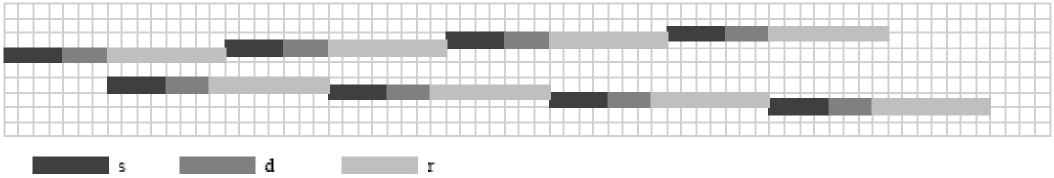


Figure 3.1: Two identical devices with D = (4, 3, 8)

It must be noted that each device has the same recovery length and thus it is not possible for a device to perform multiple transactions while another device recovers.

2. In the second case, $r < (n-1)(s+d)$ and thus a device is ready before all other devices have finished their transaction. Figure 3.2 illustrates this example.

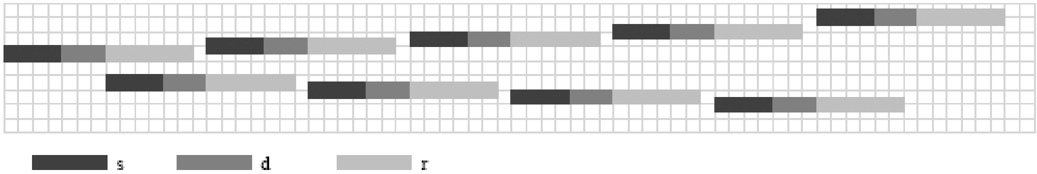


Figure 3.2: Two identical devices with D = (4, 3, 6)

The combination of both cases can be used to calculate the bandwidth each device can achieve by:

$$bw = bw_{pci} \frac{d}{\max((s+d)n, (s+d+r))} \tag{3.4}$$

The two cases above can also be used to calculate the bus utilization, $U(n)$, for n active devices. The bus utilization is defined by the ratio of the number of cycles where the bus is occupied and the number of elapsed bus cycles which is calculated by:

$$U(n) = \frac{(s+d)n}{\max((s+d)n, (s+d+r))} \tag{3.5}$$

From Equation 3.4, the bandwidth generated by *n* devices, where each device sends at its maximum rate is calculated by:

$$bw(n) = bw_{pci} \frac{dn}{\max((s+d)n, (s+d+r))} \tag{3.6}$$

Therefore, the bandwidth of a system with a single device is calculated by *bw*(1). If *r* is set to zero and *s* is set to the overhead, *ov*(*d*), then Equation 1.1 is realized. It must be noted that the minimum value of *s* during a read is 2 (address plus turn-around cycle), while the minimum value for a write is 1 (address cycle). Arbitration does not require an extra cycle under high bus load since arbitration is hidden.

Finally, the maximum number of devices that can be connected to the bus, n_{max}, where each device can send at its maximum possible bandwidth is calculated by:

$$n_{\max} \left\lfloor \frac{r}{s+d} \right\rfloor + 1 \tag{3.7}$$

3.1.3 Model for Arbitrary Devices

Modeling a PCI system with identical devices employs an ideal situation. In real-life, devices have different properties and perform varying functionality. Thus, the values of our device descriptor *s*, *d*, and *r* may vary which might not cause the neat interleaving caused by modeling identical devices. Overall bus utilization is less than theoretically possible since it is less than the sum of the individual device's utilization. The author models this problem from Liedtke *et al*. [20] and proposes a new technique which we summarize in this following section.

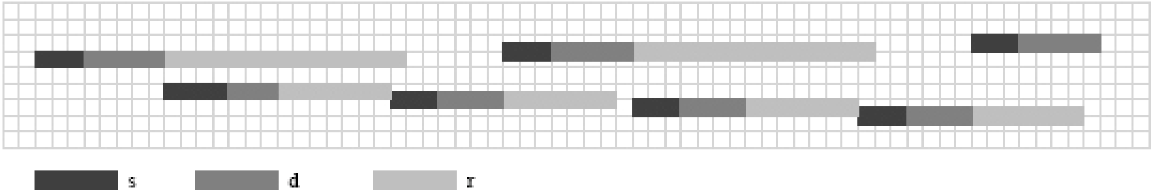


Figure 3.3: Two sample devices with $D_1 = (4, 3, 6)$ and $D_2 = (3, 4 ,7)$

Figure 3.3 illustrates two arbitrary devices where one device can perform multiple transactions while the other device recovers. The following paragraphs develop models for two devices and then generalize for multiple devices.

3.1.3.1 Two Devices

Given two devices with descriptors D_1 and D_2, in order for D_2 to perform multiple transactions, the following must hold:

$$s_2 + d_2 + r_2 < r_1 \tag{3.8}$$

Therefore it is shown that D_2 can at least start another transaction while D_1 has yet to finish its recovery phase. It is assumed that $s_1 + d_1 + r_1 \geq s_2 + d_2 + r_2$. The *displacement* of $s_2 + d_2$ in relation to $s_1 + d_1$ is denoted by a:

$$a \in [0, r_2 - (s_1 + d_1)] \tag{3.9}$$

Where $r_2 - (s_1 + d_1)$ returns 0 when $s_1 + d_1 > r_2$.

The *block time* (*b*) is defined as the distance between two starts of D_1 at $s_1 + d_1 > r_2$ which depends on the displacement (*a*) between the end of the transaction of D_1 and the start of the transaction of D_2 by *b* = *b*(*a*). The number of transactions D_2 can perform with length $s_2 + d_2$ during the interval $[s_1 + d_1 + a, s_1 + d_1 + r_1)$ is denoted by *v* = *v*(*a*).

Figures 3.4 and 3.5 illustrate two cases, explained below, where the second device D_2 can perform at least two (possibly more) transactions during the recovery phase of D_1. It is of note that ~ represents a period of time no data can be transferred since it is in use by another device.

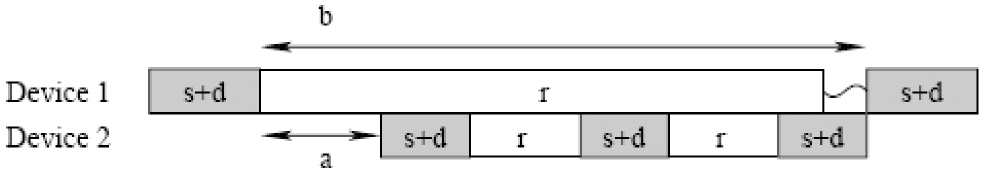


Figure 3.4: Two devices, multiple transactions, with delay

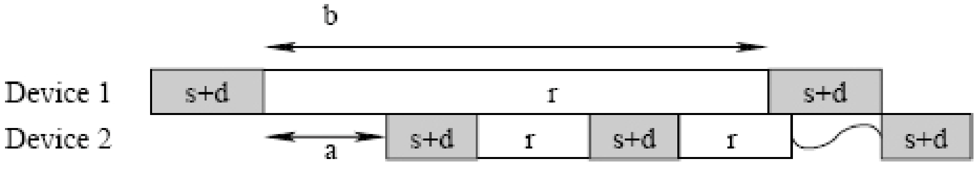


Figure 3.5: Two devices, multiple transactions, without delay

The following are the two cases to be considered:

1. In Figure 3.4, D_2 started its last transaction but it exceeds the recovery phase of D_1. Thus, D1 must wait until D_2 finishes its final transaction.

2. In Figure 3.5, the last work phase of D_2 is shorter than the recovery phase of D_1. Thus, D_1 is granted immediate access to the bus since D_2 is in its recovery phase.

The number (v) of transactions that can be performed by D_2 during the recovery phase of D_1 is calculated by:

$$\max\{v \mid r_1 > (v-1)(s_2 + d_2 + r_2) + a_i\}$$

$$v - 1 < \frac{r_1 - a_i}{s_2 + d_2 + r_2} \text{ with } r_1 > a_i \geq r_2$$

$$v = \left\lceil \frac{r_1}{s_2 + d_2 + r_2} \right\rceil \tag{3.10}$$

By setting $a_0 = 0$, the block time b_i and the displacement time a_i is calculated by:

$$b_i = s_1 + d_1 + \max(r_1, (v_1 - 1)(s_2 + d_2 + r_2) + a_{i-1} + (s_2 + d_2))$$

$$a_i = a_{i-1} + v * (s_2 + d_2 + r_2) - b_i \tag{3.11}$$

While Equation 3.11 calculates how often a device is scheduled, there is no closed formula to determine the exact bandwidth and latency. It should be noted that the worst-case latency for D_1 is $\tau_{lat,1} = s_2 + d_2$ and for D_2 is $\tau_{lat,2} = s_1 + d_1$.

3.1.3.2 Multiple Devices

The number of access combinations can be easily described by the formulas given in the previous section. However, this number increases polynomially with an increasing number of devices. In most cases it is sufficient to know the worst-case bandwidth latency. To obtain the formula for worst case latency, it must be assumed that exactly one cycle before the recovery phase of device *x* ends, all other devices request the bus and the first device is chosen per the round-robin arbitration scheme. The worst-case latency when all *n* devices receive the bus for $s_i + d_i$ cycles for device *x* is given by:

$$\tau_{lat,x} = \sum_{i=1, i \neq x}^{n} (s_i + d_i) - 1 \tag{3.12}$$

Using Equation 1.1, the bandwidth a device can achieve can be calculated by:

$$bw_x = \frac{d_x bw_{pci}}{s_x + d_x + r_x + \tau_{lat,x}} \tag{3.13}$$

Therefore, combining Equation 3.12 with 3.13 will result in the worst-case bandwidth of device *x*:

$$bw_x = \frac{d_x bw_{pci}}{s_x + d_x + r_x + \left(\sum_{i=1, i \neq x}^{n} (s_i + d_i) - 1 \right)} \tag{3.14}$$

3.1.4 Concluding Remarks

This paper provided a unique look at modeling the PCI and providing equations to calculate worst-case bandwidth and latency. One of the key aspects of our simulator is the notion of recovery period, which was borrowed from the author’s work as well as the idea of a descriptor. However, our simulator divides the descriptor into a master and target device descriptor, each with its own set of parameters. In addition, the authors work aided us in describing results obtained in our simulator, specifically in Section 8.2. To verify the results, the author used the PCI Pamette [21] and custom software to generate PCI cycles and analyze the results.

3.2 Simulation Software

This section analyzed three papers on simulation software. While only the last paper describes a PCI simulator, we discuss two other software-based simulators that are analyzed and evaluated.

3.2.1 JSIM: A Java-Based Simulation and Animation Environment

A Java-based simulation and animation environment, JSIM, is described in [25]. JSIM is a collection of java classes, which are grouped into five library packages (queue, statistic, variate, process, and event) that can be extended by a user to model complex simulations. As an example, the authors created a simple M/M/1 Bank simulation model utilizing the JSIM simulation library. The concept of the simulation library is built around graph theory, representing models as directed graphs with varying node types. To minimize the amount of coding needed to create a simulation, a graphical design environment was implemented which allows users to model simulations using point and click operations.

However, one of the key features is the ability to recall previous simulation results based on information of the current simulation. This approach, described as Query Driven Simulation (QDS), significantly reduces simulation times and cost by retrieving simulation results that were previously stored.

JSIM is a generic simulation environment which does not model any one particular system; in contrast our simulator only models PCI/PCI-X systems. However, it provides us with insight on implementation details contrasting it with our approach. Our implementation will also use Java; however, we use it only as a middle-tier framework that connects to the simulator and database. Our design also stores detailed information about the simulation in a database, however, it is not query driven. Instead, the user can manually recall previous run simulations and review the results. While both the JSIM design and our design provide a graphical user interface, our design requires no user coding and does not need to be installed onto the user's PC.

3.2.2 Building a Web-Based Federated Simulation System with Jini and XML

In Huang and Miller [26], a collaborative approach to building and running complex distributed simulation models based on JSIM is explored. The design is based on the High Level Architecture (HLA) developed by the Department of Defense (DoD). It defines the concept of a federation – a simulation system built from components called federates. Early implementations of JSIM involved using Enterprise Java Beans (EJB) [27]; however, its thin-client/thick-server oriented design was deemed an imperfect solution for HLA-style simulation systems. The authors found a solution in Jini [28], a set of API's and protocols that allow the development and deployment of distributed systems organized as a federation of services rather than a central controller like EJB. These services form a peer network made available via a lookup service. The loosely-coupled architecture allows objects to be deployed, removed, and relocated without the need to redeploy the entire system. Object mobility is easily obtained through the use of Remote Method Invocation (RMI), allowing services to communicate through the network.

While Jini allows simulation models and model agents the ability to collaborate, one obstacle the authors posed was how they would all understand each other. One solution was to use the eXtensible Markup Language (XML) [29] as the carrier of event data. XML is a general-purpose markup language whose primary purpose is exchange structured data across differing systems. To achieve interoperability, the authors propose devising a common Document Type Definition (DTD) defining a simulation schema. By using Jini and XML, the authors' goal is to turn JSIM into a web accessible distributed simulation system. The GUI is presented as a Java Applet with simulation data.

Since this simulation environment is also based on JSIM, we note that it does not simulate any particular system, however it does provide implementation details which are similar to our system. While our implementation is also web-based, we decided to use the EJB architecture to store and retrieve simulation data. This solution was stated by the author [26] to be "very useful for managing simulation data on the database side." Our proposed design uses HTML and JavaScript to render the GUI in a browser in contrast to the Java Applet design proposed by the author.

3.2.3 Design and Implementation of PCI Bus Based Systems

In this section, we summarize a PCI simulator described in Section 7.5 of [14]. The simulator was designed in part to compare results based on modifications of the PCI protocol by the author in order to improve performance and efficiency. The author offers the introduction of the Split Cycle (not to be confused with the Split Cycle introduced by the PCI-X protocol) which allows masters to send read requests to targets. Once the request has been filled by the target, it then uses a normal write transaction to provide data to the original requestor. This method eliminates wait cycles initiated by the target while it waits for the data, thus wasting valuable bandwidth that could be used by other devices.

3.2.3.1 Collecting PCI Traffic

In order to accurately model PCI traffic it is necessary to retrieve bus cycles of a typical system. There are three methods identified by the author which would prove to be useful in evaluating our simulator. We briefly describe the three methods in the following sections.

3.2.3.1.1 Logic Analyzer

A logic analyzer allows collection of detailed bus cycle information. This information can be analyzed to extract high level information needed for modeling purposes. However, logic analyzers offer limited buffer space and would not accommodate the sheer amount of PCI transactions needed for proper modeling.

3.2.3.1.2 PCI Bus Monitor

A PCI bus monitor allows transactions to be analyzed passively. However, data acquisition is usually stored in raw form which provides additional information that may not be needed. Secondly, the monitor has to be designed such that it can connect to a mass storage device or some other connection for data storage.

3.2.3.1.3 PCI Monitor Card

While the previous two methods looked at collecting transactions on a cycle by cycle basis, another approach is to collect statistics about individual transactions. A PCI monitor card could be attached to the bus loaded with the configuration header of each device attached to the bus. Thus, the monitor would only need to look at the destination address of each transaction to distinguish between PCI transactions. The PCI monitor would collect the following information:

1. Average, minimum, and maximum initial latency cycles – This can be obtained by measuring the time from **FRAME#** being asserted until the first data word.

2. Average, minimum, and maximum burst latency cycles – This is obtained by measuring the number of cycles between consecutive words in a burst transfer.

3. Average, minimum, and maximum transfer burst length – This is obtained my counting the number of data transfers during a single transaction.

4. Average, minimum, and maximum wait states before a retry cycle generation – This is obtained by counting the number of wait cycles until a retry cycle appears.

5. Average, minimum, and maximum burst length before a retry cycle generation – This is obtained by counting the number of data cycles until a retry cycle appears.

6. Relative frequency of each base address access and the cycle type (read or write).

Collecting this information gives a good statistical profile for each device which can be used to simulate bus cycles within a simulator. In addition, statistical profiling requires only a fixed amount of memory. Profiling data can be collected almost indefinitely (provided enough memory is set aside).

3.2.3.2 Generating PCI Bus Traffic Simulation

Generating bus traffic can be based on the transactions cycles as described in the first two methods above, or by the statistical information gathered which would generate random cycles. We note that the author modified the PCI protocol and the transaction data used to generate cycles. The modifications are as follows:

1. The modified PCI protocol allows for a new transaction to be started while the previous transaction is being retried, thus simulating the Split Transaction proposed by the author.

2. All idle bus cycles are removed since it is necessary to test the efficiency of the protocol and not the chipset.

3. Captured transactions are optimized based on PCI optimization rules (byte merging, consecutive word merging). This is also necessary in order to test bus efficiency and not chipset efficiency.

3.2.3.3 PCI Simulation Implementation

Implementation of the PCI simulation framework was developed exclusively using C++. While writing the simulator in Verilog was considered by the author, it was ruled out in favor of C++ for the following reasons:

1. The cycle based approach used in the C++ simulator yielded simulations times that were orders of magnitude quicker when compared with the Verilog simulator.

2. By exploiting class inheritance and object oriented design, C++ allowed for better customization and no duplication.

3. Verilog licensing could limit availability.

The simulation is divided into three parts:

1. The simulator provides a cycle based library which supports synchronous modules. Each module can contain one or more register objects, storing a single bit, which are connected by wire objects. Both register and wire objects are represented by one of six logic levels: low, high, high-Z, unknown, pull-high, and pull-low.

2. The simulator also provides generic models consisting of a PCI arbiter, PCI target, PCI master, PCI backplane, and a PCI monitor. The modules allow for fine grained control over their behavior including master and target burst length, wait states, initial latency, burst latency, and retry generation by a target. Different arbitration schemes may be specified and new ones may be added by the user.

3. In addition to the generic models, modified models are also included to allow for support of the proposed enhancements.

3.2.3.4 PCI Simulation Environment

This section outlines the environment used to simulate PCI bus transactions. Instead of using actual transactions, statistics were extracted from the data gathered. The hypothetical system used to perform the simulation is illustrated in Figure 3.6.

The system is based off the Intel 440FX PCI chipset [22] and uses the following data for the simulation:

- For system memory:
    - Average initial latency for reads is between 8 and 12 cycles long.
    - Average initial latency for writes is between 3 and 4 cycles long.
    - No wait states for cache line transfer on 32 byte boundary, and 1 cycle latency for transfers which cross a cache line boundary.
    - Burst length up to 4K page boundary. Transfer terminates at a cache line boundary.
- For the CPU:
    - Average initial latency for writes to the PCI bus is between 1 and 3 cycles with a burst length of up to 7 words.
    - No master initial latency for reads from the PCI bus (1 word per transaction).
- Arbitration:
    - Round-robin arbitration scheme.
    - Bus granted to next master after current master releases the **REQ#** line, or when the Multi-transaction Timer (MTT) expires.

The system consists of the following PCI devices to be simulated: a target representing system memory, a target representing a VGA card, a master representing a SCSI controller, a master representing an Ethernet card, and a master representing the CPU. The performance characteristics for the system memory and the VGA card are summarized in Table 3.1. Performance characteristics for the Host Bridge, SCSI controller, and Ethernet card are summarized in Table 3.2. The CPU reads directly from the VGA card and will not support bursts. Conversely, the CPU writes directly to the VGA card and bursts are allowed.

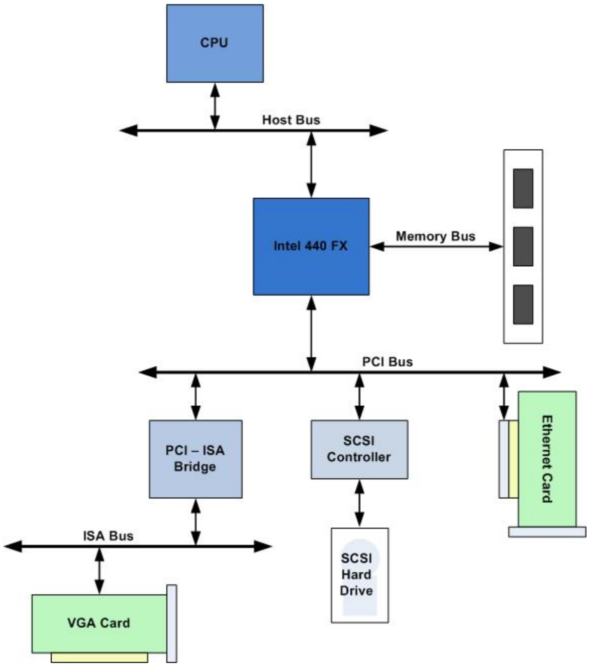


Figure 3.6: Hypothetical system used for PCI simulation

Table 3.1: Simulation target parameters (continued on pg. 40)

Type	VGA Target	System Memory Target
Decode Speed	Medium	Fast
Initial Wait States (cycles)	Random (0 to 40)	Read: Random (8 to 12) Write: Random (3 to 4)
Burst Wait States (cycles)	0	32-byte Boundary: 1

		Other: 0
Burst Length (cycles)	Random (0 to 10)	Stop at 4k Boundary
Initial Retry Threshold (cycles)	16	16
Burst Retry Threshold (cycles)	8	8

Table 3.2: Simulation target parameters (continued from pg. 39)

A description of each of the target parameters listed in Table 3.1 is provided below:

Decode Speed: A target has four possible speeds for asserting **DEVSEL#** after **FRAME#**: fast, medium, slow, and subtractive. A fast target responds within one clock cycle, a medium target within two cycles, and a slow target within three cycles. If no agent claims the transaction within three cycles, then a subtractive-decode agent may claim it on the fourth cycle.

Initial Wait States: This is the number of clock cycles before the target is ready to send or receive the first word.

Burst Wait States: The number of clock cycles before the target is ready to send or receive burst words, after the first word.

Burst Length: The number of words the target is either able to send or accept.

Initial Retry Threshold: A target that has to wait *n* number of cycles can either (1) wait *n* clock cycles with **TRDY#** deasserted and then assert **TRDY#**, or (2) retry the cycle. If *n* is greater than the initial retry threshold, then a retry will be generated, else, wait states will be added. This only applies to the first word in a burst and is limited to 16 clock cycles per the PCI 2.1 specification.

Burst Retry Threshold: Similar to the initial retry threshold, this applies to subsequent burst words and is limited to 8 per the PCI 2.1 specification.

Type	CPU Host Bridge	Ethernet Card	SCSI Controller
Read/Write Ratio	80/20 (Random)	80/20 (Random)	20/80 (Random)
Initial Wait States (cycles)	0	0	0
Burst Wait States (cycles)	0	0	0
Burst Length (cycles)	Read: 1 Write: Random (1 to 8)	Random: 8 to 384	128
Master Latency Timer (cycles)	48	48	48
Retry Overhead (cycles)	0	N/A	N/A
Transaction Count	200	100	100

Table 3.3: Simulation master parameters

A description of each of the master parameters listed in Table 3.2 is provided below:

Read/Write Ratio: This is the ratio between the percentage of PCI read commands and write commands for this master.

Initial Wait States: The number of clock cycles before the master is ready to send or receive the first word.

Burst Wait States: The number of clock cycles before the master is ready to send or receive burst data, after the first word.

Burst Length: The number of words the master is either able to send or accept.

Master Latency Timer: This is the amount of bus burst cycles the master is permitted per operation. If the master is bursting data and the **GNT#** line is deasserted, it must release the bus once the latency timer expires. The latency timer decrements by one on each burst cycle.

Retry Overhead: This is for a latency hint aware master, which is part of the author's modified protocol. It is the number of cycles measured from **REQ#** going active until **FRAME#** goes active.

Transaction Count: This is the total number of transactions before the master becomes inactive. Setting this to zero permanently disables the master.

3.2.3.5 Concluding Remarks

Our simulator also uses the simulation master and target parameters as described in tables 3.1 and 3.2. However, our original performance parameters were culled from the PCI and PCI-X specifications. Specifically, we borrowed the burst length and transaction count from the author’s work presented. We also did not use the retry overhead, since that parameter is specific to the author’s work. We also added two key components of injection rate and recovery period which aid in accurately describing a devices behavior on a bus. In addition, we also use a PCI analyzer to extract statistics and then use those statistics to feed our simulator and compare the results. One of the major differences of our approach is the design of our GUI. The referenced implementation strictly uses a command-line interface. In addition, adding devices and changing parameters requires knowledge of C++ as well as the architecture. With each alteration, a recompile is necessary which necessitates a C++ compiler. In our proposed design, adding devices and changing parameters is easily accomplished via a graphical web user interface.

CHAPTER 4

PERIPHERAL COMPONENT INTERFACE

The *Peripheral Component Interconnect* (PCI) is a *local bus* designed to allow components to connect to an industry standard high-speed bus with minimal cost. Work on PCI originally began at Intel's Architecture Lab circa 1990 and by 1992 the first component-level specification was released, named PCI 1.0. Since the original release of the PCI specification, successive versions were released including the latest version PCI 3.0. This chapter gives an overview of the PCI bus covering version 2.2.

4.1 PCI Bus Characteristics

The Peripheral Component Interconnect, or PCI, bus is a 32 or 64-bit synchronous multiplexed bus which is designed to interconnect high-performance components. Data transfers in PCI are always between an *initiator device*, one which initiates the data transfer, and a *target device*, the receiver of the data. Every device that is able to initiate a data transfer is called a *master*. Coordinating bus access between multiple devices requires the use of an *arbiter* which uses an arbitration algorithm to decide which device has the right to use the bus. Once the bus has been granted to the device, a data exchange can take place. The basic bus transfer mechanism in PCI is known as a *burst* which is composed of an address and one or more data words. A *PCI host bridge* is used when main memory is addressed by a device. The host bridge routes data between the main memory and the PCI bus. However, main memory cannot initiate a data transfer and thus is only considered a target device and cannot become a bus master. To increase efficiency of the bus, PCI supports *posting* which allows a transaction to be complete on the originating bus before being completed on the target bus. For example, when a device wants to write to main memory, the data is written to a buffer on the PCI host bridge. The data is then streamed to main memory as long as the buffer is not empty.

A write operation allows data to be transferred from the initiator to a target device. After receiving access to the bus, the initiator drives the physical target address onto the signal lines and then immediately starts sending the data. To perform a read operation, the initiator puts the source address on the bus. At this point, a one cycle delay occurs to avoid contention, called a *turnaround cycle*. The target device, with the corresponding address, will handle the request from this point forward by delivering the data. The transfer of data happens without the involvement of the CPU, whether it happens between two devices or between a device and main memory.

There are four *interrupt* lines connected to each device connected to the bus. A *multifunction device*, consisting of two or more logical functions, may use any combination of the four. A single-function device, which embodies exactly one logical function, may use only a single designated line. The mapping of system interrupt lines to PCI interrupt lines, through the PCI host bridge, is implementation dependent and is configured by the BIOS upon system initialization.

The PCI bus allows three types of addressing modes which target three different types of address spaces. To allow for configuration and plug-and-play capabilities, PCI offers 256 bytes of *configuration space*. The configuration space contains information such as vendor identification, vendor name, device type, as well as protocol-specific information such as a *latency timer*, the interrupt line, addresses and sizes of other spaces. The PCI Special Interest Group (PCISIG) assigns a unique 16-bit number to each vendor. A 16-bit number is freely chosen by the vendor and used as the device name. Combining both the vendor and device number forms a unique identifier for any device. Configuration of the device is addressed by *configuration cycles* which are generated by the PCI host bridge.

To exchange data, PCI devices incorporate both *memory space* and *I/O space*. The main memory and the memory space of PCI devices belong to a single physical address space. Therefore, the physical address is sufficient in order to select a device. Memory space can be accessed by any bus master; however, I/O can only be accessed via special I/O cycles. In general, I/O transactions offer poor performance and are not recommended for large data transfers. This is echoed by the PCI specification which recommends using memory space over I/O space.

4.2 Used Terms and Conventions

In order to build a basis for the following sections, we provide a list of terms in Section 4.2.1 as well as some common naming conventions in Section 4.2.2.

4.2.1 Used Terms

Arbitration: Arbitration is the process of selecting one device amongst others which are trying to gain access to the bus. The device selected then is granted ownership of the bus as determined by the arbitration algorithm. Arbitration is a hidden process, in a sense that arbitration is performed during the current initiator’s bus transfer and thus does not consume bus cycles.

Burst Transfer: A burst transfer is the basic transfer mechanism. It is the composition of an address and multiple data words with contiguously increasing addresses. The length of the burst can vary for each transaction. The maximum burst length is limited by the PCI specification.

Central Resource: The central resource is an element of the host system and is usually part of the host processor’s chipset. It provides bus support such as clock and reset generation as well as bus arbitration and pull-up resistors.

Initiator Device: The initiator device is a master that has been granted bus access by the arbiter which is then allowed to initiate bus transactions.

Latency: Latency is the number of bus clocks between when the master requests access to the bus and when it is granted ownership by the arbiter. This is caused, in general, by an occupied bus. Both initiators and targets are limited as to how much time they can occupy the bus in order to achieve a low latency. The amount of latency takes into account the length of the burst as well as the number of wait states.

Latency Timer: Every master that is capable of bursting more than two words during a single transaction must support a latency timer that is accessible to the host processor in the device’s configuration space. This is a programmable timer that limits the amount of time a device has access to the bus. The latency timer automatically is loaded into the device’s register once a transaction starts. The timer is then decremented with each bus cycle. If the latency timer expires before the current transaction is finished, then the device must terminate the transaction within three bus cycles.

Master: A master is any device that is capable of initiating bus transactions.

