Thesis for M.Sc degree in Internetworking

Design and Implementation of PCI Multi-Bus Simulator

Department of Computer Science
University of Alberta

Student Name: XIAO YONG LIAO
Advisor Name: Dr. Mike MacGregor

September 2007

Acknowledgments

Design and implement a software system is not easy. Especially for a big software project, it will be the result of the effort of a sizeable team of people. Although our graduation project (MINT 709) is not big, I still get my team members' help and support, also other senior technicians' guide and help.

I'd espcially like to thank my advisor, Dr. Mike MacGregor. He helped me get this very good project, gave me very nice and patient guide step by step and lent us books generously. On the other hand, I will thank PhD. student, Qinghua Ye, who gives me lots of useful guide and help me correct many important problems. Also, I will Thank my another two team members, Yile Sun and Ligang Wang, who help me overcome lots of technical problems and give me many excellent idea to make this project go smoothly. Every meeting, I can get lots of new development ideas which expand my idea and open me up to new ideas and perspectives. Working with them, I can share their excellent development experience.

Finally, I will thank my wife, Qian Yang, who provides unstinting support for whatever I happen to be doing and cheerfully accepts my complaints about the workload that I freely elected to undertake. She always manages to be on hand whenever I need sustenance or sympathy, or both, and undoubtedly I would never develop my project smoothly without her.

Xiaoyong Liao

Abstract

PCI Bus is perhaps the most successful Bus design, both on the technical and the marketing levels. From the earlist general PCI Bus to current powerful PCI-E Bus, we can see that PCI Bus has become the most popular Bus system throughout the world.

Our project is consisted of three sections: source coding, GUI design and implementation, and testing. Finally, we will give some analysis and statistics according to the data generated by the PCI Multi-Bus simulator.

In the coding section, I firstly introduce the development history of PCI Bus and then I introduce two important new PCI Bus, including PCI-X and PCI-E. Finally, I introduce our simulator's design and implementation. As for the development eviroment, I use Linux Red Hat OS, G++ compiler, and OOP technology to do the coding work. On the other hand, we can use the generated data to plot by GNUPLOT tool under Linux OS and get a good statistical result.

Till now, my coding work goes smoothly. Although there still exists some bugs not being found, I believe that this simulator will become better and better gradually in the future.

Table of Contents

List of Figures and Tables

1. Computer Bus

It is widely recognized that the computer system bus affects the system characteristics in several important ways:

(1) The bus bandwidth and transfer parameters place a limit on the system performance.

(2) The system bus is an interface that connects hardware components produced by different vendors and provides interoperability.

(3) The wide variety of configuration options supported by increasingly complex and sophisticated I/O devices make manual configuration a difficult and error-prone task. Support for software based automatic configuration has become a necessity.

(4) When multiple processors share a bus with common resources, some form of support for multi-processing is required to arbitrate the use of shared resources.

Even though memories are getting faster, CPUs get faster and quicker. Although the memory burst speed can be increased by using interleaving, the initial latency cannot be reduced, and in fact becomes the dominant factor in bus usage. This is just one of a number of parameters, other than demand for raw bus bandwidth, that have changed in recent years and must be considered in modern system bus design.

From historical perspective aspect till now, we can review all kind of computer busses such as ISA, Micro Channel, EISA, NuBus, Future Bus+, VME64, PCI, PCI-X and PCI Express Bus. Today I here introduce more detail about PCI, PCI-X and PCI Express Busses, as for other busses, I will not introduce. Figure 1 shows us the PCI family history.

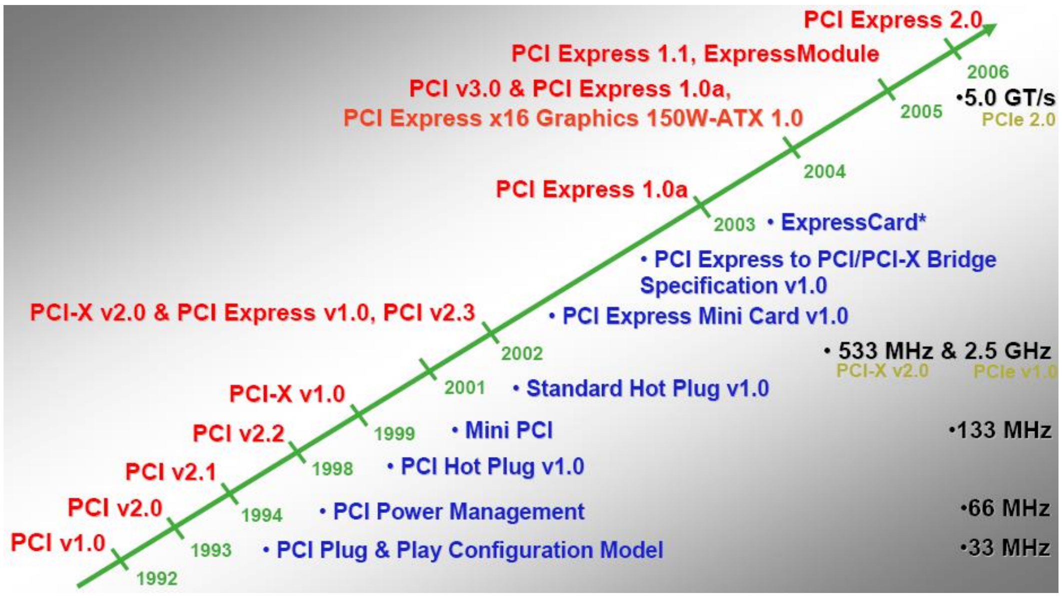


Figure1. PCI Family History

2. Introduction to PCI Bus

2.1 PCI Bus Application

The PCI Local Bus has been defined with the primary goal of establishing an industry standard, high performance local bus architecture that offers low cost and allows differentiation. While the primary focus is on enabling new price-performance points in today's systems, it is important that a new standard also accommodates future system requirements and be applicable across multiple platforms and architectures. Figure2 shows the multiple dimensions of the PCI Local Bus.

While the initial focus of local bus applications was on low to high end desktop systems, the PCI Local Bus also comprehends the requirements from mobile applications up through servers. The PCI Local Bus specifies 3.3 volt signaling requirements.

The PCI component and add-in card interface is processor independent, enabling an efficient transition to future processor generations and use with multiple processor architectures. Processor independence allows the PCI Local Bus to be optimized for I/O functions, enables concurrent operation of the local bus with the processor/memory subsystem, and accommodates multiple high performance peripherals in addition to graphics (motion video, LAN, SCSI, FDDI, hard disk drives, etc.). Movement to enhanced video and multimedia displays (i.e., HDTV and 3D graphics) and other high bandwidth I/O will continue to increase local bus bandwidth requirements. A transparent 64-bit extension of the 32-bit data and address buses is defined, doubling the bus bandwidth and offering forward and backward compatibility of 32-bit and 64-bit PCI Local Bus peripherals. A forward and backward compatible PCI-X specification (see the *PCI-X Addendum to the PCI Local Bus Specification*) is also defined, increasing the bandwidth capabilities of the 33 MHz definition by a factor of four

The PCI Local Bus standard offers additional benefits to the users of PCI based systems. Configuration registers are specified for PCI components and add-in cards. A system with embedded auto configuration software offers true ease-of-use for the system user by automatically configuring PCI add-in cards at power on.

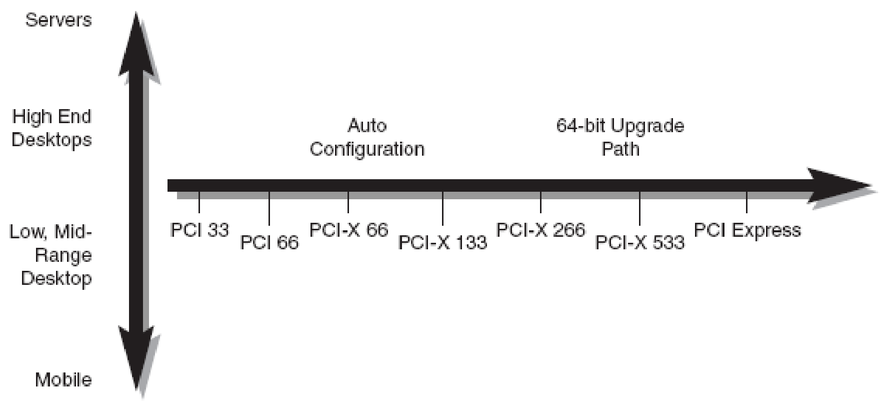


Figure2. PCI Local Bus Applications

2.2 PCI Bus Architecture

The PCI (Peripheral Component Interconnect) local bus [(PCISIG, 1995), (Shanley, 1995), (Kendall, 1994)] is a high speed bus. The PCI Bus was proposed at an Intel Technical Forum in December 1991, and the first version of the specification was released in June 1992. The current specification of the PCI bus is revision 3.0, which was released on February 3, 2004.

Since its introduction, the PCI bus has gained wide support from all the computer industry. Almost all PC systems today contain PCI slots, as well as the Apple and IBM Power-PC based machines, and the Digital Alpha based machines. The PCI standard has become so popular it influenced the creation of more than one related standard based on leveraging PCI technology.

The PCI Bus is designed to overcome many of the limitations in previous buses. The major benefits of using PCI are:

(1) High speed

The PCI bus can transfer data at a peak rate of 132MBytes/sec using the current 32 bit data path and a 33MHz clock speed. Future implementations featuring a 64 bit data path and 66MHz clock speed may transfer data as fast as 524MBytes/sec. Even at its basic mode, PCI delivers more than tenfold the performance levels offered by its predecessor in the PC world, the ISA bus.

(2) Expandability

The PCI bus can be expanded to a large number of slots using PCI to PCI bridges. The bridge units connect separate small PCI buses to form a single, unified, hierarchical bus. When traffic is local to each bus, more than one bus may be active concurrently. This allows load balancing, while still allowing any PCI Master on any bus to access any PCI Target on any other PCI bus.

(3) Low Power

Motherboards can lower their power requirement by reducing the clock rate as low as 0Hz (DC). All PCI compliant cards are required to operate in all frequency ranges from 0Hz to 33MHz.

(4) Automatic Configuration

All PCI compliant cards are automatically configured. There is no need to set up jumpers to set the card's I/O address, IRQ number, or DMA channel number. The PCI BIOS software is responsible for probing all the PCI cards in a system, and assigning resources to every card, as required.

(5) Future expansion

The PCI specification can support future systems by incorporating features such as an optional 64 bit address space, and 133MHz bus speed. The specification defines enough reserved fields in all the bus definitions (configuration space registers, bus commands, addressing modes), that any unforeseen enhancement will not hinder compatibility with present systems.

(6) Portability

By incorporating the (optional) OpenBoot standard, any device with OpenBoot Firmware can boot systems containing any microprocessor and O/S. Even without OpenBoot, it is common to see drivers for many video cards and SCSI controller for multiple CPU architectures and operating systems.

(7) Complex memory hierarchy support

The PCI Bus supports advanced features such as bus snooping to allow cache coherency to be kept even with multiple bus masters, and a locking mechanism to support semaphores.

(8) Interoperability with existing standards

The PCI Bus allows interoperability with existing ISA cards by supporting subtractive decoding of addresses (allowing addresses not decoded by PCI cards to be routed to an ISA backplane). The standard also supports the fixed legacy addresses for VGA cards and IDE controllers (required for system boot). Another feature supporting backward compatibility allows different devices to respond to different I/O byte addresses even if they share the same 64 bit word.

PCI is a *local* bus, sometimes also called an intermediate local bus, to distinguish it from the CPU bus. The concept of the local bus solves the downward compatibility problem in an elegant way. The system may incorporate an ISA, EISA, or Micro Channel bus, and adapters compatible with these buses. On the other hand, high-performance adapters, such as graphics or network cards, may plug directly into PCI. PCI also provides a standard and stable interface for peripheral chips. By interfacing to the PCI, rather than to the CPU bus, peripheral chips remain useful as new microprocessors are introduced. The PCI bus itself is linked to the CPU bus through a PCI to Host Bridge.

Figure3 shows a typical PCI Local Bus system architecture. This example is not intended to imply any specific architectural limits. In this example, the processor/cache/memory subsystem is connected to PCI through a *PCI bridge*. This bridge provides a low latency path through which the processor may directly access PCI devices mapped anywhere in the memory or I/O address spaces. It also provides a high bandwidth path allowing PCI masters direct access to main memory. The bridge may include optional functions such as arbitration and hot plugging. The amount of data buffering a bridge includes is implementation specific.

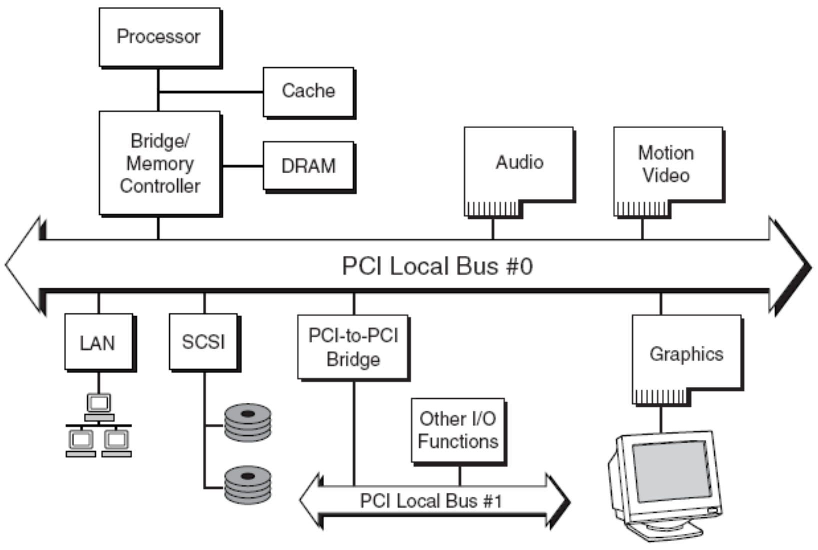


Figure3. PCI System Architecture

The basic PCI transfer is a burst. This means that all memory space and I/O space accesses occur in burst mode; a single transfer is considered a "burst" terminated after a single data phase. Addresses and data use the same 32-bit, multiplexed, address/data bus. The first clock is used to transfer the address and bus command code. The next clock begins one or more data transfers, during which either the master, or the target, may insert wait cycles.

