TEL AVIV UNIVERSITY

THE IBY AND ALADAR FLEISCHMAN FACULTY OF ENGINEERING

Department of Electrical Engineering - Systems

Subject

DESIGN AND IMPLEMENTATION OF PCI BUS BASED SYSTEMS

Thesis submitted toward the degree of

Master of Science in Electrical and Electronic Engineering

In Tel Aviv University

by

Ehud Finkelstein

October 1997

TEL AVIV UNIVERSITY

THE IBY AND ALADAR FLEISCHMAN FACULTY OF ENGINEERING

Department of Electrical Engineering - Systems

Subject

DESIGN AND IMPLEMENTATION OF PCI BUS BASED SYSTEMS

Thesis submitted toward the degree of

Master of Science in Electrical and Electronic Engineering

In Tel Aviv University

by

Ehud Finkelstein

This research work was carried out at Tel-Aviv University
in the Department of Electrical and Electronic Engineering,
Faculty of Engineering
under the supervision of Dr. Shlomo Weiss

October 1997

Acknowledgments

I would like to thank a few people who helped me during this work:

Dr. Shlomo Weiss, for his helpful feedback, and his excellent examples of clear writing style.

My family, for their encouragement and support.

Dov Freund, for answering all my C++ questions when I needed it the most.

And finally, I would like to thank Stephen Craimer and FourFold technologies Ltd., for kindly letting me use all the PCI related research I have done (and still do) while working there.

Abstract

PCI is perhaps the most successful bus design ever made, both on the technical and the marketing levels. Our work is divided into several sections: reviewing computer busses in general, related bus protocols, bus design principles, reviewing the PCI protocol, discussing PCI implementations using programmable logic chips, and finally, we propose PCI protocol improvements, simulate them, and analyze the results.

In the first section present an introduction to computer busses and their various parameters. We also briefly review the following busses: ISA, LPC, MicroChannel, EISA, VESA Local Bus, VME, NuBUS, FutureBus+.

In the second section, we also review the following bus standards, derived from the basic PCI spec: PMC, CompactPCI, PXI, PCI-ISA, PISA, CardBus, AGP, HiRelPCI, PC/104-Plus, and SmallPCI.

In the third section we present background material including: synchronous logic design principles, synchronous vs. asynchronous bus overview, and backplane physics.

The next section contains a description of the PCI protocol, explaining the basic protocol operations and commands, configuration registers, and PCI to PCI bridges.

Section 5 contains background material on FPGA and CPLD chips, as well as design considerations for PCI masters and targets that are specific to CPLD and FPGA implementations (as opposed to ASIC implementations). Most of these design considerations are design tips, used to reduce the design size, and to achieve the 33MHz speed requirement. We also show a typical design flow used for designing and simulating PCI devices. Both the design flow and the CPLD implementation notes are based on an actual working implementation of a PCI card designed, built, and debugged by the author.

In the last section, we discuss one of the drawbacks of PCI, which is efficient handling of targets with a long initial read latency. We suggest three alternatives for solving this problem.

One alternative is to return a “retry hint” during the target retry cycle, indicating how long it would take the target to complete the transaction. Now the master can retry the cycle only when the data is available. Between these events, other bus masters may use the bus.

The other solution we propose is a new PCI command which posts multiple short read requests to the target. The data is returned by the target as one or more bus master write transactions. This has the effect of adding split bus transactions to PCI.

We present a complete simulation environment, including synchronous logic simulation, PCI protocol simulation, and simulation of our PCI extensions. We use the simulation environment to simulate the latency hint protocol extension using a test bench modeled as close as possible to a typical PCI bus configuration as found on common PCs.

The results we have shown are that on a typical PC load the “retry hint” protocol extension gains up to 4% improvement on the bus utilization.

Table of Contents

Table of Figures

List of Tables

1. Introduction to the 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 2.1, which was released on June 1, 1995. Revision 2.2, which is due to be published soon, would add support for Hot insertion and removal of cards, power management, and incorporate several other ECRs that have been accumulated since Revision 2.1.

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:

- 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.
- 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.
- 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.
- 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.
- Future expansion
  The PCI specification can support future systems by incorporating features such as an optional 64 bit address space, and 66MHz 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.
- 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.
- 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.
- 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 32 bit word.

2. Computer busses

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

- The bus bandwidth and transfer parameters place a limit on the system performance.
- The system bus is an interface that connects hardware components produced by different vendors and provides interoperability.
- 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.
- When multiple processors share a bus with common resources, some form of support for multiprocessing is required to arbitrate the use of shared resources.

Even though memories are getting faster, CPUs get faster 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.

We describe design principles and tradeoffs in modern microprocessor system buses (FutureBus+, VME64, PCI) as well as some of their predecessors (ISA, MicroChannel, EISA, and NuBus) to provide perspective and a basis for comparison. The section consists of two main parts: description of bus architectures, and comparison and design tradeoffs. We begin the next section with a brief historical perspective.

2.1 Historical Perspective

It is interesting to review the way microprocessor system buses have evolved beginning with the ISA and up to new modern buses, such as PCI, FutureBus+, and VME64, and how various tradeoffs and compatibility issues were addressed in each design.

ISA Bus

The ISA Bus [(Messmer, 1995), (Shanley & Anderson, 1995), (Solari, 1992)] originated in the IBM PC, introduced in 1981. In it's first version, the ISA Bus was only 8 bit wide, with a 1Mbyte addressing range. When the IBM AT was introduced, another connector was added along the original XT Bus connector, adding 8 additional data lines, 4 additional address lines, and more interrupt/DMA lines. There was no organization defining the ISA bus, so when more AT compatible machines appeared on the market, the ISA Bus became a de-facto standard. Only in a later stage the IEEE defined the ISA bus as IEEE 996, but this was done as an afterthought. In fact, the objective of new systems having ISA slots is to be compatible with as many ISA cards as possible, IEEE compliant or not. Later on, Plug and Play was added to ISA bus. There are even a few non x86 based machines using the ISA bus (some SGI machines).

MicroChannel (MCA)

When IBM introduced it's PS/2 line of computers in 1987, one of the key features in IBM’s attempt to recapture the PC market was the MicroChannel bus [(IBM, 1989), (Pescatore, 1989)]. This was a completely new bus that had improved characteristics such as automatic configuration (no hardwired addresses), faster speed, and optional 32 bit data and address support. The MicroChannel was incompatible with the ISA Bus, however. IBM's licensing agreement required manufacturers to pay IBM for every MicroChannel based machine they sold, a policy which, in retrospect, might have been a major reason for the MicroChannel’s failure to gain a substantial market share. The MicroChannel was also used on some of IBM’s RS6000 workstations.

EISA

Compaq decided to solve the same problem solved by IBM's MicroChannel bus, using a different approach. The EISA Bus [(Messmer, 1995), (Solari, 1992)] was designed to be backward compatible with the ISA Bus by using a unique connector which was compatible with the ISA edge connector, but had additional pins for its extended functionality on a second row of contact fingers on the PCB's edge connector. This allowed EISA based machines to use old ISA cards as well as the new EISA cards in the same slots. Old ISA cards would simply connect only to the top row of connectors on the EISA socket, while EISA cards has longer fingers reaching the lower row of connector fingers. EISA cards could not be used on ISA machines, even in ISA mode, due to the longer EISA edge connector. Like MicroChannel, EISA hasn't really caught on, since EISA cards were much more expensive. EISA was used mainly on server machines which required multiple network cards and multiple disk controllers, which made efficient use of the bus mastering features and the extra bandwidth of this bus.

VME

VME [(Peterson, 1989), (VITA, 1994)] is the standard bus type on high performance multiprocessor servers, and high end embedded systems. VME began it's life as a 16 data bit/24 address bit bus for 680X0 based machines called VERSAbus. VERSAbus was later modified to use the EuroCard form factor, and was renamed to VME (Versa Module Eurocard). Today VME supports 32 data bit/32 address bit cards, and VME64 supports 64 bit cards. VME system supports all popular microprocessors today, including the 680X0, the SPARC, the Alpha, and the X86.

NuBus

The NuBus [(Byte, 1987), (MR, 1987)] standard originated at MIT in 1979. Western Digital bought the rights to the NuBus technology from MIT in 1981, but later sold the rights to Texas Instruments, which used it for it's Explorer line of LISP machines and System 1500 Unix system. When Apple introduced the Mac II, it incorporated NuBus slots. NuBus is also an IEEE standard, IEEE 1196. It uses the same DIN connector as VME, but has a card form factor similar to that of a PC card. NuBus bears some similarity to PCI. Both buses are 32 bit, multiplexed, synchronous, and support an automatic configuration mechanism.

FutureBus+

The FutureBus+ [(Aichinger, 1992), (IEEE, 1990), (Theus, 1992)] standard has a long history. Work began on FutureBus in 1979 (prior to VMEbus). The specification produced was very similar to VME, but the sponsoring committee felt it wasn't "good enough" and sent the specifications back to the FutureBus committee. This led to the collapse of the working group, and only in 1984 the foundation for a new specification was laid. The specifications were prototyped in 1985 and the first FutureBus+ (notice the name change!) systems were shipped in Tektronix 32032 based workstation in 1986. In 1987, the standard was published as IEEE 896.1-1987 . In 1989 FutureBus+ was independently selected as the baseline specification by the VME International Trade Association (VITA) for it's "Next Generation Architecture Bus Standard". It was also chosen for the Telecom industry's "Rugged Bus", and the Navy's "Next Generation Computer resources" (NGCR) program for mission-critical computing.

Local Buses: VESA and PCI

When 486 machines became widely available, it was necessary to have a standard for 32 bit cards that could provide fast video and disk performance. Since EISA was expensive, a much simpler design appeared on the market, the VESA (Video Electronics Standards Association) Local Bus [(Messmer, 1995)]. VESA Local Bus is mostly an extension of the 486 CPU bus, routed to an external connector. The VESA Local Bus was added as the third connector in the XT/AT ISA connector row, which means that all VESA Local Bus cards are long cards. The VESA Local Bus had a very limited goal (accelerating video and hard disks), so it's design is simple and cheap. The VESA Local Bus disappeared from the market when the Pentium and the PCI bus [(Shanley, 1995)] have arrived, for a few reasons:

- Local Bus cards are inherently more expensive than PCI cards, because they are longer (use more PCB material) and use a very long edge connector with golden fingers. They are also mechanically less reliable.
- The Local Bus specification is less advanced than PCI (limited burst capabilities, no plug and play, no bridges, limited number of slots).
- 486 based machines did not require an interface chip for the local bus, making local bus motherboards cheaper than PCI motherboards. The introduction of the Pentium with its entirely different system bus required a Local Bus interface chip, which made Pentium Local Bus motherboards as expensive as PCI based motherboards.

LPC - Low pin count

LPC [(Intel, 1997)] is a very recent standard proposed by Intel (V1.0 is dated Sep 27, 1997) designed to replace ISA for motherboard peripherals (serial ports, parallel port, floppy interface, on-board audio). Instead of the hard-to-design ISA interface, which also has many lines (at least 36), a new interface with a minimum of 6 signals for a host, and 7 signals for a target was proposed. The interface was inspired by PCI and is synchronous (Usually slaved to the PCI clock rate), and has only a clock line, reset line, one control line (LFRAME#), and 4 bi-directional address/data pins. The new interface is slightly faster than ISA and supports all the legacy ISA functions (memory and I/O R/W, interrupt request, DMA request, bus masters), all at their legacy addresses. With the new LPC standard it is possible to design a Super-IO chip (The industry nickname for the chip containing the serial/parallel/floppy interfaces) with a reduced pin count (up to 88 pins, down from 160 pins), thereby reducing cost and board space. The same can be applied to an on-board SoundBlaster compatible audio interface, also using legacy ISA resources.

LPC contains a few improvements over ISA, including support for 4GB memory addressing (unlike ISA’s 16MB), support for power down modes, and I/O & Memory cycle retries in an SMM handler.

LPC does **not** support the ISA Plug and Play mechanism, since LPC is intended for motherboard-only peripherals. These peripherals are known to the BIOS writer, and there is no need to probe the bus, looking for these devices.

LPC has no defined connector pinout, and there is **no** intention to produce LPC cards. The only expansion slot type in the next generation PCs would be PCI (and one AGP slot for a video card).

2.2 Bus Architectures

ISA

An ISA system consists of a single bus through which the CPU is connected to all the peripherals. In early PC systems, memory was also on the ISA bus, As shown in Figure 1. When CPU clock rates have broken the 8MHz barrier, the main CPU memory was moved to the local CPU bus to run at the CPU clock speed, instead of the ISA bus speed (8MHz), as shown in Figure 2. Some of the peripheral chips were integrated into the chipset, and are no longer directly on the ISA bus.

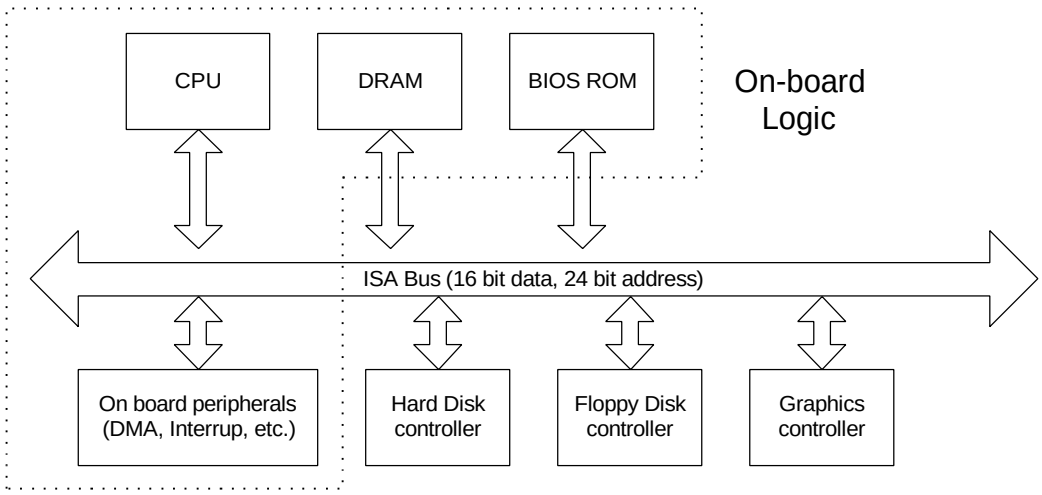


Figure 1 - Classic ISA system

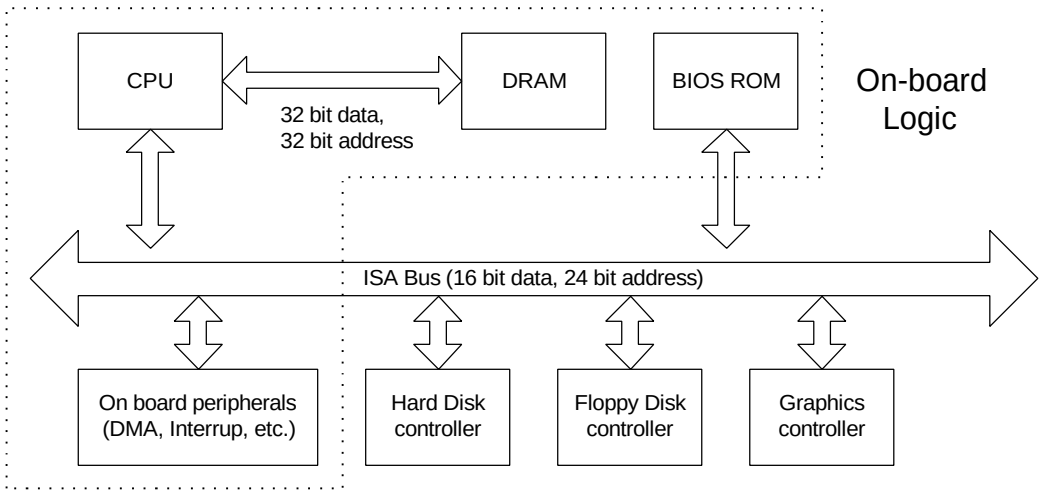


Figure 2 - Modern ISA system

The bus signals are asynchronous, and are derived from the original 8088 and 80286 CPU bus. The bus is normally controlled by the CPU or DMA controller, which means that almost all ISA cards are simple slaves. The ISA bus also supports external bus masters, but almost no cards use this feature. The ISA bus does not support bursts and until recently did not support Plug and Play.