Stalling Cycle: If the initiator is not able to send data or the target is not able to receive data during a transaction, a stalling cycle is inserted. Although no data is transferred, the bus is still occupied. A device can insert up to three stalling cycles, after which the device must terminate the transaction.

Target Device: A target device is one which responds to the transaction initiated by the initiator by recognizing its address during the address phase.

Transaction: A transaction, in the context of PCI, is an address phase followed by one or more data phases.

4.2.2 Document Conventions

Asserted / Deasserted	The terms *asserted* and *deasserted* refer to the actual state of the signal on the clock edge and not to the signal transition.

Edge / Clock Edge	The edge, or clock edge, refers to the rising clock edge unless specified otherwise.
#	A **#** sign at the end of a signal name indicates that it is active, or asserted, in the low voltage state. A signal absent of the **#** sign indicates the signal as being asserted in the high voltage state.
[n::m]	The notation [n::m], where n and m are integers such that $m > n$, represents an array of signals with $n - m + 1$ members. Therefore, **AD[31::0]** represents a 32-bit address with **AD[0]** the least significant bit.

4.3 PCI Bus Signal Descriptions

A PCI target interface requires a minimum of 47 signals, while a master interface requires a minimum of 49 signals as shown in Figure 4.1. This is sufficient for a 32-bit bus running at a maximum clock rate of 33 MHz. An additional 51 signals define optional features such as interrupts, interface control, a JTAG interface, and 64-bit data transfers. Our implementation implements all signals as specified in this section.

4.3.1 Signal Types

The PCI standard defines the following signal types:

in	*Input* is a standard input signal.

out	*Output* is a standard output signal.
o/d	*Open Drain* signals are wire-ORed and allow multiple drivers to drive a signal to a low state. If no agent is driving the signal, a pull-up register is required to keep it in a high state.
t/s	Tri-State is a bi-directional input/output signal and is a shared bus signal which may be driven by only one driver at a time. When an agent stops driving a tri-state signal, a new agent must wait at least one cycle, known as a *turnaround*, before it can drive the same signal to prevent bus contention.
s/t/s	*Sustained Tri-State* is an active low tri-state that is owned and driven by only one agent at a time. In contrast with **t/s** signals, the **s/t/s** signal must be driven high for at least one clock cycle before tri-stating.

4.3.2 Signal Groups

The PCI specification organizes signals into functional groups shown in Figure 4.1. The signal name, signaling method, and description are described in the following sections.

4.3.2.1 System Signals

CLK	**in**	*Clock* provides timing for all PCI transactions and it is an input to all devices. Signals are sampled on the rising edge,

except for **RST#**, **INTA#**, **INTB#**, **INTC#**, and **INTD#**.

RST#	**in**	*Reset* brings PCI registers, sequencers, and signals into a consistent state. Whenever **RST#** is asserted, all PCI output signals must be tri-stated, or driven to their *benign* state.

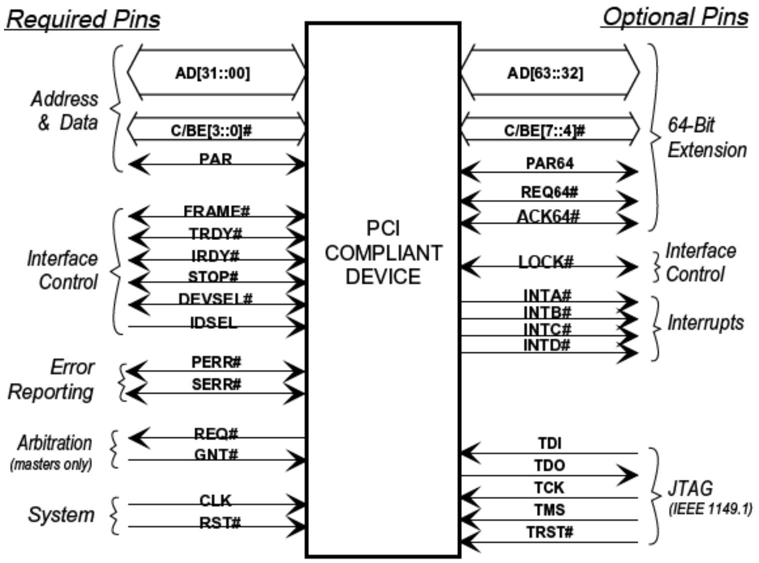


Figure 4.1: PCI bus signals as defined by the PCI standard

4.3.2.2 Address and Data Signals

AD[31::0]	**t/s**	*Address* and *Data* signals are multiplexed on the same set of pins. A PCI bus transaction consists of an address phase followed by one or more data phases. All PCI transactions start with the address signals being driven on the first clock cycle. During a read transaction, the second clock cycle is a turnaround at which point the target enables its buffers and drives the results onto the bus. Data is transferred starting on

		the third clock cycle except during a write transaction where data can be transferred on the second clock cycle since no turnaround if necessary.
C/BE[3::0]#	**t/s**	*Bus Commands* and *Byte Enable* signals are multiplexed on the same pins. During the address phase, **C/BE[3::0]#** define a bus command (Table 4.1). During the data phase, **C/BE[3::0]#** are used as byte enables to determine which byte lanes carry valid data.
PAR	**t/s**	PAR provides even parity across the **AD[31::0]** and **C/BE[3::0]#** signals. These signals are stable and valid in the following situations: 1. Once clock cycle after an address phase. 2. One clock cycle after master asserts **IRDY#** on write transaction. 3. One clock cycle after target asserts **TRDY#** on a read transaction.

C/BE#3	**C/BE#2**	**C/BE#1**	**C/BE#0**	**Command Type**

0	0	0	0	Interrupt Acknowledge
0	0	0	1	Special Cycle
0	0	1	0	I/O Read
0	0	1	1	I/O Write
0	1	0	0	Reserved
0	1	0	1	Reserved
0	1	1	0	Memory Read
0	1	1	1	Memory Write
1	0	0	0	Reserved
1	0	0	1	Reserved
1	0	1	0	Configuration Read
1	0	1	1	Configuration Write
1	1	0	0	Memory Read Multiple
1	1	0	1	Dual-Address Cycle
1	1	1	0	Memory Read Line
1	1	1	1	Memory Write and Invalidate

Table 4.1: PCI bus command types

4.3.2.3 Interface Control

FRAME#	**s/t/s**	*Frame* is driven by the current master indicating the beginning of a PCI transaction. When the bus is idle, **FRAME#** is driven high (deasserted). This signal is driven low (asserted) when a master begins a new transaction. Data transfer continues while **FRAME#** is asserted. Deasserting **FRAME#** signals the transaction is in its final data phase or the transaction has completed.
IRDY#	**s/t/s**	*Initiator Ready* is used by the bus master to indicate that it is able to complete the current data phase. During a read transaction **IRDY#** is asserted to indicate that it is prepared to accept data. During a write transaction, **IRDY#** is asserted to

		signal valid data is present on **AD[31::0]**. The actual data transfer will only take place when both **IRDY#** and **TRDY#** are asserted.
TRDY#	**s/t/s**	*Target Ready* is used by the selected target to indicate that it is ready able to complete the current data phase. When a master device is performing a write transaction, the target will deassert **TRDY#** until it is ready to accept data. During a read transaction, the target will assert **TRDY#** indicating valid data is present on **AD[31::0]**. The actual data transfer will only take place when both **IRDY#** and **TRDY#** are asserted.
STOP#	**s/t/s**	*Stop* indicates that the selected target requests to terminate the current transaction. The state of **TRDY#** and **DEVSEL#** while **STOP#** is driven determines the termination type: disconnect with data, disconnect without data, target retry, or target abort.
LOCK#	**s/t/s**	*Lock* (optional) indicates an *atomic* operation to a bridge that may require multiple transactions to complete. Non-exclusive transactions may proceed to a bridge when **LOCK#** is asserted. One must note that a grant to start a transaction

		does not guarantee control of **LOCK#.** Control is obtained in conjunction with **GNT#**.
IDSEL#	**in**	*Initialization Device Select* is used by the bus master during configuration transactions. Configuration transactions can only be accepted by a target when **IDSEL#** is driven high on the first clock cycle. Every PCI slot has a unique **IDSEL#** line and thus can be accessed before a device is configured.
DEVSEL#	**s/t/s**	*Device Select* indicates the driving device has decoded its address as the target of the current transaction. During a read or write transaction **FRAME#** is asserted on the first clock cycle in conjunction with the requested address on **AD[31::0]**. Target devices on the bus latch the **AD[31::0]** and compare it with their base address registers. The target device, whose address is in the range of **AD[31::0]**, must respond within 4 clock cycles by asserting **DEVSEL#**. If no target responds within the four clock cycles, then it is assumed that no target exists.

4.3.2.4 Arbitration (Bus Masters Only)

REQ#	**t/s**	*Request* indicates to the central arbiter that a bus master wants access to the bus. Every potential master has its own point-to-point **REQ#** line connected to a central arbiter.

GNT#	**t/s**	*Grant* indicates to a device that has its **REQ#** asserted that it has been granted bus access by the central arbiter. Every potential master has its own point-to-point **REQ#** line connected to a central arbiter. When a master’s **GNT#** is asserted, access to the bus is allowed only after the current cycle taking place has ended which is indicated by the deassertion of both **FRAME#** and **IRDY#**.

4.3.2.5 Error Reporting

PERR#	**s/t/s**	*Parity Error* is used to report data parity errors during all PCI transactions except for a *Special Cycle*. The bus master drives **PERR#** during a read and a target drives it during a data write.
SERR#	**o/d**	*System Error* is used to signal errors such as address parity errors, data parity errors on a Special Cycle command, or any other potentially catastrophic system error. In contrast with **PERR#**, **SERR#** can be driven by any target device at any time.

4.3.2.6 Interrupt

INTA#	**o/d**	*Interrupt A* is used to request an interrupt.
INTB#	**o/d**	*Interrupt B* is used to request an interrupt and is only used in

		multi-function devices.
INTC#	**o/d**	*Interrupt C* is used to request an interrupt and is only used in multifunction devices.
INTD#	**o/d**	*Interrupt D* is used to request an interrupt and is only used in multifunction devices.

4.3.2.7 64-bit Bus Extension

AD[63::32]	**t/s**	*Address* and *Data* are multiplexed onto the same set of pins and supply an extra 32 bits during 64-bit data transfers.
C/BE[7::4]#	**t/s**	*Command* and *Byte Enables* are multiplexed onto the same pins and are used during the data phase of a 64-bit transfer to indicate valid byte lanes. During the address phase, **C/BE[7::4]#** is unused.
REQ64#	**s/t/s**	*Request 64-bit Transfer*, when asserted by a bus master, indicates a request for a 64-bit transfer. **REQ64#** obeys the same timing rules as **FRAME#**.
ACK64#	**s/t/s**	*Acknowledge 64-bit Transfer* indicates that the selected target is ready to accept a request for a 64-bit transfer. **ACK64#** obeys the same timing rules as **DEVSEL#**.

PAR64	**t/s**	*Parity Upper DWORD* is similar to **PAR** except it is even parity over **AD[63::32]** and **C/BE[7::4]#**. **PAR64** obeys the same timing rules as **PAR**.

4.3.2.8 JTAG (IEEE 1149.1) / Boundary Scan

TCK	**in**	*Test Clock* is the JTAG clock used to control data shifting in and out of the device.
TDI	**in**	*Test Data Input* is used to shift data and instructions into the device.
TDO	**out**	*Test Output* is used to serially shift data out from the device.
TMS	**in**	*Test Mode Select* is used to select the state of the controller of the device.
TRST#	**in**	*Test Reset* is used to reset the device.

4.3.2.9 Additional Signals

PRSNT[1:2]#	**in**	The *Present* signals are provided for add-in boards and is used to indicate the presence of an add-in board as well as the power requirements to the motherboard.
CLKRUN	**in,**	*Clock Running* is an optional input signal used to determine

	o/d, **s/t/s**	the state of **CLK**. It is used as an output to control the state of the clock. When **CLKRUN** is asserted it indicates the clock is running at its normal speed. When deasserted, it indicates a request to slow or stop the clock. This is used to as a power saving mechanism in mobile devices.
M66EN	**in**	The *66MHZ_ENABLE* indicates to a device whether the bus is operating at 66 MHz or 33 MHz.
PME#	**o/d**	The *Power Management Event* signal allows for a device to request a change in the device or system power state.
3.3Vaux	**in**	*The Auxiliary 3.3 Volt Power* signal allows a PCI add-in card to generate power management events even if main power has been turned off.

4.4 PCI Bus Commands

In this section, we take a closer look at the 12 different commands defined by the PCI Specification as shown in Section 4.3.2.2. Unless otherwise noted, we implement all PCI bus commands specified in this section.

Interrupt Acknowledge	The *Interrupt Acknowledge* is a read implicitly addressed to the system interrupt controller. The address bits are irrelevant

	during the address phase and the byte enable indicate the size of the returned vector during the corresponding data phase.
Special Cycle	The *Special Cycle* command is used as a broadcast mechanism on PCI. The predefined event, which includes the *x*86 shutdown and halt events, is identified by an event code. In contrast with other cycles, the special cycles are not acknowledged by bus targets and thus, **DEVSEL#** is not asserted for this cycle.
I/O Read	The *I/O Read* command is used by a bus master to read data from a device mapped to I/O Space.
I/O Write	The *I/O Write* command is used by a bus master to write data to a device mapped to I/O Space.
Memory Read	The *Memory Read* command is used by a bus master to read data from a target device mapped in the Memory Address Space. The target device may perform an anticipatory read as long as there are no side effects. Also, the target must ensure cache coherency of data that is retained in temporary buffers until the PCI transaction completes.

Memory Write	The *Memory Write* command is used by a bus master to write data to a target device mapped in the Memory Address Space. The master assumes responsibility of the coherency of the data to be transferred.
Configuration Read	The *Configuration Read* is used to read the one or more registers in the Configuration Space.
Configuration Write	The *Configuration Write* command is used to write data to one or more registers in the Configuration Space.
Memory Read Multiple	The *Memory Read Multiple* command is similar to that of the Memory Read command with the exception that the bus master indicates that it intends to read one or more cache lines. This is used as a hint to the target device which it may use to prefetch data or ignore the hint and treat the request as a normal Memory Read. It is recommended that the Memory Read Multiple command be used when reading more than cache lines.
Dual Address Cycle	The *Dual Address Cycle* command is used to transfer a 64-bit address to devices in 32-bit PCI slots. The Dual Address Cycle is sent in conjunction with the upper 32 address bits on

	AD[31::0] followed by the command (i.e.: Memory or I/O Read/Write).
Memory Read Line	The *Memory Read Line* command is similar to that of the Memory Read command with the exception that the bus master indicates that it intends to read a complete cache line. As with the Memory Read Multiple command, this provides a hint to the target device which may be used to prefetch data. The target may also chose to ignore the hint and treat it as a Memory Read command.
Memory Write and Invalidate	The *Memory Write and Invalidate* command is similar to the Memory Write command except that the initiator guarantees to write a full cache line in a single transaction. This command is useful when a transaction hits a dirty cache line in a writeback cache. It may simply invalidate the cache line without writing it back since the initiator is updating the entire cache line.

4.5 PCI Address Structure

CPU and PCI devices need access to system memory which is used by device drivers to pass information between them. The shared memory typically contains control and status registers. Although the CPU’s system memory could be used to store this information, access to this portion of the shared memory would need to be controlled by the CPU and thus would cause a significant decrease in performance. PCI devices, therefore, have their own independent memory spaces. The PCI Bus Specification defines three physical address spaces: Memory, I/O, and Configuration – all of which are implemented in our simulator. The following section provides a brief discussion of each of these address spaces.

4.5.1 Memory Address Space

The Memory Address Space consists of mapped device control status registers (CSR), mapped device buffers, and system memory space. A PCI address is either 32-bits or 64-bits wide and can be generated by both master and target devices. In order to generate a 64-bit address from a 32-bit device, a Dual Address Cycle is used. The PCISIG recommends that resources used by a device should be mapped into memory space.

AD1	AD0	Burst Order	Description
0	0	Linear Incrementing	The next address in the burst is the next sequential address
0	1	Reserved	
1	0	Cacheline Wrap Mode	Same as Linear Incrementing except the next address wraps to the beginning of the cacheline when a boundary is crossed. The next address is incremented by cacheline size.
1	1	Reserved	

Table 4.2: PCI burst ordering modes

During the address phase of either a read or write, the **AD[31::02]** provides a DWORD aligned address in the Memory Address Space. While **AD[1::0]** is not part of the address decode, it is used to indicate the order of the data being transferred. The burst ordering modes are indicated in Table 4.2.

4.5.2 I/O Address Space

I/O Address Space is 32 bits wide, although most systems do not support more than 16 bits, and consists of mapped device CSRs. PCI I/O Address Space is mostly used for backward compatibility with legacy hardware devices and is not recommended for use in new designs.

Unlike Memory Address Space, I/O **AD[31::0]** provides the full byte address during the address phase. When issuing an I/O transaction, the master must ensure that **AD[1::0]** consist of the least significant valid byte.

The byte enable lines indicate the transfer size and the affected bytes within the DWORD. Valid combinations of **AD[1::0]** and **BE#[3:0]** for the initial data phase is shown in Table 4.3.

AD[1::0]	Starting Byte	Valid BE#[3:0] Combinations
00	Byte 0	xxx0 or 1111
01	Byte 1	xx01 or 1111
10	Byte 2	x011 or 1111
11	Byte 3	0111 or 1111

Table 4.3: Valid byte combinations of BE#[3:0] and AD[1::0] for I/O read/write commands

4.5.3 Configuration Address Space

One of the major advantages of PCI over other bus architectures is its configuration mechanism, allowing plug and play capability. In addition to the two previously discussed address spaces, memory and I/O, each PCI device contains 256 bytes of configuration space. This space is allocated for each logical function and is addressable by an 8-bit bus, 5-bit device, and 3-bit function (referred to as the BDF). The PCI standard divides the 256-byte space into a 64-bytes predefined header region and a 192-byte device dependent region. To access configuration space, a configuration command is sent to a device while **IDSEL** is asserted. In contrast to other PCI signals, IDSEL is uniquely driven and is connected to exactly one of **AD[31::11]** as shown in Figure 4.2.

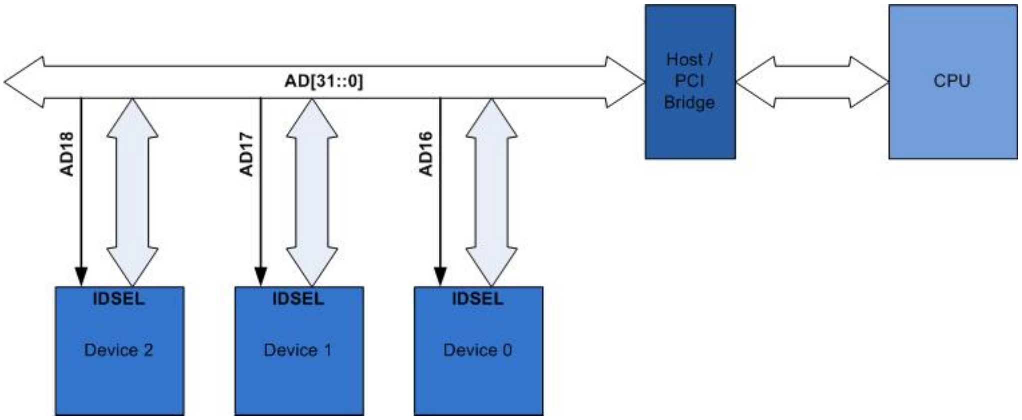
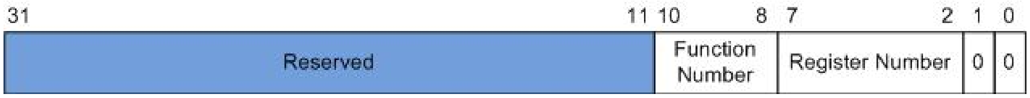


Figure 4.2: IDSEL during a configuration command

When performing configuration read/write commands, two types of transactions are defined. Type 0 configuration is issued from a master whose target is located on the same bus. Type 1 configuration is used by a master whose target is located behind one or more PCI-to-PCI bridges. The PCI-to-PCI Bridge decodes the bus number field to determine if the target of the configuration transaction is located behind the bridge. If the bus number is not on the secondary bus, then the transaction is forwarded unchanged. If the bus number matches the secondary bus, the bridge converts the transaction into a Type 0 request. The format of both Type 0 and Type 1 transactions are shown in Figure 4.3.



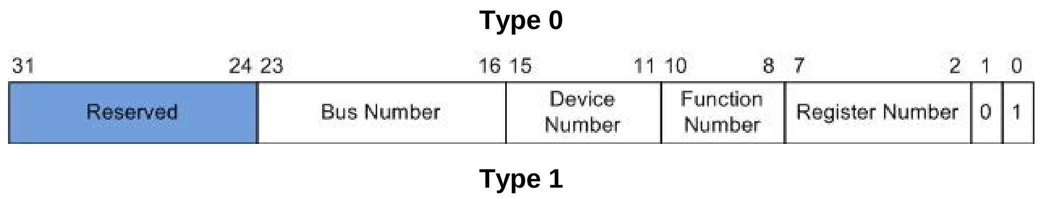


Figure 4.3: Address phase interpretation during a configuration transaction

4.6 PCI Configuration

Prior to plug-and-play, system configuration would generally use jumpers on each add-in card allowing the user to select operating characteristics such as address space, interrupt vectors and DMA channels. However, correctly configuring a system required detailed knowledge of the system itself and its respective hardware. Incorrect configuration would often lead to conflicts with other hardware, leading to unusual system behavior.

PCI allows for plug and play capability, greatly increasing the ease of system configuration. In order to realize this capability, every PCI device must provide functions, via registers in the PCI Configuration Space, which configuration software can utilize. The configuration space is a 256-byte record structure divided into a 64-byte Configuration Header and a 192-byte device-specific region. In this chapter, we focus our attention on the PCI Configuration Header.

4.6.1 Configuration Header

The predefined configuration header helps uniquely identify a device allowing it to be generically controlled. It is divided into two parts – the first 16 bytes are common to all devices while the remaining bytes can have differing layouts depending on what functions are supported by the device. To determine the layout, a Header Type field (located at 0Eh) is defined. There are currently three header types defined: Type 0 which is shown in Figure 4.4, Type 1 which is used for PCI-to-PCI bridges, and Type 2 which is defined for Cardbus Bridges. We currently only implement Type 0, as this is the most common. Information on Type 1 and Type 2 headers can be found in [16]. Support for these headers could be implemented as future work.

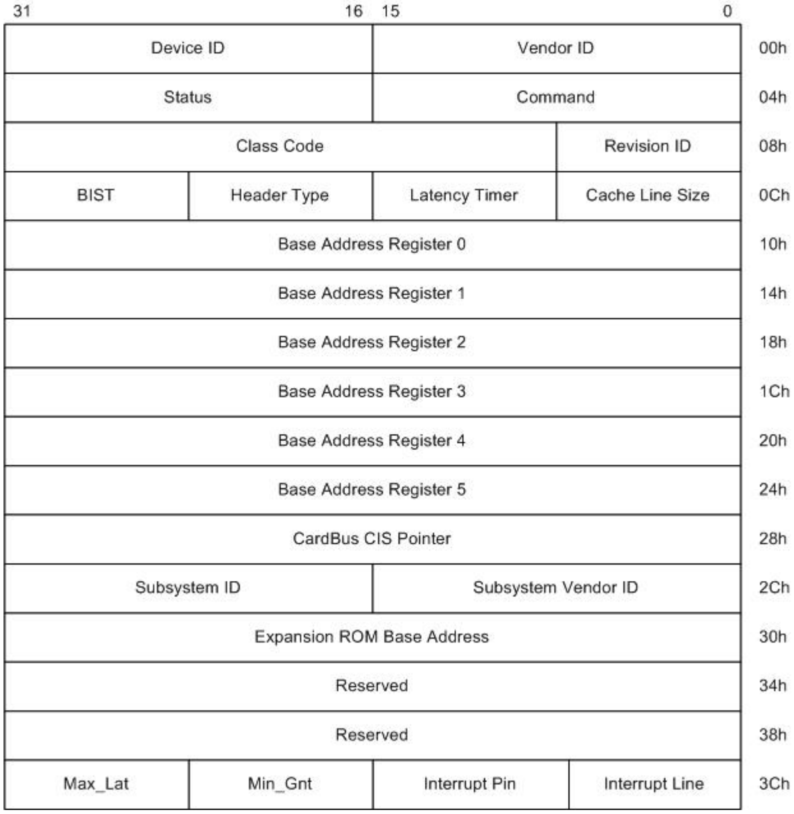


Figure 4.4: Type 0 configuration header

We now proceed to define each of the fields listed in the configuration header, which is broken into sections based on functionality.

4.6.1.1 Identification Registers

There are five fields in the header that serve to identify a device and other operational characteristics. These registers are read-only and must be implemented on all PCI devices.

Vendor ID	This field identifies the manufacturer. It is assigned by the PCI SIG.
Device ID	This field contains a device identifier. It is allocated by the vendor.
Revision ID	This field is assigned by the vendor to provide a revision level for the device.
Header Type	The value of this byte defines the layout of the second part of the header starting at 10h. It also identifies whether or not this device contains multiple functions. If the value of bit 7 is a 0, then it is a single function device. If the value of the bit is a 1, then it is a multifunction device.
Class Code	This field identifies the basic function of the device. It is broken into three byte size fields. The *Base Class* (located at 0Bh) classifies the functional category. The *Sub-class* (located

at 0Ah) more specifically identifies the device type or implementation with the Base Class. The *Programming Interface* (located at 09h) identifies specific register-level implementations.

4.6.1.2 Command Register

The command register is a 16-bit read/write field that provides coarse control over the ability for a device to generate and respond to PCI cycles. When a 0 is written to the register, it denotes that the device is logically disconnected from the PCI bus except for configuration accesses. The layout of the register is shown in Figure 4.5 with a description of the bits in Table 4.4.

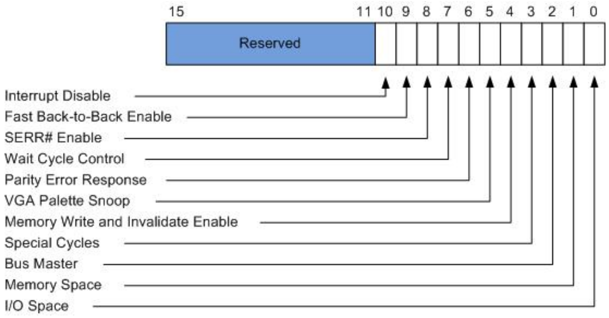


Figure 4.5: Configuration command register layout

Table 4.4: Configuration command register bits (continued on pg. 71)

Bit Location	Description
0	I/O Space access enable bit. Writing a 1 allows the device to respond to an I/O space access. This bit defaults to 0 after **RST#** and is hardwired to 0 if the device has no I/O.
1	Memory Space access enable bit. Writing a 1 allows the device to respond to memory space accesses. This bit defaults to 0 after **RST#** and is hardwired to 0 if the device is not memory mapped.
2	Master enable bit. Writing a 1 enables this device to act as a bus master. This bit defaults to 0 after **RST#** and is hardwired to 0 if the device is target only.
3	Special Cycle enable bit. Writing a 1 enables the device to respond to

	special cycles. This bit defaults to 0 after **RST#** and is hardwired to 0 if the device does not respond to special cycles.
4	Memory Write and Invalidate enable bit. Writing a 1 allows the master to use the Memory Write and Invalidate command if it is capable of doing so. Writing a 0 forces the master to use Memory Write. This bit defaults to 0 after **RST#** and is hardwired to 0 if the device does not support Memory Write and Invalidate.
5	VGA Palette Snoop enable bit. When 1, palette snooping is enabled. When 0, palette access is treated like any other access. This bit is hardwired to 0 if the device does not support VGA Palette Snooping.
6	Parity Error enable bit. When 1, the device responds to parity errors by asserting **PERR#**. When 0, the device ignores parity errors. This bit is hardwired to 0 if the device does not support parity error checking.
7	Address/Data Stepping enable bit. This is hardwired to 1 if the device performs address/data stepping and hardwired to 0 if it is not supported.
8	**SERR#** enable bit. When 1, the device is allowed to assert SERR#. When 0, the device cannot generate **SERR#**. This defaults to 0 after **RST#** and is hardwired to 0 if the device does not support **SERR#**.
9	Fast Back-to-Back enable bit. When 1, enables master to generate fast back-to-back cycles to different target devices. When 0, fast back-to-back cycles are only allowed to the same target. This bit is initialized to 1 by the BIOS if and only if all targets are fast back-to-back capable.
10	**INTx#** disable bit. When 1, the device is prevented from asserting an **INTx#** signal. When 0, the device is able to assert an **INTx#** signal.
11 - 15	Reserved.

Table 4.5: Configuration command register bits (continued from pg. 70)

4.6.1.3 Status Register

The Status register is a read-only field that contains status information related to PCI bus events. While reads to the register behave as normal, writes operate differently in that bits can be cleared, but not set. A bit is set to 1 by the occurrence of an event and is cleared by writing a 1. The layout of the register is shown in Figure 4.6 with a description of the bits in Table 4.5.