PCI supports *posting*. A posted transaction completes on the originating bus before it completes on the target bus. For example, the CPU may write data at high speed into a buffer in a CPU-to-PCI bridge. In this case the CPU bus is the originating bus and PCI is the target bus. The bridge transfers data to the target (PCI bus) as long as the buffer is not empty, and asserts a not ready signal when the buffer becomes empty. In the other direction, a device may post data on the PCI bus, to be buffered in the bridge, and transferred from there to the CPU via the CPU bus. If the buffer becomes temporarily full, the bridge de-asserts the target ready signal.

In a read transaction, a turnaround cycle is required to avoid contention when the master stops driving the address and the target to begin driving the data on the multiplexed address/data bus. This is not necessary in a write transaction, when the master drives both the address and data lines. A turnaround cycle is required, however, for all signals that may be driven by more than one PCI unit. Also, an idle clock cycle is normally required between two transactions, but there are two kinds of back-to-back transactions in which this idle cycle may be eliminated. In both cases the first transaction must be a write, so that no turnaround cycle is needed, the master drives the data at the end of the first transaction, and the address at the beginning of the second transaction. The first kind of back-to-back occurs when the second transaction has the same target as the first one. Every PCI target device must support this kind of back-to-back transaction. The second kind of back-to-back occurs when the target of the second transaction is different than the target of the first one, and the second target has the Fast Back-to-Back Capable bit in the status register set to one, indicating that it supports this kind of back-to-back.

For arbitration, PCI provides a pair of request and grant signals for each PCI unit, and defines a central arbiter whose task is to receive and grant requests, but leaves to the designer the choice of a specific arbitration algorithm. PCI also supports *bus parking*, allowing a master to remain bus owner as long as no other device requests the bus. The default master becomes bus owner when the bus is idle. The arbiter can select any master to be the default owner.

PCI provides a set of configuration registers collectively referred to as "configuration space." By using configuration registers, software may install and configure devices without manual switches and without user intervention. Unlike the ISA architecture, devices are re-locatable - not constrained to a specific PCI slot. Regardless of the PCI slot in which the device is located, software may bind a device to the interrupt required by the PC architecture. Each device must implement a set of three registers that uniquely identify the device: Vendor ID (allocated by the PCI SIG), Device ID (allocated by the vendor), and Revision ID. The Class Code register identifies a programming interface (SCSI controller interface, for example), or a register-level interface (ISA DMA controller, for example). As a final example, the Device Control field specifies whether the device responds to I/O space accesses, or memory space accesses, or both, and whether the device can act as a PCI bus master. At power-up, device independent software determines what devices are present, and how much address space each device requires. The boot software then relocates PCI devices in the address space using a set of base address registers

2.3 Introduction to PCI Bridge architecture

A PCI-to-PCI bridge provides a connection path between two independent PCI buses. The primary function of the bridge is to allow transactions to occur between a master on one PCI bus and a target on the other PCI bus. PCI-to-PCI bridges provide system and expansion board designers the ability to overcome electrical loading limits by creating hierarchical PCI buses.

Figure4 illustrates two typical applications for a bridge. The first application is the use of a bridge to create a second PCI bus segment to which additional PCI connectors are added. This bus segment is labeled in the figure as PCI Bus 1. In this example, the primary interface of bridge 1 is connected to PCI bus 0 while its secondary interface is connected to PCI bus 1. The second application example is the use of a bridge to create a PCI bus segment on an expansion board that allows multiple PCI devices to reside on a single expansion board. In this example, the primary interface of bridge 2 is connected to PCI bus 1 and its secondary interface is connected to PCI bus 2. Note that the number assigned to the bridge corresponds to the number of the bus segment spawned by the bridge. In this example, the host bridge is considered to be bridge number 0 and spawns PCI bus segment 0.

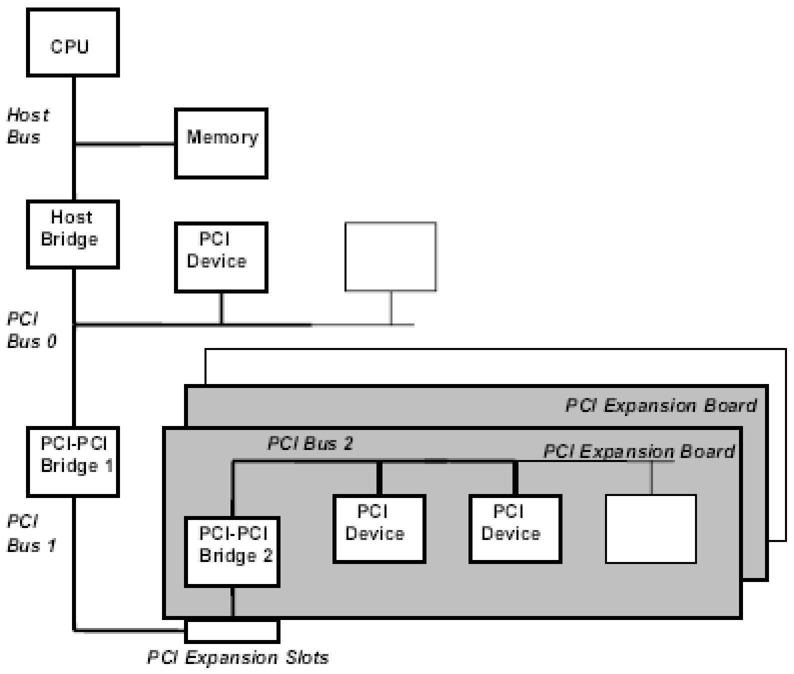


Figure4. Typical PCI Bridge Application

A bridge allows transactions between a master on one PCI interface and a target on the other interface as illustrated in Figure5. The target interface on one bus is connected to the master interface on the other bus. The blocks between the data path of the primary and secondary interfaces provide any necessary transaction address and data buffering. The target block connected to the primary PCI interface must support PCI configuration space. The bridge basically consists of four state machines¾two masters and two targets. Each of the master and target interface state machines must adhere to the requirements of the PCI Local Bus Specification.

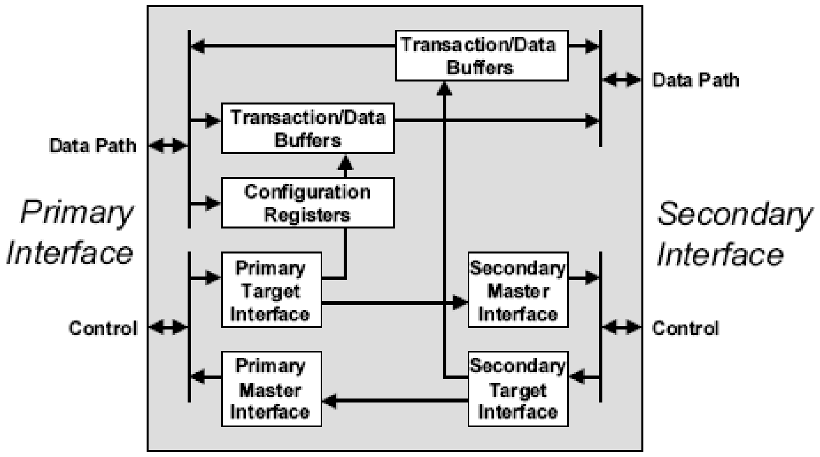


Figure5. PCI Bridge Block Diagram

The PCI Local Bus Specification requires all devices, including a PCI-to-PCI bridge, to implement a 256-byte configuration register address space. The first 64 bytes in each device's PCI Configuration Space must adhere to a standard configuration header format. The remaining 192 bytes of the Configuration Space may be used for additional capabilities as defined by the Capabilities Pointer or for device-specific purposes.

The 64-byte header format for a bridge is defined in Figure6. The first 16 bytes of the bridge header implement the common format for all devices as required by the PCI Local Bus Specification. The next 48 bytes of the device's Configuration Space are Header Type specific. A Header Type value of 1 indicates that the device follows the bridge register layout, which is defined in this specification. The following sections define the basic behavior of configuration registers and how reset affects them. Then a brief review of the common registers is presented, followed by a detailed specification of the bridge specific registers.

Presently, The PCI Bridge Architecture defines two kinds of PCI-to-PCI Bridge (P2PB). The first one is transparent P2PB by which all devices work under synchronous clock, the other is un-transparent P2PB by which all devices can work under different clock cycle. In our simulator, we just simply simulate transparent P2PB situation.

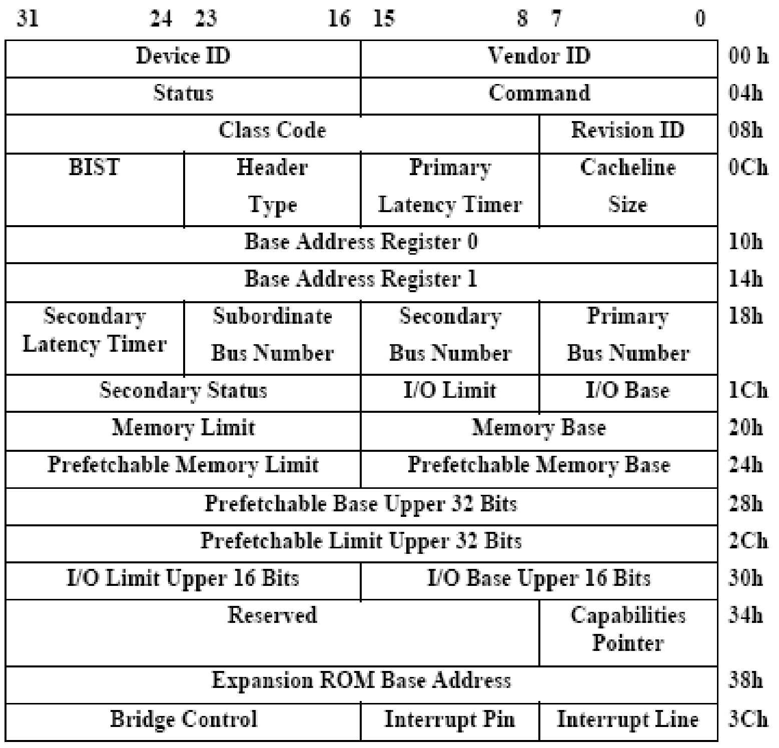
3. Simple introduction to PCI-X Bus

3.1 PCI-X 1.0/2.0

PCI-X (*Peripheral Component Interconnect Extended*) is a computer bus and expansion card standard designed to supersede PCI. It is essentially a faster version of PCI, running at twice the speed, and is otherwise similar in physical implementation and basic design. PCI-X basically uses the same PCI architecture, the same base protocols, the same BIOS, the same connector, the same driver models as PCI. Presently, PCI-X has itself been replaced in modern designs by the similar-sounding PCI Express, which features a very different logical design. Figure7 shows us

PCI-X different modes and speeds.

PCI-X was developed jointly by IBM, HP, and Compaq. PCI-X is a revision to the PCI standard that doubles the clock speed (from 66 MHz to 133 MHz) and hence the amount of data exchanged between the computer processor and peripherals. Standard PCI supports up to 64-bit at 66 MHz (though anything above 32-bit at 33 MHz is only seen in high-end systems) and additional bus standards move 32 bits at 66 MHz or 64 bits at 133 MHz. The theoretical maximum amount of data exchanged between the processor and peripherals with PCI-X is 1.06 GB/s, compared to 532 MB/s with standard PCI. PCI-X is generally backward compatible with PCI, meaning that, a PCI-X card can be installed in a PCI slot provided it has the correct voltage keying for the slot and (if inserting in a 32 bit slot) nothing obstructs the overhanging part of the edge connector. PCI and PCI-X cards can be intermixed on a PCI-X bus, but the speed will be limited to the speed of the slowest card (for this reason and the voltage compatibility issue most systems with PCI-X will have a normal PCI bus as well) . PCI-X improves the fault tolerance of PCI allowing, for example, faulty cards to be reinitialized or taken offline. Figure8 shows us that PCI-X is more efficient protocol than PCI



31 24	23 16	15 8	7 0	
Device ID		Vendor ID		00 h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Primary Latency Timer	Cacheline Size	0Ch
Base Address Register 0				10h
Base Address Register 1				14h
Secondary Latency Timer	Subordinate Bus Number	Secondary Bus Number	Primary Bus Number	18h
Secondary Status		I/O Limit	I/O Base	1Ch
Memory Limit		Memory Base		20h
Prefetchable Memory Limit		Prefetchable Memory Base		24h
Prefetchable Base Upper 32 Bits				28h
Prefetchable Limit Upper 32 Bits				2Ch
I/O Limit Upper 16 Bits		I/O Base Upper 16 Bits		30h
Reserved			Capabilities Pointer	34h
Expansion ROM Base Address				38h
Bridge Control		Interrupt Pin	Interrupt Line	3Ch

Figure6. PCI-to-PCI Bridge Configuration Register

IBM, HP, and Compaq designed PCI-X for servers to increase performance for high bandwidth devices such as Gigabit Ethernet, Fibre Channel and Ultra3 SCSI cards, and to allow processors to be interconnected in clusters. Figure9 shows us networking is moving to 10 Gigabit technologies. Compaq, IBM, and HP submitted PCI-X to the PCI Special Interest Group (Special Interest Group of the Association for Computing Machinery) in 1998. PCI SIG approved PCI-X, and it is now an open standard that can be adapted and used by all computer developers. PCI SIG controls technical support, training and compliance testing for PCI-X. IBM, Intel, Microelectronics and

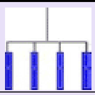
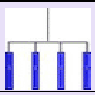
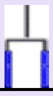
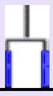
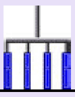
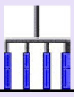
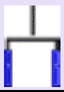
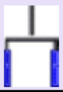
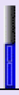
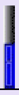
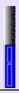
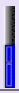
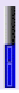
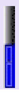
		64-Bit		32-Bit		16-Bit
Mode	$V_{I/O}$	Slots	MB	Slots	MB	
PCI 33	5V/3.3V		266		133	N/A
PCI 66*	3.3V		533		266	N/A
PCI-X 66	3.3V		533		266	N/A
PCI-X 133 (operating at 100 MHz)	3.3V		800		400	N/A
PCI-X 133	3.3V		1066		533	N/A
PCI-X 266	1.5V		2133		1066	533
PCI-X 533	1.5V		4266		2133	1066