This structure is simple and requires very little logic to control and access the bus. On the other hand, at 8MHz the bus performance is not high, with a theoretical rate of 8MB/s for write, and 5.33MB/s for read.

Comparing the ISA bus to PCI, its only advantage is its simplicity. Simple cards can be built from cheap TTL components. The slow speed of the ISA Bus, and the low pin count of TTL devices makes it possible to build cards using Wire-Wrapping techniques.

The ISA has numerous drawbacks compared to PCI: Until recently, it had no Plug and Play standard, it can't share interrupt lines, it is only 16 bit wide, it is slow, and even then it is almost impossible to realize the full ISA bandwidth potential, due to limitations of the definition of the ready and wait state bus lines.

Since the ISA Bus is widely used, it is of special importance, and therefore we will discuss the ISA Bus structure in more detail in Appendix A.

MicroChannel

The MicroChannel bus supports a 16-bit I/O address space, as in the AT, and a 24-bit (optional 32-bit) memory address space. Data ports participating in the transfer may be 8-, 16-, or 32-bit wide, and the port size may change dynamically, in every cycle. The MicroChannel supports single or burst transfers, and a bus arbitration method with 16 priority levels. A DMA controller provides DMA services to devices connected to the bus. Finally, a software-based configuration method eliminates the need for DIP switches and jumpers.

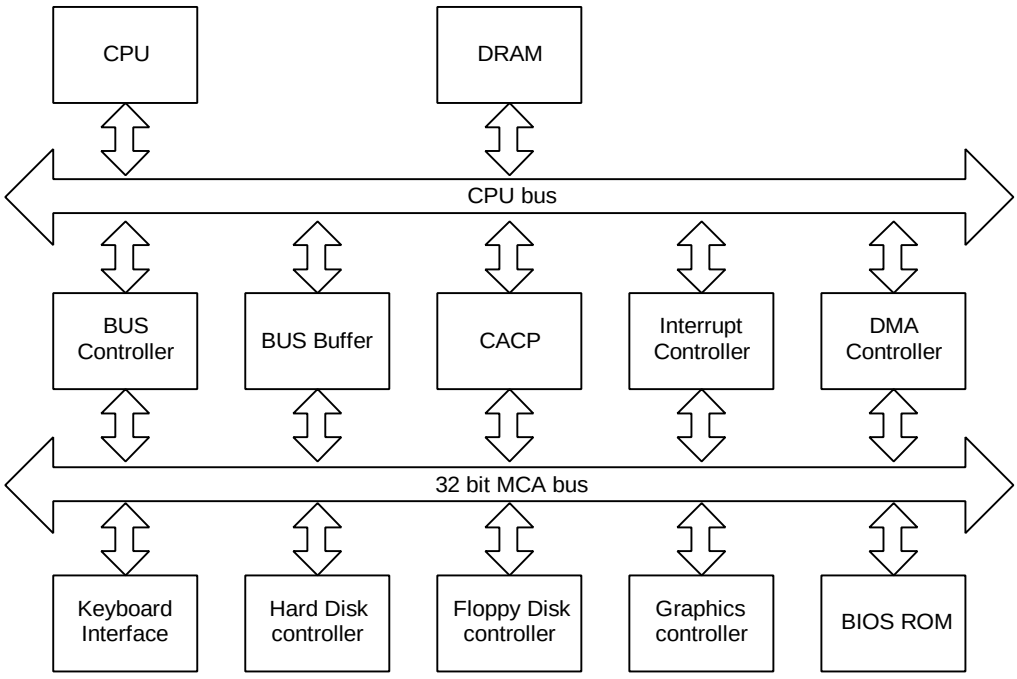


Figure 3 - MicroChannel architecture

Figure 3 illustrates a typical MicroChannel system. Transfers occur between a *master* and a *slave.* A master drives the address and control signals, such as timing strobes and the direction of the transfer. To perform a transfer, a master must become bus owner. To that end, a master must request the bus, and wait until it is granted the bus after one or more arbitration cycles. Then it can execute one or more transfer cycles, and relinquish the bus. The MicroChannel supports 16- and 32-bit masters. In addition to bus masters, there are two special masters: the system master and the DMA controller. The system master configures all MicroChannel adapters. Normally, the system master is the default master. It has the lowest arbitration priority, and it owns the bus when nobody else does.

The DMA controller transfers data between two slaves. The DMA controller is a master in the sense that it drives the address and control signals. The arbitration, however, is done by the slaves participating in the transfer.

A slave can participate in a transfer in either direction, under the control of a master. There are 8-, 16-, and 32-bit slaves. A DMA slave is a special kind of slave, which performs a transfer to another slave via the DMA controller. Like all slaves, a DMA slave does not drive address and control signals. Unlike other slaves, a DMA slave requests the bus, participates in arbitration cycles, and is granted the bus according to its own arbitration level.

The MicroChannel architecture defines a Central Arbitration Control Point (CACP), whose function is to initiate arbitration cycles when one or more requests are pending, and to allow sufficient time for resolution. Every adapter capable of requesting the bus is allocated a unique, four-bit arbitration level, at system start up. Arbitration levels range from 0000 to 1111 in descending priority order, with 1111 (lowest) assigned to the default master.

When the CACP begins an arbitration cycle, each requesting adapter drives its arbitration level on the four-bit arbitration bus. The resolution is decentralized and is performed by arbitration logic implemented in each adapter that can request the bus. An adapter that identifies a higher priority level on the arbitration bus must withdraw its request from the current arbitration cycle. The CACP allows sufficient time for this process, at the end of which the level of the device with the highest priority level remains on the arbitration bus. That device becomes the next bus owner. This logic is very similar to the NuBUS arbitration logic, which can be seen in Figure 80.

In burst mode, a device may remain the bus owner as long as no other devices request the bus. If there is another device requesting the bus, the bursting device must release the bus with the time limit specified by the MicroChannel specification. To prevent situations in which low priority devices wait indefinitely for the bus, likely to occur in systems that support high-priority bursting devices, the MicroChannel defines a fairness algorithm that work as follows. When an adapter is preempted, it cannot participate in another arbitration cycle as long as other devices are waiting for the bus. This guarantees that all pending requests are granted in their priority order, and no device will be granted the bus for the second time until all requesting devices have received access to the bus. Only bursting masters are required to implement the fairness algorithm. This algorithm is also used by NuBUS bus masters.

EISA

The EISA architecture is quite similar to the MicroChannel architecture shown in Figure 3. To remain compatible with ISA, EISA allows the use of ISA cards, and also retains the maximum 8.33 MHz clock rate. The EISA bus controller (EBC) supports 32- and 16-bit EISA adapters, and 16- and 8-bit ISA adapters. Transfers may occur between ports with different widths. For example, assume a 32-bit EISA bus master performs a write to an 8-bit ISA slave. The bus master gains access to the bus and drives address and data signals on the bus. The 8-bit ISA slave indicates it can only perform 8-bit transfers. At this point, the EISA bus controller takes over and drives on the bus the same signals as the bus master. Then the bus master deactivates its drives, and the signals, now driven only by the bus controller, remain steady on the bus. Finally, the bus controller splits the 32-bit data into four ISA byte writes. Read accesses are accumulated and combined into 32-bit reads in a similar way. The EBC implements the logic that splits/combines data and generates multiple access cycles, and bus masters are not required to duplicate it.

EISA supports burst cycles. There is also a more recent specification of an enhanced burst cycle, that provides higher transfer rates while keeping the bus clock at 8.33 MHz for compatibility. A 66Mbyte/s rate is achieved by performing 32-bit burst transfers on both edges of the clock. A 133Mbyte/s rate is achieved by multiplexing the 32-bit address bus to transfer 64 bits (using both the address and data buses), on both edges of the clock.

A common problem in ISA machines is the lack of interrupt lines. EISA solves this problem by allowing interrupt levels to be level-sensitive, instead of edge-triggered. Multiple devices may share the same interrupt line. While one device is serviced, the interrupt line remains active indicate pending interrupts from other devices. ISA adapters are edge-triggered, and require separate interrupt lines even if plugged into an EISA bus. Only multiple EISA adapters can share the same line.

VME

VME systems are usually based on a passive backplane with one or more CPU cards, and one or more I/O cards. VME backplanes may have as many as 21 I/O cards thanks to its asynchronous bus structure. VME cards come in 3 sizes: 3U, 6U, and 9U. 3U cards has a single 96 pin connector, and only 16 bits of data and 24 bits of address. 6U cards has two 96 pin connectors. The extra connector is used to add 8 extra address lines, 16 data lines, and 32 user pins. The user pins allows extra user cables to be connected to the card through the back of the backplane, allowing rapid switching of cards without removing the cables from the card first. The user pins are also used for routing additional private busses between cards in addition to the VME bus. Most standard cards today are 32 bit wide 6U cards. 9U cards are even bigger, and have a third user connector.

The actual bus width is dynamic, and set during the transaction. A cycle begins by driving the address, a 6-bit address modifier code specifying the address width (16/24/32) and operation type (r/w), and the address select line. The data bus width is set using the DS[1:0], LWORDA and A01 lines. A 32 bit capable card will also accept 8 and 16 bit cycles. An 8 or 16 bit wide card may choose to accept 32 bit transaction, but it will have to ignore the extra data. VME cannot resolve data bus width like the 680X0 bus can.

A master card may choose to do address pipelining by starting a new address cycle before the current data cycle is finished. This is possible thanks to the non-multiplexed bus. The arbitration logic is daisy chained, and has 4 levels.

VME also optionally supports hot insertion and removal of cards which is a must for fault tolerant systems.

Some recent additions to the standard includes the VXI standard (VME based measurement instruments on a card controlled by software), allowing complex CPU controlled test and measurement systems containing multiple instruments. VXI inspired the creation of its PCI equivalent, the PXI bus (see section 3.1.5).

NuBus

NuBUS is a 32 bit wide, 10Mhz synchronous bus. NuBUS uses standard VME type DIN connectors with 3 rows of 32 pins each. The clock signal is 75ns high, and 25ns low, with signals changing on the rising edge, and latching on the falling edge. This allows 25ns for hold time and clock skew. Every card has a unique slot ID which is used to map a unique 16MB memory area at the top 256MB of the memory map and is used for configuration. The bus itself is multiplexed with 32 bit for data and address. A simple, non-burst transfers involves driving the address and command lines for one cycle. The two lower bits of the command lines TM[1:0] and the two lower bits of the address lines AD[1:0] indicate the transfer size (byte/word/longword), alignment, and operation (read/write). The target acknowledges the cycle and returns status information (OK, error, timeout, or retry).

The theoretical non-burst data rate on NuBUS is 20MB/s by doing a 2 cycle read/write operation. In practice, read operations may take 3 cycles or more.

Block transfers are done in NuBUS by using two special TM[1:0] and AD[1:0] codes indicating burst read or burst write. Burst length is encoded using AD[5:2]. Possible burst lengths are 2, 4, 8, 16 longwords, and the starting address must be naturally aligned[1]. During burst, words are acknowledged by asserting TM[0], and ACK is generated only for the last word in the burst, with a status code. Since the longest burst is 16 longwords, the fastest possible cycle gives a peak transfer rate of 37.5MB/s. Non-burst capable boards can respond with ACK for the first word, which tells the bus master that bursts are not supported for this particular target.

Arbitration between bus masters is done via a 4-bit arbitration bus. A distributed algorithm similar to MicroChannel is used to select the winner. Bus fairness is guaranteed by not allowing the current bus master to arbitrate the next cycle while the current bus cycle is still running (other cards may do so). Bus locking may be achieved by keeping the card ID on the arbitration bus, with the request signal active, for multiple cycles.

[1] A *naturally aligned* block must be 2^N words long, and its start address must be an integer multiple of the block size. As a result, bits [N-1..0] of the block start address are always 0.

Interrupts are not supported by NuBUS. Instead, virtual interrupts can be generated by bus master cards which use the bus to write values to a predefined address. This address may be identified by the CPU support chips and be used to generate an interrupt on the local CPU bus.

Due to the close similarity between PCI and NuBUS, a complete technical description of NuBUS is found in Appendix B.

FutureBus+

The FutureBus+ standard is split into multiple sections:

- The electrical characteristics of the FutureBus+ signals such as signaling levels and load curves.
- The mechanical section defines the FutureBus+ form factor parameters such as card dimensions and the connector type used.
- The FutureBus+ protocol is the core of the standard, and defines the various signals and their behavior.

The use of separate definitions allows some parts of the standard to evolve, while staying compatible with other aspects of the bus.

Some of the special features of FutureBus+ can be summarized here:

- Architecture, Processor, and technology independence.
- No technology based upper limits. The only speed limiting factor in the standard should be physical limitations (speed of light). The result of this guideline was an asynchronous bus which can go faster as technology improves.
- Fault Tolerance: parity protection on all lines, fully distributed arbitration protocol (to reduce the risk of a single failure point), live insertion and removal of modules, dual bus operation, and fault detection and isolation mechanisms.
- Full support for cache coherency protocols, and split transactions.
- Message passing protocol for efficient multiprocessor communication.
- Full Support for bus to bus bridges. Bridges for the following buses has been defined in the standard: VMEbus, Multibus II and SCI (Scalable Coherent Interface) [(Tving, 1993)].