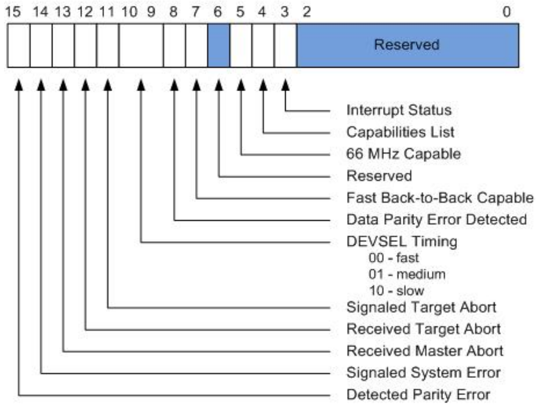


Figure 4.6: Configuration status register layout

Bit Location	Description
0 - 2	Reserved.
3	This is an optional read-only bit where a 1 indicates a device has asserted its interrupt source. If Command Register bit 10 is 0 then the appropriate INTx signal is asserted, else setting it to 1 does not change the state.
4	This is an optional read-only bit where a value of 1 indicates the extended capabilities pointer exists at offset 34h. A value of 0 indicates no extended capabilities exist.
5	This is an optional read-only bit where a value of 1 indicates the device is capable of running at 66 MHz. A value of 0 indicates 33 MHz only.
6	Reserved[1].
7	This is an optional read-only bit where a value of 1 indicates that the device supports fast back-to-back transactions when the transactions are to different targets. Set to 0 otherwise.
8	This is a read/write bit set only by masters. It is set under the following three conditions: 1) the bus agent asserted **PERR#** itself or observed **PERR#** asserted; 2) the bus agent was the master for the operation in which the error occurred; and 3) the Parity Error Response bit is set.
9 - 10	These two read-only bits encode the **DEVSEL#** as 00 for fast, 01 for medium, 10 for slow and 11 is reserved. These bits must represent the slowest time for all commands except for configuration read and write.
11	This is a read/write bit that is set by a target device when it terminates a transaction with a Target Abort.
12	This is a read/write bit that is set by a master device when its transaction is terminated by a Target Abort.
13	This is a read/write bit that is set by a master device when it terminates a transaction with a Master Abort.

---

[1] In the PCI Specification Revision 2.1, this bit indicates whether or not the device supports User Defined Features.

14	This is a read/write bit that is set whenever a device asserts **SERR#**.
15	This is a read/write bit that is set if a parity error is detected, even if parity error handling is not enabled.

Table 4.6: Configuration status register bits

4.6.1.4 Miscellaneous Registers

The following registers are device dependent and they only implemented by devices which support the described functions.

CacheLine Size

This is a read/write register which specifies the system cache line size in DWORD increments. This register is required by any master that implements the Memory Write and Invalidate commands. The value of the register determines whether Read, Read Line, or Read Multiple commands are used during memory accesses. It is also required by targets that want to allow memory bursting using cache-line wrap mode.

Latency Timer

This read/write register is required by any master capable of bursting more than two data phases. The latency timer begins when the GNT# line is deasserted and is decremented on every cycle thereafter. Once the timer expires, the master must terminate the burst within 3 clock cycles. Typical implementations hardwire the lower three bits to 0 leaving the five high-order bits as writable, which allows a granularity of eight clock cycles.

Built-in Self Test (BIST)

This is an optional read/write register which provides a mechanism for self-test on plug-in cards. If a device does not support Built-in Self Test, then a value of 0 is returned. The layout of the register is shown in Figure 4.7 with a description of the bits in Table 4.6.

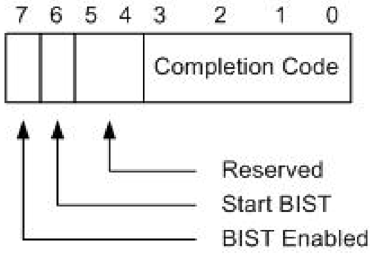


Figure 4.7: Configuration built-in self test register layout

Bit Location	Description
7	Returns 1 if the device supports BIST, else returns 0.
6	Write 1 to start BIST. Device resets back to 0 when BIST is complete.
5 - 4	Reserved and returns 0 when read.
3 - 0	Returns 0 if the device passed its test. A non-zero value specifies the device has failed and is device specific.

Table 4.7: Configuration built-in self test register bits

CardBus CIS Pointer

This is an optional read-only register implemented by devices which are both PCI and CardBus compatible. This field is used to point to the Card Information Structure (CIS) for the CardBus card.

Interrupt Line

This read/write register is used to communicate interrupt routing information and is system specific. The field is initialized during POST that contains information identifying which interrupt controller input is connected to the device's interrupt pin. The value stored is not used by the device itself; rather it is used by the device driver to determine which interrupt vector is assigned to the device.

Interrupt Pin

This read-only register identifies which interrupt is used by the device. A value of 0 is returned if a device or function does not use an interrupt pin. Any value from 1 to 4 can be used which corresponds to **INTA#** to **INTD#,** respectively.

MIN_GNT and MAX_LAT

These read-only byte registers are used to specify the desired value of the Latency Timer. The MIN_GNT register measures, in 250μs units, how long the device needs for a burst period assuming a 33 MHz clock. The MAX_LAT register specifies, in 250μs units, how often the device needs access to the PCI bus.

Subsystem Vendor ID and Subsystem ID

These read-only registers are used to uniquely identify a product where the device resides. These registers allow software device drivers to take advantage of features offered in some cards using a generic PCI chip. The Subsystem Vendor ID is assigned by the PCI SIG while the Subsystem ID is assigned by the vendor.

Capabilities Pointer

This optional read-only register points to a linked list structure of new capabilities offered by the device if bit 4 is set to 1. The value is a byte offset into the device-specific configuration space.

4.6.1.5 Base Address Registers

The Base Address registers are the key to providing Plug-and-Play configurability. This allows configuration software to determine what memory and I/O resources are required by the device as well as knowing what other devices are present. Once this information has been gathered, the device software builds a consistent map of all devices into a set of reasonable address ranges. The corresponding starting address is then written into the Base Address Registers. There are six Base Address Registers for a Type 0 configuration header allowing devices to support up to six independent address ranges.

There are two layouts for Base Address registers, Memory Space and I/O Space, determined by read-only bit 0. Base Address registers mapped to Memory Space, shown in Figure 4.8, must return a 0 in bit 0. Base Address registers that map to I/O Space, shown in Figure 4.9, must return a 1 in bit 0.

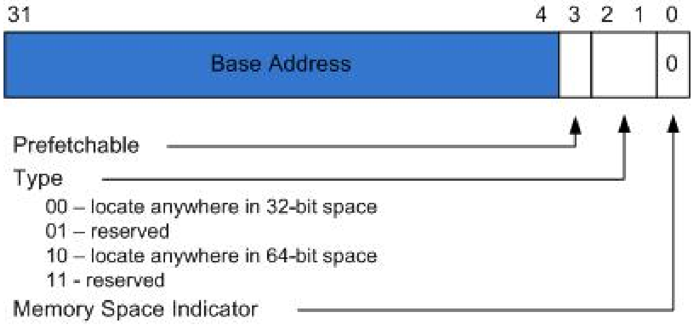


Figure 4.8: Base Address Register for Memory Space

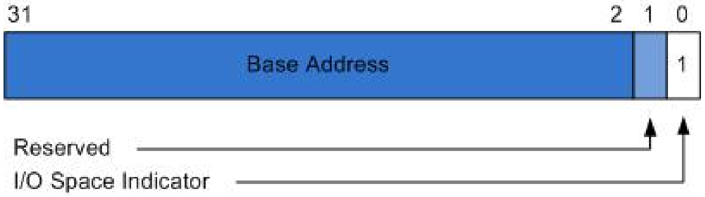


Figure 4.9: Base Address Register for I/O Space

For Base Address registers that map to Memory, read-only bits 1 and 2 determine how much memory space is to be mapped and the size of the Base Address. Memory can be mapped to either 32 or 64-bit address space with the 64-bit address space occupying two adjacent Base Address Register locations. We note that, prior to PCI Revision 2.1, the combination of 01 supported memory space below one megabyte. Bit 3 is read-only and set to 1 if data is prefetchable, else it is reset to 0.

For Base Address registers that map to I/O Space, the read-only bit 0 is hardwired to 1. The I/O Base Address Registers are always 32 bits wide.

4.6.1.6 Expansion ROM Base Address Register

There are some PCI devices that require local EPROM for Expansion ROM, providing a mechanism to embed run-time and device specific initialization code on board the system. The Expansion ROM Base Address register, shown in Figure 4.10, functions similarly to the 32-bit Base Address register described in the previous section. The upper 21 bits determine the amount of memory space required by the device in 2k increments. Bits 1 to 10 are reserved and bit 0 is used as a ROM enabled.

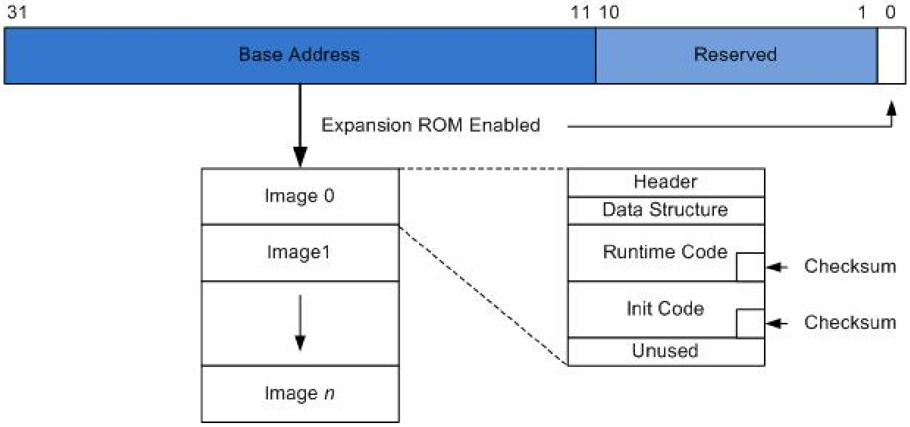


Figure 4.10: Expansion ROM Base Address Register

The expansion ROM is organized into one or more images, as shown in Figure 4.10, each with a specific format based on headers for ISA, EISA, and Microchannel adapters. While each image may contain the exact same code, each image may target a different processor.

4.7 PCI Bus Transactions

This section provides timing diagrams for common PCI bus transactions. We make a few notes about timing diagram notation. A solid line denotes the signal is being actively driven high by the current master or target. A signal drawn as a dashed line indicates no agent is actively driving it, however, it is assumed to be a stable value if shown at the high rail. A dashed line drawn between the two rails indicates tri-stated signals that contain indeterminate values. A solid line that turns into a dashed line indicates a signal that was actively driven and is now tri-stated. Finally, a solid line that transitions from low to high and then becomes a dashed line indicates the signal was actively driven high and then tri-stated.

4.7.1 Read Transaction

The timing diagram, shown in Figure 4.11, illustrates a typical read transaction. We now follow it cycle-by-cycle:

Clock

1 The bus is initially idle with most signals tri-stated. The master receives the **GNT#**, detects the bus idle, and initially asserts **FRAME#**.

2 This cycle signifies the start of the address phase. The initiator asserts **FRAME#** and places a valid address on the **AD** bus along with a bus command on the **C/BE#** bus. On the rising edge of clock 2, all targets latch both the address and the command.

3 This is the start of the first data phase. The initiator drives the appropriate **C/BE#** signals to indicate the byte lanes involved and asserts **IRDY#** to indicate it is ready to accept data from the target. The device that recognizes its address as the target asserts **DEVSEL#**. The first data phase requires a turnaround cycle since the **AD** lines were previously driven by the master and will subsequently be driven by the target.

4 The target places valid data on the **AD** bus and asserts **TRDY#**. On the rising edge of clock 4, the initiator latches the data.

5 The target inserts a wait cycle by deasserting **TRDY#** while the target continues to drive the **AD** bus, preventing it from floating. We note that the byte enables are valid on clock 5 until the data phase completes on clock 8.

6 The target places a piece of data on the **AD** bus and asserts **TRDY#**. Since both

IRDY# and **TRDY#** are asserted, the initiator latches the data.

7 The initiator inserts a wait cycle by deasserting **IRDY#**.

8 The last data transfer occurs, as indicated by the initiator asserting **IRDY#** and deasserting **FRAME#,** indicating a master-initiated termination. The target responds by deasserting **AD**, **TRDY#**, and **DEVSEL#**. The initiator then deasserts **C/BE#** and **IRDY#**.

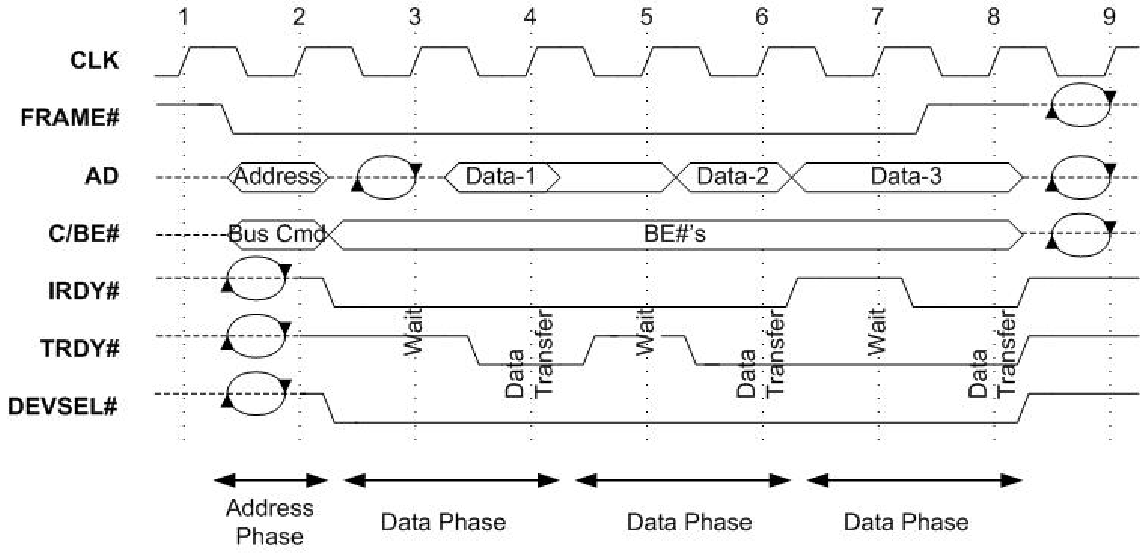


Figure 4.11: Timing diagram for a typical PCI read transaction

4.7.2 Write Transaction

The timing diagram, shown in Figure 4.12, illustrates a typical write transaction. We now follow it cycle-by-cycle:

Clock

1 The bus is initially idle with most signals tri-stated. The initiator receives the **GNT#**, and initially asserts **FRAME#**. It also places a valid address on the **AD** bus along with a bus command on the **C/BE#** bus.

2 This cycle signifies the start of the address phase. On the rising edge of clock 2, all targets latch both the address and command.

3 The initiator places a valid address on the **AD** bus along with a bus command on the **C/BE#** bus. It also drives the appropriate **C/BE#** signals to indicate the byte lanes involved and asserts **IRDY#** to indicate that it is ready to write data to the target. The device that recognizes its address as the target asserts **DEVSEL#** and asserts **TRDY#** indicating that it is ready to receive data. Unlike a read transaction, a write transaction does not require a turnaround cycle.

4 The initiator places valid data on the **AD** bus for the second data phase. Both **IRDY#** and **TRDY#** are still asserted and therefore, on the rising edge of clock 4, the target latches the data.

5 Both the master and target insert a wait cycle by deasserting **IRDY#** and **TRDY#**. The initiator continues to drive the **AD** bus, preventing it from floating.

6 The initiator places a piece of data on the **AD** bus and asserts **IRDY#**. However, since **TRDY#** is deasserted, no data is latched and a wait cycle occurs. We note that the target deasserts **FRAME#** indicating master-initiated termination.

7 The target inserts another wait cycle by deasserting **TRDY#**.

8 The last data transfer occurs as the target asserts **TRDY#**. The initiator responds by deasserting **AD**, **TRDY#**. The target then deasserts **C/BE#**, **TRDY#**, and **DEVSEL#**.

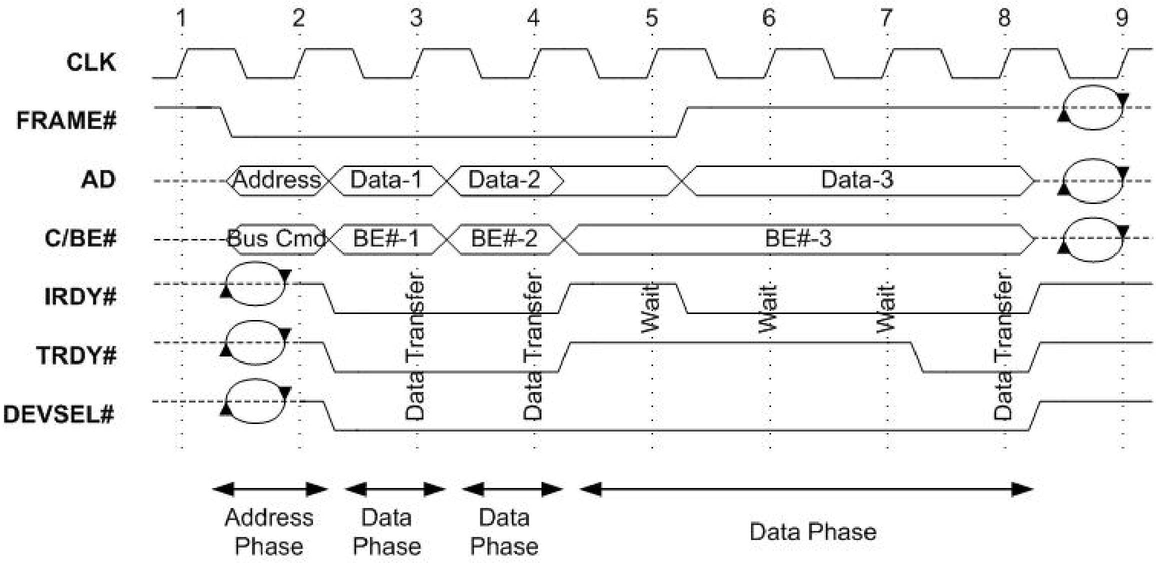


Figure 4.12: Timing diagram for a typical PCI write transaction

4.8 PCI Bus Arbitration

The PCI bus is able to support many masters, each of which may request the bus at any time. In order to allocate resources and resolve conflicts, the PCI bus implements a central arbitration scheme. Each bus master has an individual pair of **REQ#** and **GNT#** lines connected directly to the central arbiter as shown in Figure 4.13. If a master needs access to the bus, it asserts its **REQ#** signal and waits until the arbiter asserts the corresponding **GNT#** signal.

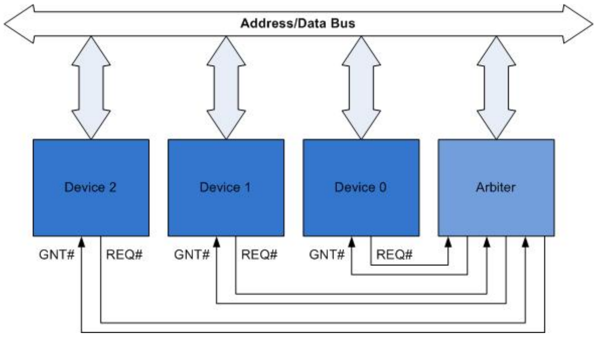


Figure 4.13: PCI arbitration

To reduce latency, *hidden arbitration* is performed allowing the arbitration process to take place while another transaction is occurring. The PCI specification does not define a specific algorithm, other than to say it must be “fair”. Fair, in this case, is defined so that each master must have a chance at access to the bus. This does not mean, however, that every agent has equal access to the bus. A fair algorithm may implement a priority scheme similar to the one shown in Figure 4.14. In this example, agents that are assigned to Level 1 have a greater need to use the bus than agents assigned to Level 2. We note that agents in Level 2 have equal access to the bus with respect to the other second level agents. However, our implementation features a simple round robin scheme. Other implementations may be added as necessary.

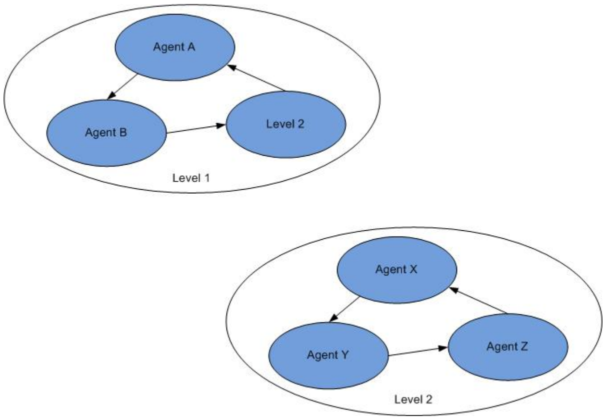


Figure 4.14: Priority scheme algorithm for fair arbitration

CHAPTER 5

PERIPHERAL COMPONENT INTERCONNECT EXTENDED

Throughout the years, bandwidth requirements for peripheral devices have grown. To keep pace with these needs, the Peripheral Component Interconnect Extended (PCI-X) standard was designed to improve performance by enhancing the existing PCI protocol.

Initially developed by HP, IBM, and Compaq, the PCI-X protocol was designed as a revision to the PCI standard. PCI-X offers several improvements over the original PCI standard including a faster clock (133 MHz versus 66 MHz), lower latency (*Split Transaction* versus Delayed Transaction), and improved fault tolerance amongst other enhancements. By increasing the clock to 133 MHz, a theoretical bandwidth of 1.06 GB/s can be achieved using 64-bit bus path in contrast to the 532 MB/s offered by PCI. In addition to offering higher performance, PCI-X is generally backwards compatible to PCI. However, while both PCI and PCI-X devices may both be intermixed, the bus speed is determined by the slowest device.

This chapter focuses on the PCI-X Revision 1.0 protocol, specifically the additional enhancements over the original PCI protocol. While PCI-X Revision 2.0 is available, none of the components are specified in this document since it will not be added to the simulator. The main reason being, since the introduction of PCI Express, PCI-X 2.0 devices are far and few between. Therefore, the additional work required to add it to the simulator would not be beneficial.

5.1 PCI-X Bus Commands

To support additional features for PCI-X, commands have been added or changed from the original PCI protocol. In Table 5.1 we provide the PCI-X command encodings along with their respective PCI commands. Our simulator implements all respective PCI-X bus commands.

C/BE#[3::0] or C/BE#[7::4]	PCI Command	PCI-X Command	Length
0000b	Interrupt Acknowledge	Interrupt Acknowledge	DWORD
0001b	Special Cycle	Special Cycle	DWORD
0010b	I/O Read	I/O Read	DWORD
0011b	I/O Write	I/O Write	DWORD
0100b	Reserved	Reserved	N/A
0101b	Reserved	Device ID Message	Burst
0110b	Memory Read	Memory Read DWORD	DWORD
0111b	Memory Write	Memory Write	Burst
1000b	Reserved	Alias to Memory Read Block	Burst
1001b	Reserved	Alias to Memory Write Block	Burst
1010b	Configuration Read	Configuration Read	DWORD
1011b	Configuration Write	Configuration Write	DWORD
1100b	Memory Read Multiple	Split Completion	Burst
1101b	Dual-Address Cycle	Dual-Address Cycle	N/A
1110b	Memory Read Line	Memory Read Block	Burst
1111b	Memory Write and Invalidate	Memory Write Block	Burst

Table 5.1: PCI versus PCI-X bus command encodings

5.2 PCI-X Bus Transactions

This section provides timing diagrams for common PCI-X bus transactions. When describing the PCI-X transaction, we will note where PCI-X differs from standard PCI.

5.2.1 Write Transaction

In Section 4.7.2, we described a typical PCI write transaction in detail. We now provide a description of the PCI-X write transaction, shown in Figure 5.1., and note the differences. We now follow it cycle-by-cycle:

Clock

1 The bus is initially idle with most signals tri-stated. The master receives the **GNT#**, detects the bus idle, and initially asserts **FRAME#**.

2 This cycle signifies the start of the address phase. On the rising edge of clock 2, all targets latch both the address and command.

3 This is the first difference between PCI and PCI-X, the addition of the *Attribute Phase*. The Attribute Phase always follows immediately after the Address Phase and provides additional information about the transaction.

4 This is known as the *Target Response Phase*. Although this occurs in the PCI protocol, it is not explicitly stated. This is the time required for the target to assert **DEVSEL#** thereby claiming the transaction.

5 This is the start of the first data phase. This cycle is the same as the corresponding cycle for the PCI protocol (i.e.: data is transferred on every clock that both IRDY# and TRDY# are asserted). However, unlike PCI, once data transfer has started, neither initiator nor target is allowed to insert wait states.

6 This is the start of the second data phase

7 This is the start of the third data phase. Note that the initiator deasserts **FRAME#** indicating an initiator-initiated termination two clocks before the end.

8 This is the start of the last data phase.

9 This is a turnaround cycle as in the PCI protocol.

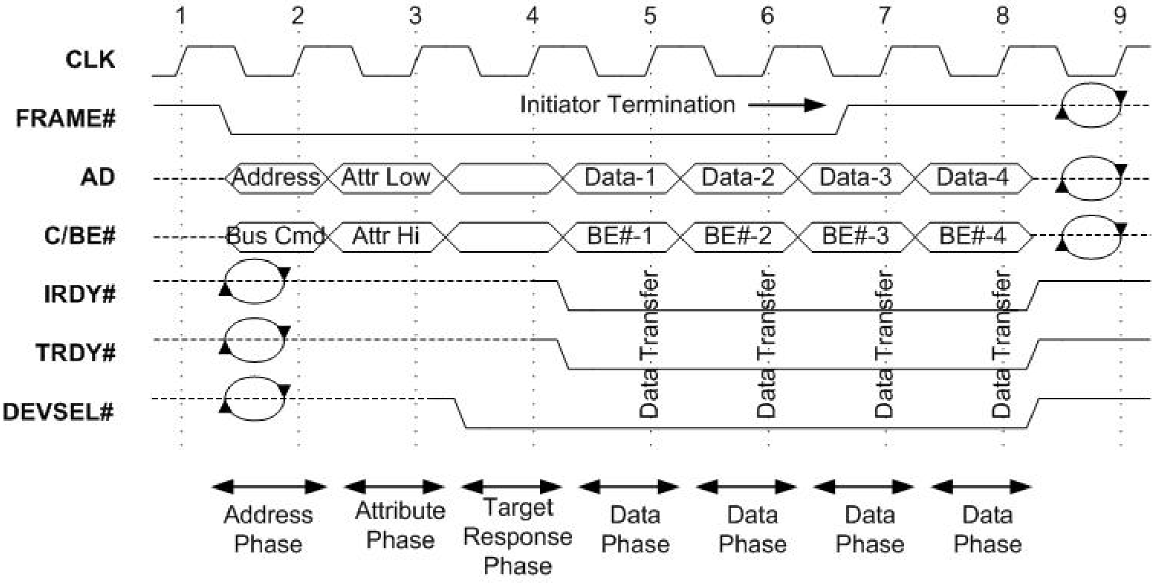


Figure 5.1: Timing diagram for a typical PCI-X write transaction

5.3 Burst and DWORD Transactions

Similar to PCI, PCI-X supports transactions that have one or more data phases which are divided into two categories. We support both DWORD and burst transactions in our implementation. Burst transactions are used by the memory commands (see Table 5.1), excluding Memory Read DWORD, and may be of any length from 1 to 4096 bytes. The Attribute Phase conveys the burst length and may be 64-bits wide (**REQ64#** asserted). DWORD transactions are used by the other PCI-X commands and are limited to a single data phase of 32-bits or less (**REQ64#** cannot be asserted).

In contrast to PCI, PCI-X treats byte enables differently. In PCI, the byte enable is conveyed during the data phase in a DWORD transaction, while PCI-X uses the Attribute

Phase. During burst transactions, byte enables are not used except for Memory Write where byte enable information is transferred similar to PCI.

5.4 Allowable Disconnect Boundaries

Conventional PCI allows a target to terminate a transaction at any time by asserting **STOP#**. In contrast, burst transactions in PCI-X can only terminate on a naturally aligned 128-byte address called an *Allowable Disconnect Boundary* (ADB). A disconnect is permitted by both the initiator and target. There are some exceptions to the ADB rule. A target abort may happen at any time by asserting **STOP#**, however, the data transfer continues until the *next* ADB. Also, a target can disconnect directly after the first data phase which is called a *Single Data Phase Disconnect*. Both of these features are implemented as part of our simulator.

5.5 Attributes and the Attribute Phase

In PCI-X, attributes are additional information conveyed during each transaction. The Attribute Phase, unique to PCI-X, lasts a single clock cycle and always follows the Address Phase of a transaction. During every transaction, the initiator conveys attribute information on the **AD[31::0]** and **C/BE[3::0]#** buses during the Attribute Phase. Burst transaction ascribe a value of 1 (high-true) to all attribute bits on both **AD[31::0]** and **C/BE[3::0]#** buses as high logic voltage, and a value of 0 to appear as a low logic voltage. DWORD transaction ascribe all attributes as high-true, except the byte enables which are low-true. For 64-bit devices, the upper buses (**AD[63::32]** and **C/BE[7::4]#**) are reserved and driven high during the Attribute Phase.

There are currently four different attribute formats – three formats for requesters and one for completers. This section describes Requester Attribute formats for burst and DWORD transactions. Section 5.6 describes Completer Attributes for Split Transactions and Section 5.7 describes Requester Attribute formats for Type 0 configuration transactions. Currently, we support all formats except those for Split Transactions, which may be implemented at a later time.