Figure7. PCI/PCI-X Modes and Speeds

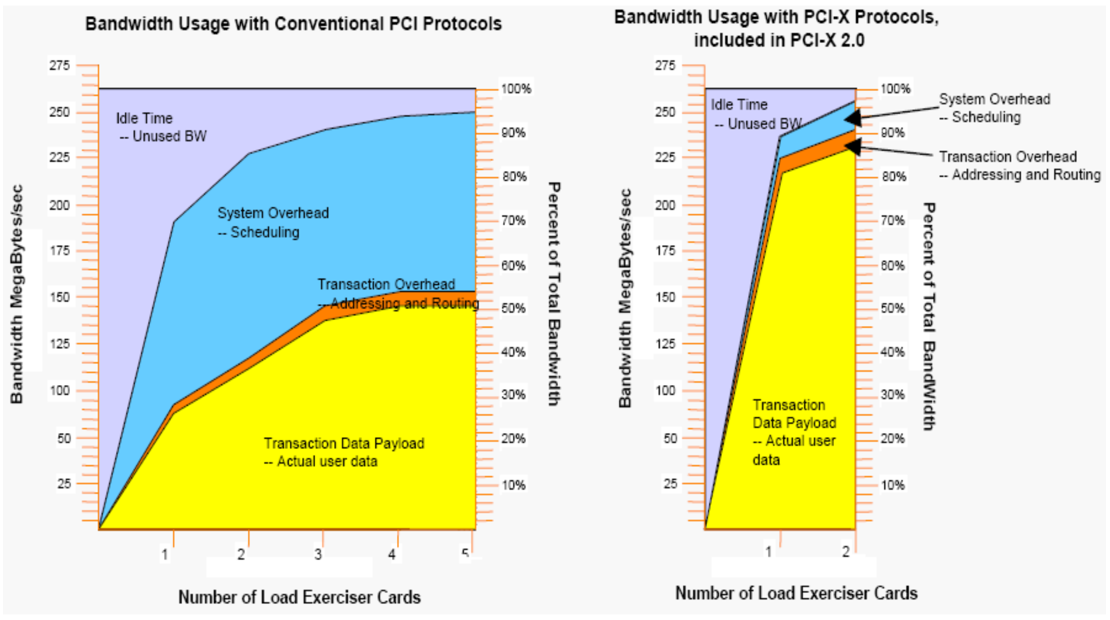


Figure8. PCI-X is more efficient protocol than PCI

Mylex plan to develop chipsets to support PCI-X. 3Com and Adaptec intended to develop PCI-X peripherals. Figure10 shows us PCI-X 2.0 can provide enough bandwidth to support all applications.

In 2003 PCI SIG ratified PCI-X 2.0 which adds 266 MHz and 533 MHz variants. These variants give roughly 2.15 GB/s and 4.3 GB/s throughput, respectively. PCI-X 2.0 makes additional protocol revisions that are designed to help system reliability and add error correction ECC to the bus to avoid resends. To deal with one of the most common complaints of the PCI-X form factor, the 184 pin connector, 16-bit ports were developed to allow PCI-X to be used in devices with tight space constraints. Similar to PCI-Express, PtoP functions were added to allow for devices on the bus to talk to each other without burdening the CPU or bus controller.

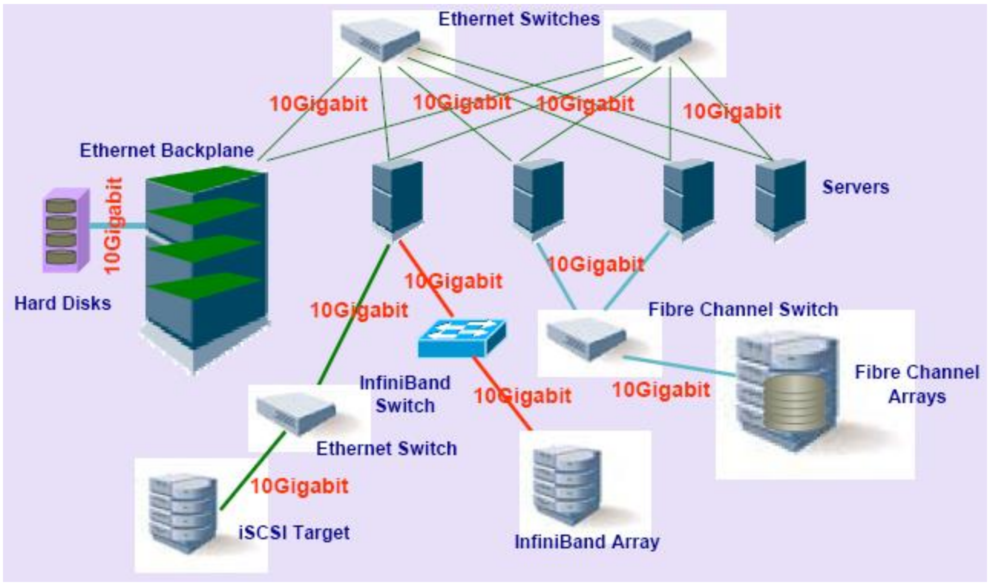


Figure9. Networking is moving to 10 Gigabit technologies

Despite the various theoretical advantages of PCI-X 2.0 and its backward compatibility with PCI-X and PCI devices, it has not been implemented on a large scale (as of 2006). This lack of implementation is primarily because hardware vendors have chosen to integrate PCI-Express instead. Currently, the PCI-X workgroup is developing PCI-X 1066 which will double PCI-X 533 bandwidths, maintain backward compatibility, and include new hardware/software extensions. Figure11 shows the PCI-X roadmap.

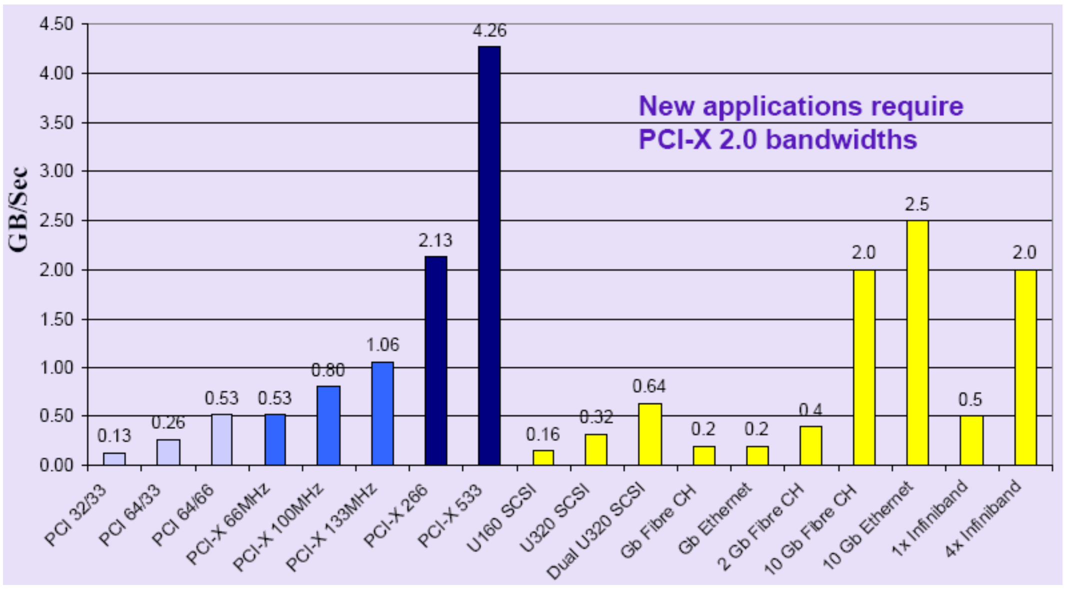


Figure10. PCI-X 2.0 can provide enough bandwidth to support all applications

3.2 Difference between PCI-X and PCI-E

PCI-X is often confused with PCI-Express, commonly abbreviated as PCI-E or PCIe. While they are both high-speed computer buses for internal peripherals, they differ in many ways. The first is that PCI-X is a parallel interface that is directly backward compatible with all but the oldest (5 volt) PCI devices. PCIe is a serial bus that offers no compatibility with older buses. In the future PCI-X and PCI buses may run off a PCIe bridge, similar to the way ISA buses ran off PCI buses in some computers. This should not be confused with compatibility. PCIe also matches PCI-X and even PCI-X 2.0 in maximum bandwidth. PCIe x1 offers 250 MB/s in both directions, and currently supports up to an x32 standard at 8 GB/s.

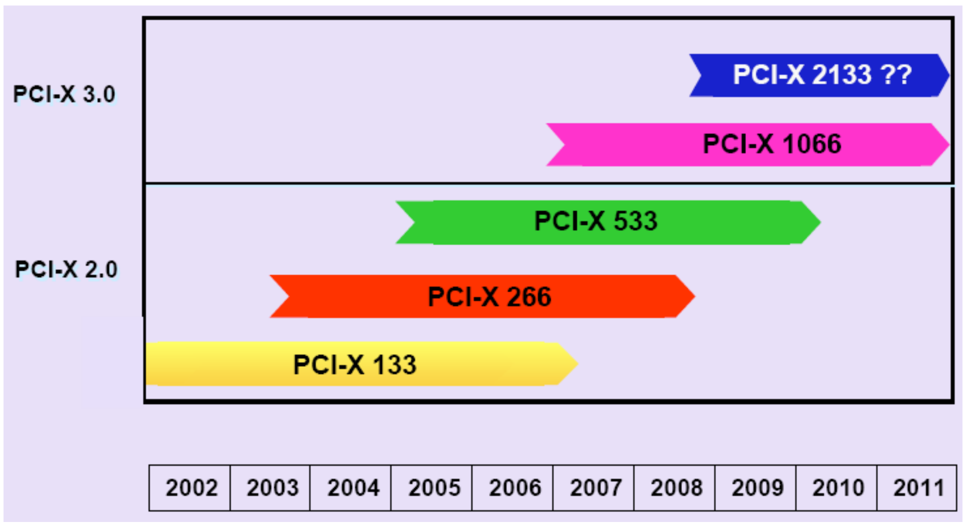


Figure11. PCI-X roadmap

PCI-X has a number of technological and economical disadvantages to PCI-Express. The 64-bit parallel interface requires inherently difficult trace routing, because as with all parallel interfaces, the signals from the bus must arrive simultaneously or within a very short window, and noise from adjacent slots may cause interference. The serial interface of PCIe suffers fewer such problems and therefore requires less complex and less expensive designs. PCI-X buses, like PCI, are half-duplex bidirectional whereas PCIe buses are full-duplex bidirectional. PCI-X buses run only as fast as the slowest device; PCIe devices are able to independently negotiate the bus speed

4. Simple introduction to PCI-E Bus

PCI Express, officially abbreviated as PCI-E or PCIe, is a computer expansion card interface format. It was designed as a much faster interface to replace PCI, PCI-X (abbreviated from PCI eXtended) for interface cards as well as AGP interfaces for graphics cards. PCIe is based around serial links called lanes. The PCIe 1.1 specification supports x1 (pronounced "by one"), x2, x4, x8, x16, and x32 lanes. In each lane, the most common version PCIe 1.1 carries 250 MB/s in each direction. Every lane of the PCIe is a dual simplex link; simultaneously receiving and transmitting. The PCIe 1.1 bus runs at 2.5 GHz. Since an explicit clock is not used, rather the clock is recovered from the data stream a special encoding called 8b/10b is used. This coding ensures that there are a sufficient number of transitions with a single character (of 10 bits) to properly and reliably recover the clock. The astute reader will notice that there are 4 times more combinations than characters used. Some of these additional characters are either discarded due to an insufficient number of edges within the 10 bit packet to extract the clock. Others are used to encode error commands. Some may be used to provide "DC balancing" so that the wire doesn't acquire an electrical charge. The remainder are simply not used. Therefore, each lane transmits 250 MB/s. The most number of lanes supported is x32, so 250MB/s x 32 x 2 (bi-directionality) is 16GB/s for a theoretical maximum transfer rate.

PCI-E uses fabric topology. A fabric is composed of point-to-point Links that interconnect a set of components – an example fabric topology is shown in Figure12. This figure illustrates a single fabric instance referred to as a hierarchy – composed of a Root Complex (RC), multiple Endpoints (I/O devices), a Switch, and a PCI Express-PCI Bridge, all interconnected via PCI Express Links. Each of the components of the topology is mapped in a single flat address space and can be accessed using PCI-like load/store accesses transaction semantics.

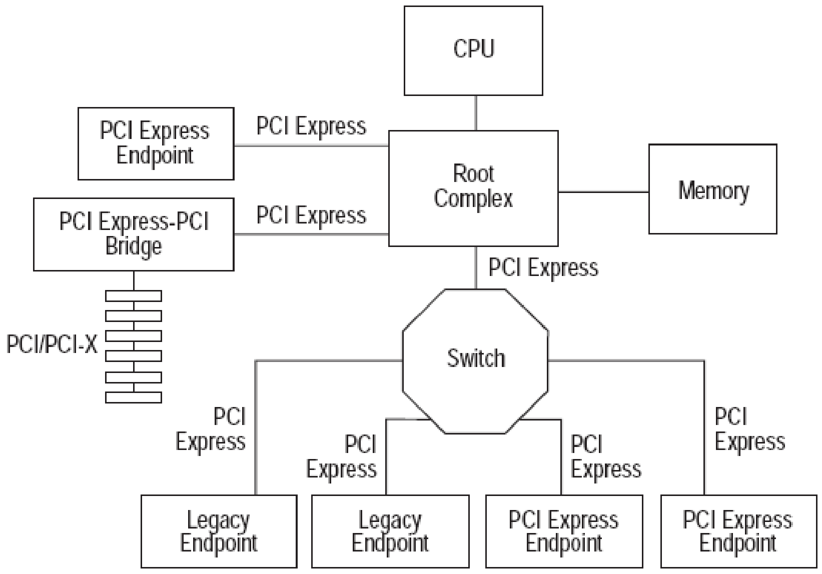


Figure12. PCI-E fabric topology

A Switch is defined as a logical assembly of multiple virtual PCI-to-PCI Bridge devices as illustrated in Figure13. All Switches are governed by the following base rules.

(1)Switches appear to configuration software as two or more logical PCI-to-PCI Bridges.

(2)A Switch forwards transactions using PCI Bridge mechanisms; e.g., address based routing.

(3)Except as noted in this document, a Switch must forward all types of Transaction Layer Packets between any set of Ports.

(4)Locked Requests must be supported as specified in Section 6.5. Switches are not required to support Downstream Ports as initiating Ports for Locked requests.

(5)Each enabled Switch Port must comply with the flow control specification within this document.