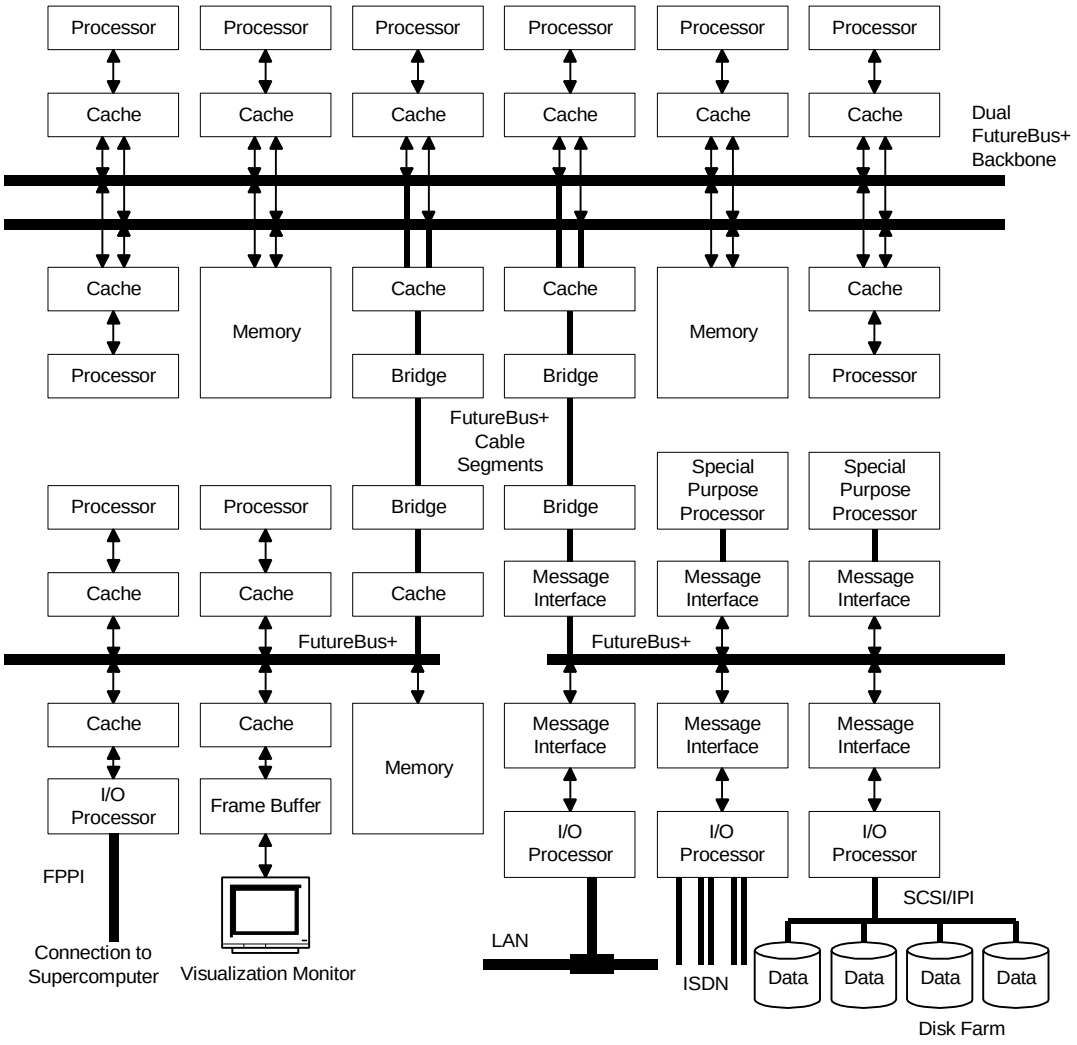


Figure 4 - FutureBus+ architecture

One of the major applications of FutureBus+ is multiprocessing, as illustrated in Figure 4. The most striking feature in Figure 4 is the hierarchy of buses interconnected by a set of bridges. The parallel protocol supports split transactions, required for efficient communications across buses. FutureBus+ integrates a MESI cache coherence protocol, and supports snarfing. While somewhat more complex to implement than a read invalidate protocol, snarfing allows cache-to-cache transfers of modified cache blocks without updating the memory. Snarfing can save significant bus bandwidth as it performs a single transfer of the modified cache block to the requester, rather than two transfers (memory write of the modified block followed by a memory read by the requester) in the simpler read invalidate protocol.

PCI

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 MicroChannel 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 (see Figure 5). 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 [(Wang et. Al., 1995)].

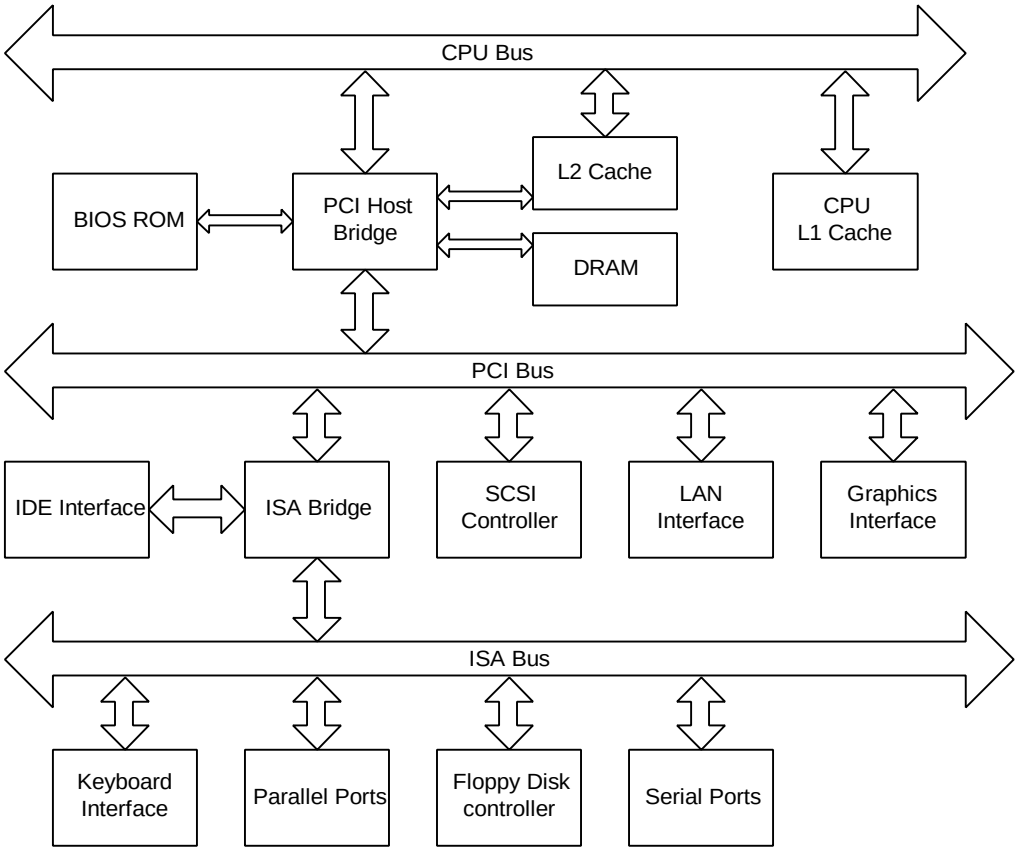


Figure 5 - PCI architecture

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 deasserts 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 begins 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.

To reduce data transfer time on PCI, a bridge may combine, merge, or collapse data into a larger transaction. *Combining* refers to converting sequential memory writes into a single PCI burst transaction. For example, a write sequence of (32-bit) double words 1, 3, and 4 can be combined into a burst with four data phases, the second data phase having the byte enables off. Transactions whose order is not sequential, for example 4, 3, and 1, must remain as separate transactions. *Merging* refers to converting a sequence of memory writes (bytes or 16-bit words) into a single double word. Unlike combining, merging can occur in any order. For example, bytes 1, 0, and 3 can be merged into the same double word with byte enables 0, 1, and 3 asserted.

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.

Four interrupt lines are connected to each PCI slot. A multifunction unit may use all four, other units may use only a designated line (one of the four). The value in the Interrupt Line register determines which IRQ should be activated by the interrupt signal of the PCI unit. (For example, IRQ14 of the AT architecture, if the PCI unit is a disk adapter). The BIOS fills this field during system initialization. Later, when the operating system boots, the device driver reads this field to find out to which interrupt vector the driver's interrupt handler should be bound to.

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 relocatable - 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 Comparison and Design Tradeoffs

Address and Data Transfers

Usually the number of address pins (either separate address pins or multiplexed address/data pins) determines the amount of addressable memory on a bus. Some buses (and PCI specifically) can generate addresses larger than the bus width by splitting the address phase across two bus cycles. A card using a wider data bus can transfer data faster, but requires more pins, it is less reliable, requires more components, and is more expensive. Non only that, but any CPU with an N-bit wide data bus can hardly make effective use of a bus which is wider than N bits.

A multiplexed bus shares the same lines for both address and data. In a non-multiplexed bus, there are separate address and data lines. Non-multiplexed buses require more lines, but they are slightly faster on write (since address and data can be sent at the same time), and they can start a new read cycle (by driving the new address) while the current read cycle is still taking place. On the other hand, if a burst cycle is taking place, there are multiple data phases for every address phase, so the extra speed improvement by having a non multiplexed bus is relatively small. If the same lines used by a non multiplexed bus for addressing were used as extra data lines on a multiplexed bus, burst transfer would be faster on the multiplexed bus, using the same number of pins. Most new buses (PCI, FutureBus+, VME64) are multiplexed.

Name	ISA	EISA	MicroChannel	VME	NuBus	FutureBus+	PCI
Min. Data Bits	8 (XT)	32	16	16	32	32	32
Max. Data Bits	16 (AT)	32	32	64	32	256	64
Min. Addr. Bits	20 (XT)	32	24	24	32	32	32
Max. Addr. Bits	24 (AT)	32	32	64	32	64	64
Multiplexed Addr/Data Bus	No	No	No	Both	Yes	Yes	Yes

Table 1 - Data and Address Bus

In the first row of Table 1, the number of data bits refers to the minimum number of bits required on the implementation, not on the actual data transfer. For example, all EISA cards have 32 data bits, but not all data cycles are 32 bit. Some cycles may be 8 or 16 bit.

VME uses non-multiplexed 32 bit address/32 bit data bus. VME64 combines these address/data lines to a multiplexed 64 bit address/ 64 bit data bus. Note that Max. VME Addr./Data Bits (second and fourth row of Table 1) refer to VME64 only.

Throughput

Bus throughput calculations can be quite complex, so we mention only a few parameters here. The single word throughput measures the theoretical rate (in MB/s) at which an arbitrary single word read requests can be performed. The burst throughput is the theoretical rate at which reads can be performed (in MB/s) using the largest possible burst cycle possible on the bus.

When an address is transferred on the bus for every data word read or written, the bus is called non-bursting. When a bus can have a single address phase followed by multiple data phases (usually for consecutive addresses) the bus can do burst transfers. Burst data transfers lowers the overhead for large transfers (such as video screen manipulation and disk/networking I/O), by allowing the slave device to pipeline the subsequent data words while transferring the current word on the bus. It also removes the address phase overhead for every word.

Name	ISA	EISA	MicroChannel	VME	NuBus	FutureBus+	PCI
Max throughput, Non Burst	8MB/s	16.66MB/s	13.11MB/s	N/A	20MB/s	200MB/s	33.33MB/s
Max throughput, burst	N/A	33.33MB/s	21.05MB/s	80 MB/s	37.5MB/s	4GB/s	133.33MB/s
Multiple Bus Masters	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Split Transactions	No	No	No	No	No	Yes	Some

Table 2 - Throughput

Referring to Table 2, the ISA throughput figure in the second row is based on no wait state cycles (2 clocks/cycle) at 8MHz with 16 bit data. This is a purely theoretical number, based on zero delay between the data request and the data acknowledge signals. The MicroChannel throughput figures are based on typical parameters. The maximum throughput, burst performance assumes a 190ns burst cycle time. A burst cycle in MicroChannel is the same as a non burst cycle, except for bus arbitration. Non burst cycles are allowed to use the bus only for one cycle. The VME data is for VME64. FutureBus+ specifies two transfer modes, an asynchronous (handshake) mode, and a synchronized packet mode. Packet mode is optional in FutureBus+ profile B. The max throughput non-burst data show in Table 2 are based on FutureBus+ profile B performance figures in for 64-bit asynchronous mode.

The ability to have multiple bus masters means that every card on the system can request the bus and generate read or write cycles by driving the address, data, and control pins. Cards on a bus with a single master cannot actively generate cycles on the bus. They can only be passively read or written.

Even though memories are getting faster, CPUs get faster quicker. Therefore if memory access time used to be less than 10 machine cycles, today the penalty for memory access may be a few dozen cycles. 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. More time is spent on waiting for the initial latency, and less time is spent reading the data. By splitting the read request from the data transfer, the dead time between the request and the data transfer can be used to send more requests to other memory agents or to receive data from previous requests. This way the initial latency is not reduced, but the bus is free during that time for other requests and is not wasted.

Split transactions is a mechanism allowing a master to request data, disconnect from the bus, and have the target reconnect later to the bus and send the data when it is available. This allows other bus master to use the bus during the latency time.

PCI standard 2.1 requires a master accessing a target to retry the data transfer if it is encountering a target retry on the first data word. This allows a target with a very long latency to receive requests from masters, disconnect, and get the data to an on board buffer. When the master retries the same address, it will receive the data immediately. This is not a true split transaction, because the master can't really know when the target data is ready. If the master retries too early, it wastes bus bandwidth. If it retries too late, it will end up with a longer latency than what is needed, and limit the target's ability to queue additional requests, since buffer storage is wasted, waiting for the master to empty it.

Synchronous and Asynchronous Buses

In synchronous buses all the bus signals are sampled on the edge of a system clock. All signals obey the setup and hold requirements relative to the clock signals. This has the advantage that no handshaking is needed to transfer a single word. Asynchronous buses, on the other hand, are handshaking data transfer without any central clock signal. Control signals can change anytime without any restrictions.

Each bus has its limitations. An asynchronous bus can adapt well to different number of peripherals on the bus by giving better performance on a bus with a smaller load. A CMOS based asynchronous bus can drive a large number of slots or a very long bus (SCSI-1 is a good example of a potentially long asynchronous bus) without any problems, even at the price of reduced performance. A synchronous bus on the other hand will work faster, but will not work at all if the bus is overloaded or made longer, causing the total propagation delay to exceed the clock cycle time. If a synchronous bus is designed in advance to support a large number of slots or a long backplane, then the performance will be much worse, even when the same bus is not loaded with extra slots, because the clock rate is fixed.

In section 4.2 we discuss synchronous and asynchronous busses in greater depth.

Name	ISA	EISA	MicroChannel	VME	NuBus	FutureBus+	PCI
Bus Clock	8MHz	8.33MHz	Async.	Async.	10MHz	Async.	0 - 33MHz
Number of Bus slots	Undef.	15	15	21	15	32	4

Table 3 - Bus clock and number of slots

The number of bus slots determines the expandability of the bus. Usually a bus has both architectural limitations to the number of slots, as well as electrical limitations. An example of an architectural limitation is the number of bits of a slot ID code, while an electrical limit is, for example, the 10 load maximum for a pin in the PCI standard. An electrical limit can be overcome by improving technology without hurting compatibility while an architectural limit requires changes in the standard in order to be solved.

As we just said, the PCI standard allows 10 electrical loads per bus. A PCI peripheral on a card counts as two loads, while an on-board peripheral counts as one load. A motherboard can have, therefore, up to 4 slots (see Table 3). Since PCI buses can be linked with bridges (up to 256 buses), A theoretical maximum of 2560 on-board PCI devices can share one PCI system, (or up to 2050, excluding the bridges).

The number of ISA slots is not limited by the standard, but limited by practical implementations. Typical systems use up to 8 slots. Logically, it is possible to have 15 bus slots on EISA, MicroChannel, and NuBUS. EISA is limited by the pre-assigned configuration space at `XC80h` through `XC84h`. MicroChannel and NuBUS are limited by the 4 bit distributed arbitration logic. In practice, the EISA timing specifications are limiting the number of physical slots to 7. MicroChannel and NuBUS might have the same practical limitations.