Figure 5.2 and Figure 5.3 illustrate the Requester Attribute format for burst and DWORD transactions, respectively. In Table 5.2 we define the fields associated for each transaction type.

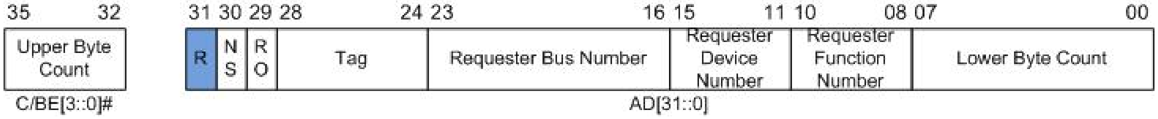


Figure 5.2: Burst transaction attributes

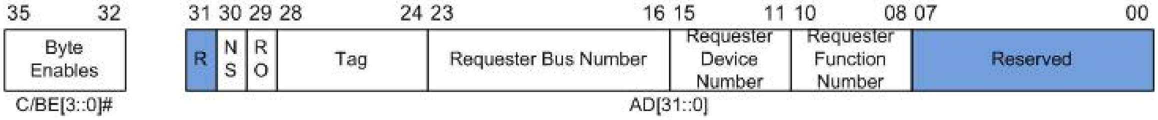


Figure 5.3: DWORD transaction attributes

Table 5.2: Description of fields associated with DWORD transactions (continued on pg. 90)

Attribute	Function
Reserved (R)	This field must be set to 0 by the requester but ignored by the completer, except for parity checking.
No Snoop (NS)	When set by the requester, it guarantees that the locations between the starting and ending address, inclusive, are not stored in any cache. This aids in performance if it is known subject locations are not cached.
Relaxed Ordering (RO)	A requester may set this bit to inform the target that memory write or read transactions must not remain in

	strict order.
Tag	This bit is necessary for a Split Completion transaction. It allows targets to identify the sequence of a transaction. It can uniquely identify up to 32 sequences from a single initiator while multiple sequences may be in progress.
Requester Bus Number	Represents the bus segment number assigned to the device.
Requester Device Number	Represents the logical device number assigned to the device.
Requester Function Number	This is assigned by the device's designer.
Upper Byte Count, Lower Byte Count	The number of bytes the initiator intends to transfer. A zero value represents 4096 bytes.

Table 5.3: Description of fields associated with DWORD transactions (continued from pg. 89)

5.6 Split Transactions

PCI uses the terms initiator and target to represent devices that participate in a transaction. The PCI-X protocol introduces two new terms which are necessary to describe Split Transactions. We note that the *requestor* is the agent that first introduces a transaction, while the *completer* is the agent that completes the transaction. This distinction must be made since during the initial request phase, the completer is the target. However, when the transaction is ready to be finished at some later point, the completer will become the initiator and the requester now becomes the target. While we currently do not support Split Transactions, it may be added at some future point. We found during our testing that data gathered from a PCI-X system generated PCI retry events rather than Split Transactions and thus the added complexity and time to implement this feature were not justified.

The following two sections, Section 5.6.1 and Section 5.6.2, describe the two commands used in a Split Transaction.

5.6.1 Split Response

The split response command is used when a target is unable to complete a transaction in a timely fashion. To notify the requestor of a split response, the target deasserts **DEVSEL#** during what would be the first data phase when **IRDY#** and **TRDY#** are asserted. Thus, it notifies the requestor that the transaction will complete at some later point in time. The target may issue a Split Response for any transaction except for a memory write.

It is noted that the Split Transaction does not completely eliminate the need for Retry termination. Devices are designed to handle only a certain number of simultaneous Split Transactions, perhaps just one. A device that is incapable of handling a Split Transaction will respond with a Retry.

5.6.2 Split Completion

A Split Completion command is used when the completer is able to complete the transfer that was started at some previous time. In Section 5.6.2.1 and Section 5.6.2.2 we describe the format used for the address phase and the Attribute Phase respectively.

5.6.2.1 Address Phase

The address phase of a Split Completion is issued by the completer, the format of which is illustrated in Figure 5.4. In this case, the address consists of the bus number, device number and function number of the original requestor. This information is supplied by the original Attribute Phase of the transaction that was terminated previously by a Split Response. Table 5.3 describes the fields associated with the address fields of a Split Completion.

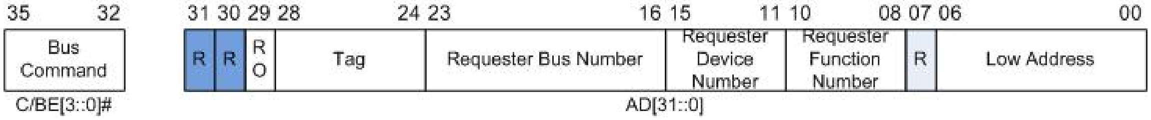


Figure 5.4: Split completion address

Attribute	Function
Reserved (R)	This field must be set to 0 by the initiator but ignored by the target, except for parity checking.
Relaxed Ordering (RO)	This field is copied from the attributes of the requester transaction.
Tag	This field is copied from the attributes of the requester transaction.
Requester Bus Number	This field is copied from the attributes of the requester transaction.
Requester Device Number	This field is copied from the attributes of the requester transaction.
Requester Function Number	This field is copied from the attributes of the requester transaction.
Lower Address	This field is copied from the least significant seven bits of the original requesting transaction if and only if the following are true: • The Split Request was a burst read. • It is the first Split Completion of the Sequence. • It is not a Split Completion Message (SCM). This field is set to zero if the Split Request was a DWORD read or the Split Completion is a SCM.

Table 5.4: Split completion address field definitions

5.6.2.2 Attribute Phase

The Attribute Phase of a Split Completion is issued by the completer, the format of which is illustrated in Figure 5.5. The address consists of the bus number, device number and function number and refers to the transaction initiator, which in the case of a Split Completion, is the completer. Table 5.4 describes the fields associated with the attribute fields of a Split Completion.

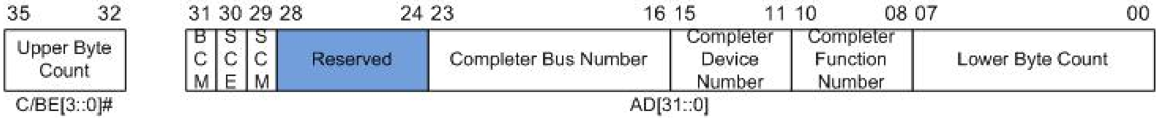


Figure 5.5: Split completion completer attributes

Attribute	Function
Byte Count Modified (BCM)	This bit is set to 1 if the Byte Count field consists of a number smaller than the remaining byte count. It is set to zero if it is a Split Completion Message, or the Byte Count field consists of the full remaining byte count.
Split Completion Error (SCE)	This field is set if there is an error in the transaction of a Split Completion Message.
Split Completion Message (SCM)	This field is set to 1 if the Split Completion is a Split Completion message and set to 0 if it contains read data.
Reserved (R)	This field must be set to 0 by the completer but ignored by the requester, except for parity checking.
Completer Bus Number	This field is supplied by the Bus Number register in the PCI-X Status register.
Completer Device Number	This field is supplied by the Device Number register in the PCI-X Status register.
Completer Function Number	This field is provided by the designer and does not need any initialization.
Upper Byte Count, Lower Byte Count	This field is copied from the corresponding bits of the requesting transaction's attributes. However, in some cases, the completer generates the values for this field.

Table 5.5: Completer attribute field definitions

5.6.2.3 Split Completion Message

During the Attribute Phase of a Split Completion, if the completer sets the Split Completion Message (SCM) bit, then the transaction includes a Split Completion Message. This message indicates to the requester the completion of a split write request as well as error conditions. While the Split Completion Message is always a burst transaction, it is only limited to a single DWORD. Figure 5.6 illustrates the format of the Split Completion message while Table 5.5 describes the attributes associated with each field.

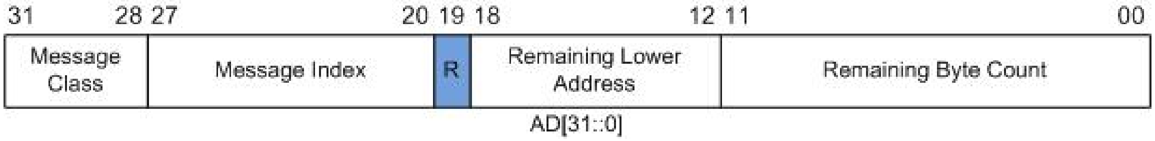


Figure 5.6: Split completion message

Attribute	Function
Message Class	This field conveys the message class identified by the following values: • 0h – Write completion. • 1h – PCI-X bridge error. • 2h – Completer error.
Message Index	This field identifies the type of message within the message class. The Write Completion class contains only one value, 0. All other values are reserved. The Completer Error class defines the following values: • 00h – Byte Count Out of Range. • 01h – Split Write Data Parity Error. • 8*x*h – Device-Specific Error.
Reserved (R)	This field must be set to 0 by the completer but ignored by the requester.
Remaining Lower Address	If the Split Request was a burst memory read, then this field contains the least significant seven bits of the first byte of a memory read data that has not previously been sent. It is set to 0 if the Split Request was a DWORD transaction.
Remaining Byte Count	If the Split Request was a burst memory read, then this field contains the number of bytes of memory read data that has not previously been sent. It is set to 4h if the Split Request was a DWORD transaction.

Table 5.6: Split completion message fields

5.7 Configuration and Initialization

This section will complete our brief explanation of the PCI-X protocol by looking at configuration and initialization.

5.7.1 Configuration Transaction Timing

PCI implementations, for the most part, connect the **IDSEL** line to **AD** bits via series resistors on the backplane. However, with PCI-X running at 133 MHz, the time required to charge the **IDSEL** input may require more than one clock. To compensate for this, PCI-X requires initiators to drive the address on the **AD** bus four clocks before asserting **FRAME#** on all configuration transactions. This is illustrated in Figure 5.7.

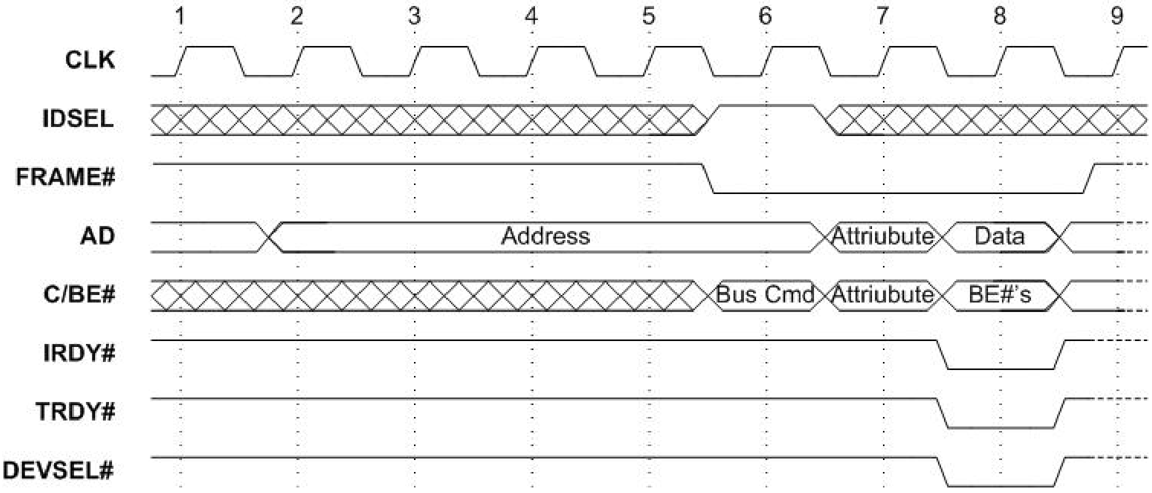


Figure 5.7: PCI-X configuration transaction timing

In a configuration transaction, the initiator can drive the address onto the **AD** bus immediately after recognizing that the **GNT#** signal has been asserted. During this time, the bus is technically idle until it is able to assert **FRAME#** four clocks later.

5.7.2 Configuration Address

One of the main distinctions between PCI and PCI-X is the addition of the Attribute Phase. The Attribute Phase conveys a device's physical address on any transaction that it initiates. This is in contrast to PCI where there is no need for a device to know its physical address since this is only relevant to configuration software.

Both PCI and PCI-X define two types of configuration transactions – Type 0 and Type 1. Figure 5.8 illustrates both types. In conventional PCI, a Type 1 configuration transaction is converted by the host bridge to a Type 0 transaction when it has reached the bus segment as conveyed in the address. This conversion also happens in PCI-X, however, unlike PCI, the Device Number is transferred to the Type 0 address according to the translation shown in Table 5.6. While we currently only support Type 0 configuration addresses, we describe Type 1 configuration addresses for completeness.

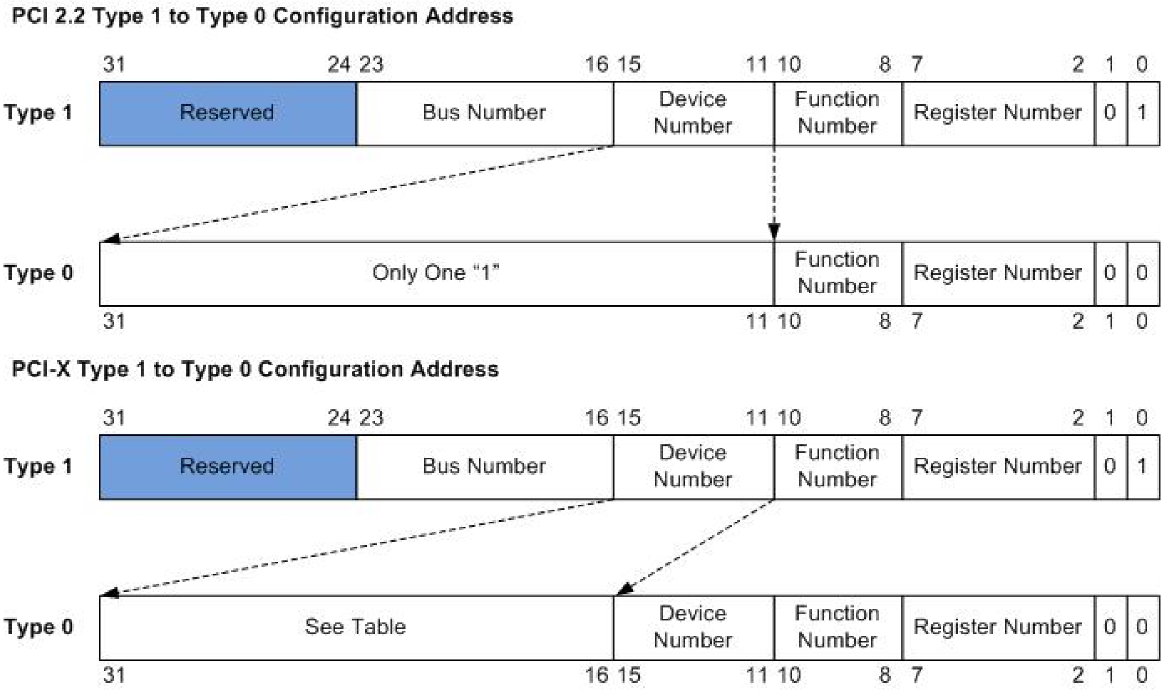


Figure 5.8: Configuration transaction address format

Device Number	Address AD[31::16]
0 0000b	0000 0000 0000 0001b
0 0001b	0000 0000 0000 0010b
0 0010b	0000 0000 0000 0100b
0 0011b	0000 0000 0000 1000b
0 0100b	0000 0000 0001 0000b

0 0101b	0000 0000 0010 0000b
0 0110b	0000 0000 0100 0000b
0 0111b	0000 0000 1000 0000b
0 1000b	0000 0001 0000 0000b
0 1001b	0000 0010 0000 0000b
0 1010b	0000 0100 0000 0000b
0 1011b	0000 1000 0000 0000b
0 1100b	0001 0000 0000 0000b
0 1101b	0010 0000 0000 0000b
0 1110b	0100 0000 0000 0000b
0 1111b	1000 0000 0000 0000b
1 xxxxb	0000 0000 0000 0000b

Table 5.7: PCI-X mapping of the device number to the upper AD lines

5.7.3 Configuration Attributes

The format of the Requester Attributes during the Attribute Phase of Type 0 configuration transactions is shown in Figure 5.9. This differs from the Requester Attributes of DWORD transactions by defining bits 0 to 7 as the Secondary Bus Number. This field contains the bus number of the segment on which the Type 0 transaction is executing.

The Secondary Bus Number and the Requester Bus Number are the same if the configuration transaction originated on same bus as the target device. In contrast, the Secondary Bus Number and Requester Bus Number are different if the transaction originated on a different bus segment and was converted from a Type 0 to a Type 1 by a PCI-X bridge.

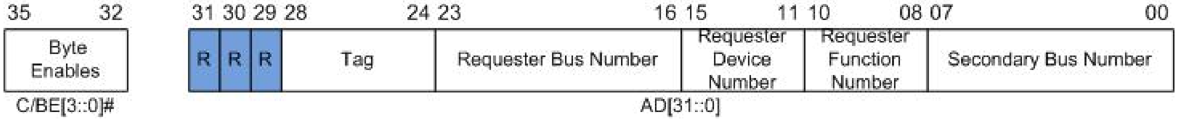


Figure 5.9: Type 0 configuration transaction attributes

The Attribute Phase of a Type 1 transaction, currently not implemented, is not shown as it is similar to the DWORD format as illustrated in Figure 5.3.

5.7.4 Configuration Header

The Configuration Space header defined by PCI-X is essentially the same as the one defined in the PCI 2.2 Specification Protocol. If a device is initialized as a PCI-X device, then the following changes apply and are currently supported:

- **Command Register**
    - *Bus Master* – This is ignored when initiating a Split Completion.
    - *Memory Write and Invalidate Enable* – This is ignored since PCI-X does not support this command.
    - *Fast Back-to-Back Enable* – This is ignored since PCI-X does not use fast back-to-back timing.
    - *Stepping Control* – This is ignored by the device.
- **Status Register**
    - *Capabilities List* – This bit is set to 1 since all PCI-X devices include the PCI-X Capabilities List item.
    - *Fast Back-to-Back Capable* – PCI-X devices do not use fast back-to-back timing, thus any value can be used when the device is in PCI-X mode.
    - *Detected Parity Error* and *Master Data Parity Error* – When in parity mode, these bits act identically as in conventional PCI.
    - *DEVSEL Timing* – This indicates the conventional PCI **DEVSEL#** timing.

- **Base Address Registers –** All Base Address registers that request memory resources (except Expansion ROM Base Address registers) are required to support 64-bit addressing per the method defined in PCI 2.2. Unless the memory address range contains locations with read side effects or locations in which the device does not tolerate write merging, the Prefetchable bit must be set. The minimum memory address range that can be requested by a Base Address register is 128 bytes. In order to conserve space, the specification recommends that devices request an address range no larger than an integral power of two that is larger than the memory range actually used by the device.
- **Latency Timer Register –** The default value is 64 in PCI-X mode.
- **Cacheline Size Register, MIN_GNT and MAX_GNT –** These registers are optionally used and must comply with requirements as specified in PCI 2.2.

5.7.5 PCI-X Capabilities List Item

To determine if a specific device is PCI-X capable, a Capabilities List item is included in the Configuration Space of each function. These additional control and status registers must be included regardless of whether or not the device is operating in PCI-X mode. Thus, we support these additional features as described in this section. Figure 5.10 illustrates the format of the PCI-X Capabilities List item. In the following sections, we describe each register.

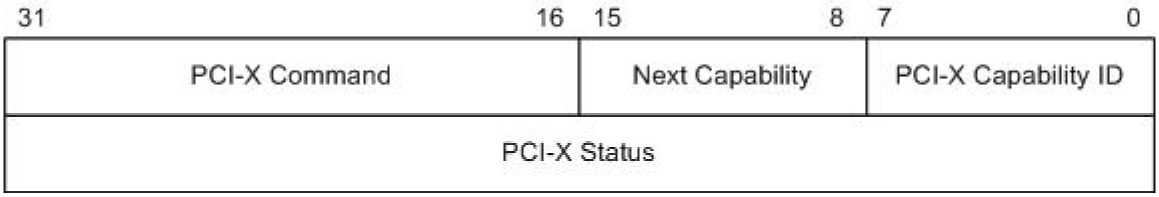


Figure 5.10: PCI-X Capabilities List format

5.7.5.1 PCI-X ID

The ID identifies this register as a PCI-X register set and always returns 07h when read.

5.7.5.2 Next Capabilities Pointer

This register points to the next capability item in the list as required by PCI 2.2.

5.7.5.3 PCI-X Command Register

This register provides control over various features and functions of PCI-X. Figure 5.11 illustrates the format of the Command Register while Table 5.7 provides a description of each item.

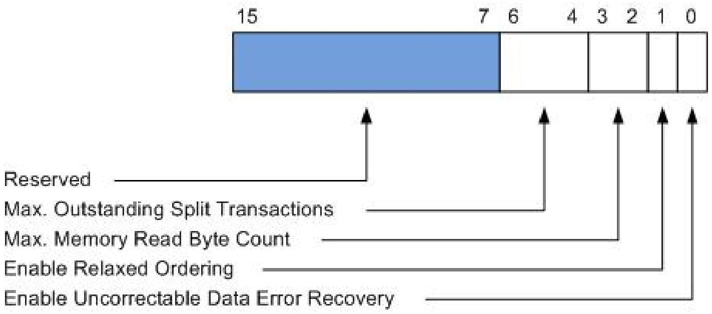

15
7
6
4
3
2
1
0
Reserved
Max. Outstanding Split Transactions
Max. Memory Read Byte Count
Enable Relaxed Ordering
Enable Uncorrectable Data Error Recovery

Figure 5.11: PCI-X Command Register format

Bit Location	Description
0	This read/write register is set to 1 by the device driver to help recover from data parity errors. If the device sets this register to 0, then the device is operating in PCI-X mode and **SERR#** is asserted. This bit is set to 0 after **RST#**.
1	If enabled (set to 1), the device can set the Relaxed Ordering bit in the

<table>
<tr><td></td><td>Requester Attributes of transactions that it initiates indicating strict ordering is not required. For devices that never use relaxed ordering, this bit is set to 0 and can be read-only. This bit is set to 1 after <b>RST#</b>.</td></tr>
<tr><td>3 - 2</td><td>This read/write register sets the maximum byte count the device can use when initiating burst memory reads. System configuration software can use this to tune performance, although modification is not recommended.
<table>
<tr><th>Register Value</th><th>Maximum Byte Count</th></tr>
<tr><td>0</td><td>512</td></tr>
<tr><td>1</td><td>1024</td></tr>
<tr><td>2</td><td>2048</td></tr>
<tr><td>3</td><td>4096</td></tr>
</table>
This bit is set to 0 after <b>RST#</b>.</td></tr>
<tr><td>6 - 4</td><td>This read/write register sets the number of outstanding Split Transactions permitted by the device at a given time. System configuration software can use this to tune performance, although modification is not recommended.
<table>
<tr><th>Register Value</th><th>Maximum Outstanding</th></tr>
<tr><td>0</td><td>1</td></tr>
<tr><td>1</td><td>2</td></tr>
<tr><td>2</td><td>3</td></tr>
<tr><td>3</td><td>4</td></tr>
<tr><td>4</td><td>8</td></tr>
<tr><td>5</td><td>12</td></tr>
<tr><td>6</td><td>16</td></tr>
<tr><td>7</td><td>32</td></tr>
</table>
The value after a <b>RST#</b> is the number of maximum Split Transactions the device can have outstanding when the Maximum Memory Read Byte Count field is set to 0 (512 bytes).</td></tr>
<tr><td>15 - 7</td><td>These bits are reserved.</td></tr>
</table>

Table 5.8: PCI-X Command Register description

5.7.5.4 PCI-X Status Register

This register provides additional information about the capabilities and current operating modes of PCI-X. Figure 5.12 illustrates the format of the Status Register while Table 5.8 provides a description of each item.

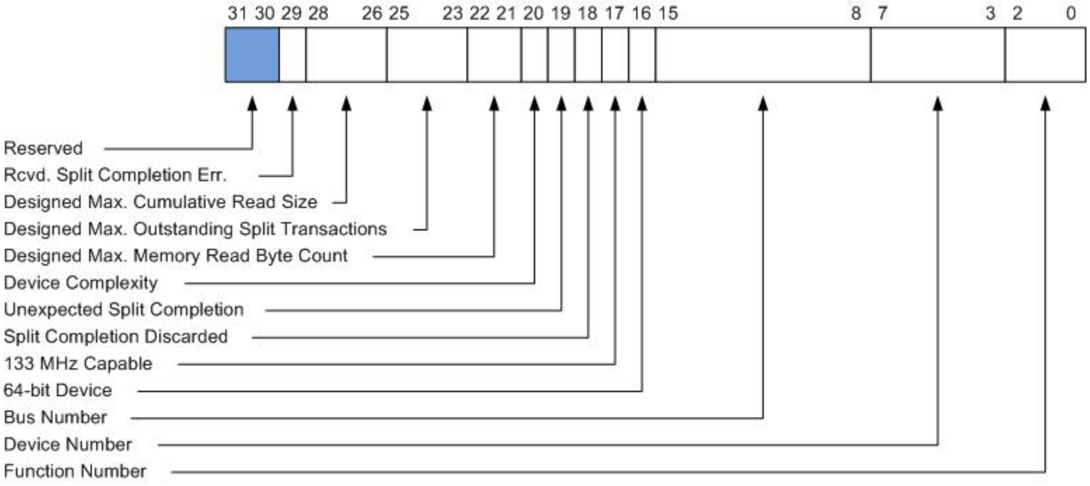

31 30 29 28 26 25 23 22 21 20 19 18 17 16 15 8 7 3 2 0
Reserved
Rcvd. Split Completion Err.
Designed Max. Cumulative Read Size
Designed Max. Outstanding Split Transactions
Designed Max. Memory Read Byte Count
Device Complexity
Unexpected Split Completion
Split Completion Discarded
133 MHz Capable
64-bit Device
Bus Number
Device Number
Function Number

Figure 5.12: PCI-X Status Register format

Table 5.9: PCI-X Command Register description (continued on pg. 103)

Bit Location	Description
2 - 0	Read-only register bit used for diagnostic purposes and indicates the number of this function.
7 - 3	Read-only register bit used for diagnostic purposes and indicates the number of the device that contains this function.
15 - 8	Read-only register bit used for diagnostic purposes and indicates the number of the bus segment for the device that contains this function.
16	Read-only register bit where a 1 indicates the device/function contains a 64-bit data-path.
17	Read-only register bit where a 1 indicates that the device’s maximum clock frequency is 133 MHz while a 0 indicates a maximum frequency of 66 MHz.
18	Write 1 to clear this register. If the value is a 1, then a Split Completion has been discarded and asserts **SERR#**. If the value is a 0, then no Split Completion has been discarded.
19	Write 1 to clear this register. If the value is a 1, then an unexpected Split Completion has been received. If the value is a 0, then no unexpected Split Completion has been received.
20	This read-only register has a value of 0 for a simple device, and a 1 for a bridge device. Specifically, this bit is used to determine if a device post write transactions.
22 - 21	This read-only register indicates the maximum byte count the device can use when initiating one of the burst memory read operations. System configuration software can use this to set the Maximum Memory Read Byte Count field in the PCI-X Command register. **Register Value** **Maximum Byte Count** 0 512 1 1024

<table>
<tr><td></td><td><table><tr><td>2</td><td>2048</td></tr><tr><td>3</td><td>4096</td></tr></table></td></tr>
<tr><td>25 - 23</td><td>This read-only register indicates the number of outstanding Split Transactions a device can have at one time. System configuration software can use this to set the Maximum Outstanding Split Transactions field in the PCI-X Command register.<table><tr><th>Register Value</th><th>Maximum Outstanding</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>2</td></tr><tr><td>2</td><td>3</td></tr><tr><td>3</td><td>4</td></tr><tr><td>4</td><td>8</td></tr><tr><td>5</td><td>12</td></tr><tr><td>6</td><td>16</td></tr><tr><td>7</td><td>32</td></tr></table></td></tr>
<tr><td>28 - 26</td><td>This read-only register indicates the maximum size of all outstanding burst memory read transactions that the device can have at one time.<table><tr><th>Register Value</th><th>Maximum Outstanding</th><th>Outstanding Bytes</th></tr><tr><td>0</td><td>8</td><td>1 KB</td></tr><tr><td>1</td><td>16</td><td>2 KB</td></tr><tr><td>2</td><td>32</td><td>4 KB</td></tr><tr><td>3</td><td>64</td><td>8 KB</td></tr><tr><td>4</td><td>128</td><td>16 KB</td></tr><tr><td>5</td><td>256</td><td>32 KB</td></tr><tr><td>6</td><td>512</td><td>64 KB</td></tr><tr><td>7</td><td>1024</td><td>128 KB</td></tr></table></td></tr>
<tr><td>29</td><td>Write 1 to clear this register. If the value is a 1, then a Split Completion error message has been received. If the value is a 0, then no Split Completion error message has been received.</td></tr>
<tr><td>31 - 30</td><td>These bits are reserved.</td></tr>
</table>

Table 5.10: PCI-X Command Register description (continued from pg. 102)

CHAPTER 6

SCRIPTSIM

ScriptSim [33] is an open source software tool that integrates Verilog with scripts such as Python and Perl. ScriptSim allows Verilog to dynamically create script processes and communicate with those processes by passing Verilog data types to the scripts. In turn, the scripts have access to any Verilog data used in the design and can perform the equivalent of any Verilog assign including: blocking, non-blocking, continuous assign, or force. In Figure 6.1 we see a block level view of the ScriptSim architecture.