(6)A Switch is not allowed to split a packet into smaller packets, e.g., a single packet with a 256-byte payload must not be divided into two packets of 128 bytes payload each.

(7)Arbitration between Ingress Ports (inbound Link) of a Switch may be implemented using round robin or weighted round robin when contention occurs on the same Virtual Channel. This is described in more detail later within the specification.

(8)Endpoint devices (represented by Type 00h Configuration Space headers) may not appear to configuration software on the switch's internal bus as peers of the virtual PCI-to-PCI Bridges representing the Switch Downstream Ports.

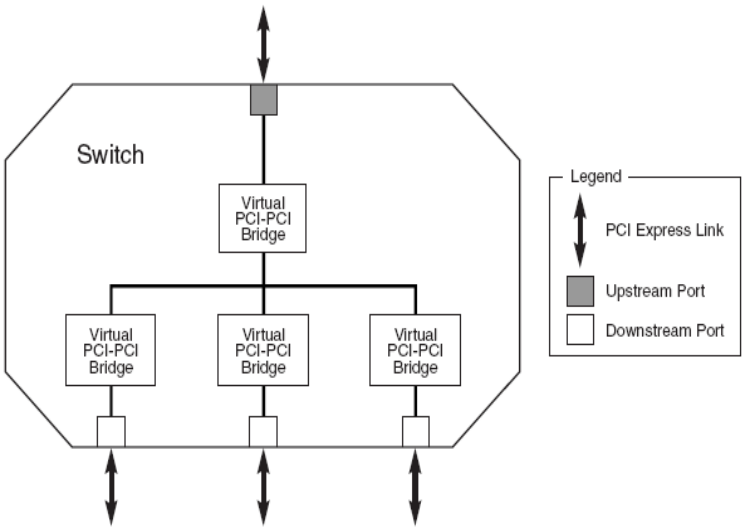


Figure13. Logical Block Diagram of a Switch

4.1 PCI-E Bus hardware protocol

PCI Express is a layered protocol, consisting of a Transaction Layer, a Data Link Layer, and a Physical Layer. Each of these layers is divided into two sections: one that processes outbound (to be transmitted) information and one that processes inbound (received) information, as shown in Figure14.The Physical Layer is further divided into a logical sublayer and an electrical sublayer. The logical sublayer is frequently further divided into a Physical Coding Sublayer (PCS) and a Media Access Control (MAC) sublayer (terms borrowed from the IEEE 802 model of networking protocol).

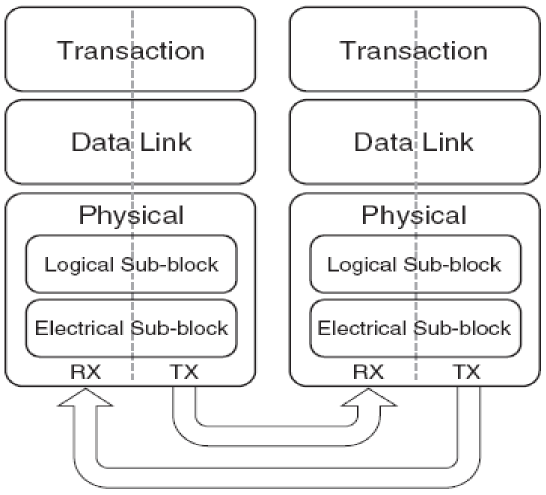


Figure14. High level layering diagram

PCI Express uses packets to communicate information between components. Packets are formed in the Transaction and Data Link Layers to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the other layers, they are extended with additional information necessary to handle packets at those layers. At the receiving side the reverse process occurs and packets get transformed from their Physical Layer representation to the Data Link Layer representation and finally (for Transaction Layer Packets) to the form that can be processed by the Transaction Layer of the receiving device. Figure15 shows the conceptual flow of transaction level packet information through the layers.

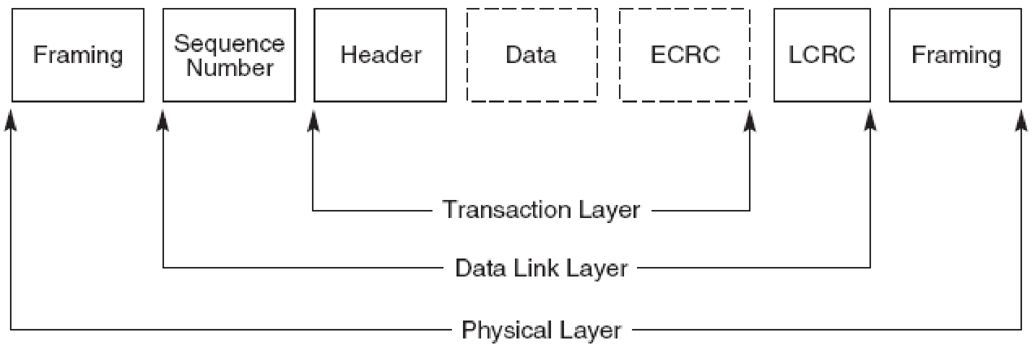


Figure15. PCI-E Packet Flow Through the Layers

The PCI-E physical Layer interface is known by the acronym PIPE which stands for "Physical Interface for PCI Express". At the electrical level, each lane utilizes two unidirectional Current Mode Logic (CML) pairs at 2.5 Gbit/s. Transmit and receive are separate differential pairs, for a total of 4 data wires per lane.

A connection between any two PCIe devices is known as a "link", and is built up from a collection of 1 or more lanes. All devices must minimally support single-lane (x1) link. Devices may optionally support wider links composed of 2, 4, 8, 12, 16, or 32 lanes. This allows for very good compatibility in two ways: A PCIe card will physically fit (and work correctly) in any slot that is at least as large as it is (e.g. an x1 sized card will work in any sized slot), and a slot of a large physical size (e.g. x16) can be wired electrically with fewer lanes (e.g. x1 or x8) as long as it provides the power and ground connections required by the larger physical slot size. In both cases, PCIe will negotiate the highest mutually supported number of lanes. It is not possible to place a physically larger PCIe card (e.g. a 16x sized card) into a smaller slot, even though the two would be signal-compatible if it were possible.

PCIe sends all control messages, including interrupts, over the same links used for data. The serial protocol can never be blocked, so latency is still comparable to PCI, which has dedicated interrupt lines.Data transmitted on multiple-lane links is interleaved, meaning that each successive byte is sent down successive lanes. The PCIe specification refers to this interleaving as "data striping." While requiring significant hardware complexity to synchronize (or deskew) the incoming striped data, striping can significantly increase the throughput of the link. Due to padding requirements, striping may not necessarily reduce the latency of small data packets on a link.

As with all high data rate serial transmission protocols, clocking information must be embedded in the signal. At the physical level, PCI Express utilizes the very common 8B/10B encoding scheme to ensure that strings of consecutive ones or consecutive zeros are limited in length. This is necessary to prevent the receiver from losing track of where the bit edges are. In this coding scheme every 8 (uncoded) payload bits of data are replaced with 10 (encoded) bits of transmit data, consuming an extra 25% of the overall electrical bandwidth.

Many other protocols (such as SONET) use a different form of encoding known as "scrambling" to embed clock information into data streams. The PCI Express specification also defines a scrambling algorithm, but it is used to reduce EMI (Electromagnetic interference) by preventing repeating data patterns in the transmitted data stream.

First-generation PCIe is constrained to a single signaling rate of 2.5 Gbit/s. The PCI Special Interest Group (the industry organization that maintains and develops the various PCI standards) plans future versions adding signaling rates of 5 and 10 Gbit/s.

The PCI-E Data Link Layer implements the sequencing of the Transaction Layer Packets (TLPs) that are generated by the Transaction Layer, data protection via a 32-bit cyclic redundancy check code (CRC, known in this context as LCRC) and an acknowledgment protocol (ACK and NAK signaling). TLPs that pass an LCRC check and a sequence number check result in an acknowledgment, or ACK, while those that fail these checks result in a negative acknowledgment, or NAK. TLPs that result in a NAK, or timeouts that occur while waiting for an ACK, result in the TLPs being replayed from a special buffer in the transmit data path of the Data Link Layer. This guarantees delivery of TLPs in spite of electrical noise, barring any malfunction of the device or transmission medium. ACK and NAK signals are communicated via a low-level packet known as a data link layer packet, or DLLP. DLLPs are also used to communicate flow control information between the transaction layers of two connected devices, as well as some power management functions.

The PCI-E transaction layer implements split transactions (transactions with request and response separated by time), allowing the link to carry other traffic while the target device gathers data for the response.

PCI Express utilizes credit-based flow control. In this scheme, a device advertises an initial amount of credit for each of the receive buffers in its Transaction Layer. The device at the opposite end of the link, when sending transactions to this device, will count the number of credits consumed by each TLP from its account. The sending device may only transmit a TLP when doing so does not result in its consumed credit count exceeding its credit limit. When the receiving device finishes processing the TLP from its buffer, it signals a return of credits to the sending device, which then increases the credit limit by the restored amount. The credit counters are modular counters, and the comparison of consumed credits to credit limit requires modular arithmetic. The advantage of this scheme (compared to other methods such as wait states or handshake-based transfer protocols) is that the latency of credit return does not affect performance, provided that the credit limit is not encountered. This assumption is generally met if each device is designed with adequate buffer sizes.

First-generation PCIe is often quoted to support a data rate of 250 MB/s in each direction, per lane. This figure is a calculation from the physical signaling rate (2.5 Gbaud) divided by the encoding overhead (10bits/byte.) This means a 16 lane (x16) PCIe card would then be theoretically capable of 250 * 16 = 4 GB/s in each direction. While this is correct in terms of data bytes, more meaningful calculations will be based on the usable data payload rate, which depends on the profile of the traffic, which is a function of the high-level (software) application and intermediate protocol levels. Like other high data rate serial interconnect systems, PCIe has a protocol and processing overhead due to the additional transfer robustness (CRC and Acknowledgments). Long continuous unidirectional transfers (such as those typical in high-performance storage controllers) can approach >95% of PCIe's raw (lane) data rate. These transfers also benefit the most from increased number of lanes (x2, x4, etc.) But in more typical applications (such as a USB or Ethernet controller), the traffic profile is characterized as short data packets with frequent enforced acknowledgments [citation needed]. This type of traffic reduces the efficiency of the link, due to overhead from packet parsing and forced interrupts (either in the device's host interface or the PC's CPU.) This loss of efficiency is not particular to PCIe.

4.2 PCI-E 2.0

PCI-SIG announced the availability of the PCI Express Base 2.0 specification on 15 January 2007.[2] PCIe 2.0 doubles the bus standard's bandwidth from 2.5 Gbit/s to 5 Gbit/s, meaning a x32 connector can transfer data at up to 16 GB/s in each direction. PCIe 2.0 is still compatible with PCIe 1.1, so older cards will still be able to work in machines with this new version.

5. PCI Multi-Bus Simulator

In modern computer system, the CPU and memory communicate used in every fast system bus, for example, the network cards use DMA transfers over the PCI bus to place incoming data in buffers and in main memory or to transmit data from buffers in the main memory. In this paper, we just focus on the general PCI buses’ operation without consideration of PCI-X/PCI-E situation. We simulate not only the single PCI bus operation, but PCI multi bus simple operation by PCI-to-PCI Bridge to evaluate the simulator’s performance.

Most of modern computer systems are built up using a transaction-based I/O buses to transfer data between devices and memory or directly from one device to another. During the period of designing this PCI Multi-bus simulation system, it is rather import to check that PCI bus and PCI-to-PCI Bridge (P2PB) are able to transfer data to and from the attached devices at the rate required for correct operation of the device. This section describes a simulator for the evaluation of the use of PCI bus and P2PB. This simulator is based on the PCI Local Bus Specification and P2PB Architecture Specification.

The PCI bus is controlled by complex protocol which permits several different forms of transaction as follows:

(1) Data Transfer transaction focuses on transferring data between PCI Bus devices or between PCI Bus devices and main memory.

(2) Configuration transaction focused on data transferred to configure relative devices or to determine the properties of devices.
(3) Special Cycle transaction focuses on sending messages/information to several destinations.
(4) Interrupt Acknowledge transaction focuses on the relative operation during the process of interrupt acknowledge.

Our simple simulator just deals with regular data transfer transactions on the single PCI bus or multiple PCI bus by P2PB. It can simulate the operation of both the 32bit/64bit versions of PCI bus at synchronous clock frequencies: 33MHz, 66MHz, or 133MHz. On the other hand, this multi-Bus simulator can simply simulate 6~8 Buses' transactions.

5.1 Data Transfer Transactions-Read/Write Transaction

From hardware perspective, the local bus specification describes the principles of Read transactions and Write transactions.

The timing diagrams in this section show the relationship of significant signals involved in 32-bit transactions. When a signal is drawn as a solid line, it is actively being driven by the current master or target. When a signal is drawn as a dashed line, no agent is actively driving it. However, it may still be assumed to contain a stable value if the dashed line is at the high rail. Tri-stated signals are indicated to have indeterminate values when the dashed line is between the two rails (e.g., AD or C/BE# lines). When a solid line becomes a dotted line, it indicates the signal was actively driven and now is tri-stated. When a solid line makes a low to high transition and then becomes a dotted line, it indicates the signal was actively driven high to pre charge the bus and then tri-stated.

Figure16 illustrates a read transaction and starts with an address phase which occurs when FRAME# is asserted for the first time and occurs on clock 2. During the address phase, AD[31::00] contain a valid address and C/BE[3::0]# contain a valid bus command.