Multiprocessing Support: Locking and Snooping

When multiple processors are sharing a common bus with common resources, a semaphore is required to arbitrate a specific resource between multiple processors. A semaphore is set by the processor requesting the resource, and cleared when the resource is freed. A semaphore must be read before it is set, because it might be already set. The semaphore read and set operations must be atomic because if two processor may read the semaphore at the same time and both see the semaphore is clear, they may both try to set it simultaneously.

Bus locking allows a specific memory range to be locked by a master, causing this memory area to be accessible only by the same master until the lock is released. A more limited way of bus locking is the Read-Modify-Write cycle which allows a master to read a word, modify it, and then write the result back to the same address, with a guarantee that no other master can access the bus between the read and the write operation. These hardware primitives allows the system software to build more complex locking mechanisms for resource arbitration between multiple processors.

In multiprocessor systems we usually find one or more processors with local cache memory on a common bus, sharing common memory which is also on the bus. A data coherency problem is caused by the local caches since data modified by one processor in memory is not reflected in another processor if that processor has a copy of that word on his local cache. Not only that, but on a write-back cache system a word may be written back to memory but in fact be written only to the write back cache. Later, a different word in the same cache line is written by a different processor, and flushed to memory. If the first word with its entire cache line is also flushed to memory, it will erase the rest of the cache line in memory, destroying newer data. A possible solution is bus snooping, which allows a processor to watch the bus all the time, invalidate (or update) its local cache copy of data which appears on the bus, and even allows it to interrupt a write cycle on the bus in order to flush its own write back cache first, before allowing the cycle to restart.

Name	ISA	EISA	MicroChannel	VME	NuBus	FutureBus+	PCI
Bus Snooping	No	No	No	No	No	Yes	Yes
Bus Locking	No	Yes	No	Yes	Yes	Yes	Yes

Table 4 - Locking and snooping

Bus Locking is defined for PCI, but most targets don’t support it. PCI bridge support is optional (since 2.1), and its usage is highly discouraged. Bus Locking is implemented in VME by a Read-Modify-Write cycle, unlike the PCI locking mechanism, which allows other transfers to take place while the lock is still in place.

Bus Snooping is essential for PCI, because modern PCI based systems typically have bus master PCI cards and local CPU caches with write back capabilities. The snooping mechanism is required when a dirty cache line is partially overwritten by an external PCI bus master.

Plug and Play

A Plug and Play capability is the definition of a standard way for a bus master to interrogate all the devices on the bus and identify them (including type, manufacturer, and model), their resource requirements (memory and I/O address space, interrupts, and DMA channels), assign specific resources to each device, or disable any specific device. It also defines a common header for an optional bootstrap ROM, which can do further device specific initialization. It can even go as far as defining a neutral language for ROMs which is processor independent, allowing the same card to run its initialization code on all processor types if their system software supports an interpreter for that language.

Name	ISA	EISA	MicroChannel	VME	NuBus	FutureBus+	PCI
"Plug and Play"	Yes/No	Yes	Yes	Yes	Yes	Yes	Yes

Table 5 - Plug and Play

ISA “Plug N’ Play” (or PNP) is a new standard, supported by some ISA cards only since 1995. A fully compliant ISA “Plug and Play” system requires a PNP aware motherboard, PNP cards, and a PNP aware operating system, such as Windows 95.

A VME board may contain CR/CSR registers which carry identification information and configuration registers, but since these are defined as optional, not all boards may have them.

All PCI cards are plug and play, but most PCI cards do not support the OpenBoot standard for neutral language boot code (only some Apple Macintosh PCI cards do). Some Power-PC system manufacturers have gone the trouble of implementing a software X86 emulator to allow booting from SCSI bus masters on Power PC based PCI systems.

Interrupt Support and DMA

There are many ways to implement interrupt support. Some buses have minimal interrupt support with no interrupt lines. A device can generate an interrupt by writing to a special address on the bus, or generating some other special type of bus cycle. Interrupt priority is handled by the regular bus arbitration logic. Most buses, however, have dedicated interrupt lines, which determines the maximum number of interrupt levels on the system. On systems which use level triggered interrupts, the interrupt event is active as long as the interrupt line is asserted, while edge triggered interrupts are generated during the transition of the interrupt lines from a low state to a high state or vice versa. Interrupt sharing is harder when using edge triggered interrupts, because edge transitions on an interrupt line which is already active are invisible.

DMA channels are a special feature that makes it easier to build a bus master card. Every DMA channel has a pair of handshake lines on the bus, which are connected to the DMA controller. A DMA transfer takes place after a program initializes the DMA channel by setting the data direction, base address, and number of words. The card will then read or write to the block of memory specified in the DMA channel registers simply by handshaking word transfers using the dedicated lines. The DMA controller takes care of the actual bus cycle generation, address generation, address increment, and word count. Since every DMA channel requires at least two lines per channel, and requires dedicated DMA logic, only a limited number of DMA channels are available.

Name	ISA	EISA	MicroChannel	VME	NuBus	FutureBus+	PCI
Number of Interrupts	6/12	12	11	7	N/A	N/A	4
Interrupt type	Edge	Level/Edge	Level	Level	Virtual	N/A	Level
DMA Channels	3/7	7	N/A	N/A	N/A	N/A	N/A

Table 6 - DMA and interrupts

The 6/12 ISA interrupt lines and 3/7 DMA channels in Table 6 correspond to XT/AT cards respectively. EISA supports level-sensitive interrupts, but ISA adapters are edged-triggered, for compatibility reasons, even if plugged into an EISA bus. PCI defines 4 interrupt lines *per slot*, but these 4 lines aren't necessarily common to all slots. In theory a 4-slot PCI system may have 4 unique lines for every slot, and use as little as one or as many as 16 CPU interrupt lines.

Virtual Interrupts are predefined addresses in the memory map. A card that wishes to interrupt the CPU must act as a bus master and write to this address. Apple’s version of NuBUS added a line called NMRQ (Non Master ReQuest), which acts like the usual interrupt line.

Physically, the MicroChannel bus does not define any DMA request/grant signals, but unlike other buses with no DMA signals, the MicroChannel system architecture specifies an on board DMA controller, which works by generating normal bus cycles, reading from the source and writing to the destination. This DMA controller does not require special bus support from the bus itself other than normal read/write cycles.

3. PCI Bus related standards

Although the PCI standard has many benefits, it was designed for the desktop PC market, and cannot be directly used in most embedded systems or in industrial applications due to packaging, reliability, or maintenance considerations. As a result, a number of PCI related standards were developed. These standards bring PCI performance to other segments of the computer industry, and enable the use of PCI silicon, software, and firmware in non-desktop systems. While the proliferation of PCI related standards is certainly evidence of the success of PCI, some confusion arises because it is not always clear why are so many standards needed, what are the differences between them, and which standards apply to a specific system. This section deals with these questions. We describe standards that enable the use of PCI technology in industrial applications, embedded systems, laptops, and mobile systems. We also describe an extension of the PCI, the AGP bus, developed for graphics and multimedia applications.

3.1 Industrial Applications: Backplane Expansion

Price is the dominant design consideration in desktop PC systems. In critical embedded computing systems used in telecommunication, industrial automation, and military applications, in addition to cost there are at least two other primary factors:

- Reliability

  In desktop PCs, ISA and PCI adapters are inserted into card-edge connectors located on a motherboard. A PCI card is typically fastened only at one point of the edge opposite to the connector. This mechanical arrangement has poor shock and vibration characteristics, the card edge connectors are subject to shifting or even disconnection. This situation is compounded by the use of an “active” backplane: a motherboard on which both ICs and connectors are mounted. In an industrial or military environment, the mechanical stresses involved in supporting cards may lead to failures in the motherboard.

  In critical embedded systems, these reliability concerns are addressed by the use of pin and socket connectors, front panel retainers that lock the card to the frame, and card guides. Every card is mechanically supported on all four edges. Reliability is also enhanced by the use of a “passive” backplane, whose primary function is to provide electrical connections; mechanical stresses on the backplane are low due to the use of front panel retainers and card guides.

- Maintenance

  Low MTTR (mean time to repair) is a key requirement in critical embedded. Failures in a passive backplane are uncommon due to improved mechanical characteristics, as we have seen above, and the lack of active components. The motherboard used in desktop PCs is much more likely to fail simply because ICs mounted on it may fail. Hence the use of a passive backplane considerably simplifies maintenance; all ICs and other active components are mounted on cards plugged into the backplane that can be speedily removed. This simple replacement procedure applies to all the cards in the system, such as processor cards or peripherals. The system controller is simply one of the cards plugged into the backplane.

The VME bus has been the dominant architecture used in the telecommunications, industrial automation, military, and medical markets (see sections 0 and 0).

Although the VME architecture is expected to continue to dominate the industrial system market at least in the near future, the success of PCI has led to three standards that apply PCI technology to industrial systems: PCI-ISA passive backplane standard, PMC, and CompactPCI. A fourth standard, HiRelPCI is currently developed by an IEEE working group. A fifth standard, PISA, is very similar to PCI-ISA, but its industry support is presently very weak. These standards are summarized below and described in the rest of this section.

	PISA Passive Backplane	PCI-ISA Passive Backplane	PMC	CompactPCI	HiRelPCI
Connector	Edge Card	Edge Card	Pin & Socket	Pin & Socket	Pin & Socket
Installation	Perpendicular	Perpendicular	Parallel	Perpendicular	Perpendicular
Shock Resistance	Poor	Poor	Good	Good	Good
Off-the-shelf PCI Adapters	No	Yes	No	No	No
Off-the-shelf ISA Adapters	Yes	No	No	No	No
Organization	PISA	PICMG	IEEE	PICMG	IEEE
Dimensions (in.)	3.9×6.9	3.9×12.3	2.9×5.9	3.9×6.3 9.2×6.3	4.5×8.4 10.4×11.3

Table 7 - PCI related industrial standards

3.1.1 PCI-ISA Passive Backplane

The PCI-ISA standard specifies a passive backplane with two buses, ISA and PCI, and a CPU card whose dimensions are identical to the dimensions of the long style PCI cards. The CPU card contains all the components normally located on the motherboard (including the PCI chip set that implements the PCI-ISA bridge), and two connectors, one for the ISA bus and one for the PCI bus. Hence the "active" motherboard is replaced with a plug-in CPU card, and I/O expansion cards plug into a passive backplane that has only connectors on it. As we have seen above, the passive backplane improves reliability and simplifies maintenance.

The standards supports up to 15 ISA slots and up to 4 PCI slots, as in desktop PCs, and allows the use of off-the-shelf ISA and PCI adapters. The adapters use standard ISA and PCI edge connectors. These connectors are less reliable than the pin-and-socket connectors used in VME and CompactPCI systems. Some backplanes offer up to 13 PCI slots by using one or more PCI to PCI bridge chips on the backplane.

3.1.2 PISA Passive Backplane

This standard is very similar to PCI-ISA, and has the same objectives. The method, however used by the PISA standard is to pack both the PCI and the ISA bus on a new high density edge connector, very similar to the EISA connector. This edge connector has two rows of contacts. The upper row on each side are ISA signals, while the lower row on each side contains the PCI signals.

As a result, PISA single board computers may use half length cards. This makes PISA based systems even cheaper than PCI-ISA. It also means PISA system with its short CPU card may take half the volume of a PCI-ISA system. This makes PISA systems almost as small as PC/104 based systems.

Not only that, a PISA slot can accommodate both an ISA based CPU card, and a PISA CPU card.

The PISA standard, however, is relatively new, and has far less support than PCI-ISA. It is yet to be seen whether this standard will gain industry acceptance like PCI-ISA.

3.1.3 PMC - PCI Mezzanine Card

The PMC standard [(IEEE, 1995)] uses existing PCI silicon and packages them in a different form factor, suitable for mounting on existing CPU cards, mostly in VME based systems. This allows manufacturers of VME based system to enjoy VME's expandability with the wide selection of PCI based solutions. PMC is defined as an IEEE standard, IEEE P1386.1, and backed by VITA, the VME manufacturers organization.

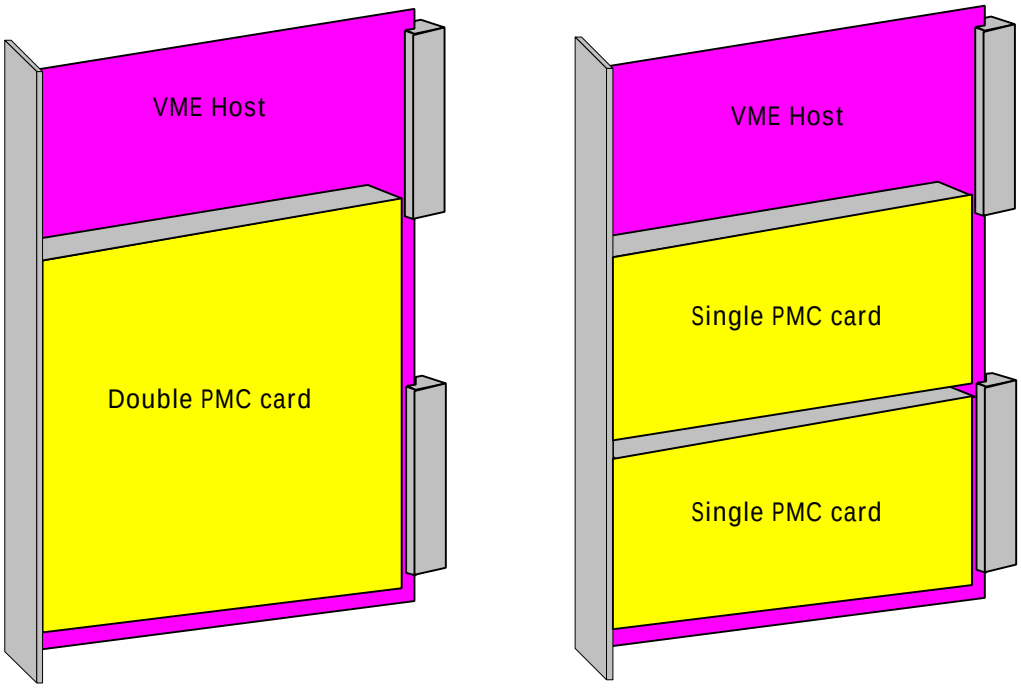


Figure 6 - PMC modules mounted on a VME64 board

PMC is basically an I/O expansion card for single board computers. As shown in Figure 6, two single PMC cards or one double PMC card may be attached in parallel to a VME board. In desktop PCs, PCI cards are placed in perpendicular to the motherboard. PMC provides a low profile configuration for systems that cannot directly use the PCI solution due to space limitations. PMC does *not* replace the VME bus.

3.1.4 CompactPCI

The CompactPCI [(Force, 1997), (PICMG, 1995)] standard attempts to replace VME directly by using PCI chips on a 3U or 6U form factor cards (called Eurocards, see Figure 7). The use of the Eurocard mechanical form factor, common in embedded systems due to the popularity of the VME bus, and the use of card guides and pin-and-socket connectors result in a rugged and reliable package; this is the main reason for the adoption of the Eurocard mechanics in the CompactPCI. An added benefit is that VME and CompactPCI cards, having the same form factor, may be mixed in the same frame. The two buses may be interconnected by a CompactPCI - VME bridge.