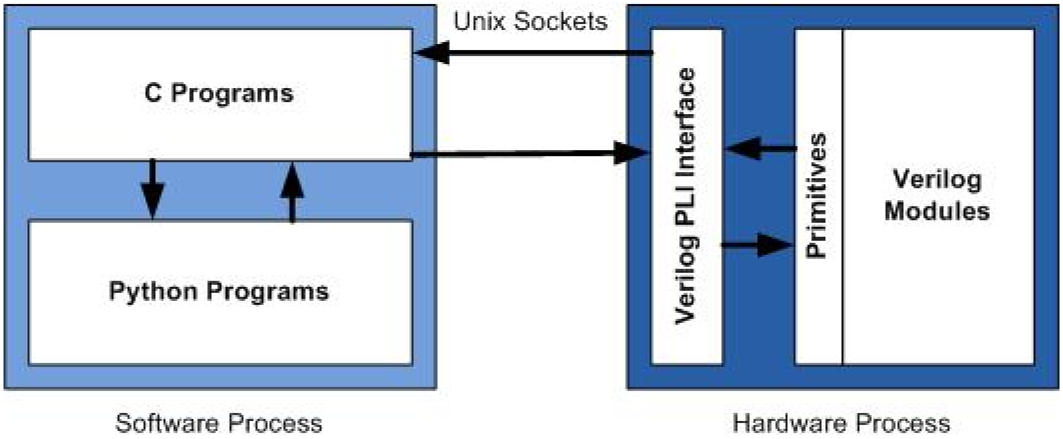


Figure 6.1: ScriptSim architecture

To communicate data between the various Verilog modules and the scripts, ScriptSim uses Verilog's *Programming Language Interface* (PLI). The PLI interface allows user supplied C programs to interact with the simulation environment and access Verilog's internal data structures. One major issue with PLI is when Verilog calls a PLI routine, execution is suspended until the C subroutine is terminated. This causes problems since the software may generate multiple access cycles, each of which must be handled by the

Verilog module. To solve this problem, ScriptSim runs the software and simulator as two different processes, communicating through Unix sockets. The scripts then use the C programs as a proxy to send and receive data.

While ScriptSim is flexible enough to create even the most complex simulations, we use its pre-built PCI bus model as a basis for our own simulator. The remainder of this chapter describes the ScriptSim PCI bus model.

6.1 PCI Functional Bus Model

The ScriptSim PCI model is written using Python and can be used with any PLI compliant Verilog simulator. The PCI model supports several important features:

- Includes all mandatory features as documented in the PCI Local Bus Specification Version 2.2 as described in Section 4 of this document
- Arbitration, master, and target functionality included
- Masters and targets both support unlimited burst length
- Targets can support up to five address spaces which may include a mix of memory or I/O space

6.2 PCI Bus Model Architecture

As we have stated previously, ScriptSim uses PLI to integrate Verilog (hardware) with Python scripts (software). In Figure 6.2, we illustrate the ScriptSim PCI bus architecture. The hardware portion is comprised of the Verilog module that interfaces the PLI shared object. Communication between the Verilog module and the software is performed using Unix sockets which is handled by subroutines located in `agent.c`. Data exchange between `agent.c` and the various Python scripts is handled using a custom C interface, `agent_python.c`. This program embeds Python code directly into the application allowing the various scripts access to subroutines and objects.

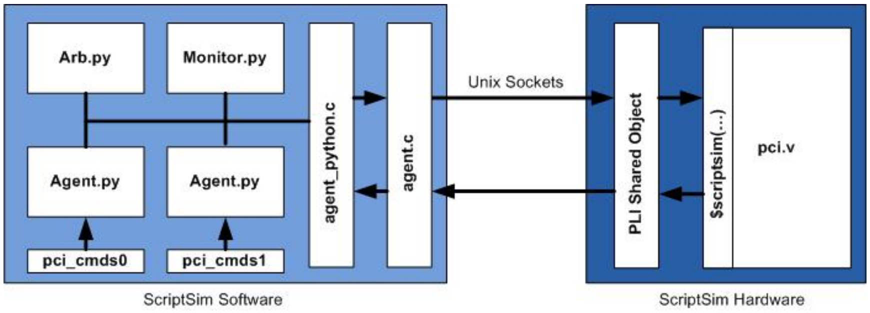


Figure 6.2: ScriptSim PCI bus model architecture

ScriptSim provides us with several Python scripts to simulate the PCI bus. The PCI bus is comprised of a bus monitor (`Monitor.py`), arbiter (`Arb.py`), and one or more agents (`Agent.py`). The agents represent the devices on the bus and use command files (`pci_cmds#`) to configure and generate activity.

6.3 PCI Bus Simulation

This section describes how to configure and generate PCI bus activity using ScriptSim. The arbiter, monitor, and agent Python scripts have been pre-created based on the PCI Local Specification Version 2.2 and thus only need to be included in our Verilog module as described in Section 6.3.1. Once instantiated, each agent must then be configured to generate bus activity as described in Section 6.3.2.

6.3.1 Verilog Instantiation

Instantiating Python scripts to be called in Verilog is as simple as adding the `$scriptsim()` user function along with the proper arguments. We describe in the following sections the arguments needed to instantiate an agent, arbiter, and monitor. Appendix A provides a sample Verilog test bench using an arbiter, monitor, and 4 agents.

6.3.1.1 Agent Instantiation

The agent both drives and receives PCI signals. All outputs must be connected to dedicated Verilog register bits. If all agents output 'z', in general, an effective 'pull-up' should be present to force an inactive level. To create an agent, we place the following code inside an `initial begin` block including the parameters as described in Table 6.1:

```
$scriptsim("agent.py",ss_id,"pci_cmds0",clk,reset_l,...)
```

Parameter	Description
1	ScriptSim program file name
2	ScriptSim ID which is an integer value returned by ScriptSim and can be used on later calls to allow Verilog code to generate bus cycles
3	Command file name for master and target configuration
4 .. *n*	The remaining parameters are the PCI signals

Table 6.1: ScriptSim agent parameters

6.3.1.2 Arbiter Instantiation

As with the agent, the arbiter both drives and receives PCI signals. To crate an arbiter, we place the following code inside an `initial begin` block including the parameters as described in Table 6.2:

```
$scriptsim("arb.py",clk,reset_l,req_l,gnt_l,frame_l,irdy_l)
```

Parameter	Description
1	ScriptSim program file name
2 - 7	PCI signals relevant for arbitration

Table 6.2: ScriptSim arbiter parameters

6.3.1.3 Monitor Instantiation

The PCI monitor is a passive display of PCI bus activity. It uses the Python/Tk GUI to create a window showing bus traffic which includes the following information:

- The start time of the transfer
- The name of the master, as determined by which agent won the arbitration process
- The name of the target, as determined by which agent drives **DEVSEL#**
- The operation type
- A decode of the address. For configuration cycles, the device number is decoded, and for standard PCI configuration space registers, the register name is shown
- The physical data being transferred
- Error messages, if appropriate

To create a monitor, we place the following code inside an `initial begin` block including the parameters as described in Table 6.3:

```
$scriptsim("monitor.py","req0_agent_name,req1_agent_name,...",clk,reset_
l,req_l,gnt_l,frame_l,irdy_l,trdy_l,stop_l,lock_l,idsel,devsel_l,ad,cbe_
l,par_l,perr_l,serr_l);
```

Parameter	Description
1	ScriptSim program file name
2	List of the agent names
3 - 18	PCI signals relevant for the monitor

Table 6.3: ScriptSim monitor parameters

6.3.2 Configuration and Activity Generation

There are three methods to specify the agents’ configuration and activity. The first, and easiest, method is to create a text control file to specify the target capabilities and the master cycles. This is shown in Section 6.3.2.1.

The second method also uses a text control file to specify the target capabilities. However, a Verilog task is used to request a model to perform PCI bus cycles as described in Section 6.3.2.2.

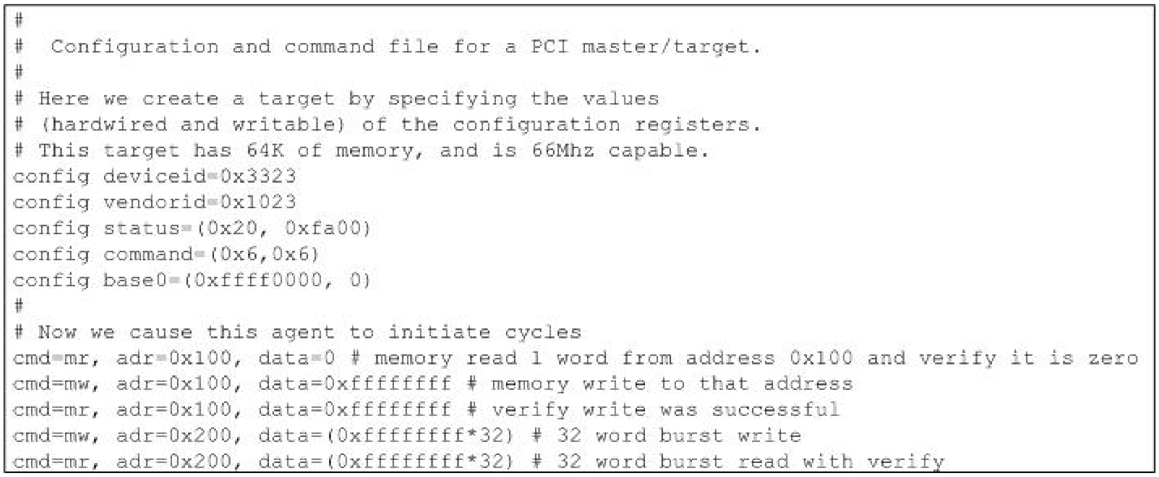
The third method is to use Python to generate the activity. We do not describe this method as it is much more complex and beyond our scope.

6.3.2.1 Control File

In Section 6.2, we noted that each device we wish to simulate on the bus must include the `Agent.py` python module along with its corresponding command file (Figure 6.2). Each agent reads a command file to determine whether it should act as a bus master and/or target. There can be multiple agents instantiated in the Verilog simulation with each agent usually reading from a different command file. For example, if we wish to simulate two devices on a bus, we would first create two command files (e.g.: `pci_cmds0` and `pci_cmds1`) and instantiate them within the Verilog file as shown:

```
initial begin: main_block
    …
   $scriptsim("agent.py",ss_id0,"pci_cmds0",...)
   $scriptsim("agent.py",ss_id1,"pci_cmds1",...)
   …
end
```

While each bus master may contain one or more bus commands within a file, if there are no bus master commands present in the file, then the agent will only act as a target. Also, if no base registers are defined in configuration space, then the agent will never respond as a target except for configuration cycles. In Figure 6.3, we show a sample control file for a master device. We note that master devices are also target devices since they can both act as a master and target and thus be configured as such. All configuration commands begin with the keyword `config` while all other commands are in the form `keyword=value`. A command occupies one line and the order of keywords is unimportant. Comments are allowed and begin with a # or `//`. Commas and blank lines are ignored by the parser and may be added for readability.



```
#
#  Configuration and command file for a PCI master/target.
#
# Here we create a target by specifying the values
# (hardwired and writable) of the configuration registers.
# This target has 64K of memory, and is 66Mhz capable.
config deviceid=0x3323
config vendorid=0x1023
config status=(0x20, 0xfa00)
config command=(0x6,0x6)
config base0=(0xffff0000, 0)
#
# Now we cause this agent to initiate cycles
cmd=mr, adr=0x100, data=0 # memory read 1 word from address 0x100 and verify it is zero
cmd=mw, adr=0x100, data=0xffffffff # memory write to that address
cmd=mr, adr=0x100, data=0xffffffff # verify write was successful
cmd=mw, adr=0x200, data=(0xffffffff*32) # 32 word burst write
cmd=mr, adr=0x200, data=(0xffffffff*32) # 32 word burst read with verify
```

Figure 6.3: ScriptSim control file for a master device

Both master and target devices have different keyword and value pairs which are described in Table 6.4 and Table 6.5 respectively. We note that target keywords follow the `config` command.

Table 6.4: Master keyword descriptions (continued on pg. 112)

Keyword	Value	Description
adr	Number	This specifies the address. If the command is a configuration read or write, bits 15:11 designate the device number. For device *n*, bit *n*+16 of the address will be asserted.
be	Number(s)	This specifies the byte enables for a single cycle or burst transfer and can either be decimal or hex values. Byte enables are for 32-bit quantities, even if 64-bit transfers are performed and can vary during the burst. A '1' bit indicates the byte is enabled. The default is 0xf, i.e. all bytes enabled.
burst	Number	This value specifies the burst length in terms of 32-bit data blocks. If burst exceeds the amount of write data specified, then extra blocks will be 0. If burst exceeds the amount of data specified on a read, no read data checking will occur for the extra blocks. Defaults to the number of data blocks, or 1 if no data blocks are specified.
cmd	2 letter command abbreviation	Specify the command to be driven on the **C/BE#** lines. Valid commands are: `ia` - interrupt acknowledge `sc` - special cycle

		`ir` - I/O read `iw` - I/O write `mr` - memory read `mw` - memory write `cr` - configuration read `cw` - configuration write `mm` - memory read multiple `ml` - memory read line `mi` - memory write and invalidate
data	Number(s)	This value specifies the write data, or read verify data. To specify a burst, multiple values are placed in parenthesis. Multipliers are allowed and values are a maximum of 32 bits. If 64-bit transfers are performed, two values will be transferred per cycle. The first data value will appear on bits 31:0 of the **AD** bus.
nodevsel	None	Verify that no device responds (**DEVSEL#** is not asserted).

Table 6.5: Master keyword descriptions (continued from pg. 111)

Table 6.6: Target keyword descriptions (continued on pg. 113)

Keyword	Value	Description
deviceid	Number	The Device ID specifies the hardwired value in the configuration space register at address 0x0 (upper 16 bits).
vendorid	Number	The Vendor ID specifies the hardwired value in the configuration space register at address 0x0 (lower 16 bits).
status	Number(s)	This value specifies the hardwired capability and written value of the configuration space Status register.
command	Number(s)	This specifies the hardwired capability and written value of the configuration space Command register.
base0	Number(s)	This specifies the hardwired capability and written value of the base register at address 0x10.
base1	Number(s)	This specifies the hardwired capability and written value of the base register at address 0x14.
base2	Number(s)	This specifies the hardwired capability and written value of the base register at address 0x18.

base3	Number(s)	This specifies the hardwired capability and written value of the base register at address 0x1c.
base4	Number(s)	This specifies the hardwired capability and written value of the base register at address 0x20.
base5	Number(s)	This specifies the hardwired capability and written value of the base register at address 0x24.

Table 6.7: Target keyword descriptions (continued from pg. 112)

6.3.2.2 Verilog Tasks

In the previous section, Python modules were used to generate PCI operations. However, it is also possible to allow Verilog code to directly generate PCI operations. This is done by including the following call inside the Verilog code:

```
$scriptsim(id,command,address,bytes,options,data,results,done)
```

Parameter	Description
Id	The unique ID of the agent
command	4-bit PCI command code
address	PCI signals relevant for the monitor
bytes	Byte count
options	Specified various options
data	Write or read-compare data
results	Indicates errors
done	Set to 1 when the PCI command completes

Table 6.8: Verilog task call parameters

CHAPTER 7

IMPLEMENTATION OF THE PCI SIMULATOR

The goal of the PCI Simulator is to provide an accurate representation of a real life PCI bus. We noted previously that commercially available PCI simulators focus mainly on verification of the PCI protocol. Also, one of the major drawbacks amongst PCI simulators is the lack of a graphical user interface (GUI). In most cases, configuring a simulation requires editing text files or even the source code itself. Running a simulation is usually done via a command line with the results either displayed directly on the screen or saved to a text file. Additional processing may be required to extract information from the results. While this may be sufficient for research purposes, commercial applications require a full-featured intuitive interface.

The following chapter describes our unique approach to PCI simulation and is structured as follows. In Section 7.1 we describe our methodology. Section 7.2 outlines the system architecture. In Section 7.3 we provide the user interface implementation. Section 7.4 details the modification and implementation of ScriptSim.

7.1 Methodology

Accurately modeling a PCI bus can roughly be broken up into two separate components, the functional aspect and the behavioral aspect. The functional aspect relies on modeling the PCI bus in accordance the specifications set forth by the PCI-SIG. ScriptSim, described in the previous section, provides us with a complete PCI software model in which we modify to include PCI-X functionality per the PCI-X Local Bus Specification v1.0b [17]. Modeling behavior, however, relies on device specific characteristics. An 802.11 b/g wireless adapter streaming a maximum of 52 Mbps of data exudes different bus characteristics than today's high resolution frame grabbers which can easily saturate a PCI bus. While bandwidth is one characteristic of a particular device, we will see that one can decompose a device into a set of performance characteristics that will help describe a device's behavior.

7.1.1 Modeling Behavior

Our approach to modeling behavior begins with breaking down a device into a set of performance parameters. While our goal is to describe specific devices (i.e.: an Ethernet card, a frame grabber, etc), we note that these sets of parameters will allow us to describe any device regardless of what its specific function may be. These parameters were determined using several techniques including analyzing device requirements from the PCI local bus specifications [16][17], borrowing ideas from previous PCI simulators [14][15], and analyzing data collected from a bus analyzer.

We now introduce the notion of a *device descriptor* which contains parameters that describe how a device acts on a bus. We recall that a transaction occurs between a master, which initiates the data transfer, and a target which receives the data. Thus, we decompose our device descriptor into a master device descriptor and a target device descriptor.

7.1.1.1 Master Device Descriptor

The master device descriptor is made up of eight performance parameters. The read/write ratio, burst length, and transaction parameters were ideas borrowed from [14], while the latency parameters were taken from Section 3.5 in the PCI local Bus Specification [16] as well as [14]. Before describing each parameter, we first introduce our key contributions, the notion of *injection rate* and *recovery period*. The idea of recovery period was first introduced in Section 3.1.1 [13]; however, we expand the definition to provide a method for calculating the recovery period.

Formally, the injection rate is the amount of data placed into the system per unit time. We note that the injection rate in our simulator is strictly defined as the required bandwidth of a device. We acknowledge more complex methods such as queuing models based on the Poisson process which may provide a more fine grained approach. This, and other models could be implemented as needed, however, we found it sufficient to use required bandwidth.

In a shared bus environment, each device will incur a certain amount of arbitration latency, which is a function of the arbitration algorithm, the sequence in which masters are granted access, and the amount of time each is allowed access on the bus. Each device must therefore provide sufficient buffer space to match the injection or consumption rate with the amount of data that can be moved across the bus. Without loss of generality, we define $c_{req,n}$ as the first clock cycle the device is able to request the bus for transaction n. The minimum amount of time a device must wait before requesting the bus for transaction n+1, regardless of whether transaction n has completed or not, is given by:

$$recovery_{n+1} = \left\lceil \frac{bytes_{n+1}}{bandwidth_{device} \times clock\_period_{pci}} \right\rceil \tag{7.1}$$

Where $bytes_{n+1}$ are the number of bytes expected to be sent during transaction n+1, which is a function of the burst length. Therefore, transaction n+1 can start on clock cycle:

$$c_{req,n+1} = c_{req,n} + recovery_{n+1} \tag{7.2}$$

We now define the performance parameters associated with our master device descriptor:

Injection Rate	The amount bandwidth required by the device.
Read/Write Ratio	Describes the ratio between the number of reads and the number of writes.
Burst Length	The number of data words a master can send contiguously. A value of 1 places the device in normal mode where only one data word is transferred per transaction.

Initial Wait States	The number of wait states from the assertion of **FRAME#** until the first word is ready to be sent or received.
Subsequent Wait States	The number of wait states a master waits before sending or receiving the next data word, after the initial data word.
Recovery Period	The minimum number of cycles the master must wait before requesting the bus.
Master Latency Timer	The minimum number of clock cycles the master is allowed to retain ownership of the bus. This value is decremented on each clock cycle after initiating a transaction.
Transactions	The number of transactions the master is allowed to initiate.

7.1.1.2 Target Device Descriptor

Our target device descriptor shares many of the same performance parameters as a master device with a few exceptions. We note that burst length and subsequent retry threshold are parameters from [14], while the other parameters are from the PCI Local Bus Specification [16][17] as well as [14]. We now define the performance parameters associated with a target device descriptor below:

Decode Speed	The number of clock cycles required to claim a transaction.

	Both PCI and PCI-X offer four decode speeds: Fast, Medium, Slow, and Sub. Sub, or subtractive decode, only responds to decodes ignored by other targets.
Burst Length	The maximum number of data words a target can send or receive.
Initial Wait States	The number of wait states from the assertion of **FRAME#** until the first word is ready to be sent or received.
Subsequent Wait States	The number of wait states a target waits before sending or receiving the next data word, after the initial data word.
Initial Retry Threshold	The number of wait states allowed by the target before a retry is generated. This only applies to the first data word in a burst and is limited to 16 per the PCI 2.1 specification.
Subsequent Retry Threshold	This is the same as the initial retry threshold except it applies to burst cycles and is limited to 8 per the PCI 2.1 specification.

7.2 Architecture of the PCI Simulator

As shown in Figure 7.1, we decided to decompose the system into three parts: the graphical user interface (GUI), the integration module, and ScriptSim. The GUI was designed using Java built using a distributed architecture based on the J2EE platform. J2EE provides us with a well-documented, reliable, flexible and scalable architecture. The user interacts with the GUI to build simulations and store them in a database. The integration module, also written in Java, uses the stored information in the database and presents it in a manner which can be read by ScriptSim which then uses this information to run the simulation. Once the simulation is complete, the results are presented back to the integration module and then storing them in the database. The results may be viewed immediately, or retrieved from the database at a later time.

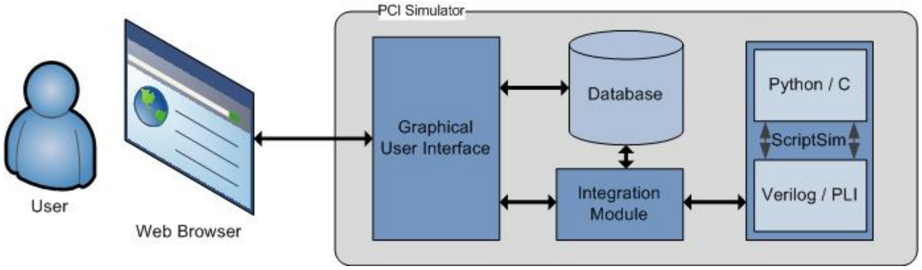


Figure 7.1: PCI Simulator system architecture

7.3 Implementation of the Graphical User Interface

7.3.1 GUI System Design

System design is essential for building successful software. Careful planning and practical use of software engineering techniques will help create robust, scalable, and reusable software. Typically, the creation of a software design begins with the assessment of functional requirements set by the customer. In our case, these requirements are predefined and provide a basis for our thesis. In most cases, however, analytical tools such as *use cases* help identify *actors* and operations. Use cases are a part of the *Unified*

Modeling Language (UML) [23], a standardized language used to model systems. In Figure 7.2 we illustrate the use case diagram for our application.

In our application, the user creates a new simulation, adds devices, and configures the system. Prior to running the simulation, the user can view the configuration and edit the simulation as needed. After running the simulation, various statistics can be viewed and analyzed. At any point, the current simulation can be saved and previous simulations can be loaded.

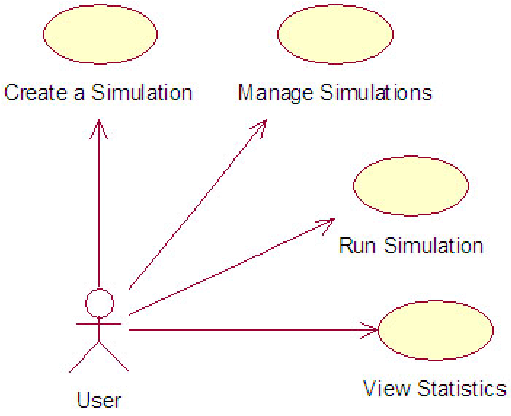


Figure 7.2: Use case diagram of the PCI simulation application

7.3.2 Web Application Architecture

One important aspect of developing an application architecture is subdividing the application into objects or components known as object decomposition. In most cases, components are consigned to a single tier, while others connect or span tiers. Careful design, therefore, is important to ensure scalability and promote object reuse. To accomplish this task, we decided to use the Model-View-Controller (MVC) architecture, shown in Figure 7.3, which divides the application into layers. The model represents the data and business logic which govern access to the data. The view obtains data through the model and renders the contents to the user. The controller defines the application behavior by accepting interactions with the view and translating them into actions performed by the model.

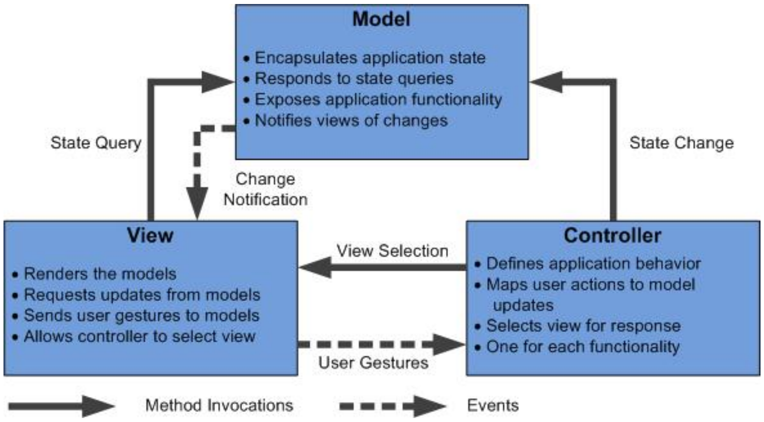

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
State Change
Change Notification
View
• Renders the models
• Requests updates from models
• Sends user gestures to models
• Allows controller to select view
View Selection
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
User Gestures
Method Invocations
Events

Figure 7.3: Model-View-Controller architecture

7.3.3 Design Patterns

Software design patters [24] promote reuse by providing a solution to a known recurring problem. Each layer of the MVC architecture uses one or more patters which, when combined, provides a high-level abstraction view of our application. Figure 7.4 decomposes our MVC architecture into patterns and other components. The components in the controller parse client requests and begin the process of devising an appropriate response. The control layer uses the Application Response and Command Factory design pattern as well as the Command Handler component. The model layer handles the business logic and acts as a bridge between the controller and the view. It consists of the following design patters: Session Façade, Business Object, Data Access Object, and Business Delegate. Lastly, the view layer formats the response and presents it to the client. It consists of the View Helper and Composite View design patters and two additional components: the Screen Flow Manager and the Screen View. We conclude this discussion by following a typical user request.

A client interacts with the user interface (Section 7.5) inside the browser and submits an HTTP request. The request is handled by the Application Controller servlet which extracts data from the request and determines the action, mapping it to a command using the services provided by the Command Factory and Command Handler. The Command Handler invokes the appropriate action on a Session Façade which controls access to our business objects by providing a coarse-grained service. Depending on the action, the Business Object may be persisted by a session or in a database in which a Data Access Object may be needed. Once the data has been retrieved, it must now be formatted and presented by the Screen View. To help build the page it uses a Composite View – a template containing the structure of the page such as the header, footer, and navigation. The View Helper provides the dynamic content using the data that was retrieved. Finally, the Screen Flow Manager keeps track of the page sequence, determining the next page in the sequence of pages.