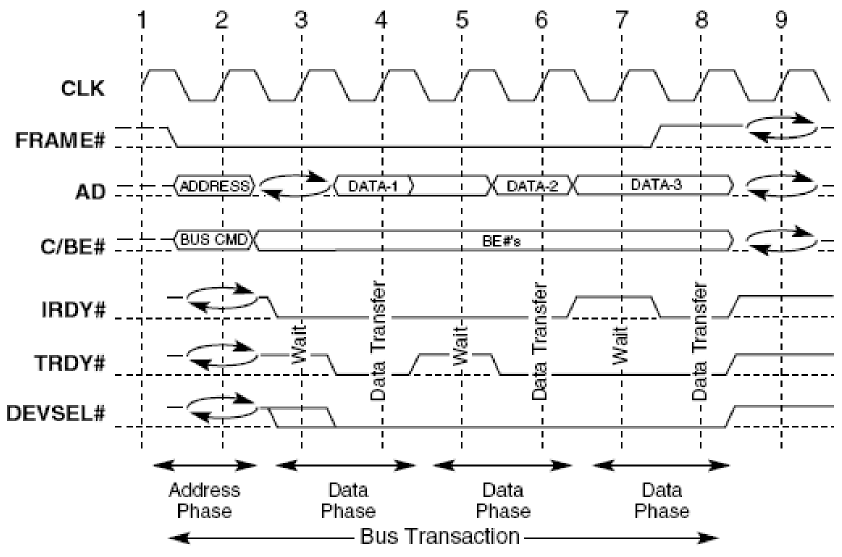


Figure16. PCI bus Basic Read Operation

The first clock of the first data phase is clock 3. During the data phase, C/BE# indicate which byte lanes are involved in the current data phase. A data phase may consist of wait cycles and a data transfer. The C/BE# output buffers must remain enabled (for both read and writes) from the first clock of the data phase through the end of the transaction. This ensures C/BE# are not left floating for long intervals. The C/BE# lines contain valid byte enable information during the entire data phase independent of the state of IRDY#. The C/BE# lines contain the byte enable information for data phase N+1 on the clock following the completion of the data phase N. This is not shown in Figure16 because a burst read transaction typically has all byte enables asserted; however, it is shown in Figure17. Notice on clock 5 in Figure17, the master inserted a wait state by de-asserting IRDY#. However, the byte enables for data phase 3 are valid on clock 5 and remain valid until the data phase completes on clock 8.

The first data phase on a read transaction requires a turnaround-cycle (enforced by the target via TRDY#). In this case, the address is valid on clock 2 and then the master stops driving AD. The earliest the target can provide valid data is clock 4. The target must drive the AD lines following the turnaround cycle when DEVSEL# is asserted. Once enabled, the output buffers must stay enabled through the end of the transaction. (This ensures that the AD lines are not left floating for long intervals.)

One way for a data phase to complete is when data is transferred, which occurs when both IRDY# and TRDY# are asserted on the same rising clock edge. (TRDY# cannot be driven until DEVSEL# is asserted.) When either IRDY# or TRDY# is de-asserted, a wait cycle is inserted and no data is transferred. As noted in Figure16, data is successfully transferred on clocks 4, 6, and 8 and wait cycles are inserted on clocks 3, 5, and 7. The first data phase completes in the minimum time for a read transaction. The second data phase is extended on clock 5 because TRDY# is de-asserted. The last data phase is extended because IRDY# was de-asserted on clock 7.

The master knows at clock 7 that the next data phase is the last. However, because the master is not ready to complete the last transfer (IRDY# is de-asserted on clock 7), FRAME# stays asserted. Only when IRDY# is asserted can FRAME# be de-asserted as occurs on clock 8, indicating to the target that this is the last data phase of the transaction.

Figure17 illustrates a write transaction. The transaction starts when FRAME# is asserted for the first time which occurs on clock 2. A write transaction is similar to a read transaction except no turnaround cycle is required following the address phase because the master provides both address and data. Data phases work the same for both read and write transactions.

In Figure17, the first and second data phases complete with zero wait cycles. However, the third data phase has three wait cycles inserted by the target. Notice both agents insert a wait cycle on clock 5. IRDY# must be asserted when FRAME# is de-asserted indicating the last data phase.

The data transfer was delayed by the master on clock 5 because IRDY# was de-asserted. The last data phase is signaled by the master on clock 6, but it does not complete until clock 8. Note:

Although this allowed the master to delay data, it did not allow the byte enables to be delayed.

Generally speaking, a transaction on the PCI bus involves an address phase, in which an address is transferred followed by a sequence of one or more data phases, in each of which a data item is transferred. Each transaction is initiated by a bus device acting as master and involves another participant known as the target. Data transfers may be oriented towards the master or towards the target, with the corresponding transactions respectively being denoted read and write transactions. The address specified at the start of the transaction is the address within the target of the first data item transferred.

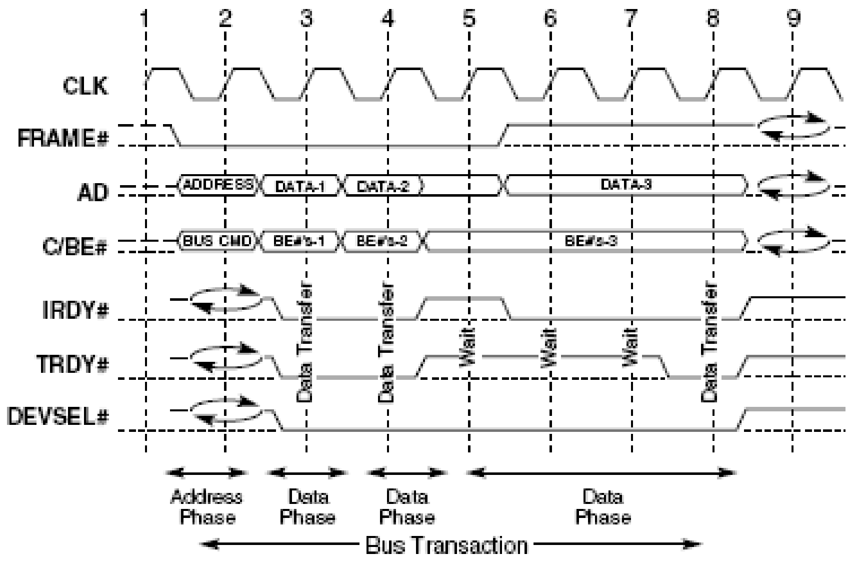


Figure17. PCI bus Basic Write Operation

The PCI bus may be implemented in a 32bit or 64 bit version. In the 32 bit version, an address is represented by 4 bytes, which are transmitted in a single bus clock cycle, and each data item can be from 0 to 4 bytes, which are likewise transmitted in a single bus clock cycle. In the 64 bit version, an address is represented by 8 bytes and each data item by from 0 to 8 bytes, which are transmitted in a single clock cycle. According to the PCI bus specification, the number of data bytes to be transferred in a given data phase is specified by byte enable bus signals presented immediately prior to the actual transfer of data and may vary from one data phase to the next within a burst transaction. The simulator makes the simplifying assumption that the first (N-1) data phases of an N data phase transaction involve transfer of data with the maximum width (4 bytes or 8 bytes) for the bus version concerned and that only the final transfer may involve fewer bytes, to allow for data transfers whose length is not an integral multiple of 4 or 8 bytes respectively.

The PCI bus specification allows the target to delay the transfer of a data item by inserting wait states, each lasting one clock cycle, in any data phase. In more detail, the first data phase of a transaction can be extended to at most 16 clock cycles and each subsequent data phase to at most 8 clock cycles. The simulator allows the user to specify a maximum number of wait states for each device. Note that these wait states are specified for the device as master, although strictly speaking they describe the behavior of the device's target in the system under consideration. The user can also specify whether the actual number of wait states used in each data phase is selected deterministically or stochastically. With deterministic selection, exactly wait states are inserted in every data phase. With stochastic selection, the simulator will for each data phase select a random whole number as the number of wait states to be inserted. These features make it possible to simulate the behavior of devices with slow response and of targets such as main memory, which may exhibit access contention.

5.2 PCI Bus device's early termination

As for hardware implementation, I here just give simple description and will not write too much about it. The PCI bus specification allows bus transactions to be terminated before the data transfers tackled by the master at the start of the transaction. Our simulator will terminate the data transfer transaction according to several situations mentioned below:

(1) New master wants to gain control right of the bus and the PCI bus arbitrator grants the bus to this new master. This situation is dealt with by the simulator according to the PCI bus specification for arbitration using rotating or fixed priority or time quantum scheme. The simulator implements the rules for dealing with the master's latency timer whose initial value is gave by the user. A single idle state lasting one clock cycle is always inserted on the bus between the current transaction and its next transaction. The simulator does not support fast back-to-back access without inserted idle states.

(2) Target device can not respond within the limit of 8 or 16 clock cycles set by the rules for insertion of wait states. The target must send disconnection information to the master which must terminate or end the transaction. The master may subsequently try to restart the transaction after a delay which is at least about two bus clock cycles. In our simulator, this situation is assumed not to occur.

(3) Target device can not respond to master device, not support a burst mode transaction requested by the master device, sends a STOP signal to terminate the current transaction, or determines that the address to be used in the next data phase lies outside the range of addresses available to it. Our simulator will not simulate this situation and also not simulate errors such as data errors, address parity errors and any other system errors.

5.3 Relative parameters of Device, Data Generation, Input, Output, and Bus

In this simulator, every device can work as a PCI Bus master which can initiate either read or write data transfer transaction by using specified block data. A device is defined as PCI Bus's logical device to work according to the PCI Bus specification. If a physical device which can both read and write is to be simulated, or if the device uses several different block sizes or buffer size, then two or more PCI Bus devices need to be defined. Every device has several basic characters mentioned in table 1.

Name character	Function
Priority	Positive integer giving the priority of the device during bus arbitration by using fixed device's priority. For rotating and time quantum priority arbitration, the value is ignored.
Transaction type	There are two kind of transactions: Read, Write
Buffer size	Positive integer giving the size in bytes of the buffer which the device as

	master will try to fill or empty during each transaction. If no new devices are active during the transaction, all the data will be transferred in a single burst. If pre-emption occurs before the end of the transaction, the rest of the buffer will be transferred in on or more following secondary transactions.
Maximum data rate	The maximum data rate is that the device can reach. If the contents of the buffer have not been completely transferred by the time the next buffer full is generated, the user is warned that a data overrun situation has occurred.
Maximum wait states	The maximum number of wait states which can be inserted in each data phase by the device's target in the system.
Wait state generator process	Stochastic if the number of wait states is selected randomly from the interval [0...n_w] for each data transfer phase, or Deterministic if the number of wait states is exactly n_w in each data transfer phase. ($0 <= n_w <= 8$)
Latency timer	The initial value is used for the latency timer at the start of each new transaction of relative the device

Table1. Device parameters introduction

As for the input parameters, they are used to set general parameters of the simulation and general bus parameters. When the simulator source code runs, it will show us some dialogue for the setting parameters' input and we will typed in interactively. The relative input parameters shown in table 2.

Input parameters	Type content
Number of masters	A positive integer, the number of PCI Bus's logical devices.
Number of clock cycles in simulation	A positive integer, the length of the simulation clock cycles
Throughput plot required(y/n)	If yes, the simulator will plot 2D diagram with bus throughput and applied load parameters. (y/n responses are not case sensitive)
……	……
Data rate plot required(y/n)	If yes, the simulator will plot 3D diagram with bus utilization and time at all kinds of applied loads. (y/n responses are not case sensitive)
Histogram of transfer times plot required(y/n)	If yes, the simulator will plot 3D histogram of transfer time at all kinds of applied load. (y/n responses are not case sensitive)

Table2. relative input parameters

If a throughput plot is wanted, the user will input a name for the file to contain data for sending to gnuplot. If a data rate plot is wanted, the user will give a name for the file to contain data for sending to gnuplot, and for the size in clock cycles of the time slot to be used to sample the utilization of the bus. There are relative points appear in the gnuplot diagram for corresponding time slot. Choosing a smaller value will create a more accurate diagram in which the bus utilization changes with time, but it will create a big file to be generated. If a histogram of transfer times is wanted, the user will input a name for the file to contain data for sending to gnuplot and also for the number of histogram bins corresponding to the relative unit normalized transfer time, which is the time taken to transfer a buffer of data for corresponding device if no pre-emption appears and no wait states are inserted during the period of transfer.

As for general bus parameters, they are showed in table 3 as follows:

Name	content
Frequency (MHz)	PCI Bus clock frequency, f, has 3 options: 33, 64, or 133 MHz
Data size (Bytes)	Size of data objects has two options: 4 or 8 bytes
Arbitration (f, r, or t)	If f, bus arbitration works according to fixed priority If r, bus arbitration works with a rotating (round robin) arbitration scheme If t, bus arbitration works with time quantum or time slot scheme

Table3. General bus parameters

Output from the simulator consists of some standard information directed to the standard output stream stdout and a number of optional forms of output, as selected by the user during the initial dialogue. Table 4 shows us the relative standard output parameters.

Name	content
The amount of generated data	It is generated by all of the PCI Bus devices during the simulation period
The amount of transferred data	It is actually transferred on the PCI Bus during the period of simulation.
The average burst lengths	It is the average burst lengths of data transferred by each of PCI Bus device

Table4. Relative standard output parameters

In this section, I simply introduce some basic input and output information. As for others, I will not give more details because of the paper space.

5.4 Introduction to design and implementation of simulation software

Our simulator is developed by C/C++/G++ and top-down method. It works or executes under Redhat Linux platform. As for the software architecture, It is consists of three sections: head file which mainly focuses on the definition of parameters and declaration of functions, main program which shows us the definition of relative functions and main function, and plot program which call relative *gnuplot* commands to draw corresponding diagrams according to users' requirements.

5.4.1 Head file section

I here just give some simple introduction to these three sections. Firstly, as for the head file section, we call some regular C/C++ library file:

```
#include <stdlib.h>                              #include <unistd.h>
#include <stdio.h>                               #include <iostream>
#include <math.h>                                #include <cstdlib>
#include <ctype.h>                               ……
```

Then, define constant value and parameters (table 5 shows some of parameters):

Parameter name	content
#define IDLE 0 #define ADDRESS 1	These 4 constant values are used to implement data transfer transaction operation including addressing, inserting overload, and data transfer.