CompactPCI uses high density connectors (defined by ANSI as IEC-1076) designed for telecom products. 3U cards use two connectors (see Figure 7). The lower connector contains 32-bit PCI signals. The upper connector is to be used by 64 bit cards. 6U cards use three additional connectors for user defined I/O. Note that the lower two connectors are identically defined in 3U and 6U cards. Hence as far as PCI bus signals are concerned, 3U and 6U cards are electrically interchangeable. Unlike normal PCI, the CompactPCI standard allows up to 8 cards on a backplane without a bridge chip, mostly due to the use of the high quality connector, and a large number of GND pins.

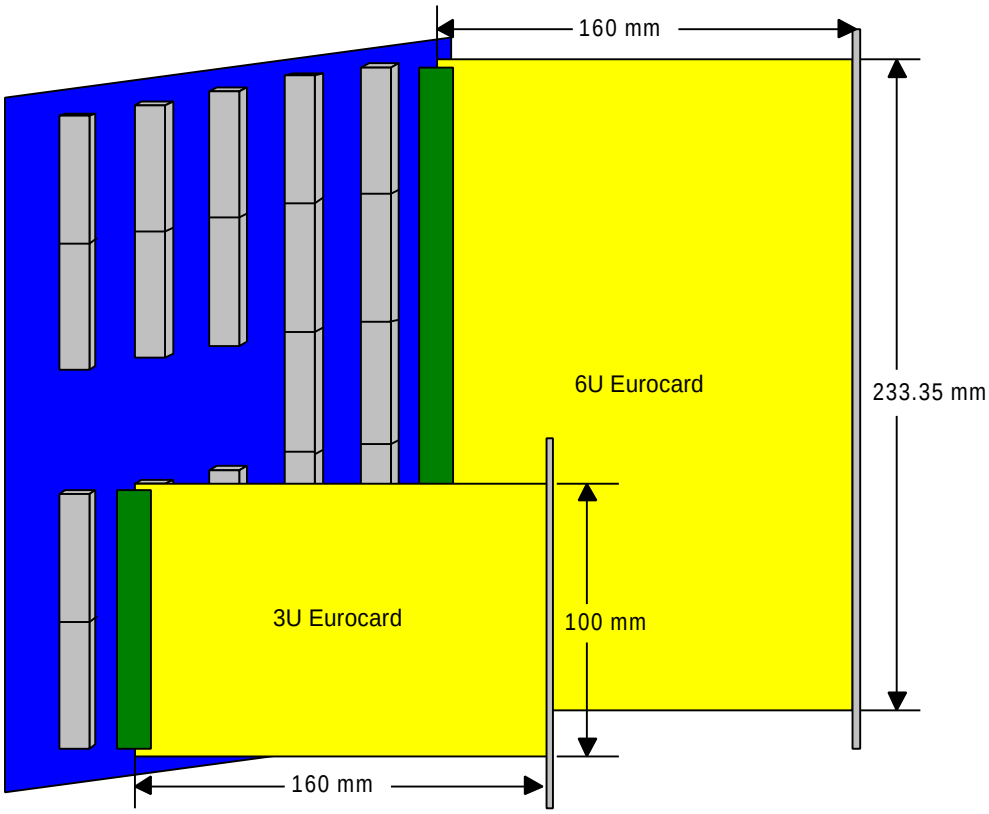


Figure 7 - 3U and 6U form factor Eurocards

In comparison with VME, the CompactPCI solution has the following benefits:

- CompactPCI systems use the same PCI chips and the same software as desktop PC systems. Operating systems, drivers, and applications that run on desktop PCs also run on CompactPCI systems. Hence several operating systems and a large number of applications are available for use on CompactPCI platforms.
- PCI chips are produced in large quantities for the global PC market. As a result, low cost silicon is available and can be used in CompactPCI systems. Due to high integration levels, a PCI chip set consists of only a few chips; this considerably simplifies system design.
- The CompactPCI peak bandwidth is 132 MByte/s for the 33 MHz 32-bit implementation, and 264 MByte/s for the 33 MHz 64-bit implementation; this is identical to the PCI bus peak bandwidth.

3.1.5 PXI - PCI eXtensions for Instrumentation

PXI [(NI, 1997)] is derived from CompactPCI, and related to CompactPCI in the same way VXI relates to VME. PXI was invented by National Instruments, a company specializing in computerized test equipment. PXI was intended to leverage the PCI technology for instrumentation users. This includes test and measurement systems embedded on a CompactPCI card, controlled by a CompactPCI CPU card running Windows 95 or Windows NT. The advantage of PXI over VXI and GPIB is the high speed bus and the lower cost of PCI peripherals.

The standard addresses three areas:

- Mechanical architecture

  The specification defines mechanical requirements for PXI modules and backplanes, including: System slot location, cooling requirements and environmental testing requirements.

- Electrical architecture

  The specification defines electrical requirements for PXI modules and backplanes, including: A reference clock, a trigger bus, a local bus and a star trigger bus.

- Software architecture

  Unlike other busses, the PXI specification requires cards to be delivered with a VISA driver (Virtual Instrument Software stAndard). VISA allows application software to access instrumentation modules in a transparent way, whether they are controlled by IEEE 488, VXI, RS232 or PXI.

PXI use the user signals on the J2 CompactPCI connector, and reassigns some of the signals used only in the CompactPCI system slot such as REQ#[6:0] and GNT#[6:0]. The extra signals are used for the following:

A 10MHz reference clock (with a 100ppm accuracy over all the supported temperature range) is delivered to all PXI modules. The clock and backplane must provide a skew of less than 1ns between slots.

The trigger bus allows intermodule synchronization and communication. The PXI_TRIG[7:0] lines are shared between all modules, and can be used for clocking and triggering independently of the PCI protocol. The trigger bus use PCI signaling levels.

The local bus is a group of 13 signals that are daisy chained between adjacent slots. The behavior of the local bus is user defined, and may be used for transmission of analog signals up to +/-42VDC and 200mA.

The star trigger bus originates from a special Star trigger slot, which is next to the system slot. The Star trigger bus use the local bus, which is unused because the star trigger slot is next to the system slot. The star trigger bus is used to route triggering and clock signals to one or more PXI frames.

3.1.6 HiRelPCI - High Reliability PCI

HiRelPCI [(IEEE, 1996)] is currently under work by the IEEE1996 working group. It uses a similar connector to CompactPCI, and adds features from SCI [(Tving, 1993)] to address a very large number of nodes. The current draft of the proposal specifies two board formats, 6SU and 12SU, both larger than the corresponding VME and CompactPCI 3U and 6U Eurocard formats (see Figure 7).

A unique feature of HiRelPCI is its support for SCI. The proposal extends PCI to include a packet mode in which packets are sent over SCI/Serial Express buses to other components of a distributed system. Packet transmissions are not visible by normal PCI transactions. To enable sustained operation with a single fault, HiRelPCI provides redundancy on several levels: redundant elements (power supplies, processors, or boards), chassis redundancy, backup serial communications on 6SU style boards, and dual PCI and TDM buses on 12SU style buses.

3.2 Embedded Systems: Single-Board Computer Expansion

3.2.1 PC/104-Plus

PC/104 is the embedded system version of the ISA bus. PC/104-based embedded systems incorporate 16- and 32-bit x86 processors running at 16 - 33 MHz clock rates. Many of these systems are upgraded versions of earlier implementations that used 8-bit microcontrollers, and their bandwidth requirements are adequately met by the 5 Mbyte/s ISA (PC/104) bus. The introduction of the Pentium CPU in embedded systems, however, led to the need to extend the PC/104 bus. The result is the PC/104-Plus standard [(PC104, 1997)], which defines a new form factor and a pass-through connector for PCI, but remains compatible with existing PC/104 cards. PC/104-Plus offers the following:

- Compact 3.6” by 3.8” cards.
- Self stacked cards provide expansion without backplanes or card cages.
- The use of pin-and-socket connectors and four corner mounting provide reliable electric connections and good shock and vibration characteristics.

As illustrated in Figure 8, a typical system may consist of PC/104 (ISA) cards and PC/104-Plus (PCI) cards stacked using pass-through connectors. As in other ISA systems, the 104-pin ISA bus is split into two connectors. The 120-pin PCI bus in PC/104-Plus modules is implemented using a high-density (2 mm) connector, hence it takes less space than the ISA connectors. The PCI connector pins are thinner and more vulnerable than the ISA connector pins. Since PC/104 systems do not use card guides, a connector pin "shroud" serves as a guide of the PCI connector and protects the PCI connector pins.

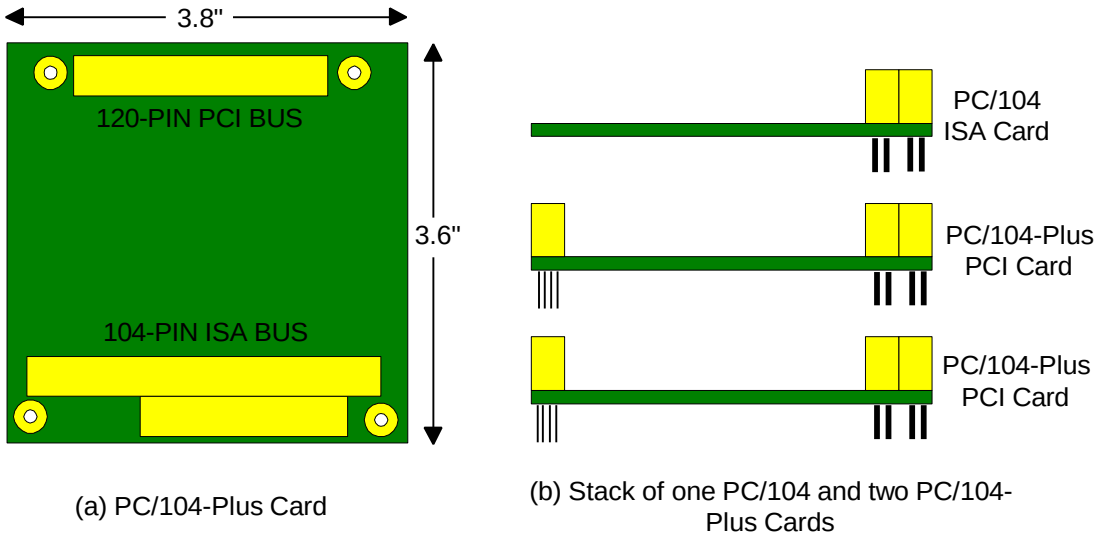


Figure 8 - PC/104 and PC/104-Plus cards

3.3 Laptops and Mobile Systems

As its name indicates, release 1.0 of the PCMCIA (Personal Computer Memory Card International Association) standard was designed for memory cards used primarily in laptop PCs. These Type I cards are about the size of a credit card and are 3.3 mm thick. The next version of the standard (release 2.0) enabled the use of network, modem, and other I/O expansion cards by introducing a higher capacity, 5 mm thick, Type II form factor. Backward compatibility is maintained, Type I cards may be inserted into Type II slots. Release 2.0 also defines a Type III form factor, with a thickness of 10.5 mm, primarily used for hard disk drives. Two vertical Type II slots may contain either two Type II cards or a single Type III device.

The version of the standard released in 1994 (and updated a few times since then) was renamed PC Card [(PCMCIA, 1997)], instead of PCMCIA release 3.0. PC Cards use the same form factor and Type I, Type II, or Type III thickness as PCMCIA release 2.0.

3.3.1 CardBus

The PC Card [(Shanley & Anderson, 1995b)] standard also defines several enhancements; the enhancement pertinent to our discussion is a 32-bit, 33 MHz, interface called CardBus. The CardBus interface is very similar to PCI, but addresses additional issues such as hot plug and power down which are essential in laptop implementations. CardBus cannot use PCI chips directly, but it is very easy to modify existing PCI chips to be CardBus compatible. CardBus uses Type I, Type II, or Type III thickness, the same form factor, and the same connector as PC Card. A PC Card may use a CardBus slot; a connector key, however, prevents a user from inserting a CardBus card into a PC Card slot.

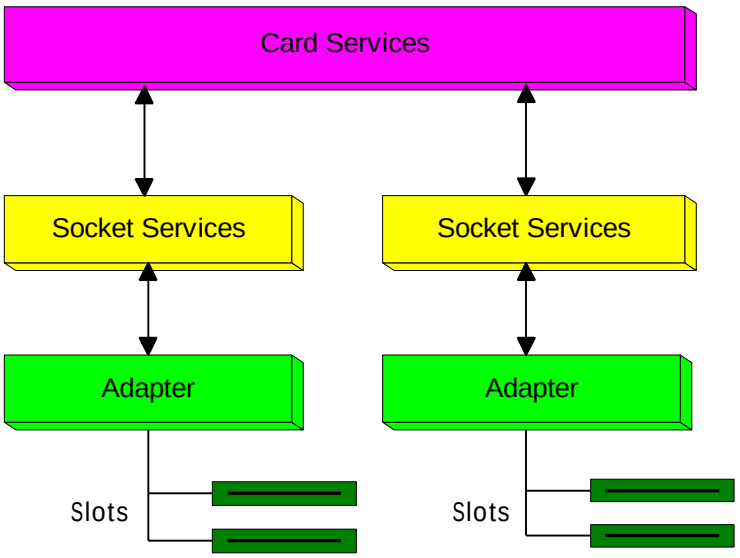


Figure 9 - PC Card and CardBus software support

Each PC Card (or CardBus) slot is connected to an adapter; an adapter can control multiple slots (called sockets in PC Card literature). Since adapters from different vendors often have different hardware interfaces, a software layer called Socket Services provides a generic interface to PC Card adapter hardware. Hence Socket Services hide inconsistencies in the hardware interface. The next software layer, Card Services, enables the use of the same slot for different PC Cards. Card Services manages a pool of resources (such as memory maps, I/O addresses, and interrupt request lines) and assigns resources to PC Cards plugged into the system.

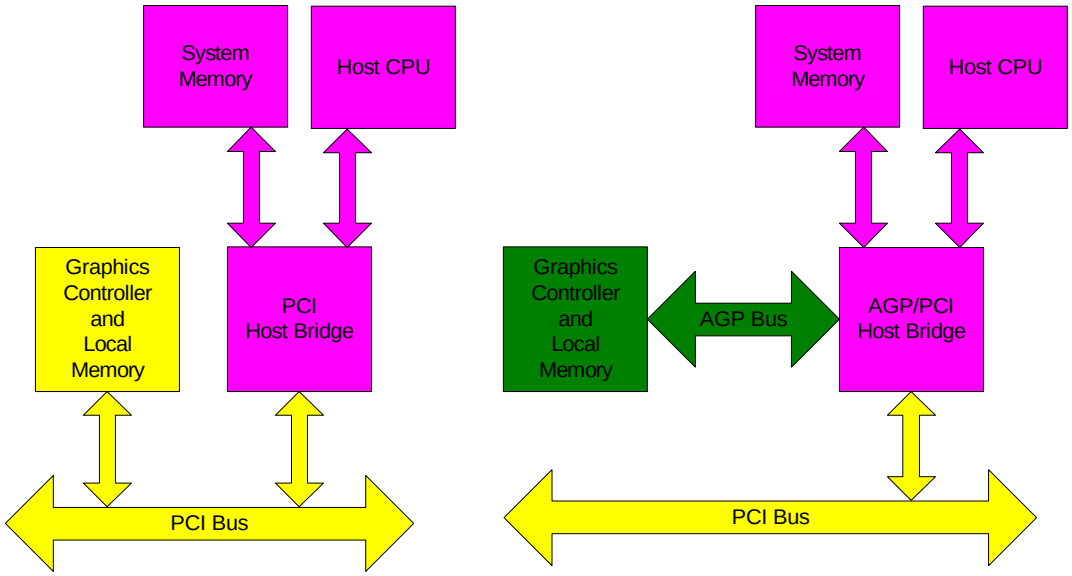
3.3.2 Small PCI

SmallPCI [(PCISIG, 1996)] is a small form factor of PCI endorsed by the PCI SIG, with dimensions and connector very close to those used by CardBus. SmallPCI and CardBus target different market segments, however. CardBus is intended to be a standard for end-user cards and as such supports hot swapping and PCMCIA backward compatibility, but, as we have seen, requires special software support for its operation (socket services). On the other hard, SmallPCI is intended to be used by OEM manufacturers who want to install small functional modules based on PCI chips inside their products. Potential applications include set-top boxes, PDAs, traffic controllers, elevator controllers, navigational systems, and other products that require small form factor packaging.