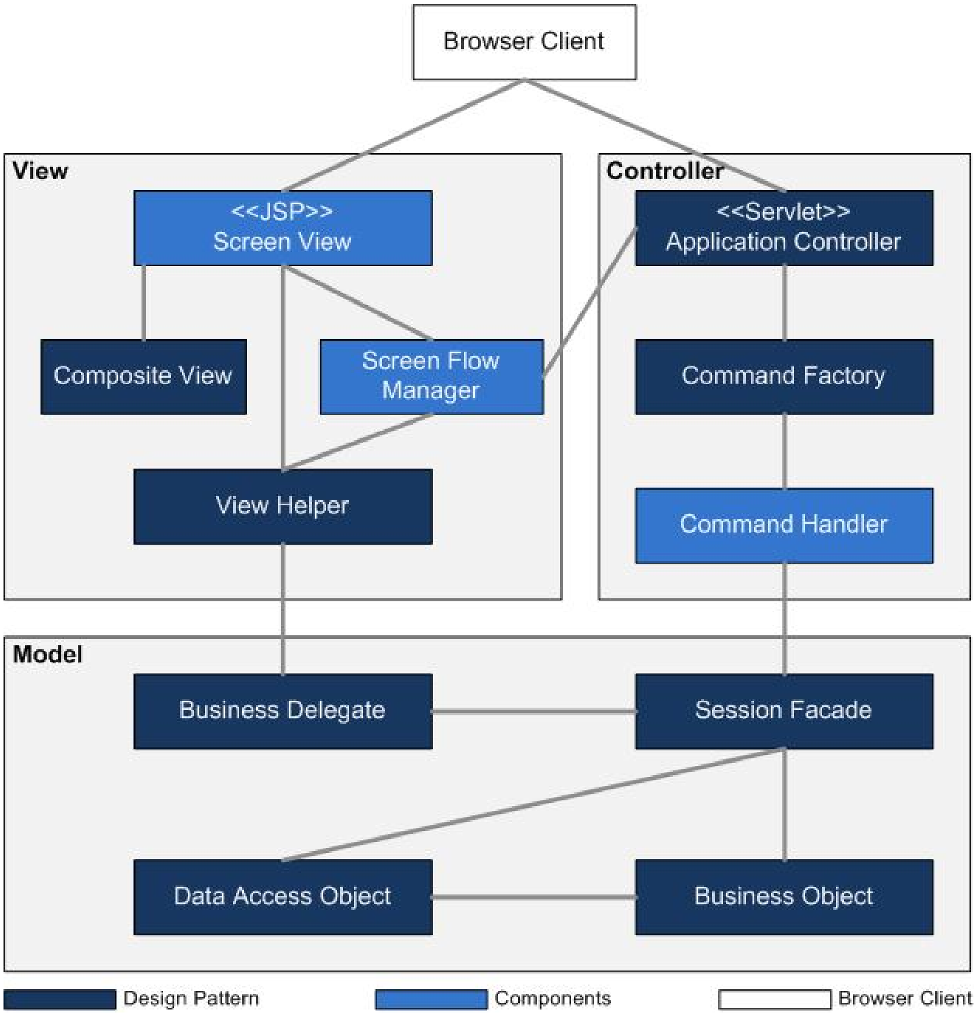
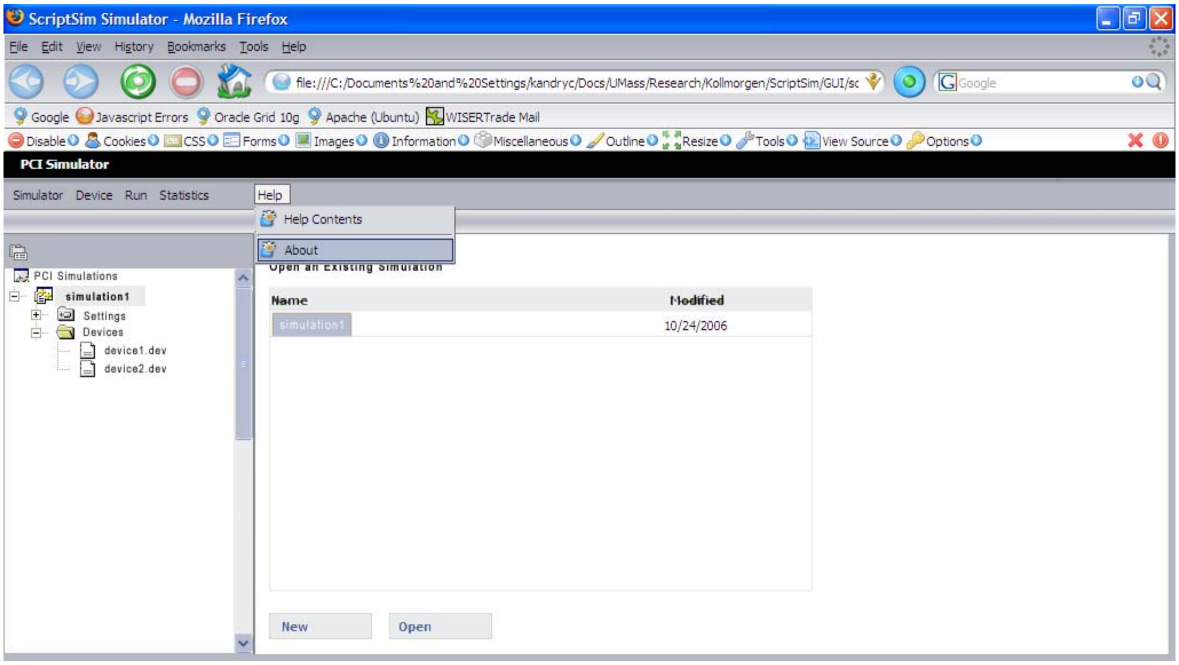


Figure 7.4: MVC architecture decomposed into design patterns and components

7.3.4 User Interface Design

We extend our discussion to include the graphical user interface that is presented to the user. An important aspect of designing user interfaces is not only the layout, but also the navigability. Menu choices should be logically grouped together and page location should always be evident. To keep a level of familiarity, we designed the interface, illustrated in Figure 7.6, with a look and feel similar to other design packages in the industry such as Altera’s Quartus II and Xilinx’s ISE.

As illustrated in Figure 7.5, the interface can be divided into sections. At the top reside the menu choices available to the user. The left side of the interface displays the active simulation project along with all the files associated with the project. A file selected in the project menu will open in the main window display. The main window allows for tabbed navigation such that multiple items may be open – enabling users to easily switch between multiple screens.


ScriptSim Simulator - Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///C:/Documents%20and%20Settings/kandryc/Docs/UMass/Research/Kollmorgen/ScriptSim/GUI/sc
Google
Google Javascript Errors Oracle Grid 10g Apache (Ubuntu) WISERTrade Mail
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize Tools View Source Options
PCI Simulator
Simulator Device Run Statistics Help
Help Contents
About
PCI Simulations
simulation1
Settings
Devices
device1.dev
device2.dev
Open an Existing Simulation
New
Open

Name	Modified
simulation1	10/24/2006

Figure 7.5: PCI Simulator design interface

7.3.5 Site Map

A site map provides a graphical hierarchical representation of how a web site is organized. In our case, it logically represents the functionality which mirrors the menu choices provided to the user. The site map for our web application is shown in Figure 7.6.

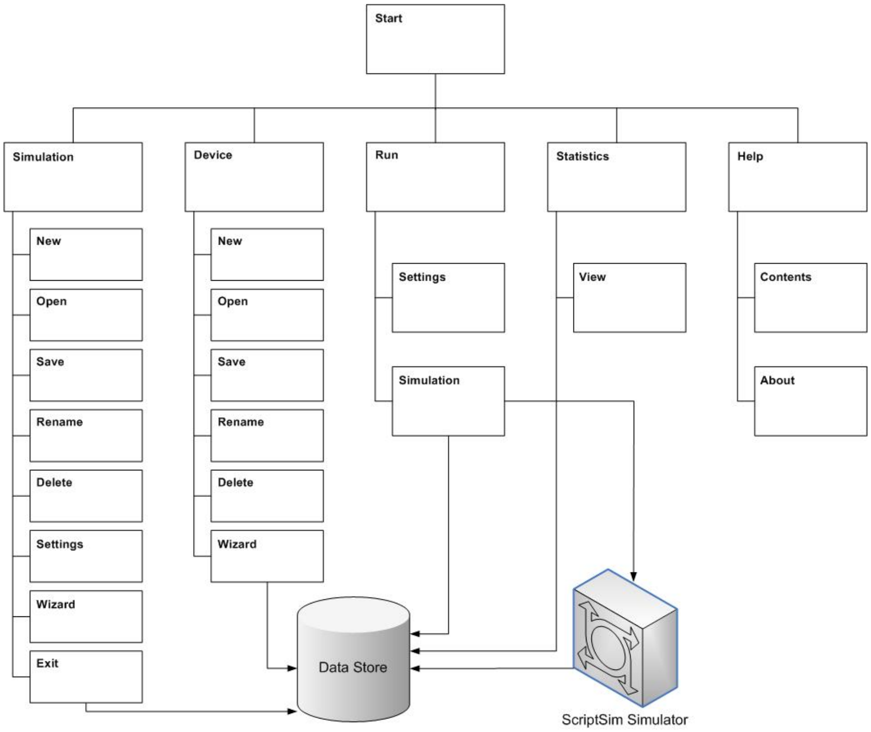


Figure 7.6: PCI Simulator site map

7.3.6 Functional Design

At the beginning of this section we discussed the functional requirements and illustrated the use case based on our requirements in Figure 7.1. This formed a high-level view of the capabilities provided by the system. The sections that followed discussed the architectural details and the user interface design. With that information in mind, we now provide the functional details of how the user interface interacts with our system. This interaction can be explained with the use of a UML *sequence diagram*. A sequence diagram is usually derived from a use case and shows the interaction of objects, the messages between them, and the order in which the messages occur. The following subsections list the use cases of the system and their respective sequence diagram.

7.3.6.1 Simulation Use Case

The simulation use case, illustrated in Figure 7.7, provides a look at the interaction between the user and the various functions provided by the system. The user may create a new simulation either manually or through a simulation wizard, open an existing simulation, save the current simulation, rename a simulation, delete a simulation, or edit the settings of a simulation.

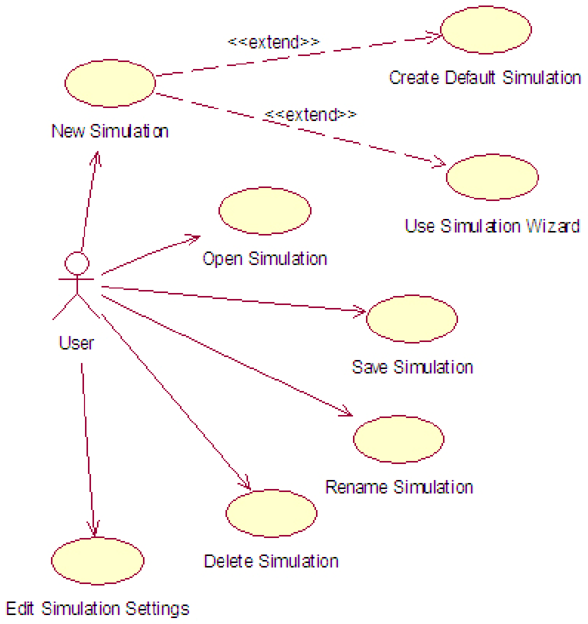


Figure 7.7: Simulation use case

7.3.6.1.1 New Simulation

When issuing a new simulation, the XML request is handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object. In turn, a new `SimulationFacade` is created in which the `newSimulation()` method is called. The `SimulationDAO` creates a new `Simulation` object, adds the data provided by the user to the object, and saves it to the database. For easy retrieval, the object is persisted in a session. A status is sent back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI. The user is then presented with a confirmation message and the user interface is updated appropriately.

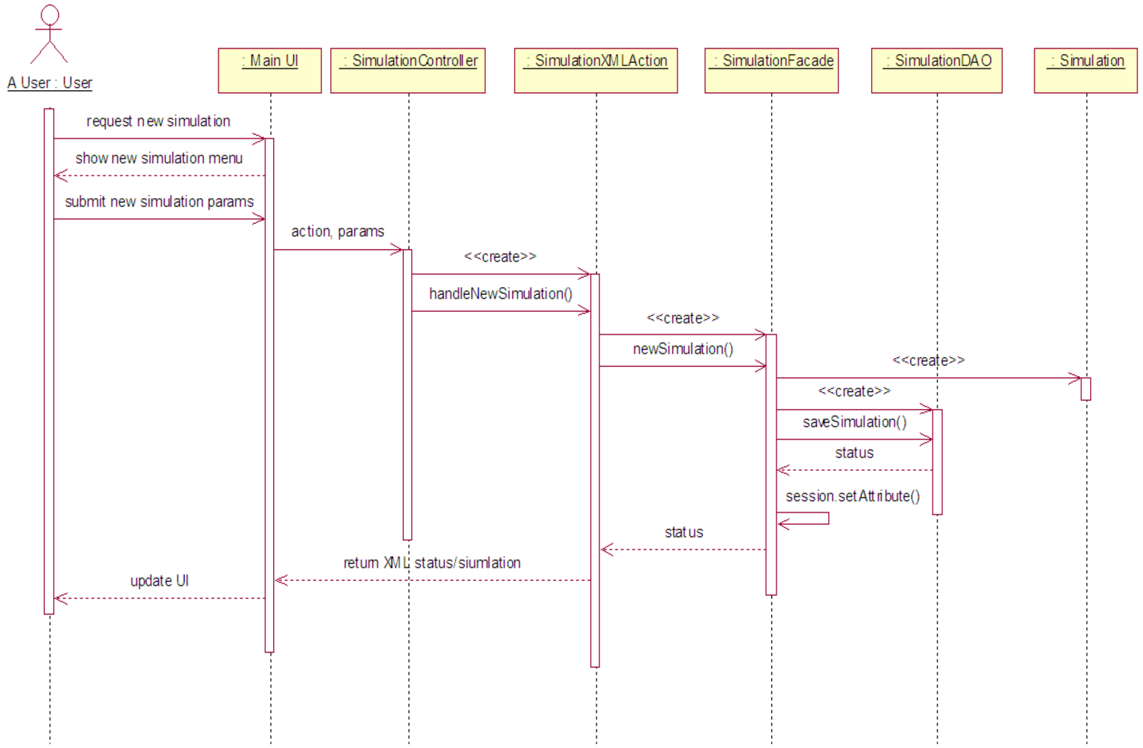


Figure 7.8: New simulation sequence diagram

7.3.6.1.2 Open Simulation

When the open simulation command is issued, the XML request data is formed and is handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object. In turn, a new `SimulationFacade` is created in which the `getSimulations()` method is called. The `SimulationDAO` retrieves a list of simulations from the database and returns it back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI. The user is then presented with a list of simulations to choose from. Once a simulation is chosen, a separate XML request is sent to the server containing the simulation id. Again, the `SimulationController` handles the request, extracts the id and calls the `SimulationXMLAction.handleOpenAction()` method. From there, the `SimulationFacade.getSimulation()` is called with the simulation id and then sent to the `SimulationDAO` object. When the simulation data is returned it gets added to the `Simulation` object. The XML response is formed and then sent back to the GUI where it is updated to reflect the opened simulation.

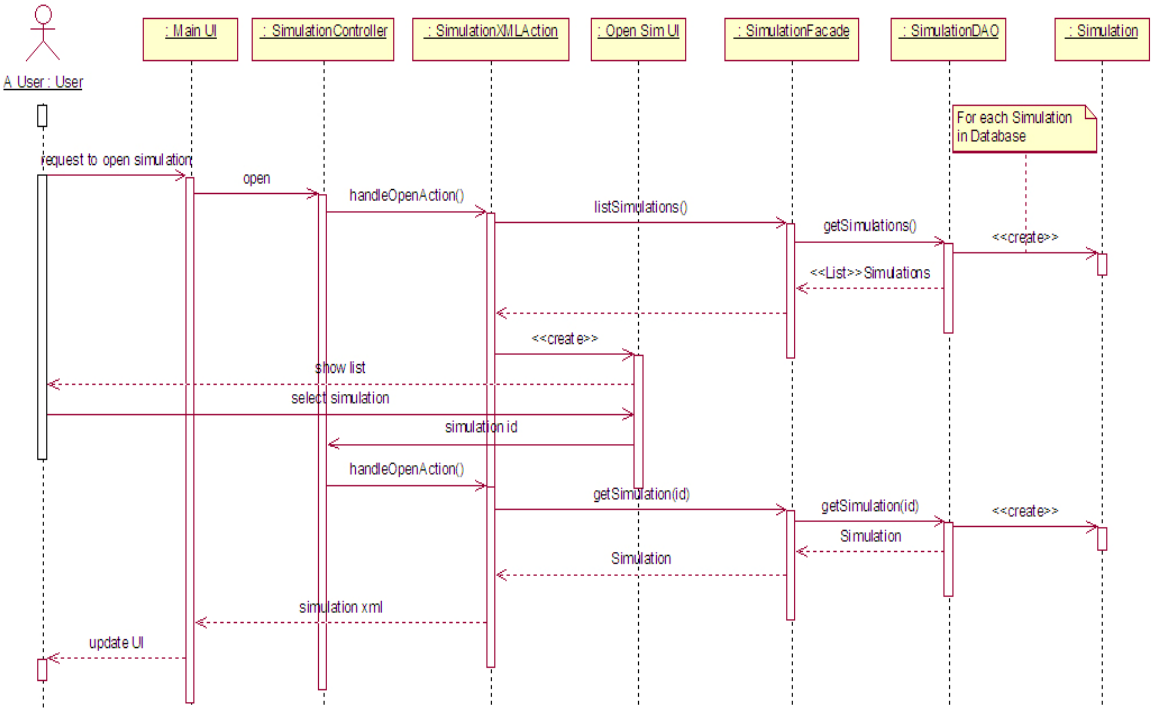


Figure 7.9: Open simulation sequence diagram

7.3.6.1.3 Save Simulation

In issuing a save simulation command, the XML request is handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object, extracts the id, and then calls the `SimulationXMLAction.handleSaveSimulation()` method. From within the `handleSaveSimulation()` method, a new `SimulationDAO` object is created and the `saveSimulation()` method is called which stores the updated information in the database. A status method is returned back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI.

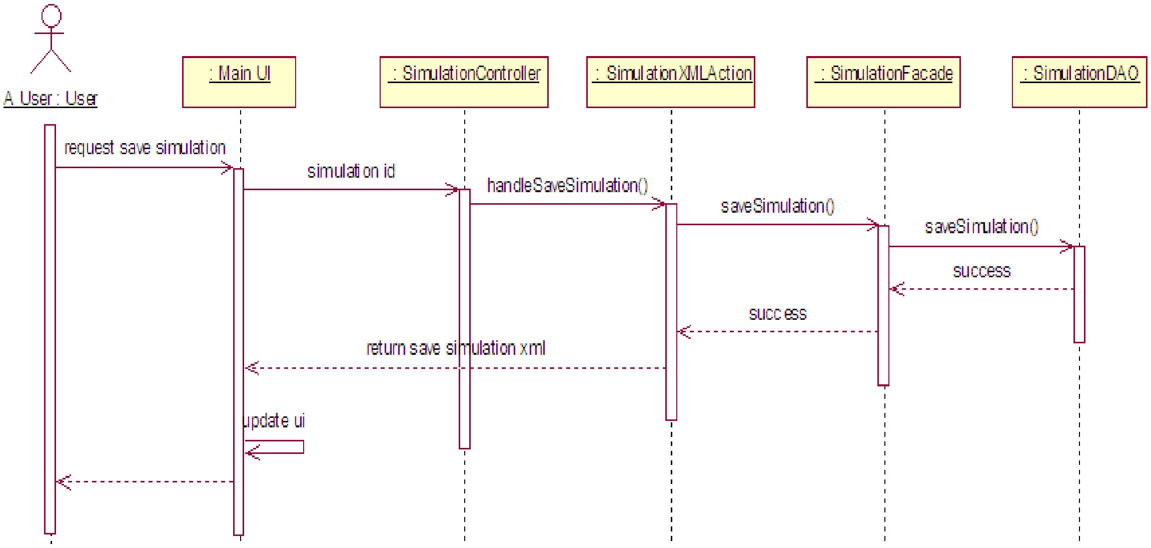


Figure 7.10: Save simulation sequence diagram

7.3.6.1.4 Rename Simulation

When a rename simulation command is issued, the XML request is formed and handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object, extracts the id and new name, and then calls the `SimulationXMLAction.handleRenameSimulation()` method. From within the `handleRenameSimulation()` method, a new `SimulationFacade` object is created and the `getSimulation()` method is called. This returns the `Simulation` object from the session. The new name is inserted into the object and then stored in the database. A status method is returned back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI.

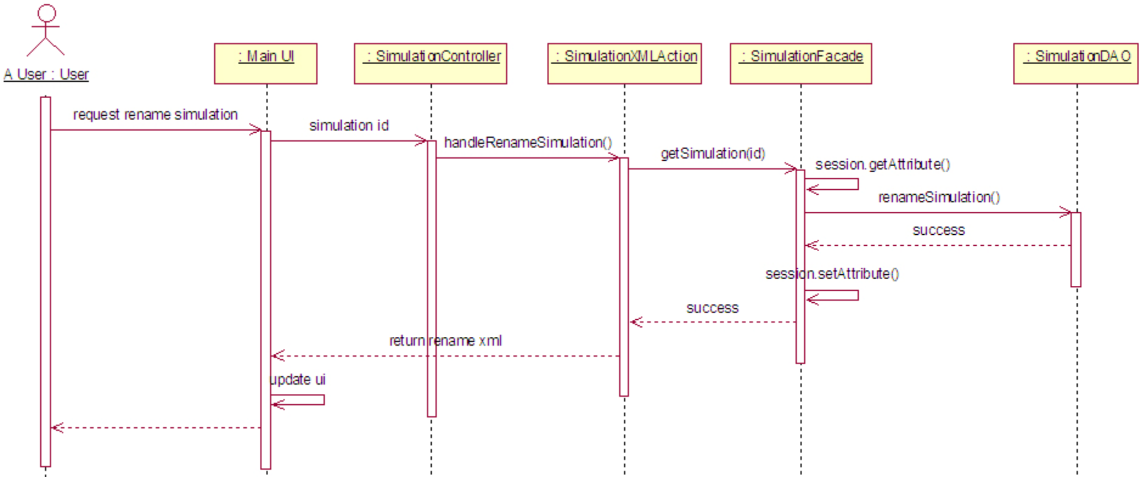


Figure 7.11: Rename simulation sequence diagram

7.3.6.1.5 Delete Simulation

When the delete command is sent, an XML request is formed and the simulation id is sent to the server and handled by the `SimulationController`. The `SimulationController` creates a new `SimulationXMLAction` object and calls the `handleDeleteSimulation()` method. From there, a new `SimulationFacade` is created and the `deleteSimulation()` method called with the simulation id as the attribute. The `SimulationDAO` deletes the simulation from the database and returns the status. The session is then invalidated removing any objects persisted with the session. The response XML is then formed along with the status (success or error) and then sent to the GUI where the appropriate action takes place.

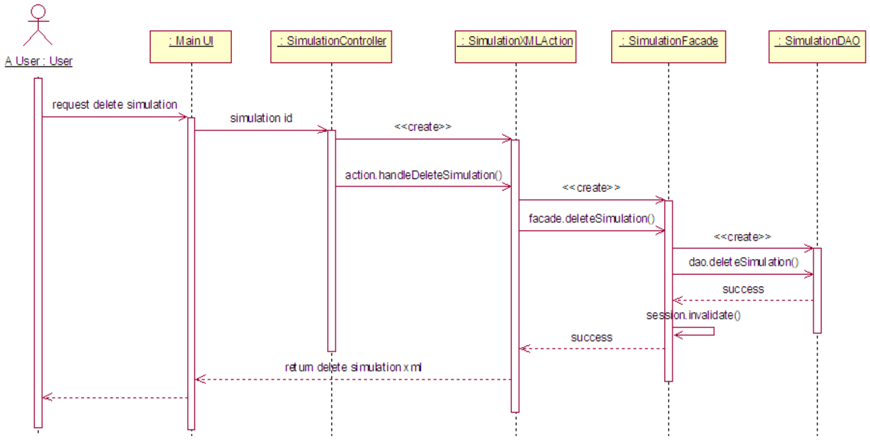


Figure 7.12: Delete simulation sequence diagram

7.3.6.1.6 Simulation Settings

When the simulation settings command is issued, the XML request data is formed and is handled by the `SimulationController` which calls the `SimulationXMLAction.handleEditSimulation()` method. From there, a new `SimulationFacade` object is created and the `SimulationFacade.getSimulation()` is called with the simulation id as the attribute. The Simulation object is returned from the session and return back to the `SimulationXMLAction` object. An XML response is formed with the simulation settings and returned back to the client. The user is presented with the existing settings and may then proceed to edit the settings. Once finished with the edit, the user submits the information back to the server, which again is handled by the `SimulationController`. The `SimulationController` object calls the `SimulationXMLAction.handleEditSimulation()` method which creates a new `SimulationFacade` object. The `saveSimulationSettings()` method is called with the simulation id and data which is stored in the database by a call to the `SimulationDAO.saveConfigurationSettings()` method. The status is then returned, and the `Simulation` object is updated within the session for later retrieval. A confirmation method is returned to the client via the `SimulationXMLAction` which forms the response XML handled by the client.

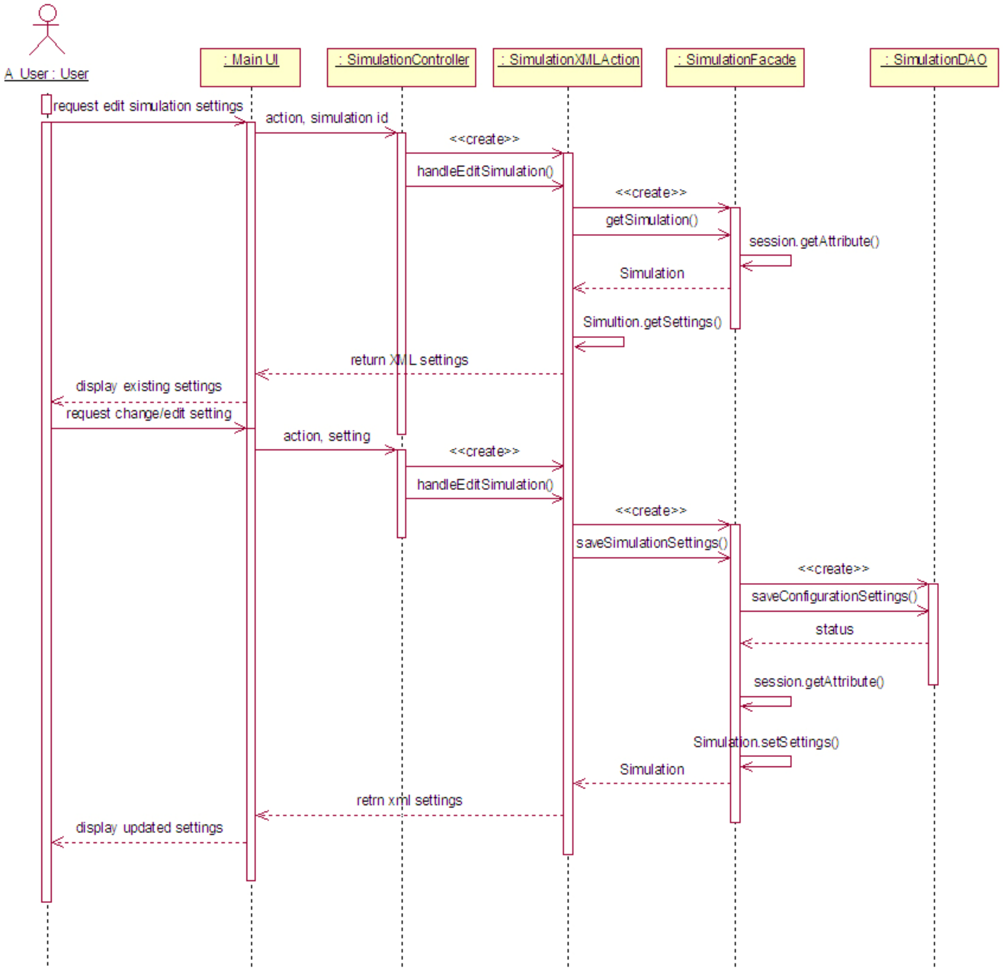


Figure 7.13: Simulation settings sequence diagram

7.3.6.2 Device Use Case

The simulation use case, illustrated in Figure 7.14, provides a look at the interaction between the user and the various functions provided by the system. The user may create a new device either manually or through a simulation wizard, open an existing device that was previously created, save a device, rename a device, or delete a device.

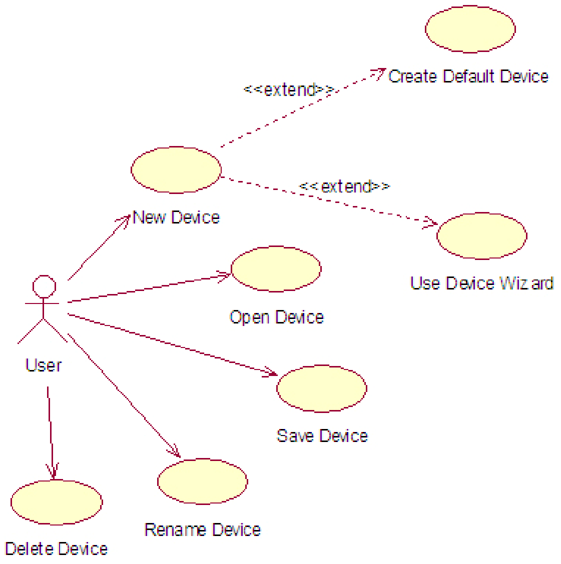


Figure 7.14: Device use case

7.3.6.2.1 New Device

When issuing a new device command, the XML request is handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object. In turn, a new `SimulationFacade` is created in which the `newDevice()` method is called. The `SimulationDAO` creates a new `Device` object, adds the data provided by the user to the object, and saves it to the database. For easy retrieval, the object is persisted in a session and added to the current simulation. A status is sent back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI. The user is then presented with a confirmation message and the user interface is updated appropriately.