#define WAIT 2 #define DATA 3	
Double f;	PCI bus clock frequency (MHz)
Int width;	PCI bus data width (Bytes)
Int arb;	PCI bus arbitration scheme
Int nm;	Number of PCI bus masters
……	……
Int buff_size[];	Buffer size for master 1..n (Bytes)
Int read_write[];	R or W tranction type for master 1..n
Char rw_both[];	Both R/W transaction type for master 1..n
Int status_selection[]	Wait state selection for master 1..n
Int LT[];	Latency timer value
Int PB[]	PCI bridge selection
Int PBAbus1[]	Bus0 device accesses Bus1 devices
Int PBAbus0[]	Bus1 device accesses Bus0 devices
……	……
Long int sim_total;	Total simulation time (clock cycles)
Long int t;	Current t;
Long int slot_width;	Width of time slot for activity graph
Long int slot_current;	Time for end of current time slot
Long int slot_data	Amount of data transferred at start of slot
……	……
Int pre_empt;	It will be true, if a new PCI bus master has requested bus
Int throughput_plot;	If throughput file is required, it will be true
Int rate_plot;	If data rate plot is required, it will be true
Int verbose;	If users want detail of transfer, it will be true
Char tqvalue;	Time quantum scheme's temporary parameter

Table5. Define constant value and parameters

Finally, I declare some of relative functions and define relative Classes (table 6 as follows)

Class/Function name	content

Class device;	class device{
	public:
	void pparams_bus1(FILE *fp);
	void pparams_bus2(FILE *fp);
	void pparams_bus3(FILE *fp);
	……
	void pparams_bus0_bus1(FILE *fp);
	void pparams_bus1_bus0(FILE *fp);
	……
	~device(); };

Class arbitrator;	class arbitrator{ public:

	int arbitrate (int current); void timequatum (); ~arbitrator(); };
Class bus;	class bus{ public: void businit(); void PCI_Bridge_Bus1(); Void PCI_Bridge_Bus2(); Void PCI_Bridge_Bus3(); Void PCI_Bridge_Bus4(); ……. Void PCI_Bus0_acces_Bus(int v1, int v2) Void PCI_Bus1_acces_Bus0() Void PCI_Bus1_acces_Bus2() Void PCI_Bus1_acces_Bus(int v1,int v2) Void PCI_Bus2_acces_Bus(int v1,int v2) …… ~bus(); };
Class graph; ……	class graph{ public: void draw_graph(); ~graph(); }; ……

Table6. Define relative functions and Classes.

5.4.2 Implementation of data transfer transaction -- main function section

Under single bus situation, our multi-Bus simulator works according to the steps as follows:

(1) PCI bus initialization operation including setting the default bus frequency, bus width, bus arbitration parameter and so on.

(2) Users will enter relative value of parameters according to the screen mentions interactively. the relative parameters including number of master, number of clock cycles of simulation, number of points on applied load axis, which kind of plot wanted by users, bus frequency, bus width, bus arbitration and so on.

(3) Start to enter device's parameter value according to the prompts including number of master, priority of master, buffer size, transaction type(read, write, or both of them), device's max. data rate, maximum wait clock cycles, latency timer's clock cycles, etc.

(4) After input relative system parameters, the program executes a big loop which implements the whole data transfer transaction. This big loop is consists of four parts:

Part 1: Do some initialization before data transfer transaction start including:

→ storing device's data production rate to the corresponding array

→ storing mean interval between buffers arrival instants to the corresponding array

→ storing the time of arrival of first buffer full for each device to array

→ setting the remaining data in every device's buffer is zero to corresponding array
→ setting every device's burst length is zero to corresponding array
→ setting accumulated number of burst is zero to corresponding array
→ some of other initialization

Part 2: If a new buffer full of data is due for a device, checking that current buffer is empty, insetting data, and deciding instant when next buffer is due

Part 3: If there are one or more devices with data in their buffers, then select the device with highest priority to do data transaction operation according to the arbitration algorithm

Part 4: When the device with highest priority start to transfer data, we use switch() statement to implement the whole data transfer process including IDLE operation to do overhead operation before transferring data, ADDRESS operation to transfer address, WAIT operation to execute wait state action during data transfer, and DATA operation to transfer data item with general method and burst method

All these parts mentioned above used by every device, when they get bus control right.

(5) During the data transfer process, the system will output related data and result to corresponding file for plotting. On the other hand, we can see the real time output on the computer screen.

(6) Finally, at the end of the execution of software, the system will call GNUPLOT automatically and start the plot of throughput, data rate, and histogram according to users' requirement.

5.4.3 Implementation of arbitration scheme

About arbitration scheme, we try to use fix priority algorithm, rotating priority algorithm, and time quantum algorithm and red robin algorithm. Below is simple introduction to these three algorithms.

Fix priority algorithm: it is simple and just arbitrates several devices according to their fixing priority. The device with high priority device always gets the bus control right for data transfer operation. For example, device 1's priority is 2, device 2's priority is 3, and device 3's priority is 4, when they request to control bus simultaneously, the arbitration scheme will decide which one will get the bus control right according to their priority's value, the smaller the better.

Rotating priority algorithm: the arbitration scheme will not consider every device's priority and just according to a sequence to operate. For example, when device 1 finish its data transaction, device 2 request the data transfer, when arbitrator finds its buffer not empty, the arbitrator will give device 2 the bus control right. Or else, device 2 will have no right to control bus and arbitrator will check device 3 for data transaction operation.

Time quantum algorithm: the arbitrator will give every device the same time quantum to transfer their data. The time quantum is a constant which equals 16 clock cycles. If a device's operation time is smaller than time slice (or time quantum), it will finish data transaction smoothly. Or else, the arbitration scheme will stop it, push its relative parameters into a stack for protection when its operation does not finish in a fixed time slice and select next device to use bus for data transfer operation according to corresponding algorithm.

Red robin algorithm: this algorithm is almost the same as time quantum algorithm. On the other hand, it is a little bit simple than the time quantum algorithm and I did not use it in my source code, but I can add it according to the special requirement.

In our simulator, we mainly use former two arbitration algorithm. As for time quantum algorithm, we just use it do some simple compare testing as supplement.

5.4.4 Implementation of plotting graph

This simulator not only simulate the PCI bus data transfer transaction, but can collect relative data to draw three kinds of 2D/3D diagrams including throughput plot, data rate plot, and histogram plot. If the users want a file containing data for a throughput plot be produced, then an output file containing data suitable for gnuplot will be generated. This file shows us the details of simulation and of the devices attached to the bus. The actual data to be plotted consist of *(Generated data, Transmitted data)* pairs, *(d_g, d_t)*, is fit for plotting by the gnuplot plot command. Figure 18 shows a throughput plot as an example.

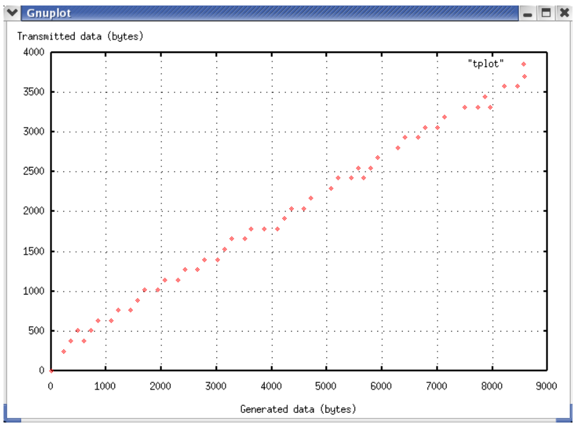


Figure18. Throughput plot

If users want a file containing data for a data rate plot be produced, then an output file containing data suitable for reading into gnuplot will be generated. The final data to be plotted consisted of *(Time, Load, PCI_Bus utilization)* triplets, *(t, l, u)*, is fit for plotting as a 3D plot by the GNUPLOT relative command. Figure19 shows us the example of 3D data rate plot.

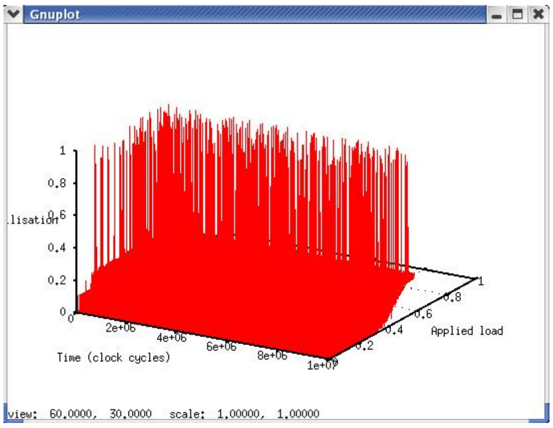


Figure19. 3D data rate plot

If users want a file containing data for a histogram produced, then an output file containing data suitable for reading into gnuplot will be generated. The relative data to be plotted is composed of *(Transfer time, Load, Samples)* triplets, (t,l,n), fit for plotting as a 3D plot via the gnuplot relative command. Figure20 shows us a 3D histogram plot. In this example, because there are several devices, the data file contains several samples columns, one for each device. When display the file, user must select the desired samples column by using gnuplot command as follow:

```
Gnuplot> load "name of file.gnp"
Gnuplot> splot "name of data file" using 1:2:4 notitle
```

In this example, the suing 1:2:4 parameter to the splot command specifies that time value are selected from column 1, load from column 2 and samples from column 4 of the file. Transfer times for several devices can be plotted on the same histogram by suing several file and using specification in the splot command. For example:

```
Gnuplot> load "name of file.gnp"
Gnuplot> splot "name of data file" using 1:2:3, "name of data file" using 1:2:6
```

Give the histograms for device 1 (column 3) and 4 (column 6).

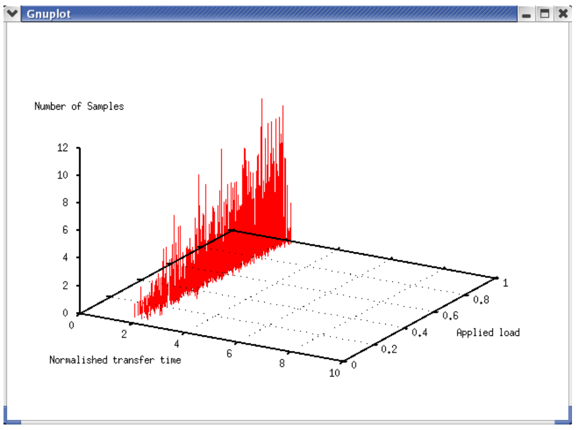


Figure20. 3D histogram plot

5.4.5 Implementation of multi PCI bus simulation

Peripheral Component Interconnect (PCI) is a stardard that describes how to connect the peripheral components of a system together. In order to design and implement our multi-Bus PCI simulator, we need to know not only the PCI loacal Bus specification, but PCI-PCI(P2P) Bridge specification. Because the PCI Buses and PCI-PCI Bridges are the glue connecting the system components together.

In figure21, we can see that the CPU is connected to PCI Bus0 and PCI Bridge connects the promary Bus to the secondary PCI Bus, PCI Bus1. In PCI Bridge specification, PCI Bus1 is described as being downstream of the PCI-PCI Bridge and PCI Bus0 is upstream of the Bridge. The SCSI and Ethernet devices are connected to the secondary PCI Bus. Physically the Bridge, secondary PCI Bus and two devices would be contained on the same PCI card. The PCI-ISA Bridge supports older, legacy ISA devices.

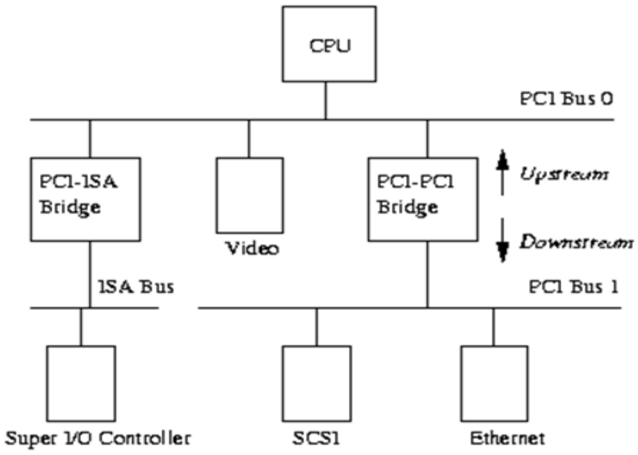


Figure21. Example PCI Based System

If we want to design and implement a PCI multi-Bus hardware product, we must spend lots of time thinking over PCI address space assignment, PCI configuration header, PCI I/O, and PCI memory address. On the other hand, we also deal with PCI-PCI Bridges' PCI configuration cylces, PCI Bus numbering, PCI I/O and PCI memory windows. *Under Linux system,* the kernel PCI data structures diagram looks like Figure 22. Each PCI device (including the P2P Bridges) is descirbed by a pci_dev data structure. Every PCI Bus is described by a pci_bus data structure. The result is a tree structure of PCI buses each of which has a number of child PCI devices attached to it. As a PCI bus can only be reached using a PCI-PCI Bridge (except the primary PCI bus, bus 0), each pci_bus contains a pointer to the PCI device (the PCI-PCI Bridge) that it is accessed through. That PCI device is a child of the the PCI Bus's parent PCI bus.

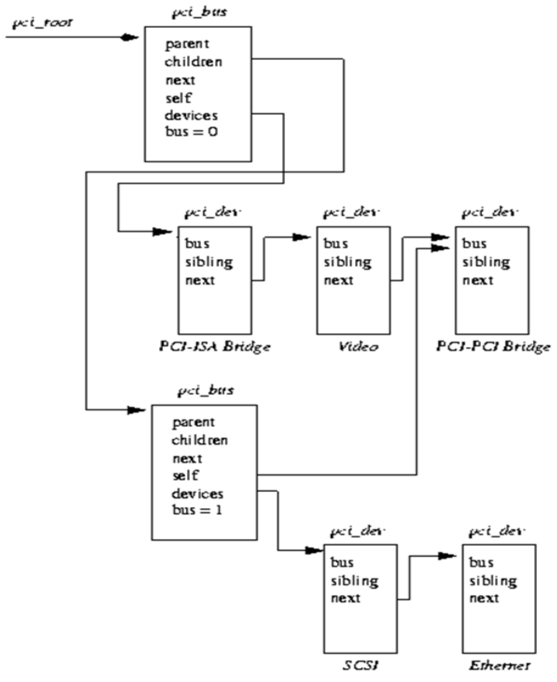


Figure22. Linux Kernel PCI Data Structures.

When Linux looks for downstream PCI Buses, it must configure the intervening P2P Bridges' secondary and subordinate Bus numbers. As for the hardware aspect, P2P Bridges need to know the primary Bus number, secondary Bus number, subordiante Bus number, PCI I/O and PCI memory windows for P2P Bridges to pass PCI I/O, PCI memory or PCI configuration address space reads/writes across them. How does Linux to deal with this problem which is at the time when you wish to configure any given PCI-PCI Bridge you do not know the subordiante Bus number for the Bridge, you do not know if there are further P2P Bridge downstream and you do not know what number will be assigned to them. The answer is to depthwise recursive algorithm and scan each Bus for any P2P Bridges assgining them numbers as they are found. As each PCI-PCI bridge is found and its secondary bus numbered, assign it a temporary subordinate number of 0xFF and scan and assign numbers to all PCI-PCI bridges downstream of it. This all seems complicated but the worked example below makes this process clearer.