SmallPCI supports all the features of a standard PCI card, without 64 bit extension, and with the added CLKRUN signal. CLKRUN controls the PCI clock frequency for reduced power mode support. A SmallPCI card has the same form factor as PC Card and CardBus, and is inserted, in parallel to the system board, into a connector mounted on the system board. The connector is keyed to prevent insertion of PC Card or CardBus devices.

3.4 3D Graphics: AGP

AGP (Accelerated Graphics Port) is a relatively new standard [(Intel, 1996)] developed by Intel to provide a high bandwidth graphics expansion slot for PCs. Because of its high speed, AGP is designed to run as a point-to-point protocol; hence it supports a single graphics expansion card. By providing a high bandwidth path between the graphics controller and the system memory (see Figure 10), some of the 3D data structures may be shifted from the local memory of the graphics card into the main memory. The graphics engine may efficiently use both the local memory and the main memory. The result is that demand for additional memory may be satisfied by simply allocating main memory space, without increasing the cost of the local memory on the graphics card.



(a) Without AGP, the Graphics Controller must access the system memory through the PCI bus.

(b) With AGP, the Graphics Controller gets direct access to the system memory through the host bridge logic.

Figure 10 - System Architecture: AGP/PCI vs. PCI only

As shown in Figure 10, AGP does not replace the PCI bus. Essentially, AGP is an additional connection point into the main memory, through the bridge chipset. This path is both logically and physically independent of the PCI bus. Most I/O devices use the PCI bus; AGP is intended for the exclusive use of graphics devices. AGP maintains compatibility with the standard PCI protocol, but uses additional (sideband) signals to add new AGP data transfer modes. With new memory technologies such as SDRAM and Rambus, future memory systems may have enough bandwidth to easily drive both the PCI and AGP busses at their peak bandwidth simultaneously.

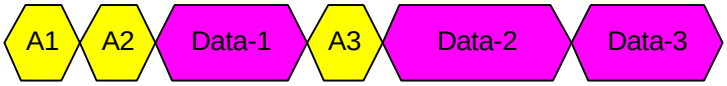
Note the distinction between the PCI bus and PCI transactions. The PCI bus may only run PCI transactions. The AGP bus, however, may run both PCI and AGP transactions, since the AGP standard is an extension of the PCI standard. Traffic on the AGP bus may be a mixture of interleaved AGP and PCI transactions.

Compared with the PCI standard, AGP offers the following benefits:

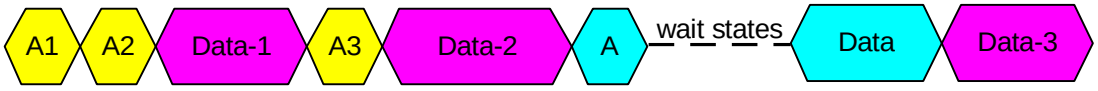
- Optional 133MHz mode, with a sustained throughput of over 500 MByte/sec.
- Demultiplexed address and data transfers. By adding an optional, 8 bit address bus, multiple read requests can be queued up through this bus, while the main address/data bus is used for data transfer, achieving 100% actual bus usage (rather than asymptotic performance level on infinitely long bursts, as in the PCI bus).
- Pipelined memory operations. By decoupling read data transfers from read requests, multiple requests may be initiated without waiting for the completion of the first request, thus hiding the memory access latency.

Pipelined Operation

Pipelined operation enables the graphics controller to insert new requests between data transfers (see Figure 11). At the end of the current data transfer, the data flow may be temporarily suspended and the bus used for either one or more address transfers, specifying one or more new requests, or for a PCI transaction. Transactions are never preempted; the data transfer may be suspended at the end of the current transaction, but not in the middle of it. PCI transactions cannot be pipelined, the data phase must always follow the transaction’s address phase.



(a) Pipelined AGP transactions. Two requests (A1, A2) are followed by a data transfer (Data-1) and a third request (A3).



(b) Interleaved AGP and PCI transactions. A PCI transaction (A, Data) is inserted between AGP data transfers Data-2 and Data-3. The PCI transaction cannot be pipelined, and wait states must be inserted when necessary.

Figure 11 - AGP bus transaction pipelining and interleaving

Demultiplexed Address and Data Transfers

In Figure 11 and in the above discussion, we have assumed that the multiplexed address/data bus is used to perform address transfers, as in the PCI bus. AGP defines a separate, optional, 8-bit address bus, and supports demultiplexed address and data transfers. In conjunction with pipelining, this feature enables 100% use of the main bus for data transfers, even when data transfers are short. (Compare this with the PCI bus, where only long burst transactions approach full utilization of the bus for data transfers.) The width of this sideband address bus is limited to 8 bits to keep the pin count down. Each transfer on the bus consists of two 8-bit phases, for a total of 16 bits.

A full AGP request is broken into three 16-bit parts, referred to as Type 1, Type 2, and Type 3. The encoding of the three types, where `A` is a bit of the address field, `L` is a bit of the transfer length field, and `C` is a bit of the command field is shown below. Note that the address field is split across the three types.

Type 1:	`0AAA`	`AAAA`	`AAAA`	`ALLL`
Type 2:	`10CC`	`CC-A`	`AAAA`	`AAAA`
Type 3:	`110-`	`AAAA`	`AAAA`	`AAAA`

Hence a full request consists of 33 address bits, four command bits (`CCCC`), and three bits (`LLL`) that determine the length of the transfer in 8-byte increments. The address bits are the 33 upper bits of a 36-bit address; memory may be accessed at 8-byte boundaries, and the lower three bits of the address are always zero. Type 1 contains the 12 least significant bits of the address. Once a full request is issued registers retain the Type 2 and Type 3 parts, and as long as these two parts do not change only the Type 1 part has to be transferred for each request. Because of locality, this is common occurrence.

4. Bus Design Principles

4.1 The physics of the backplane bus

In this section we are going to discuss the physical parameters of typical busses based on backplanes. We will show how the bus speed depends on the number of slots, and more specifically on the total capacitance of each signal.

A high speed signal on a backplane bus acts as a transmission line, whose characteristic impedance and propagation delay are given by:

$$Z_0 = \sqrt{L/C} \quad [1]$$

$$t_{po} = l\sqrt{LC} \quad [2]$$

Where:

L	Distributed inductance per unit length
C	Distributed capacitance per unit length
l	Length of the bus signal.

For a typical microstrip backplane, we can calculate the following equations:

$$Z_0 = \left(87/\sqrt{L/C}\right)\cdot \ln\left[5.98h/(0.8w+t)\right] \text{ ohms} \quad [3]$$

$$t_{po} = 1.017\sqrt{0.475e_r + 0.67} \text{ ns/ft} \quad [4]$$

Where:

ε_r	Relative dielectric constant of the PCB material (typically $\varepsilon_r = 4.7$)
r, t, h	See Figure 12.

Where:

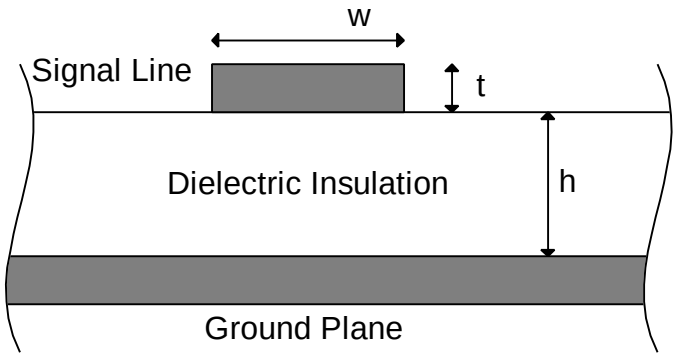


Figure 12 - Cross section of a microstrip bus line

If we substitute typical values of t = 1.4 mils, w = 25 mils, h = 1/16 inch, we get: Z_0=100 ohms and t_{po}=1.7 ns/ft.

These values correspond to an *unloaded* backplane. If we add slots to the system, the total capacitance on every signal is now C_L, and the new values for a loaded backplane are:

$$Z_L = Z_0/\sqrt{1+\left(C_L/C\right)} \quad [5]$$

$$t_{pL} = t_{po} \cdot \sqrt{1+\left(C_L/C\right)} \quad [6]$$

Where:

C_L	Distributed load capacitance per unit length

A typical unloaded backplane has a typical capacitance of 20pf/ft. However, a typical backplane bus has 15 slots per foot. With a typical 3-5pF for each connector, and 10-15pF for the actual signal transceiver, we get C_L=300 pF/ft, and the loaded backplane parameters are now:

$$Z_L = 100 \Big/ \sqrt{1+\left(300/20\right)} = 25 \text{ ohms} \quad [7]$$

$$t_{pL} = 1.7 \cdot \sqrt{1+\left(300/20\right)} = 6.8 \text{ ns/ft} \quad [8]$$

The slow propagation delay is only one problem. Another problem arises, which is the current that needs to be driven by the bus transceiver. A transceiver drives two bus lines (assuming its in the middle of the bus), at a tyical 3V swing. The current required is:

$$I_D = 3V \Big/ \left(Z_L/2\right) = 240mA \quad [9]$$

this is much more than a typical driver can supply.

If the driver cannot drive this amount of current, several round-trip delays to the nearest termination are required for the waveform to cross the receiver threshold region. With each round trip delay at $2 \cdot t_{pL}$=13.6 ns, it may take up to 100ns for the signal to settle down.

There are a few solutions to this problem:

1. Limit the number of slots.
2. Use better bus drivers, with lower capacitance. For a discussion of low capacitance driver implementation, see [(Balakrishnan, 1984)].
3. Use reflective wave signaling. With this method, the bus is *not* terminated. The electrical wave propagates down the bus, reflects off the unterminated end and back to the point of origin, thereby doubling the initial voltage excursion to achieve the required voltage level.

As we will see later, these solutions are used by PCI.

4.2 Synchronous vs. Asynchronous busses

Synchronous buses are called synchronous because all the bus signals are sampled on the edge of a system clock. All signals obey the setup and hold requirements relative to the clock signals. This has the advantage that no handshaking is needed to transfer a single word. Asynchronous buses, on the other hand, are handshaking data transfer without any central clock signal. Control signals can change anytime without any restrictions.

Let us demonstrate this with a small example:

Bus A is an asynchronous bus transferring data in one direction, with REQ# and ACK# active as handshake lines:

1. Source drives data.
2. Source asserts REQ# to signal that the data is ready.
3. Target samples data.
4. Target asserts ACK# to signal it has taken the data.
5. Source deasserts REQ# after ACK# is asserted, otherwise the Target may assume that another transfer is taking place since REQ# is still low.

6. Target deasserts ACK# after REQ# is deasserted, otherwise the Source may assume on the next transfer that ACK# is asserted because of the next transfer when it's in fact asserted from the last transfer.

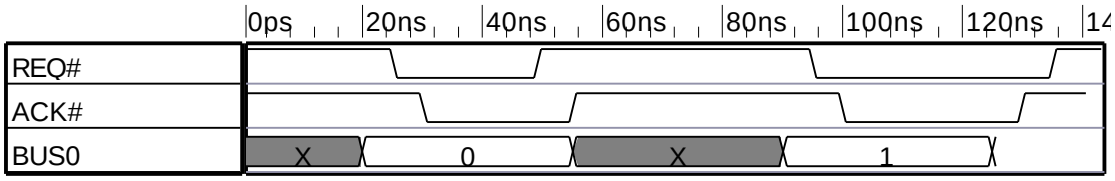


Figure 13 - 4 Phase Asynchronous data transfer

Bus B is a synchronous bus transferring data in one direction, with REQ# and ACK# active as handshake lines:

1. Source drives data and asserts REQ# to be sampled on the rising edge of the next clock pulse.
2. If the Target intends to sample the data on the next clock, it asserts ACK#. Otherwise, ACK# remains deasserted. ACK# will be sampled only on the next clock edge.

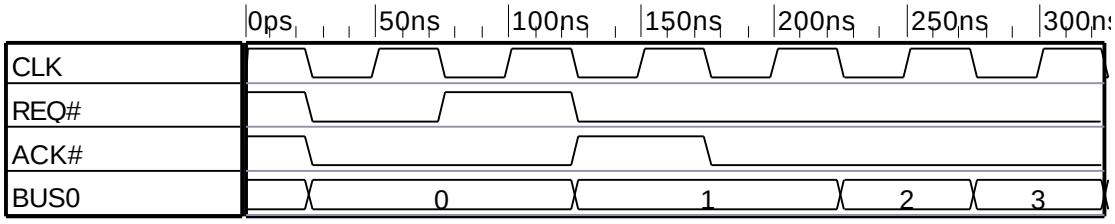


Figure 14 - Synchronous data transfer

On the 1st rising clock edge, REQ# is asserted and the data is ready. Since ACK# is also asserted, data is sampled.

On the 2nd clock rising edge REQ# is deasserted, since data is not ready yet.

On the 3rd clock rising edge REQ# is asserted, but ACK# is deasserted, so data transfer doesn't take place.

On the 4th rising edge, REQ# is still asserted with the previous clock data, but now ACK# is asserted, so data is sampled.

On the 5th rising edge, Both REQ# and ACK# are asserted, so a new data item is sampled.

On the 6th rising edge, Both REQ# and ACK# are asserted, so a new data item is sampled.

4.2.1 The relative merits of Synchronous and Asynchronous Buses

1. Asynchronous bus speed depends only on the speed of the source and the target. A faster target and a faster source means faster transfer rate.
2. Synchronous bus wait states are always an integer multiple of the clock period, so in average, half a cycle is always wasted on a synchronous bus for wait states. A synchronous bus with a 40ns clock cycle, for example, requires 2 cycles i.e. 80ns to access a 50ns RAM. An asynchronous bus might take 50ns plus some overhead caused by handshaking.
3. A synchronous bus samples all data by the clock edge, all the signals must be valid by the clock edge. Since the clock speed is fixed, this usually means that the number of loads the bus can drive is limited since load capacitance may limit the signal switching speed below the clock speed. The PCI bus, for example, is limited to 10 loads. since every expansion slot equals 2 loads (the connector counts as one load), this limitation translates to 4 expansion slots. On-board PCI peripherals count as only one load. It is possible to design synchronous buses with a large number of loads, but it requires a lower clock speed, which is really a waste if most installations uses only 2 or 3 slots. Asynchronous buses, on the other hand, will dynamically adjust to a much larger variety of bus conditions. A 20 slot asynchronous bus may be slower than a 3 slot

asynchronous bus, but it works. **We have demonstrated in section 4 that the speed difference between an unloaded bus and a 15 slot bus may be up to a factor of 4**.