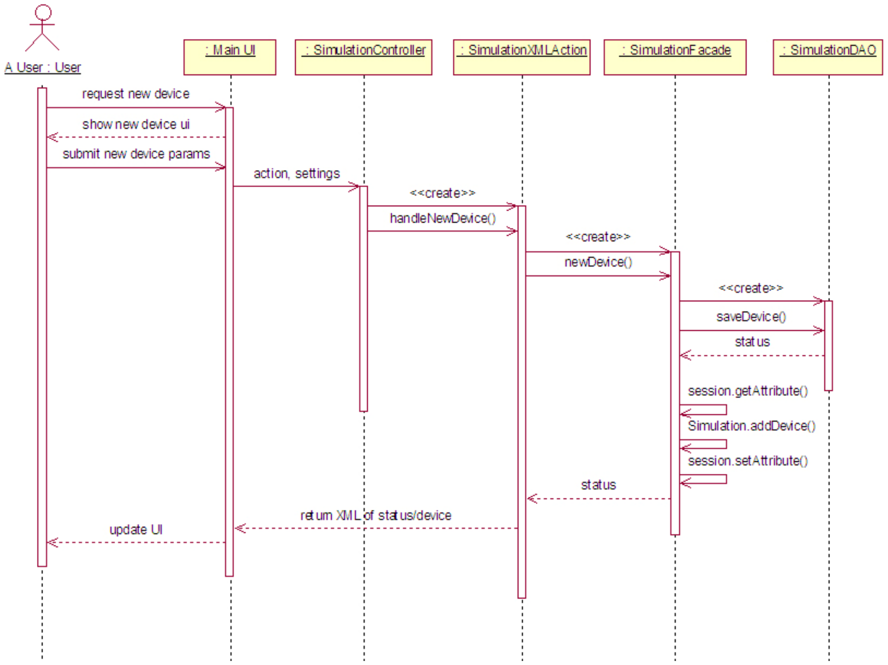


Figure 7.15: New device sequence diagram

7.3.6.2.2 Open Device

When the open command is issued, an XML request is formed and handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object. A new `SimulationFacade` is created in which the `getDevices()` method is called. The `SimulationDAO` retrieves a list of devices from the database and returns it back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI. The user is then presented with a list of devices to choose from. Once a device is chosen, a separate XML request is sent to the server containing the device id. Again, the `SimulationController` handles the request, extracts the id and calls the `SimulationXMLAction.handleOpenDevice()` method. From there, the `SimulationFacade.getDevice()` is called with the device id and then sent to the `SimulationDAO` object. When the device is returned it gets added to the

`Simulation` object. The XML response is formed and then sent back to the GUI where it is updated to reflect the opened device.

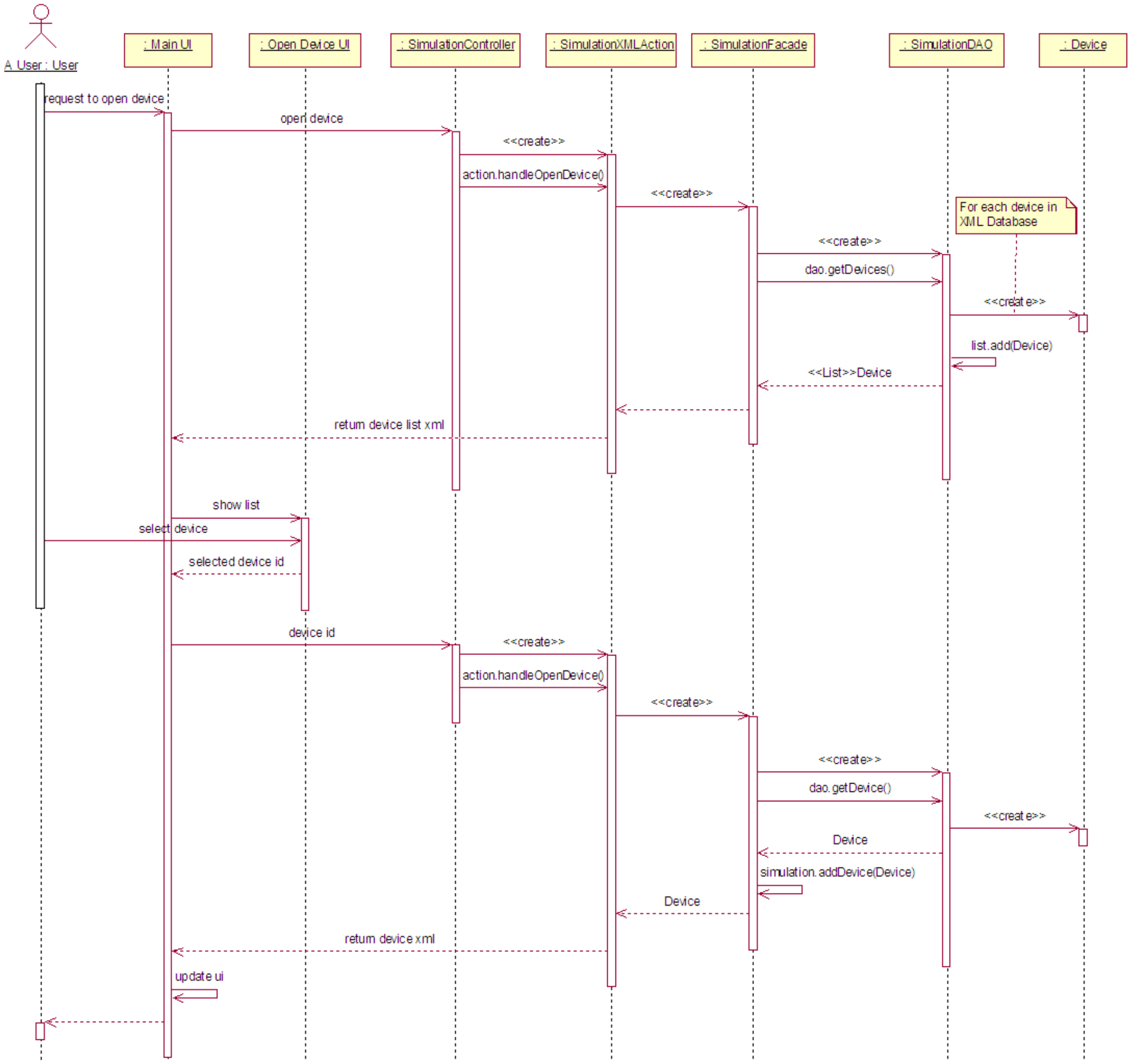


Figure 7.16: Open device sequence diagram

7.3.6.2.3 Save Device

In issuing a save device command, the XML request is handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object, extracts the id, and then calls the `SimulationXMLAction.handleSaveDevice()` method. From within the `handleSaveDevice()` method, a new `SimulationDAO` object is created and the `saveDevice()` method is called which stores the updated information in the database. A status method is returned back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI.

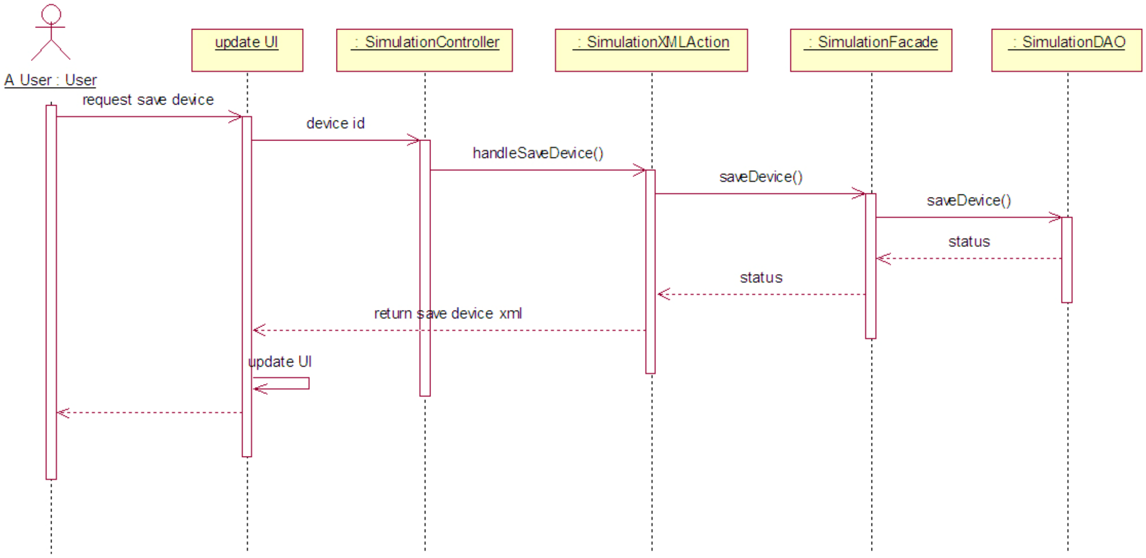


Figure 7.17: Save device sequence diagram

7.3.6.2.4 Rename Device

When a rename device command is issued, the XML request is formed and handled by the `SimulationController` which instantiates a new `SimulationXMLAction` object, extracts the id and new name, and then calls the `SimulationXMLAction.handleRenameDevice()` method. From within the `handleRenameDevice()` method, a new `SimulationFacade` object is created and the `getDevice()` method is called. This returns the `Device` object from the session. The new name is inserted into the object and then stored in the database. A status method is returned back to the `SimulationXMLAction` object where an XML response is formed and returned back to the GUI.

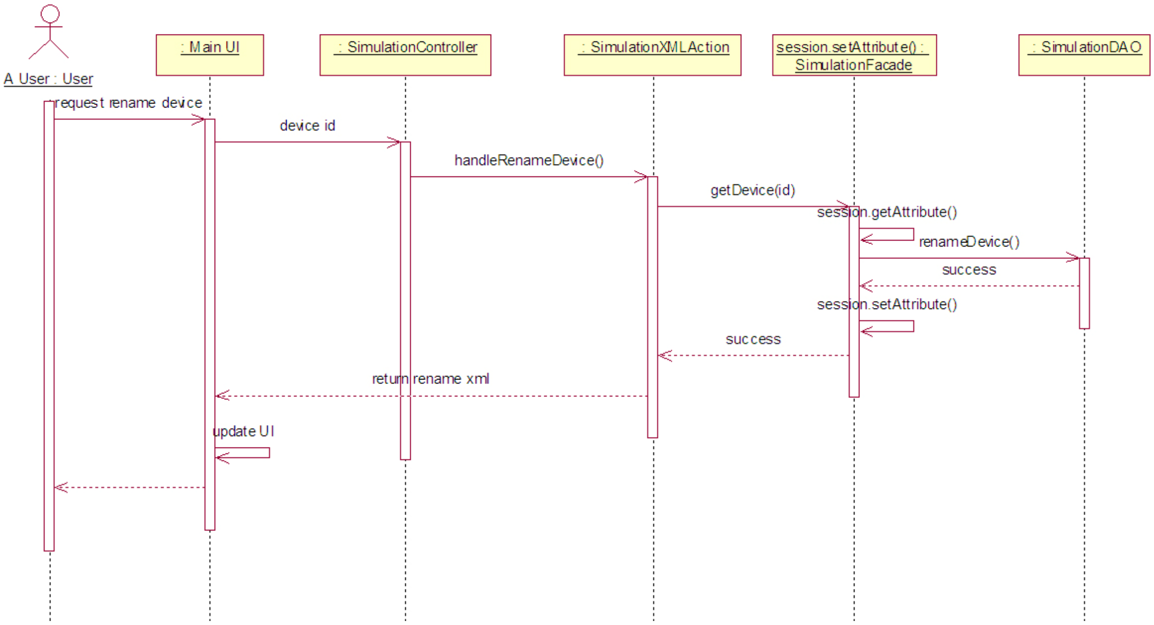


Figure 7.18: Rename device sequence diagram

7.3.6.2.5 Delete Device

When the delete command is issued, an XML request is formed and the device id sent to the server and handled by the `SimulationController`. The `SimulationController` creates a new `SimulationXMLAction` object and calls the `handleDeleteDevice()` method. From there, a new `SimulationFacade` is created and the `deleteDevice()` method is called with the device id as the attribute. The `SimulationDAO` deletes the device from the database and also updates every other simulation that references it. The device is then removed from the Simulation object. The response XML is then formed with the status (success or error) and then sent to the GUI where the appropriate action takes place.

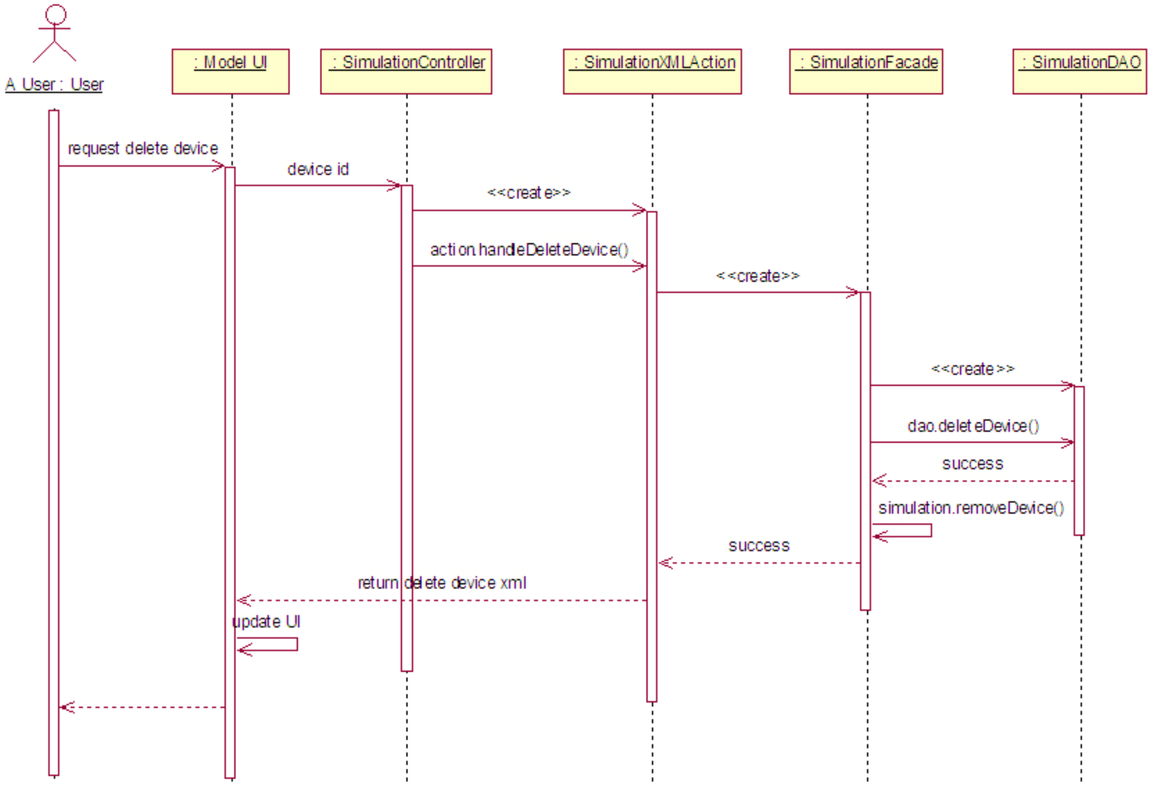


Figure 7.19: Delete device sequence diagram

7.3.6.3 Run Simulation Use Case

The run simulation use case, illustrated in Figure 7.20, provides a look at the interaction between the user and the various functions provided by the system. In this specific case, the user can only perform two functions: run simulation and view statistics. We note that the view statistics use case extends the run simulation use case.

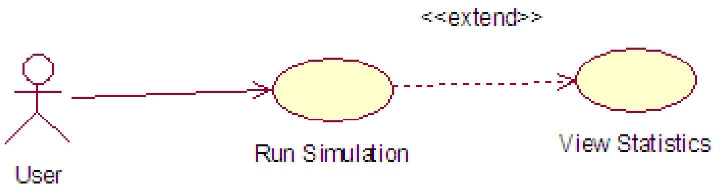


Figure 7.20: Run simulation use case

7.3.6.3.1 Run Simulation

This action allows the user to run the simulation from the currently configured system. An XML request is sent to the server with the simulation id and action. The

`SimulatorController` creates a new `SimulationXMLAction` object calling the `handleRunSimulation()` method. The `runSimulation()` method is then called with the simulation id. The `Simulation` object is then retrieved from the session by id. From the `Simulation` object, the information is retrieved and the necessary configuration scripts are created. The ScriptSim simulator is then run using the newly created scripts. Once run, the simulator monitors the status and sends updates back to the UI via an XML response. Once complete, the user is notified of the completion and the status.

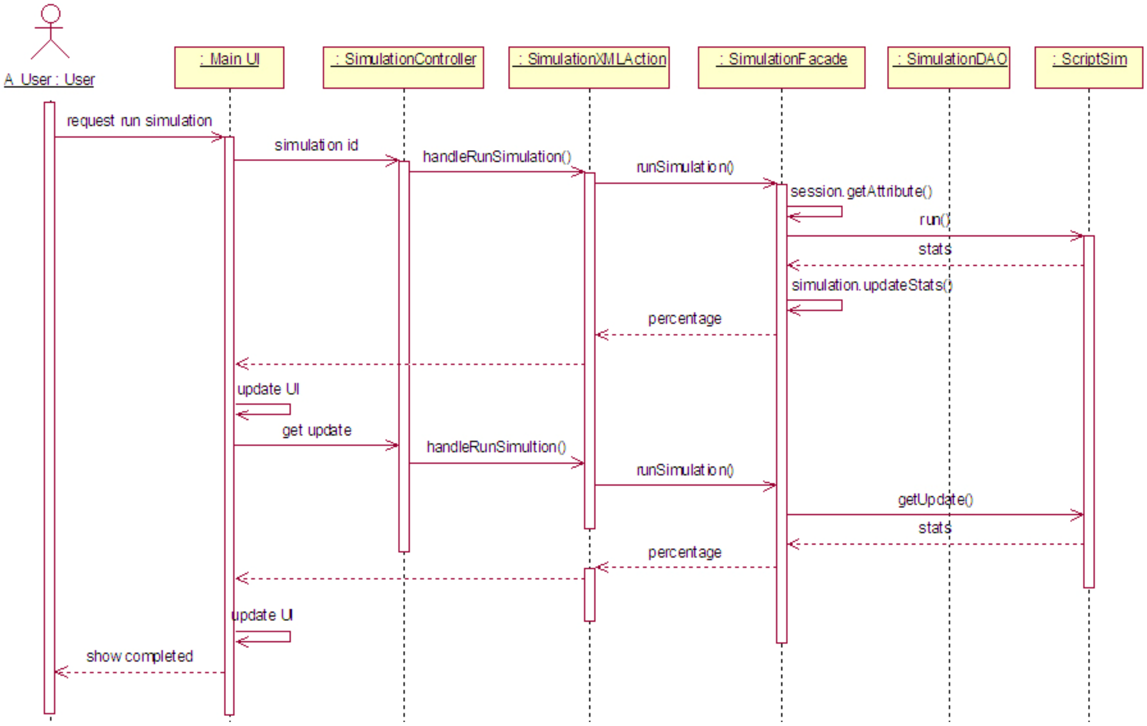


Figure 7.21: Run simulation sequence diagram

7.3.6.3.2 View Statistics

This action allows the user to view the statistics from a previously run simulation. An XML request is sent to the server with the simulation id and action. The `SimulatorController` creates a new `SimulationXMLAction` object calling the `handleViewStatistics()` method. The `getStatistics()` method is then called with the simulation id. The `Statistics` object is then retrieved once the `generate()` method is called. The `Statistics` object is returned to the controller which extracts the information and formats it into an XML response. The response is then sent back to the client, which formats the page and presents it to the user.

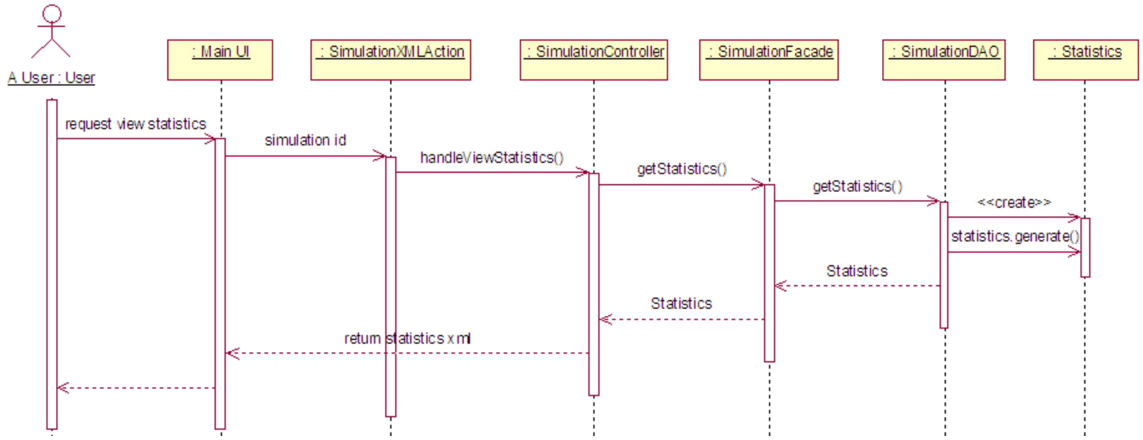


Figure 7.22: View statistics sequence diagram

7.4 Modification and Implementation of ScriptSim

We recall from Section 6 that ScriptSim provides a complete PCI model per the PCISIG Local Bus Specification Version 2.2, which is described in detail in Section 4. We noted previously that ScriptSim lacks PCI-X functionality and the ability to model device behavior. To add in the PCI-X component, as described in Section 5, we simply modified a single python file (`pci_lib.py`), the details of which are beyond the scope of this thesis. However, adding the ability to model device behavior was more complex. Previously in Section 7.1, we provided details on how to model device behavior by decomposing a master and target device into a set of performance parameters. We modified ScriptSim to accommodate the parameters of both master and target device descriptors as described in Sections 7.1.1.1 and 7.1.1.2 respectively. Before providing these details, outlined in Section 7.4.2 below, we must first understand some of the challenges of synchronization when implementing device latency as described in the following section.

7.4.1 Device Latency and Synchronization

One of the factors that affect bus performance is device latency. Both master and target devices have the ability to add wait states during a transaction if one, or both, are unable to send or receive data during a clock period. In Figure 7.23, the target device adds a wait state on clock cycle 3, which is signaled by the assertion of **TRDY#**. Before clock cycle 4 occurs, the target deasserts **TRDY#** and a data transfer occurs during that same clock cycle. It is here we note that a data transfer will occur on clock cycle 5 as long as the target deasserts **TRDY#** at some point in between clock cycle 4 and clock cycle 5. In a real system, devices are attached to a physical bus and thus, any signal that is placed on the bus is almost immediately sensed by other devices. However, this is not true in the case of ScriptSim which tries to emulate a PCI bus.

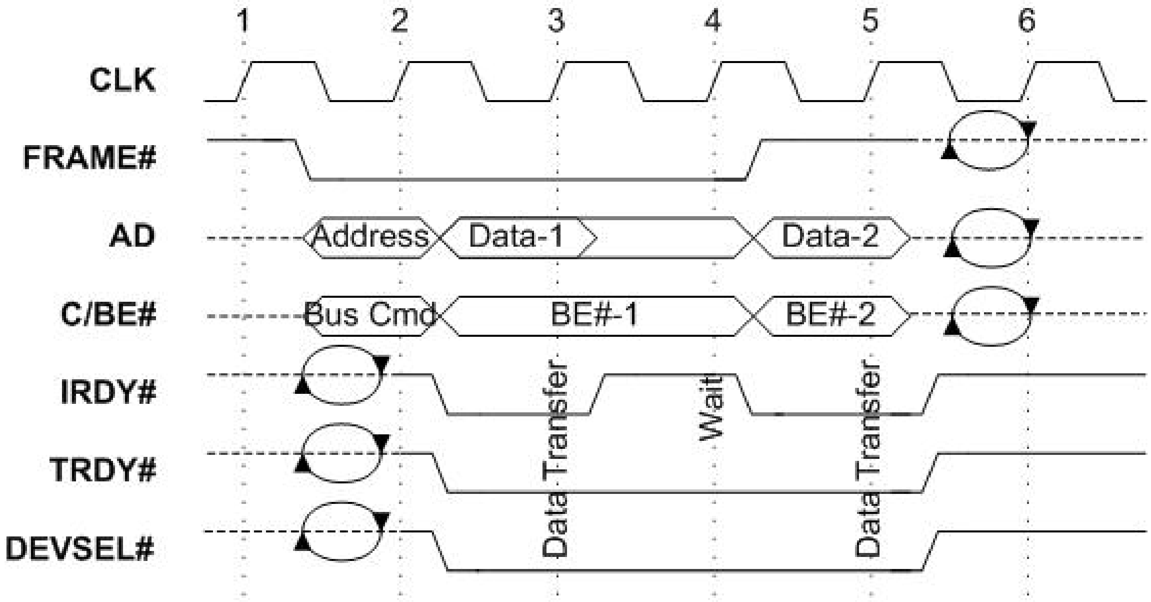


Figure 7.23: Typical PCI write transaction

To understand this more clearly, we must first understand how ScriptSim handles bus signals. Figure 7.24 provides a high-level view of how bus signals are sent and received in ScriptSim. In this figure, three devices (dev1, dev2, and dev3) are attached to the bus, each of which send and receive bus signals on every clock cycle. During a single clock period, the ScriptSim verilog program receives bus signals from every device, merges them using an `and_reduce` function, and then distributes the merged signal back to all devices. Thus, signals sent by some device on clock cycle *n* will not be received by other devices until clock cycle *n* + 1. In the case of a data transfer, devices will now receive data one clock cycle late. This can easily be solved by allowing devices to send data one clock cycle early. However, this only works if devices are always ready to send or receive data (i.e.: insert no wait states). If a device has to induce latency by inserting wait states, data transfers that occurred during a previous clock period are no longer valid.

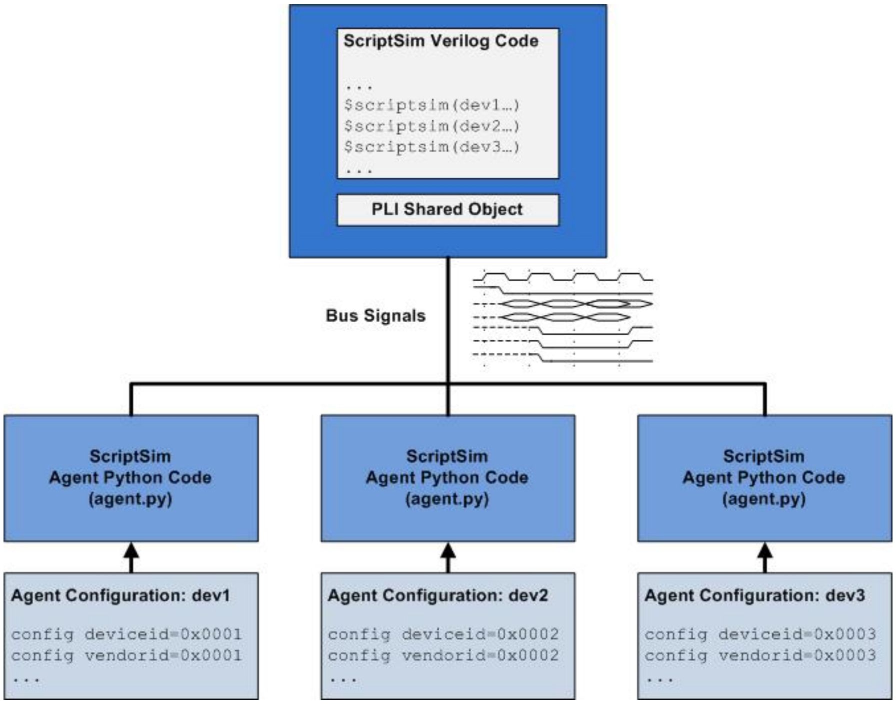


Figure 7.24: High-level view of bus signal distribution in ScriptSim

To illustrate, we present a simple example as shown in Figure 7.25, which recreates the PCI write shown in Figure 7.23. The start of a transaction begins with an idle phase on clock cycle 1, which is followed immediately by the address phase in clock cycle 2 (**AD_OUT**). At this point, we realize the address will not be received by other devices on the bus until clock cycle 3 (**AD_IN**). The first data phase will not occur until clock cycle 4 since the target asserts **TRDY_OUT#** on clock cycle 3, which will not be received by the master until clock cycle 4. While data is sent out during clock cycle 4, the target won't receive the data until clock cycle 5, at which point the target will deassert **TRDY_OUT#** indicating a wait state. Thus, the final piece of data will be sent on clock cycle 6 and received by the target on clock cycle 7. The transaction ends on clock cycle 8, two full clock cycles later than expected.

To solve this issue, we allowed devices to send signals one clock early. However, issues arrive when a device needs to insert wait states since it is possible for a device to send data during a clock period where a wait state should occur. Therefore, we allow both master and target to share latency information during a transaction. By knowing a priori how many wait states a device must incur during a specific portion of a transaction, a device can send signals at the correct time. While this does not correctly emulate signals during a PCI transaction, it does produce correct results.