Here I give a simple example for the Linux P2P Bridge nubmering process. There are 4 basic steps to simply describe the whole process:

Step1: in Figure23, The first Bridge scan would find Bridge1. The PCI bus downstream of Bridge1 would be numbered as 1 and Bridge1 assigned a secondary bus number of 1 and a temporary subordinate bus number of *0xFF*. This means that all Type 1 PCI Configuration addresses specifying a PCI bus number of 1 or higher would be passed across Bridge1 and onto PCI Bus1. They would be translated into Type 0 Configuration cycles if they have a bus number of 1 but left untranslated for all other bus numbers. This is exactly what the Linux PCI initialisation code needs to do in order to go and scan PCI Bus1.

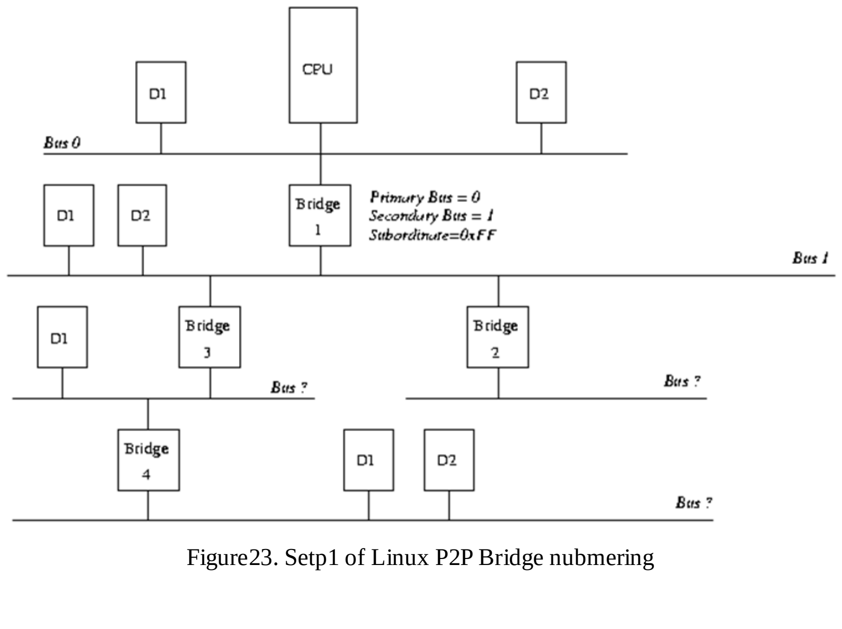


Figure23. Setp1 of Linux P2P Bridge nubmering

Step2: Linux uses a depthwise algorithm and so the initialisation code goes on to scan PCI Bus 1. Here it finds PCI-PCI Bridge2. There are no further PCI-PCI bridges beyond PCI-PCI Bridge2, so it is assigned a subordinate bus number of 2 which matches the number assigned to its secondary interface. Figure24 shows how the buses and PCI-PCI bridges are numbered at this point.

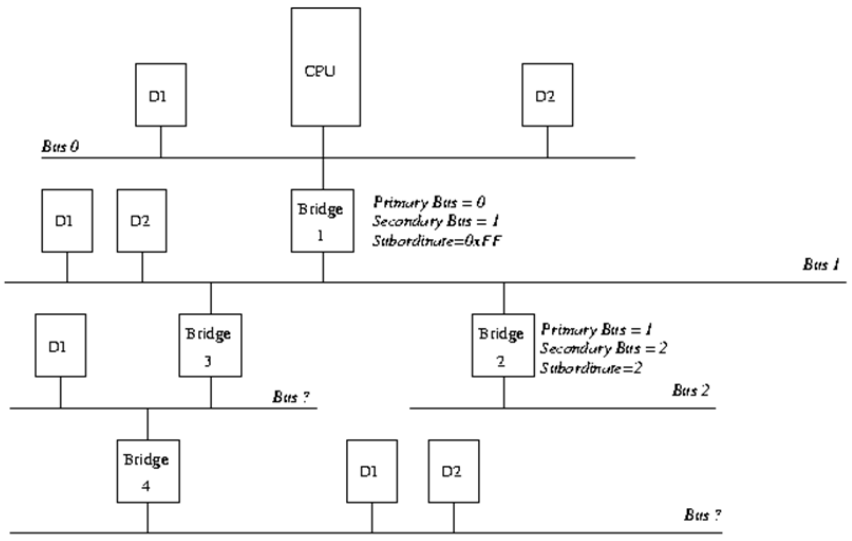


Figure24. Setp2 of Linux P2P Bridge nubmering

Step3: The PCI initialisation code returns to scanning PCI Bus 1 and finds another PCI-PCI Bridge, Bridge3. It is assigned 1 as its primary bus interface number, 3 as its secondary bus interface number and 0xFF as its subordinate bus number. Figure25 shows how the system is configured now. Type 1 PCI configuration cycles with a bus number of 1, 2 or 3 wil be correctly delivered to the appropriate PCI buses.

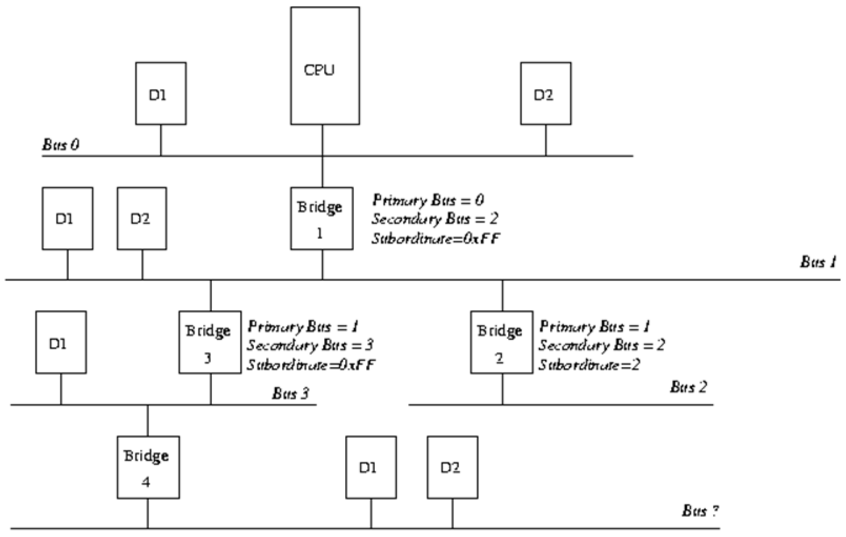


Figure25. Setp3 of Linux P2P Bridge nubmering

Step4: Linux starts scanning PCI Bus 3, downstream of PCI-PCI Bridge3. PCI Bus 3 has another PCI-PCI bridge (Bridge4) on it, it is assigned 3 as its primary bus number and 4 as its secondary bus number. It is the last bridge on this branch and so it is assigned a subordinate bus interface number of 4. The initialisation code returns to PCI-PCI Bridge3 and assigns it a subordinate bus number of 4. Finally, the PCI initialisation code can assign 4 as the subordinate bus number for PCI-PCI Bridge1. Figure26 shows the final bus numbers.

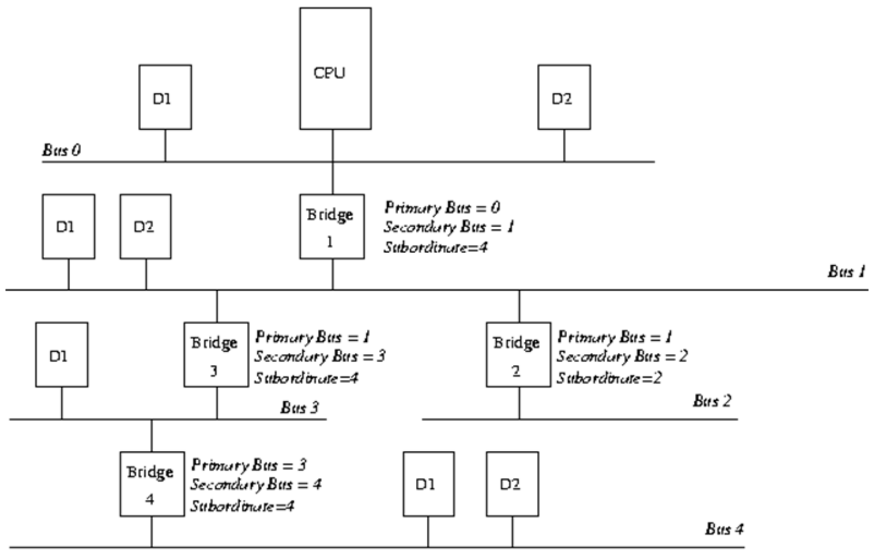


Figure26. Setp4 of Linux P2P Bridge nubmering

These 4 steps refers to not only hardware but also software, they are rather complex to be implemented. Because our PCI multi-Bus simulator is completely implemented by software under Linux Red Hat, it just do some simple operation and transaction. Figure27 shows the simple system structure. We try to design and implement 6~8 PCI Buses (including Bus0) and 5~7 P2P Bridges. Our Buses structure looks like star style (Bus1~Busn are extended from Bus0).

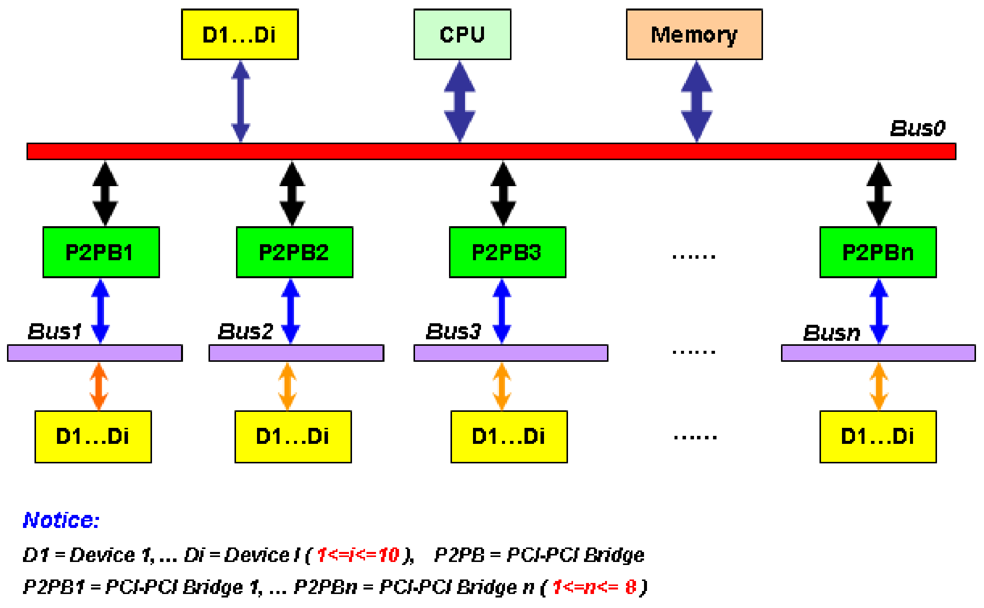

D1…Di
CPU
Memory
Bus0
P2PB1
P2PB2
P2PB3
……
P2PBn
Bus1
Bus2
Bus3
……
Busn
D1…Di
D1…Di
D1…Di
……
D1…Di
Notice:
D1 = Device 1, … Di = Device I (1<=i<=10), P2PB = PCI-PCI Bridge
P2PB1 = PCI-PCI Bridge 1, … P2PBn = PCI-PCI Bridge n (1<=n<= 8)

Figure27. PCI Bus and P2P Bridge structure of simulator

Under multi PC Bus situation, Besides the single PCI bus' relative functions, I add several new ones such as Bridge_Bus1(), Bridge_Bus2()…Bridge_Bus6(), Bus0_access_Bus(), Bus1_access_Bus()…. Bus6_access_Bus(). These functions deal with the data transfer transaction among Bus0, Bus1, Bus2…Bus6 devices. I here simply introduce these functions.

PCI_Bridge_Bus1(): this function is used to implement data transaction among bus1's devices. According to one of arbitration scheme, when PCI Bridge1 gets the bus0 control right, it will check if bus1's devices want data transfer transaction. If one or more bus1's devices want it, the Bridge1 will let them do the relative operations and when they finish, Bridge1 will release the bus0's control right and bus0's other device will get the bus0's control right to do corresponding operation. In this function, we will call PCI_Bus1_access_Bus () function which will let Bus1's devices access any other Buses' devices. On the other, PCI_Bridge_Bus1() can extend its new subordinate Buses according to our original design, although this function is still not started to be implemented.

PCI_Bridge_Bus2(): this function is used to implement data transaction among bus2's devices. According to one of arbitration scheme, when PCI Bridge2 gets the bus0 control right, it will check if bus2's devices want data transfer transaction. If one or more bus2's devices want it, the Bridge2 will let them do the relative operations and when they finish, Bridge2 will release the bus0's control right and bus0's other device will get the bus0's control right to do corresponding operation. In this function, we will call PCI_Bus2_access_Bus () function which will let Bus2's devices access any other Buses' devices. On the other, PCI_Bridge_Bus2() can extend its new subordinate Buses according to our original design, although this function is still not started to be implemented.

PCI_Bridge_Bus3(),PCI_Bridge_Bus4(),PCI_Bridge_Bus5(), and PCI_Bridge_Bus6() have the same design idea and here I will not give more details.

PCI_Bus0_access_Bus1(): This function focuses on bus0's device accessing bus1's devices. When one of bus0's devices (except Bridge) wants to access bus1's devices, we make this bus0's device and all bus1's devices as a new group. In this new group, we can use different arbitration algorithm to implement data transfer transaction among these devices. When corresponding data transaction finish, this new group will disintegrate automatically.

PCI_Bus0_access_Bus(): This function can implement that Bus0's non-Bridge devices access any other Buses' devices such as Bus1, Bus2, Bus3…Bus6 and so on. The accessing process almost the same as PCI_Bus0_access_Bus1().

*PCI_Bus1_access_Bus0():*This function focuses on bus1's device accessing bus0's devices. When one of bus1's devices wants to access bus0's devices (except Bridge), we make this bus1's device and all bus0's devices (un-including Bridge) as a new group. In this new group, we can use different arbitration algorithm to implement data transfer transaction among these devices. When corresponding data transaction finish, this new group will disintegrate automatically.