4.2.2 Estimating timing requirements of a synchronous bus

We assume that the bus control logic, which limits the effective data transfer rate of the bus is implemented in a PAL. PALs, or Programmable Logic Arrays, are general purpose, programmable components, used to implement logic equations. This is a simplified model, but is sufficent for us. In [(Kunkel & Smith, 1986)] a much more complex model can be seen, which also takes into account clock and data skew. It also breaks down the flip flop structure into gates for the timing analysis.

A timing module of a PAL in a synchronous mode is:

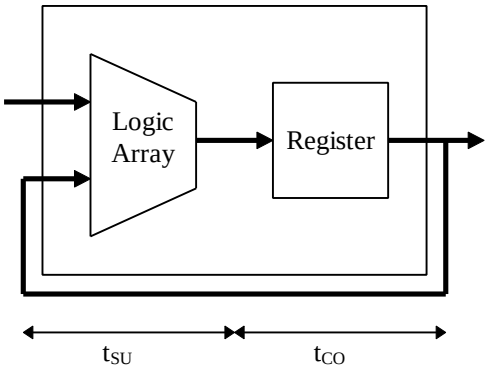


Figure 15 - Synchronous logic timing model

The parameters are:

t_{SU}	Minimum setup time required for the incoming data to be valid prior to the clock
t_{CO}	Maximum guaranteed time from the rising clock edge to output valid.

In a synchronous bus, all the signals are changed on the rising clock edge (and hence are valid after t_{CO}), travel through the bus (we assume a propagation delay of t_{BUS}), and must be valid before the next rising clock edge (hence a setup time of t_{SU}). Since a valid data might be transferred every clock cycle, the maximum data rate is:

$$F_{sync} = \frac{1}{\left(t_{CO} + t_{BUS} + t_{SU}\right)} \qquad [10]$$

4.2.3 Estimating timing requirements of an asynchronous bus

A timing module of a PAL in an asynchronous mode is:

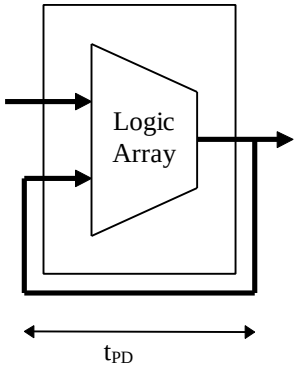


Figure 16 - Asynchronous logic timing model

The parameters are:

t_{PD}	Maximum guaranteed time from data input to data output.

In an asynchronous bus, a bus transaction is completed after N sequential bus events. A bus event can be defined as a change in one of the bus signals causing another bus signal to change. A complete bus transaction can be described as a connected graph made of bus events. In our asynchronous bus example, the bus transaction was made of 4 events (REQ# going low, causing ACK# to go low, causing REQ# to go high, causing ACK# to go high). We can assume that a bus event will take a minimum time of t_{PD}, so the data rate of our asynchronous bus example is:

$$F_{async} = \frac{1}{\left(4 \cdot t_{PD} + 4 \cdot t_{BUS}\right)} \quad [11]$$

In reality, this is a bit more complicated. When ACK# goes high, for example, we can transfer the next word of data. Since REQ# going low qualifies the data as valid, it **must** be asserted after the data is valid, so if it takes t_{DO} to produce the next data word on the bus, then REQ# will go down only after $\max(t_{PD}, t_{DO})$ after the last cycle ended. We can also assume it takes t_{DO} to sample the data on the target, so ACK# will go down only after $\max(t_{PD}, t_{DO})$. So, the data rate is really more like:

$$F_{async} = \frac{1}{\left(2 \cdot t_{PD} + 4 \cdot t_{BUS} + 2 \cdot \max(t_{PD}, t_{DO})\right)} \quad [12]$$

For most PALs, the following rule of thumb holds: $t_{SU} < t_{CO} < t_{PD}$, but: $t_{SU} + t_{CO} > t_{PD}$. Also usually: $t_{PD} < t_{DO}$.

4.2.4 Improved asynchronous bus protocols

The asynchronous bus protocol we have shown is really the slowest form of an asynchronous bus. Here are two simple methods to increase the data rate:

1. We can assume that the source and the target will not miss the REQ# and ACK# assertions, so we make them into pulses, which means that REQ# is deasserted without waiting for ACK# to be asserted, and ACK# is deasserted without waiting for REQ# to be deasserted. This cuts down the data rate to:

$$\frac{1}{\left(2 \cdot t_{PD} + 2 \cdot t_{BUS}\right)} \geq F_{async} \geq \frac{1}{\left(4 \cdot t_{PD} + 2 \cdot t_{BUS}\right)} \quad [13]$$

2. An even better method is to use both edges of REQ# and ACK#, so that we don't need to deassert them at all. For example, when REQ# goes high to low, a word is transmitted. When REQ# goes low to high, the next word is transmitted. ACK# can work the same way. This way we can increase the data rate to:

$$F_{async} = \frac{1}{\left(2 \cdot t_{PD} + 2 \cdot t_{BUS}\right)} \quad [14]$$

This can be shown on the following timing diagram:

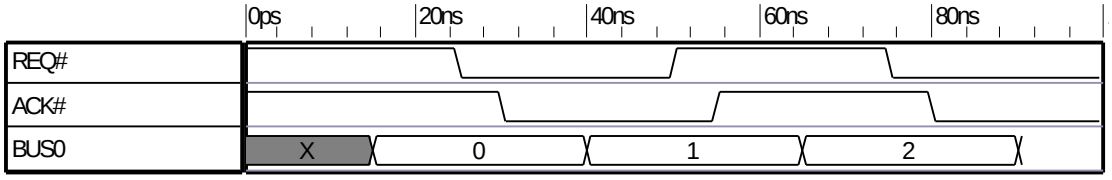


Figure 17 - 2 Phase asynchronous data transfer

4.2.5 Metastability Considerations for Asynchronous buses

Usually, even when asynchronous buses are used, they end up driving a synchronous microprocessor. This means that in practice, the asynchronous bus events must be synchronized to a local clock on each target. This limits performance even further for two reasons:

1. As stated, the time it takes to response to a synchronous event is usually longer than responding to an asynchronous event since $t_{SU} + t_{CO} > t_{PD}$.

2. In order to synchronize an asynchronous signal, it must be sampled by a at least one register, in order to prevent metastability[2]. This adds even further delay. A good example is the MC68020 and MC68030 microprocessors which use an asynchronous bus externally, and synchronizes it internally to the processor's clock, adding one extra clock cycle delay for every input signal. This is one of the reasons the MC68040 uses a fully synchronous bus instead.

4.2.6 Selecting a bus type

As we have demonstrated, both bus types have limitations. An asynchronous bus can adapt well to different number of peripherals on the bus by achieving better performance on a backplane with a smaller load. A CMOS based asynchronous bus can drive a large number of slots or a very long bus (VME is a good example of an asynchronous bus with many slots) without any problems, even at the price of reduced performance. A synchronous bus on the other hand will work faster, but will not work at all if the bus is overloaded or made longer, causing the total propagation delay to exceed the clock cycle time. If a synchronous bus is designed in advance to support a large number of slots or a long backplane, then a worst case design rule dictates lower performance, even when the same bus is not loaded with extra slots, because the clock rate is fixed.

4.3 Synchronous design methodologies

Designing synchronous circuits can be easier when general design techniques are applied. We will review some of these techniques and see how it can help us design synchronous circuits.

4.3.1 The general synchronous models

A synchronous circuit can be described in a general way by the following diagram.

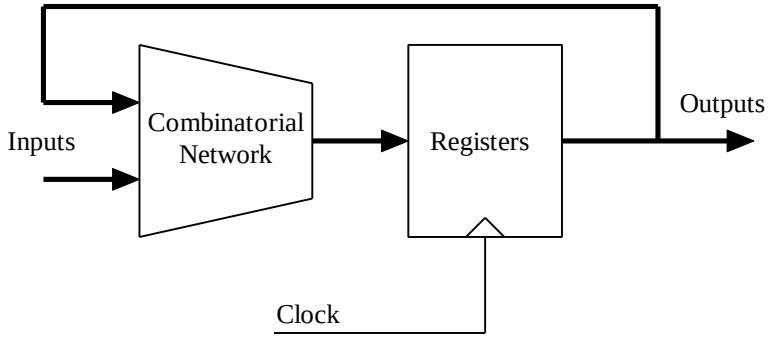


Figure 18 - The general synchronous model

The combinatorial block in the diagram represents a network of basic gates with no feedback paths. Loops may be closed only through synchronous registers. Synchronous circuits work by latching a new state only on a clock edge. This means that as long as the slowest path in the combinatorial network has a faster propagation delay than the clock cycle time, the synchronous timing requirements will be held. If the clock cycle is faster, the setup timing requirements on the register inputs may be violated.

[2] Metastability occurs when a signal that is latched by a clock edge on a register is changing within the defined t_{setup} and t_{hold}. In this case, the register output is **undefined**, and may exhibit unusual effects such as oscillations, for an arbitrarily long time. Metastability cannot be prevented, but it can be statistically reduced to a low probability, well below the expected lifetime of the product.

The maximum clock frequency of the above circuit, is therefore:

$$F_{\max} = \frac{1}{t_{co} + t_{su} + t_{pd}} \quad [15]$$

4.3.2 Synchronous Logic design

Figure 18 represents the general structure of a synchronous logic circuit. We will now provide an example of a typical circuit, as implemented using programmable logic chips.

The original combinatorial block in Figure 18 was broken into multiple combinatorial blocks, representing the basic units supported by the logic devices in use. These blocks are typically a 4- to 6-input arbitrary expression in FPGA chips, while in PLD and CPLD chips these expression might have up to 20 to 36 inputs, but with a limited number of product terms. For a general introduction to CPLDs and FPGAs, see sections 6.1.1 and 6.1.2.

When designing synchronous logic circuits, the circuit must obey a few technical requirements:

1. Setup and hold time requirements on input pins.
2. Clock to output (t_{CO}) requirements on output pins.
3. Maximum path length between registers inside the chip, and between chips.

It is also important to remember that CPLDs, due to their fixed timing model usually include the signal routing delay when calculating t_{PD}, while FPGA timing calculations must include this term separately, since it depends on the specific routing that was done. Different timing can be achieved by placing the same logic at different places across the chip.

4.3.3 Optimizing Synchronous Logic circuits

In the following section, we will show a few techniques to optimize a synchronous circuit. **The prime objective of all these optimizations is to increase the maximum clock rate at which the synchronous circuit can operate**. We will discuss three main techniques for achieving this goal:

1. Retiming.
2. Fanout control
3. Pipelining.

We will draw a sample circuit which will be used to illustrate some of the techniques. The circuit below is a parity generator using both parallel and serial techniques. The circuit front end is a parallel 8 to 1 parity generator, followed by a single bit serial parity generator.

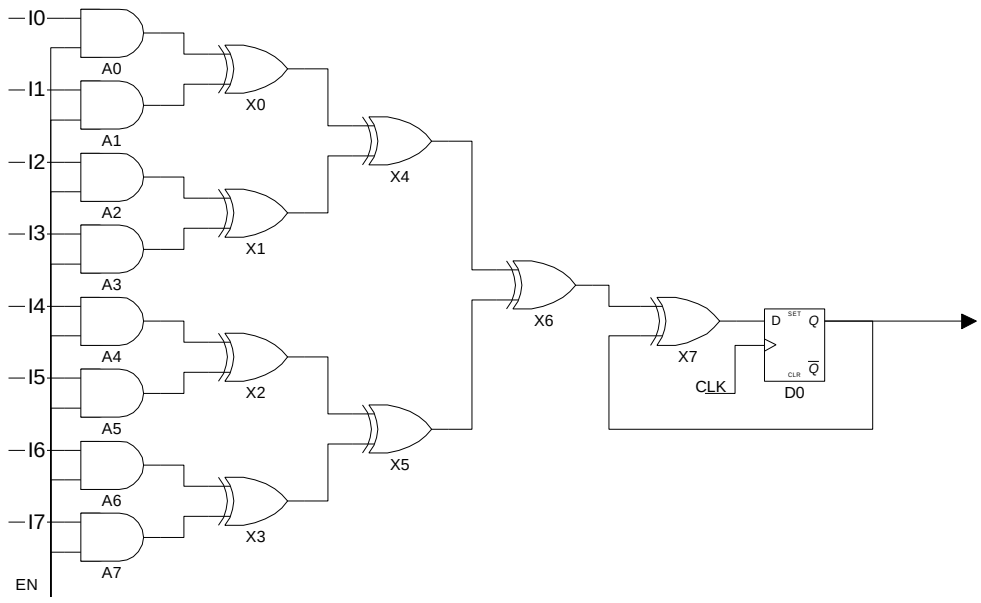


Figure 19 - Parity generator example

As we can see, data is received in groups of 8 bits, which are passing a single combinatorial level (for the enable logic), 3 more combinatorial levels (producing a single parity bit), and then another combinatorial level, mixing the previous parity bit with the new bit. The result is then sampled in the output register. Ignoring routing delays, the maximum frequency of this circuit is:

$$F_{\max} = \frac{1}{\max\left(t_{su} + 5t_{pd}, t_{su} + t_{pd} + t_{co}\right)} = \frac{1}{t_{su} + t_{pd} + \max\left(4t_{pd}, t_{co}\right)} \quad [16]$$

In the next few sections we will see how we can increase this frequency by using the 3 techniques mentioned above.