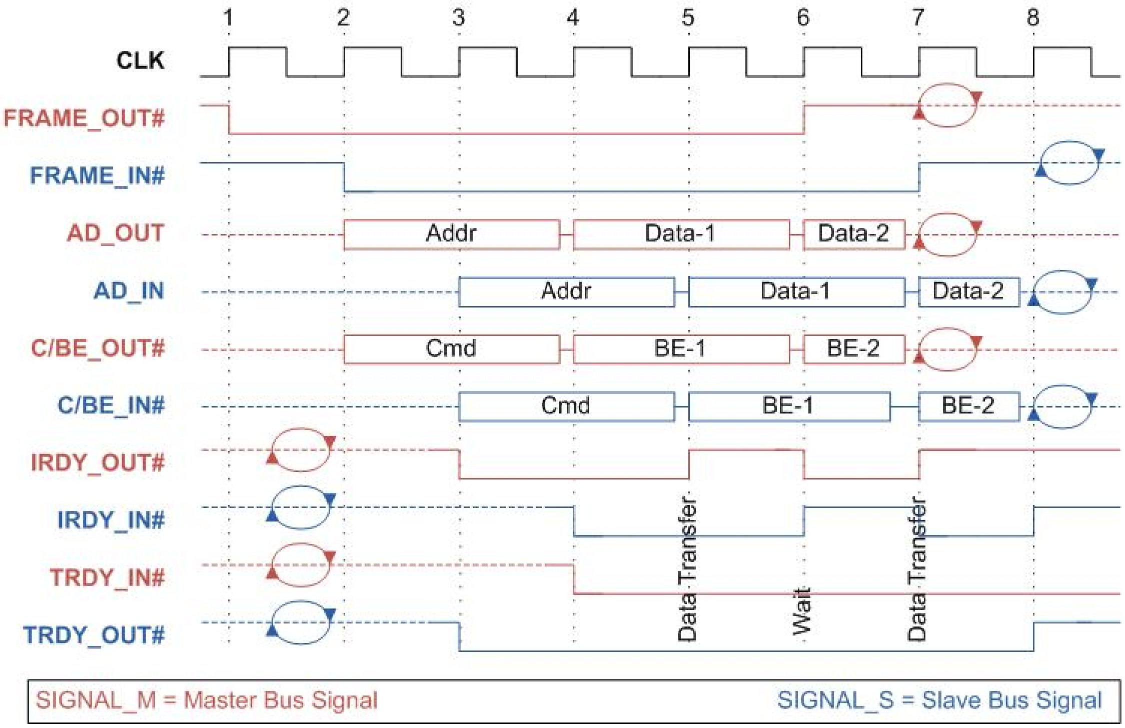


Figure 7.25: PCI write transaction in ScriptSim

7.4.2 Implementing Device Behavior

We recall that ScriptSim uses the notion of a control file to configure each device that is to be simulated. Within the control file are a set of keyword-value pairs that describe the device depending on if it is a master device or target device. These keyword descriptions are detailed in Tables 6.4 and 6.5 for master and target devices, respectively. To implement the parameters associated with our device descriptor, as discussed in Section 7.1.1, we extend the control file to include additional parameters. These extended parameters for master devices are outlined in Table 7.1 while extended target device parameters are outlined in Table 7.2.

Table 7.1: Extended master keyword descriptions (continued on pg. 147)

Keyword	Value	Description
cmd	2-4 letter command abbreviation	Specify the command to be driven on the **C/BE#** lines. **PCI values:** `ia` – interrupt acknowledge `sc` – special cycle `ir` – I/O read `iw` – I/O write `mr` – memory read `mw` – memory write `cr` – configuration read `cw` – configuration write `mm` – memory read multiple `ml` – memory read line `mi` – memory write and invalidate **PCI-X values:** `ia` – interrupt acknowledge `sc` – special cycle `ir` – I/O read `iw` – I/O write `id` – Device ID `mr` – memory read `mw` – memory write `amrb` – alias to memory read block `amwb` – alias to memory write block `cr` – configuration read `cw` – configuration write `spc` – split completion `dac` – dual address cycle `mrb` – memory read block `mwb` – memory write block
miws	Number	This value specifies the number of master initial wait states.
msws	Number(s)	Specifies the number of master subsequent wait states.

tiws	Number	This specifies the number of initial waits states we expect for the target.
tsws	Number(s)	This specifies the number of subsequent waits states we expect for the target.
tirt	Number	Specifies the initial retry threshold of the target we are communicating with.
tsrt	Number	Specifies the subsequent retry threshold of the target we are communicating with.
start	Number	This value specifies the start time, in cycles. A master will not initiate a transaction until the current clock cycle is greater than or equal to this value.

Table 7.2: Extended master keyword descriptions (continued from pg. 146)

Keyword	Value	Description
cap_pntr	Number	The Capability Pointer specifies the hardwired value in the configuration space register at address 0x34 (upper 16 bits). The address specified in this register is a pointer to the first capabilities list item
capability	Number(s)	Specify the hardwired capability and written value. Example: capability=(0x84, 0x7, 0x0, 0x0, 0x0) specifies the item is located at address 0x84, that it is a PCI-X capability list, has no other capabilities in the list, has an ID of 0x0, and a status set to 0x0.

Table 7.3: Extended target keyword descriptions

While we acknowledge there were several additions and changes to the source code to accommodate the above parameters, the implementation details are beyond the scope of this thesis.

CHAPTER 8

EXPERIMENTAL RESULTS

In this section we discuss the experimental results obtained from our PCI simulator. To verify the accuracy of the simulator, we gathered PCI and PCI-X cycle snapshots from a variety of configurations generated by Vanguard's VMetro Bus Analyzer [19]. Statistical information was then extracted and used in our simulator to generate results.

For experiments, the setup was organized as shown in Figure 8.1. Modern PC architectures split the memory controller and I/O controller into separate chips labeled as the Northbridge and Southbridge respectively. The Northbridge, or memory controller hub, handles high speed communication between devices such as the CPU, RAM, and AGP or PCI Express. The Southbridge, or I/O controller hub, typically handles less performance critical I/O devices such as the PCI or PCI-X bus, LPC Bus and other I/O devices. The PCI/PCI-X bus will contain one or more PCI/PCI-X devices as well as the VMetro Bus Analyzer which will be used to passively monitor the bus. Data collected by the analyzer will be then sent to the PC via USB which can be analyzed through VMetro's BusView software.

In Table 8.1 we outline the various configurations used for testing. Each test configuration was first run on a physical system using the VMetro Bus Analyzer to capture the data. We then calculated the following statistics which will be used as a comparison:

Utilization	Indicates how the bus is being used and is calculated by dividing the number of bus transactions by the total number of clocks.
Efficiency	This the duration of data transfers versus duration of transactions and is calculated by dividing the data total percentage by the transactions percentage.
Bandwidth	Amount of data sent over the bus per unit time and is calculated by dividing the total number of bytes sent divided by the total time.

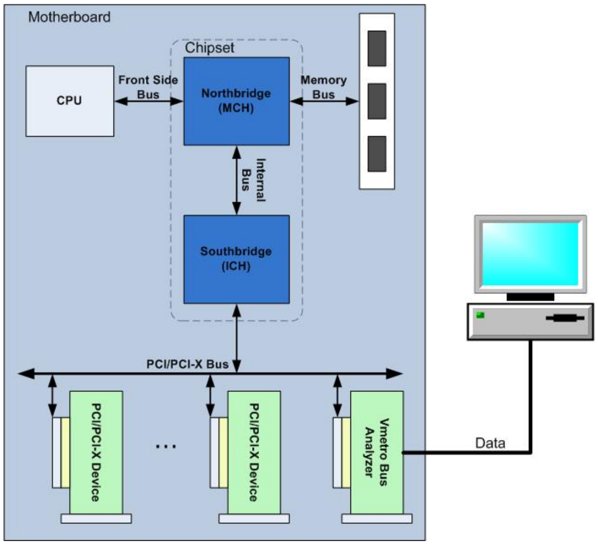


Figure 8.1: Experimental system setup

Sim. Number	**Bus**			**PCI Devices**		**PCI-X Devices**		**Hostbridge**
	Type	**Speed**	**Width**	**Master**	**Target**	**Master**	**Target**	
1	PCI	33 MHz	32-bit	1	1	0	0	Intel [30]
2	PCI	33 MHz	32-bit	3	2	0	0	Intel [30]
3	PCI-X	133 MHz	64-bit	0	0	1	1	Intel [31]

Table 8.1: Summary list of experiments

8.1 Simulation 1: Single PCI Master

Our first simulated system consists of a 32-bit, 33 MHz PCI bus using Intel's 82801DB I/O controller hub. A single bus master device, Foresight Imaging's PCI frame grabber, can inject data onto the bus at a rate of 110 MB/s (calculated using a resolution of 1280 by 1024 with 24 bit depth at 28 frames per second) and exhibits performance characteristics as described in Table 8.2. The frame grabber writes to and reads from system memory, acting as the target device, via the host bridge. The target device incurs no initial wait states on writes and exhibits an average of 15-34 initial wait states on reads. It can sustain long bursts (up to a 4K page boundary) with no subsequent wait states. The performance characteristics of system memory are described in Table 8.3. The frame grabber will perform 34 burst writes at the maximum speed (i.e.: no subsequent wait states) for each transaction until it hits a cache line which falls on a 4KB page boundary, in which case a read will occur. A round robin arbiter is modeled after the PCI scheduler found in Intel's 82801 ICH with a MTT (multi-transaction timer) set to 20.

The following graph (Figure 8.2) shows the results of our simulation versus the statistics gathered by the VMetro Bus Analyzer. We incur minimal error with respect to bus utilization and efficiency, 2.21% and 0.3% respectively. However, there was a 7.76% difference in the bandwidth, with our simulator producing a larger bandwidth value. We believe this is due to our optimistic calculation of the recovery period. We recall that our definition of recovery period is the minimum amount of time between transaction requests, calculated using Equations 7.1 and 7.2. Generally this is the amount of time it takes to refill the buffers, although, there may be other device specific factors involved which may delay requests. Our calculation only depends on the device’s required bandwidth and the burst length of the next transaction while ignoring other outside factors.

Device	Frame Grabber
Device Type	32-bit PCI Master
Injection Rate	110 MB/s
Read/Write Ratio	Perform write until 4K boundary and then 1 read transaction
Burst Length	*Read*: 4 *Write*: 34
Initial Wait States	*Read*: 0 *Write*: 0
Subsequent Wait States	*Read*: 0 *Write*: 0
Master Latency Timer	64
Recovery Period	Calculated using Equation 7.1
Transaction Count	200

Table 8.2: Master performance specifications for simulation 1

Device	Host Bridge (System Memory)
Device Type	32-bit PCI Target
Decode Speed	Medium
Burst Length	Stop at 4K page boundary
Initial Wait States	*Read*: Random (15 - 24) *Write*: 0
Subsequent Wait States	*Read*: 0 *Write*: 0
Initial Retry Threshold	16
Subsequent Retry Threshold	8

Table 8.3: Target performance specifications for simulation 1

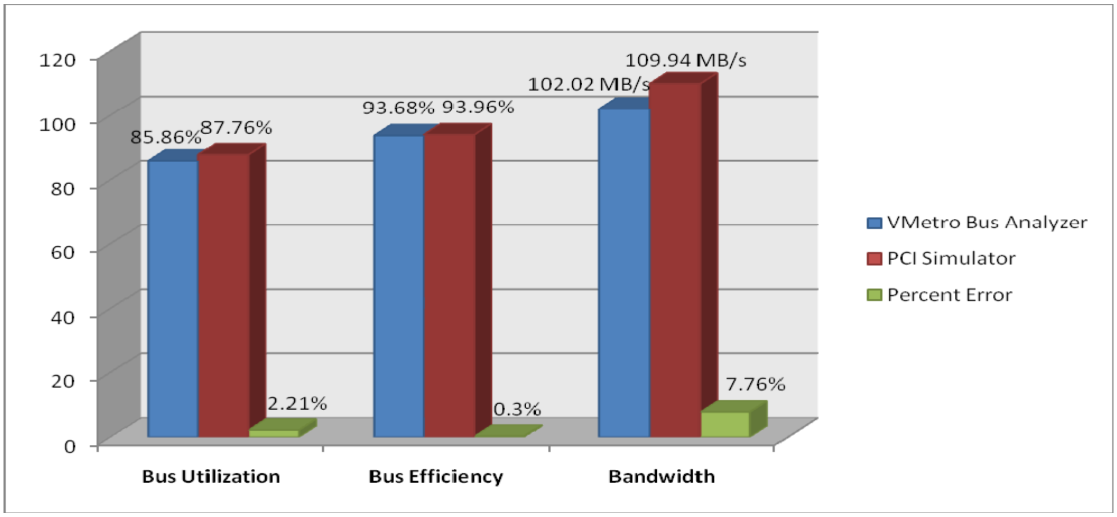


Figure 8.2: VMetro analyzer versus PCI simulator results for simulation 1

8.2 Simulation 2: Three PCI Masters

As with the previous simulation, our system consists of a 32-bit, 33 MHz PCI bus using Intel's 82801DB I/O controller hub. However, in order to generate more interesting results, we now include three PCI master devices. In addition to the frame grabber we used in the previous simulation, we include Engineering Design Team's Camera and PCI interface streaming live data which can inject approximately 110 MB/s of data onto the bus. The camera performs burst writes for 66 cycles at the full data rate (i.e.: no wait states). We note that, while the device's master latency timer is set at 64, the PCI specification allows two extra cycles before a device must complete the transaction. After each burst write, a memory read is performed followed by an I/O read of 4 cycles and 1 cycle, respectively. Periodically, a CPU read of the camera's PCI interface is performed and thus injects approximately 1 MB/s of data. Reads and writes to memory are still performed via the host bridge and assume the same parameters. However, I/O reads via the host bridge now incur initial wait states of between 5 and 8 cycles. We also assume a round robin arbiter with the MTT set to 20 cycles.

Device	Frame Grabber	Camera	CPU
Device Type	32-bit PCI Master	32-bit PCI Master	32-bit PCI Master
Injection Rate	110 MB/s	110 MB/s	1 MB/s
Read/Write Ratio	Perform write until 4K boundary and then 1 read transaction	Repeat memory write followed by read and then I/O read	All memory reads
Burst Length	*Read*: 4 *Write*: 64	*Read*: 4 *Write*: 66	*Read*: 1
Initial Wait States	*Read*: 0 *Write*: 0	*Read*: Random (13 – 15) *Write*: 0	
Subsequent Wait States	*Read*: 0 *Write*: 0	*Read*: 0 *Write*: 0	
Recovery Period	Calculated using Equation 7.1	Calculated using Equation 7.1	Calculated using Equation 7.1
Master Latency Timer	64	64	64
Transaction Count	200	200	200

Table 8.4: Master performance specifications for simulation 2

Device	Host Bridge (System Memory)	Host Bridge (I/O Device)
Device Type	32-bit PCI Target	32-bit PCI Target
Decode Speed	Medium	Medium
Burst Length	Stop at 4K page boundary	1
Initial Wait States	*Read*: Random (15 to 24) *Write*: 0	*Read*: Random (5 - 8) *Write*: 0
Subsequent Wait States	*Read*: 0 *Write*: 0	*Read*: 0 *Write*: 0
Initial Retry Threshold	16	16
Subsequent Retry Threshold	8	8

Table 8.5: Target performance specifications for simulation 2

Shown on following graph (Figure 8.3) are the results of our simulation versus the statistics gathered by the VMetro Bus Analyzer. What is interesting to note is the fact that we find a significant decrease in error (87.09%) with respect to the bandwidth from our previous simulation. We theorize this may be due to the fact that the recovery period may be hidden by another device transferring data on the bus. Consider a simple example with two devices on a bus, D_1 and D_2, where D_2 has been granted access to the bus and D_1 is beginning its recovery period. If the recovery period for D_2 is less than or equal to the bus access time required by D_1 to complete its transfer, then D_2 will be able to immediately start after D_1 completes. If we denote the overhead associated to D_2 as $D_{2,overhead}$ and the following holds true:

$$D_{2,overhead} \leq D_{1,access} - D_{2,recovery} \tag{8.1}$$

Then the amount of error associated with the additional overhead is eliminated since in our simulated case and in the physical system D_2 will access the bus as soon as D_1 completes. In fact, we see that in Section 3.1.2 and in 3.1.3, Schönberg provides a detailed explanation of why this may be the cause. Specifically, see Figures 3.2 and 3.3.

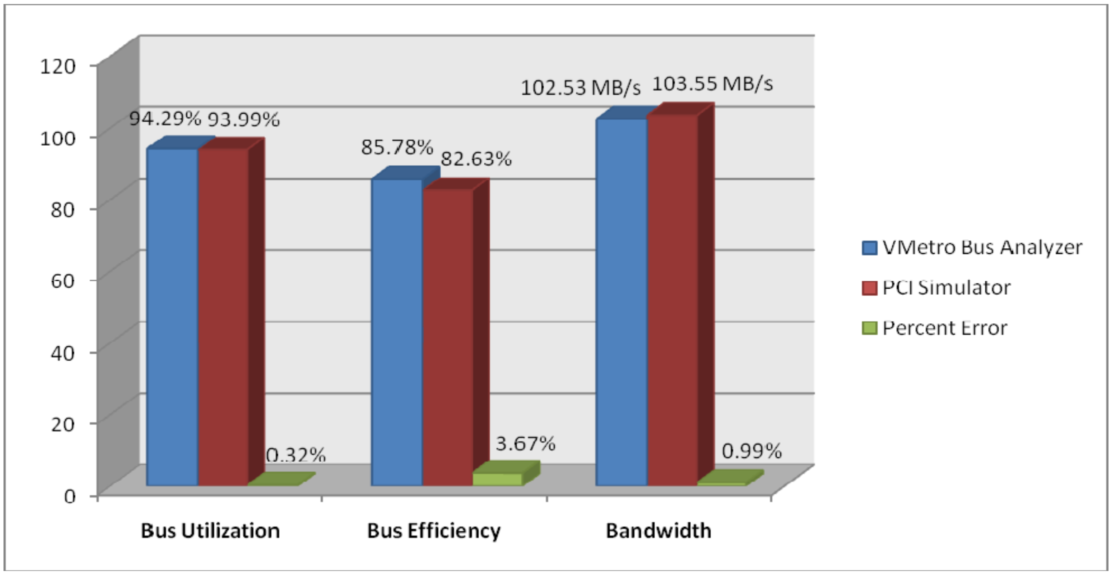


Figure 8.3: VMetro analyzer versus PCI simulator results for simulation 2

8.3 Simulation 3: Single PCI-X Master

Our final simulated system consists of a 64-bit, 133 MHz PCI-X bus using Intel's 6700PXH 64-bit PCI hub. A single bus master device, a Nallatech 64-bit 133 MHz PCI-X FPGA computing motherboard [32], can inject data onto the bus at a rate of 192 MB/s and exhibits performance characteristics as described in Table 8.6. The FPGA can perform burst writes to system memory, acting as the target device, via the host bridge. The target device incurs no initial wait states on writes and can sustain long bursts (up to a 4K page boundary) with no subsequent wait states. The performance characteristics of system memory are described in Table 8.7. The FPGA will perform 1,024 burst writes with no subsequent wait states for each transaction until it hits a cache line, which falls on a 4KB page boundary.

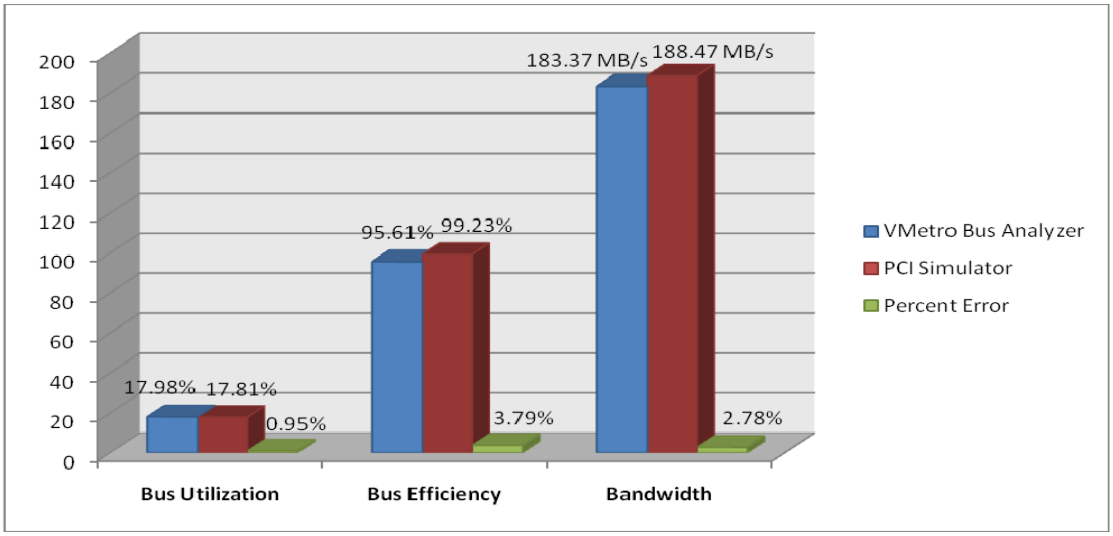


Figure 8.4: VMetro analyzer versus PCI simulator results for simulation 3

The results of our experiment are shown in Figure 8.4. As we have discussed previously, the simulator calculates a minimum for the recovery period, thus leading to a higher bandwidth. However, we also notice that our simulator has a higher efficiency than the actual system. The bus efficiency is calculated using the following formula:

$$efficiency = \frac{percent_{data}}{percent_{utilization}} \tag{8.2}$$

Where the data percentage is calculated by:

$$percent_{data} = \frac{clock\_cycles_{data}}{clock\_cycles_{total}} \tag{8.3}$$

Keeping the total number of clock cycles the same, as we increase the amount of data cycles, we increase the data percentage, thus leading to a higher efficiency.

Device	Nallatech FPGA
Device Type	64-bit PCI-X Master
Injection Rate	192 MB/s
Read/Write Ratio	Perform write until 4K boundary and then 1 read transaction
Burst Length	*Write*: 1024
Initial Wait States	*Read*: 0 *Write*: 0
Subsequent Wait States	*Read*: 0 *Write*: 0
Recovery Period	Calculated using Equation 7.1
Master Latency Timer	1024
Transaction Count	100

Table 8.6: Master performance specifications for simulation 3

Device	Host Bridge (System Memory)
Device Type	64-bit PCI-X Target
Decode Speed	Medium
Burst Length	Stop at 4K page boundary
Initial Wait States	*Read*: Random (15 - 24) *Write*: 0
Subsequent Wait States	*Read*: 0 *Write*: 0
Initial Retry Threshold	16
Subsequent Retry Threshold	8

Table 8.7: Target performance specifications for simulation 3

CHAPTER 9

SUMMARY

9.1 Conclusion

In this thesis we have developed a novel approach to PCI simulation using ScriptSim, an open-source PCI simulator. We extended ScriptSim to include PCI-X functionality and developed a web-based graphical user interface which provides users with a high level of configurability to model real-life systems. The architecture and design of the system employ well known software engineering techniques that ensure scalability. By using well known design patterns we promote reuse while decreasing overall design time.

In order to achieve a high-level of accuracy in our simulations, we developed techniques that allow devices to exhibit individualized behavior on the bus. This was done by decomposing a device into a set of performance parameters that, together, make up a device descriptor. While techniques were borrowed from the PCI specifications and a previous simulator, we introduce two unique parameters, injection rate and recovery period allowing us to specify how much data a device is able to put onto the bus and the minimum amount of time it can start a subsequent transaction, once one has already started. Experimental results show we achieve a high level of accuracy in bus utilization, efficiency, and bandwidth versus system data captured by a bus analyzer.

9.2 Future Use

Simulators have become a popular item in a designer’s toolbox. They allow for fast prototyping with minimal expense helping to reduce costly mistakes that may arise in the future. For those who are looking for the PCI bus as a solution to their problem, we believe our tool could be of great benefit. Designing, building, and testing custom hardware for the PCI bus can become both timely and costly. Using our PCI web simulator, designers can easily simulate a PCI system based on their architecture to immediately determine if the PCI bus is an appropriate choice. Users may also wish to determine how different combinations of devices on the bus affect their custom device, thereby determining if it is scalable. In addition, they may wish to alter parameters of their simulated device to determine the best performance.

9.3 Future Work

While we have achieved an accurate PCI/PCI-X simulation tool with a flexible web-based GUI, there are several additions which could be implemented allowing for better ease of use and performance. Most of the GUI is text based; implementing a window which allows device objects to be dragged and placed into the window would allow users to easily visualize the architecture of the system they are modeling along with how devices may interact. For performance, simulations could be distributed amongst many load balanced systems.

While we believe that we have developed a rich and robust PCI simulation environment, we can also see room for improvement and how others might contribute. The open-source nature of ScriptSim allows developers to integrate current and future implementations of PCI. For instance, while we currently only implement PCI-X v1.0, one can augment the code to implement PCI-X v2.0. Furthermore, one might also wish to alter the code to test for possible improvements in the PCI protocol.

By developing the application interface using Java's enterprise architecture, we provide an extensible architecture for one to build upon. While most of the GUI is text based viewed via a web browser; implementing a window which allows device objects to be dragged and placed into the window would allow users to easily visualize the architecture of the system they are modeling along with how devices may interact. For performance, simulations could be distributed amongst many load balanced systems. One feature of the simulator is the ability to store devices in the context of parameters. Each developer needs to describe a device (such as an Ethernet card, or video card) in the context of these parameters which are usually derived from manufacturer specifications. It may be useful to allow designers to download devices created by others. In fact, we could easily see a repository where by devices can be stored and automatically updated by the simulator without the interference of the user. This would allow access to all the latest devices, in addition to fixes of current devices. Another contribution may provide the ability to recall results based on current input data. Simulations may take an excessive amount of time. However, it would be beneficial to know if someone has run a similar simulation, and thus only the results would be displayed. Along with the results would be the parameters of the simulation allowing the designer to adjust the simulation for comparison purposes.

In conclusion, we foresee many contributions that can be made by the community to allow our PCI simulator to grow and become an important aspect in the research community.

REFERENCES

[1] T. Shanley and D. Anderson, *ISA System Architecture*. MindShare, Inc., 1995.

[2] W. Fischer, “IEEE P1014 – a standard for the high-performance VME bus,” *IEEE Micro*, vol. 5, pp. 31-41, Feb. 1985.

[3] B.P. Aichinger, “Futurebus+ as an I/O Bus: Profile B,” in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (Gold Coast, Australia), pp. 300-307, ACM SIGARCH and IEEE Computer Society TCCA, May 19-21, 1992.

[4] *IEEE Standards for Futurebus+ - Logical Protocol Specification*, std 896.1-1991 ed., Mar. 1992.

[5] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, and Philips, *Universal Serial Bus Specification 2.0*, 2.0 ed., Apr. 2000.

[6] *IEEE Standard for a High Performance Serial Bus*, std. 1394-1995, Dec. 1995.

[7] *IEEE Standard for a High Performance Serial Bus – Amendment 1*, std. 1394a-2000, Mar. 2000.

[8] *IEEE Standard for a High Performance Serial Bus – Amendment 2*, std. 1394b-2002, Dec. 2002.

[9] Intel Corp., *Accelerated Graphics Port Interface Specification*. Santa Clara, May, 1998. Revision 2.0.

[10] Intel Corp., *AGP V3.0 Interface Specification*. Santa Clara, Sept., 2002. Revision 1.0.

[11] S.K. Dewan, “Introduction to PCI Express - A New High Speed Serial Data Bus,” *Nuclear Science Symposium Conference Record*, vol. 2, pp. 687-691, Oct. 2005.

[12] InfiniBand Trade Association, *InfiniBand Architecture*, rel. 1.2, Oct. 2004.

[13] E. Solari and G. Willse, *PCI & PCI-X Hardware and Software Architecture & Design*. Research Tech Inc., 6th ed., 2005.

[14] E. Finkelstein, “Design and Implementation of PCI Bus Based Systems.” Master’s Thesis, Tel Aviv University, 1997.

[15] S. Schönberg, "Using PCI-Bus Systems in Real-Time Environments." PhD Thesis, Technische Universität Dresden, 2002.

[16] PCI Special Interest Group, *PCI Local Bus Specification Revision 2.2*, Dec. 1998.

[17] PCI Special Interest Group, *PCI-X Addendum to the PCI Local Bus Specification*, rev. 1.0b, July 2002.

[18] "DesignWare PCI and PCI-X Solutions." `http://www.synopsys.com/products/designware/pci_solutions.html`.

[19] VMETRO. `http://www.vmetro.com/`.

[20] J. Liedtke, M. Völp, and K. Elphinstone, "Preliminary thoughts on memory-bus scheduling." In *9th SIGOPS European Workshop*, (Kolding, Denmark), Sept. 2002.

[21] "PCI Pamette V1." `http://www.hpl.hp.com/downloads/crl/pci/index.html`.

[22] Intel Corp., Santa Clara, *Intel 440FX PCISET 82441FX PCI and Memory Controller (PMC) and 82442FX Data Bus Accelerator (DBX)*, May 1996. Order # 290549-001

[23] M. Fowler and K. Scott, *UML Distilled Second Edition*. Reading, MA: Addison-Wesley, Jan. 2000.

[24] D. Alur, J. Crupi, and D. Malks, Core J2EE Patterns. Upper Saddle River, NJ: Prentice Hall, 2nd ed., 2003.

[25] J. A. Miller, R. S. Nair, Z. Zhang, and H. Zhao, "JSIM: A Java-Based Simulation and Animation Environment," in *Proceedings of the 30th Annual Simulation Symposium*, pp. 31-42, Atlanta, George, April, 1997.

[26] X. Huang and J. A. Miller, "Building a Web-Based Federated Simulation System with Jini and XML," in *Proceedings of the 34th Annual Simulation Symposium*, pp. 143-150, Seattle, Washington, April, 2001.

[27] J. Stearns, R. Chinnici, and Sahoo, "An Introduction to the Java EE 5 Platform." Sun Developer Network, May 2006. `http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/`.

[28] “Jini Network Technology.” `http://java.sun.com/developer/products/jini/index.jsp`.

[29] “Extensible Markup Language (XML).” `http:// www.w3.org/XML/`.

[30] Intel Corp., Santa Clara, *Intel 82801DB I/O Controller Hub 4 (ICH4),* May 2002. Order # 290744-001

[31] Intel Corp., Santa Clara, *Intel 6700PXH 64-bit PCI Hub,* July 2004. Order # 302628-002

[32] Nallatech. `http://www.nallatech.com/`.

[33] ScriptSim. `http://www.nelsim.com/scriptsim/intro.html`.