*PCI_Bus1_access_Bus():*This function can implement that Bus1's non-Bridge devices access any other Buses' devices such as Bus0, Bus2, Bus3…Bus6 and so on. The accessing process almost the same as PCI_Bus1_access_Bus0(). From PCI_Bus0_access_Bus(), we can know Bus2, Bus3, Bus4, Bus5 and Bus6's accessing process.

All these functions mentioned above are the main ones in the multi PCI bus simulator. Figure28 shows the source code executive process and you can get the simulation process according to the diagram. As for others, I do not give more details.

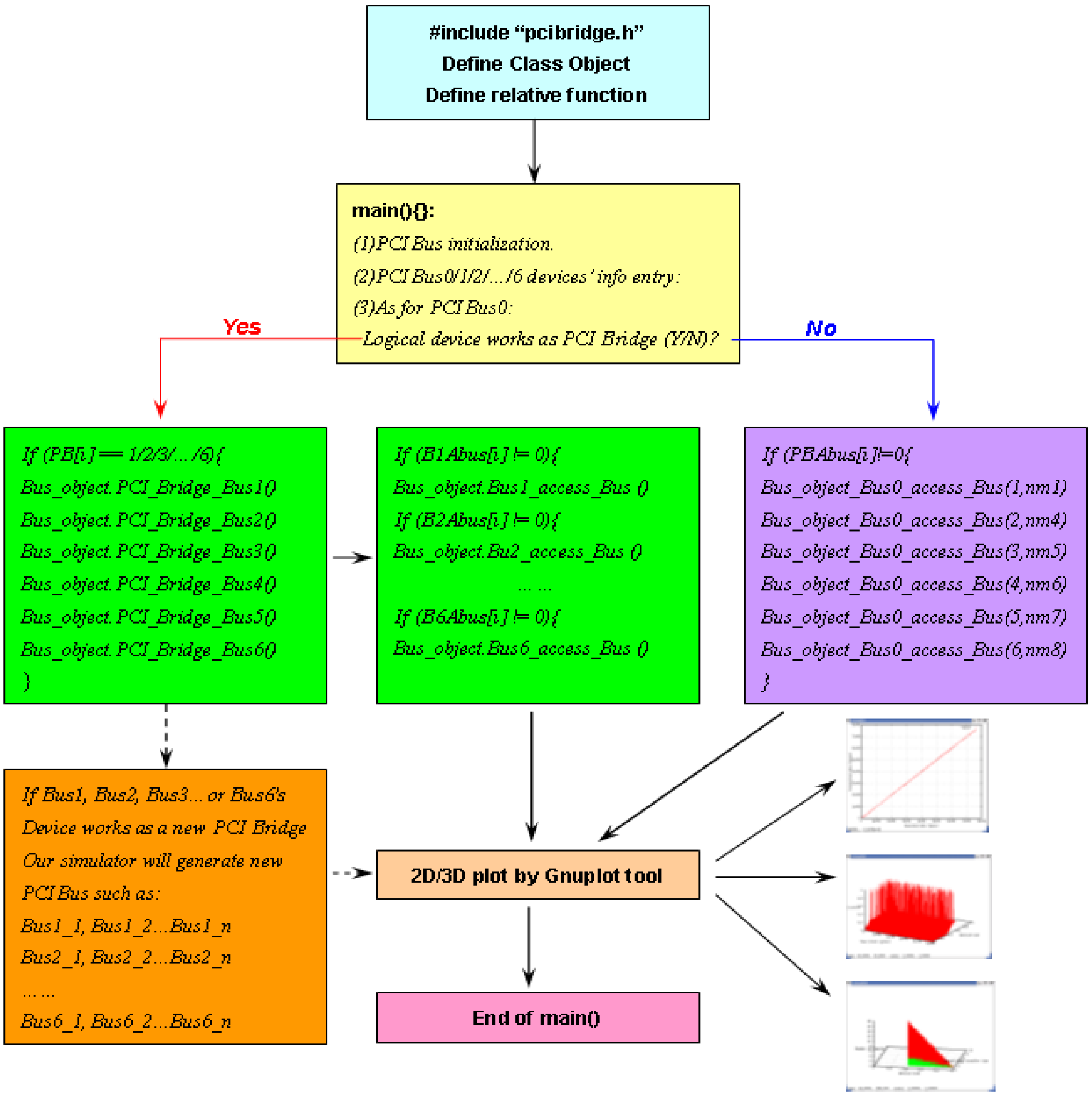

#include "pcibridge.h"
Define Class Object
Define relative function
main(){}:
(1)PCI Bus initialization.
(2)PCI Bus0/1/2/.../6 devices' info entry:
(3)As for PCI Bus0:
Logical device works as PCI Bridge (Y/N)?
Yes
No
If (PB[i] == 1/2/3/.../6){
Bus_object.PCI_Bridge_Bus1()
Bus_object.PCI_Bridge_Bus2()
Bus_object.PCI_Bridge_Bus3()
Bus_object.PCI_Bridge_Bus4()
Bus_object.PCI_Bridge_Bus5()
Bus_object.PCI_Bridge_Bus6()
}
If (B1Abus[i] != 0){
Bus_object.Bus1_access_Bus ()
If (B2Abus[i] != 0){
Bus_object.Bu2_access_Bus ()
… …
If (B6Abus[i] != 0){
Bus_object.Bus6_access_Bus ()
If (PBAbus[i]!=0{
Bus_object_Bus0_access_Bus(1,nm1)
Bus_object_Bus0_access_Bus(2,nm4)
Bus_object_Bus0_access_Bus(3,nm5)
Bus_object_Bus0_access_Bus(4,nm6)
Bus_object_Bus0_access_Bus(5,nm7)
Bus_object_Bus0_access_Bus(6,nm8)
}
If Bus1, Bus2, Bus3... or Bus6's
Device works as a new PCI Bridge
Our simulator will generate new
PCI Bus such as:
Bus1_1, Bus1_2...Bus1_n
Bus2_1, Bus2_2...Bus2_n
… …
Bus6_1, Bus6_2...Bus6_n
2D/3D plot by Gnuplot tool
End of main()

Figure28. Simulator function diagram

As for the whole PCI Multi-Bus simulator, I just introduce the basic content of design and implementation. If you want more detail, please review my source code. On the other hand, this simulator is just used for PCI Bus research and it still exists lots of logic bugs which causes many problems. Therefore, this simulator maybe can't give you accurate simulation of PCI bus data transfer transaction and all these stuffs are just used as reference. However, I believe that this simulator will become more and more robust in the coming future under further development.

REFERENCE

[1] PCI Local Bus Specification, revision 3.0, August 12, 2002

[2] PCI-to-PCI Bridge Architecture Specification, revision 1.1, December 18, 1998

[3] http://en.wikipedia.org/wiki/PCI-X

[4] http://en.wikipedia.org/wiki/PCI_Express#_note-2

[5] http://tldp.org/LDP/tlk/dd/pci.html

[6] PCI Local Bus (ppt)

[7] PCI Family History (pdf)

[8] PCI-X 2.0 Overview (pdf)

[9] PCI Local Bus Specification, revision 3.0, Feburary 3, 2004

[10] PCI BIOS Specification, revision 2.1, August 26, 1994

APPENDIX

//Part of source code of PCI Multi-Bus Simulator

Call head file: pcibridge.h”

```
#include “pcibridge.h”
```

Define classes’ object:

```
device device_object;
bus bus_object;
arbitrator arbitrator_object;
graph graph_object;
Test test_object;
```

Defnine relative function:

```
//these function mentioned below used to enter
//Bus1,Bus2…Bus6’s device information to
//relative file.
Device::pparams(FILE *fp)
Device::pparams_bus1(FILE *fp)
Device::pparams_bus2(FILE *fp)
Device::pparams_bus3(FILE *fp)
Device::pparams_bus4(FILE *fp)
Device::pparams_bus5(FILE *fp)
Device::pparams_bus6(FILE *fp)
Device::pparams_bus0_bus1(FILE *fp)
Device::pparams_bus1_bus(FILE *fp)
Device::pparams_bus2_bus(FILE *fp)

// arvitration function defination
Int arbitrator::arbitrate(int current)
//Bus initialization function
Void bus::businit()
Void bus::PCI_Bridge_Bus1(){
   for (int r1=0;r1<=nm1;r1++){
          if (bus1_access_bus[r1]==0){
                bus_object.PCI_Bus1_acces_Bus(0,nm);          //access the BUS0's devices
                };
          if (bus1_access_bus[r1]==2){
                bus_object.PCI_Bus1_acces_Bus(2,nm4);        //access the BUS2's devices
                };
          if (bus1_access_bus[r1]==3){
                bus_object.PCI_Bus1_acces_Bus(3,nm5);        //access the BUS3's devices
                };
          if (bus1_access_bus[r1]==4){
                bus_object.PCI_Bus1_acces_Bus(4,nm6);        //access the BUS4's devices
                };
          if (bus1_access_bus[r1]==5){
                bus_object.PCI_Bus1_acces_Bus(5,nm7);        //access the BUS5's devices
                };
          if (bus1_access_bus[r1]==6){
                bus_object.PCI_Bus1_acces_Bus(6,nm8);        //access the BUS6's devices
                };
    };

   // Perform simulation for increasing fractions of max. data rate
   printf("\n----------\nSimulation results:\n");
   printf("\n        Generated data    Transmitted data       ");
   printf("Average burst lengths (data cycles)");
   fflush( stdout );

   // PCI BuS1 device start to R/W operation
   …………….
}

Void bus::PCI_Bridge_Bus2()
Void bus::PCI_Bridge_Bus3()
Void bus::PCI_Bridge_Bus4()
Void bus::PCI_Bridge_Bus5()
Void bus::PCI_Bridge_Bus6()

void bus::PCI_Bus0_acces_Bus(int v1, int v2)
void bus::PCI_Bus1_acces_Bus0()
void bus::PCI_Bus1_acces_Bus2()
void bus::PCI_Bus1_acces_Bus(int v1,int v2)
void bus::PCI_Bus2_acces_Bus(int v1,int v2)
void bus::PCI_Bus3_acces_Bus(int v1,int v2)
void bus::PCI_Bus4_acces_Bus(int v1,int v2)
void bus::PCI_Bus5_acces_Bus(int v1,int v2)
void bus::PCI_Bus6_acces_Bus(int v1,int v2)

void graph::draw_graph()
Functions' defination of all Time Quatum Class and objects (omited here!)

Main(){
Bus_object.businit();

Input bus0's devices parameters:
Nm/simtot/.../priority/buffer size/Max. data rate/wait states/stochastic or determ/latency timer/...

  // Copy data to data throughput plot file if required
  if (tplot) {
          device_object.pparams(tfile);
          fprintf(tfile,"# Generated data    Transmitted data\n");
         };

  // Copy data to data rate plot file if required
  if (rplot){
          device_object.pparams(rfile);
          fprintf(rfile,"#      Time        Load        Bus utilisation\n");
         };

  // Copy data to histogram file if required
  if (xhisto){
           device_object.pparams(hfile);
           fprintf(hfile,"#Transfer time      Load        Samples\n");
         };

Input bus1's devices parameters:
Input bus2's devices parameters:
Input bus3's devices parameters:
Input bus4's devices parameters:
Input bus5's devices parameters:

// prepare to display data transaction info on the computer screen
  printf("\n----------\nSimulation results:\n");
  printf("\n      Generated data    Transmitted data      ");
  printf("Average burst lengths (data cycles)");
  fflush( stdout );
```

```
for (idr=1; idr<=ndr; idr++){
    for (i=1; i<=nm; i++){
        if (PB[i]==1){
                printf("\n BUS0 device %d (PCI Bridge) gets the Bus control right!\n\n",i);
                bus_object.PCI_Bridge_Bus1();      //run the BUS1's devices
        };
        if (PB[i]==2){
            printf("\n BUS0 device %d (PCI Bridge) gets the Bus control right!\n\n",i);
            bus_object.PCI_Bridge_Bus2();        //run the BUS2's devices ... master number is nm4
        };
        if (PB[i]==3){
            printf("\n BUS0 device %d (PCI Bridge) gets the Bus control right!\n\n",i);
            bus_object.PCI_Bridge_Bus3();        //run the BUS3's devices...master number is nm5
        };
        if (PB[i]==4){
            printf("\n BUS0 device %d (PCI Bridge) gets the Bus control right!\n\n",i);
            bus_object.PCI_Bridge_Bus4();      //run the BUS4's devices...master number is nm6
        };
        if (PB[i]==5){
            printf("\n BUS0 device %d (PCI Bridge) gets the Bus control right!\n\n",i);
            bus_object.PCI_Bridge_Bus5();      //run the BUS5's devices...master number is nm7
        };
        if (PB[i]==6){
            printf("\n BUS0 device %d (PCI Bridge) gets the Bus control right!\n\n",i);
            bus_object.PCI_Bridge_Bus6();      //run the BUS6's devices...master number is nm8
        };

        if (PBAbus[i]!=0){
                printf("\n BUS0 device %d is communicating with BUS%d devices!\n\n",i,PBAbus[i]);
            if (PBAbus[i]==1){
                    bus_object.PCI_Bus0_acces_Bus(1,nm1);      //access the BUS1's devices
                    };
            if (PBAbus[i]==2){
                    bus_object.PCI_Bus0_acces_Bus(2,nm4);      //access the BUS2's devices
                    };
            if (PBAbus[i]==3){
                    bus_object.PCI_Bus0_acces_Bus(3,nm5);      //access the BUS3's devices
                    };
            if (PBAbus[i]==4){
                    bus_object.PCI_Bus0_acces_Bus(4,nm6);      //access the BUS4's devices
                    };
            if (PBAbus[i]==5){
                    bus_object.PCI_Bus0_acces_Bus(5,nm7);      //access the BUS5's devices
                    };
                    if (PBAbus[i]==6){
                              bus_object.PCI_Bus0_acces_Bus(6,nm8);        //access the BUS6's devices
                              };
                    if (PBAbus[i]==7){
                              bus_object.PCI_Bus0_acces_Bus(7,nm9);        //access the BUS7's devices
                              };
                    if (PBAbus[i]==8){
                              bus_object.PCI_Bus0_acces_Bus(8,nm10);        //access the BUS8's devices
                              };
                }
           else{
                    printf("\n BUS0 device can not access this BUS, because it is not exist!\n");
             };
             … …
           };
              … …
        while (t < simtot){
            // implement Bus0's data transfer transaction including Read and Write
                  ………..}
// 2D/3D plotting
graph_object.draw_graph();
}    // end of main()
```