Retiming

The basic principle behind retiming is balancing the various combinatorial delays in the system. A combinatorial delay begins at the input to a combinatorial gate, which can be either a circuit input (`I0-I7`, `EN`), or the output of an internal register (Output `Q` of `D0`). The combinatorial delay ends at either the circuit output (none here), or at the input of an internal register (Input `D` of `D0`). If we measure all the possible paths and sort them according to their delay, our path delays will range from t_{min} to t_{max}. Unfortunately, the maximum frequency is determined solely by we t_{max}. It is obvious from here that if the total amount of delay in a system is fixed, it is best to distribute it evenly among all paths. By using *retiming* techniques, we can do exactly that. We can short some paths at the expense of some other paths. Retiming we can use the following basic transform:

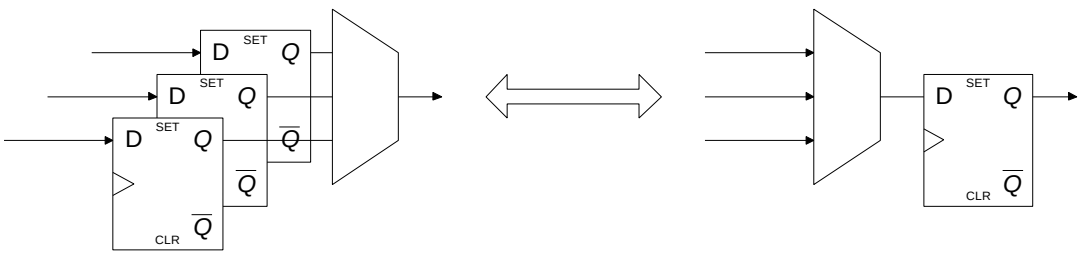


Figure 20 - Retiming transformation

Using this transformation, we can re-arrange Figure 19 like this:

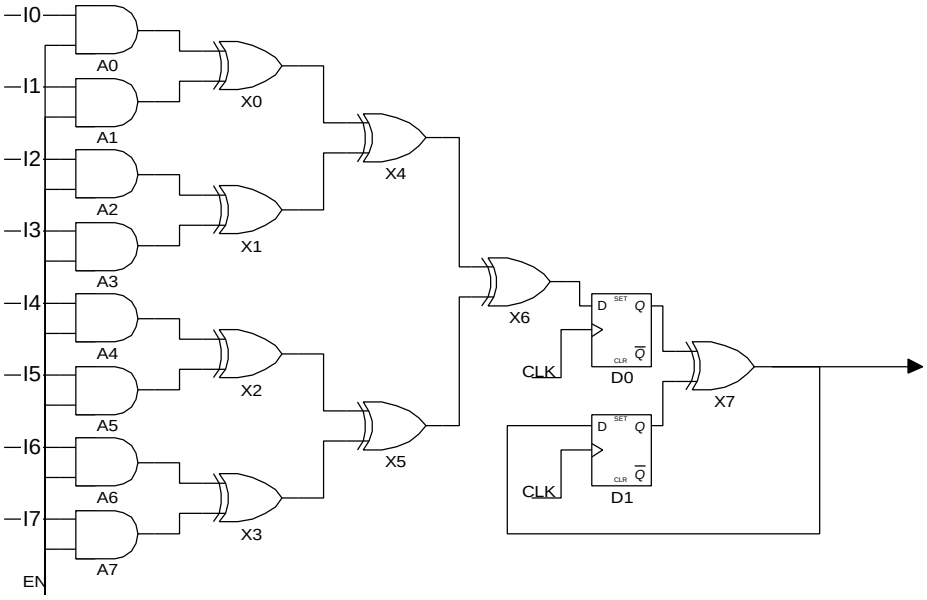


Figure 21 - Parity generator example - retiming example no. 1

We have now reduced the longest path , and now we get:

$$F_{\max} = \frac{1}{\max\left(t_{su} + 4t_{pd}, t_{su} + t_{pd} + t_{co}\right)} = \frac{1}{t_{su} + t_{pd} + \max\left(3t_{pd}, t_{co}\right)} \qquad [17]$$

We have to remember that this is at the expense of the output path which is now slightly slower. We can do this trick again, and end up with:

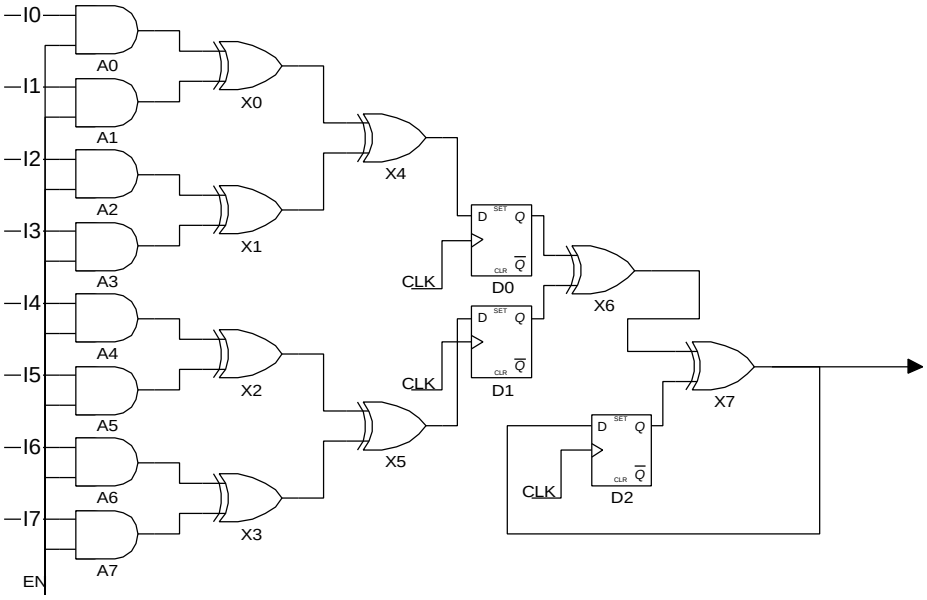


Figure 22 - Parity generator example - retiming example no. 2

We have now reduced the longest path again, at the expense of the output path.

To summarize, we use retiming to redistribute gate delays among different paths, without changing the circuit behavior, provided we use a clock slow enough.

Fanout control

Fanout control is used when a single signal has to drive a large number of inputs. When a large number of inputs are driven from the same signal, there are two main problems:

1. The accumulated capacitance of all the inputs is causing the signal to propagate slower.
2. The signal routing across an FPGA or an ASIC becomes longer.

In order to solve the problem, we usually use the following transformation:

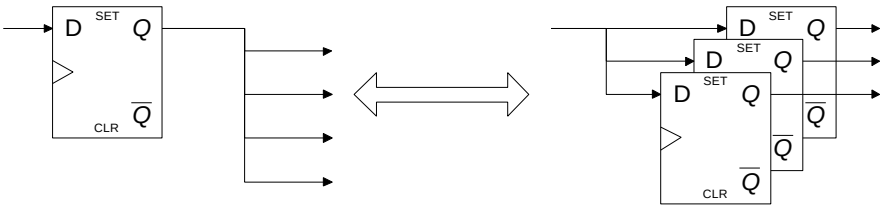


Figure 23 - Fanout control by replicating registers

By using redundant registers, long paths are broken into multiple, parallel paths.

Here is an example of a sample chip layout, and how it is affected by replicating pipeline stages:

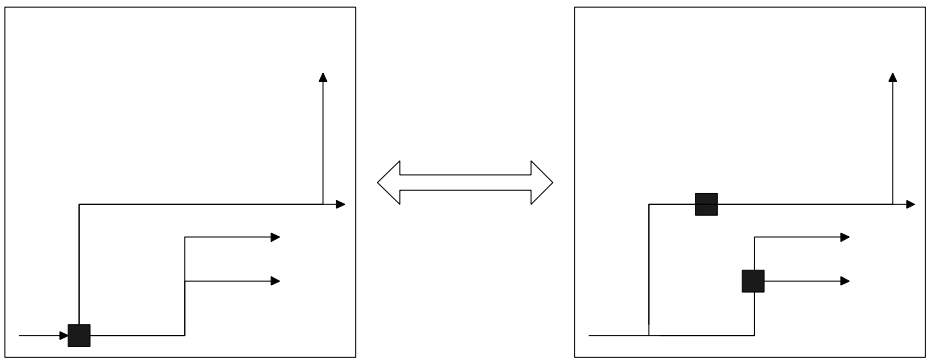


Figure 24 - The effect of register replication on FPGA/ASIC path length

As we can see, the layout on the left has one very short path and four very long paths. The layout on the right has replicated the single register into two separate copies, creating six medium length paths, thus reducing the *maximum* path length over the original layout on the left.

To summarize, we control fanout to redistribute routing delays across more, shorter paths, without changing the circuit behavior.

Pipelining

System performance can be measured by more than one parameter. Latency is the time it takes an input signal to propagate through the system to the output. Throughput is the rate at which the system can process new input signals. For example:

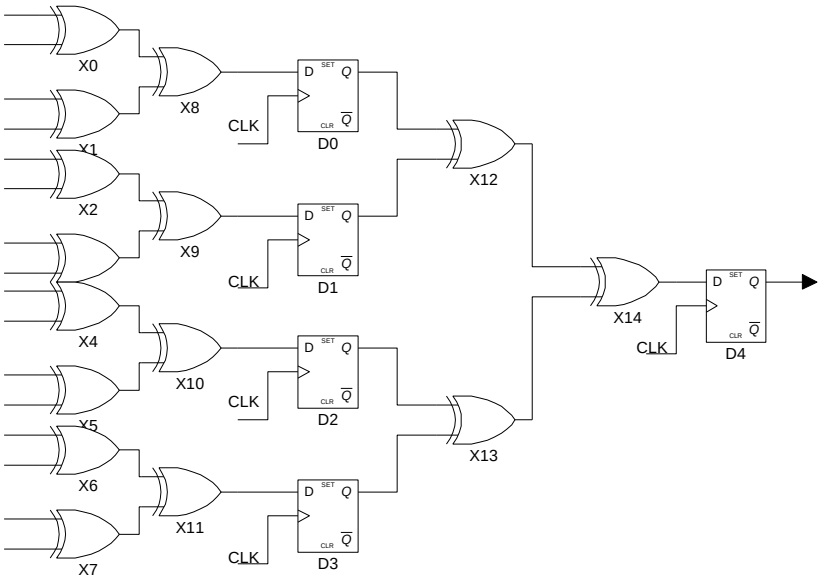


Figure 25- Latency vs. throughput

This circuit can accept new data every $t_{co}+2*t_{pd}+t_{su}$, but the input to output latency is $2*t_{co}+4*t_{pd}+2*t_{su}$. If `D0-D3` were removed, the latency would be shorted to $t_{co}+4*t_{pd}+t_{su}$, but the throughput would drop, and the circuit would accept new data only every $t_{co}+4*t_{pd}+t_{su}$.

If a system has N combinatorial levels, its latency and throughput is N * t_{PD}. When a pipeline stage is introduced, the latency is increased to t_{CO} + N * t_{PD} + t_{SU} but the throughput can drop to $t_{co}+N/2*t_{pd}+t_{su}$, if the pipeline stage is placed after N/2 combinatorial levels. Since t_{PD} is usually only slightly more than t_{CO} + t_{SU}, it usually pays of to introduce pipeline delays for N>2, if the design objective is to increase throughput. If we include routing delays, the effective t_{PD} is even larger, making pipeline even more important.

A pipeline stage can be easily added to any system, by adding a single delay to the system outputs which can be delayed. By using retiming techniques, these delays can be moved backward along some of the paths as needed. Since the system throughput is dominated by the longest path, it is best to distribute the delay among all the paths as evenly as possible, as discussed above,

To summarize, we use pipelining to redistribute routing and gate delays by splitting long paths into multiple, shorter paths, at the cost of changing the system behavior, albeit only by adding a fixed output delay.

4.3.4 Synchronous systems with multi-phase clocks

All the pipeline examples we have shown so far, assumed that a single clock is driving the whole pipeline at exactly the same time. In reality, this becomes an obstacle. Unlike the pipeline which usually uses very short, point to point lines, the clock signal is distributed across the whole pipeline, and it is not always practical to expect a uniform clock without any phase difference across all the pipeline.
Instead, we may use a different approach. We will bisect the pipeline in such a way that some registers (register group A) will be driven by clock A, while the rest (register group B), will be driven by clock B, which is a delayed copy of clock A (obeying a few timing rules). We will make sure that no signal is sampled by both register group A and register group B.
When different clocks are used for two neighboring stages, it is impossible to have overlapping setup/hold timing, so a minimal clock delay must be guaranteed, to make sure that the clock delay is greater than register setup time, and the clock-to-output time. This means that every pipeline stage is updated separately, while the other register is idle. This requirements limits the maximum frequency of such a pipeline, and it's minimum clock cycle time is $2(t_{SU} + t_{CO} + t_{PD})$.
We can illustrate the pipeline in the following schematic diagram:

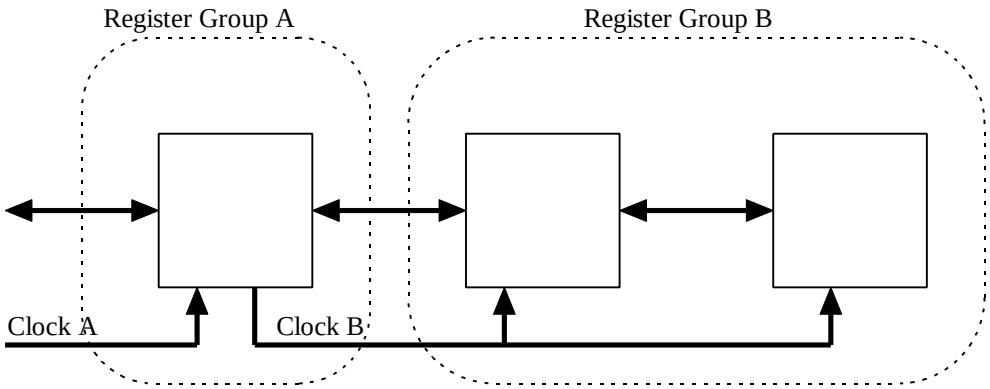


Figure 26 - Grouping different synchronous clock zones

The timing diagram below demonstrates the timing calculations done in order to ensure proper timing margins. Since all registers are the same type, all registers share t_{SU}, t_{CO}. The timing diagram shows two identical clocks, A and B, with a phase difference. **Data_A_Out** drives **Data_B_In** and vice versa. t_{BA} and t_{AB} are the timing margins that must be kept.

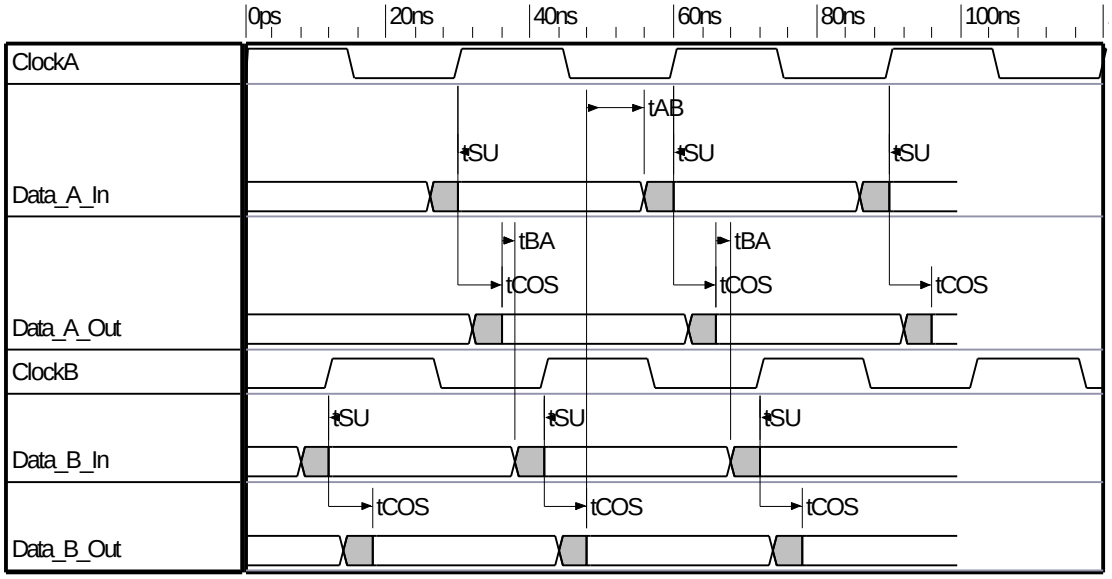


Figure 27 - Timing analysis for multi phase synchronous systems

We can further divide Register Group B into two groups using two clocks, and so on. The result is that we can have up to N different groups as long as any signal in the pipeline drives only one register group.

5. The PCI bus operation

5.1 The PCI Bus Signal Description

The PCI Bus is a multiplexed, synchronous, 32 bit bus. A minimal PCI target interface consists of 47 signals, while a master or master/target interface requires a minimum of 49 signals. The following paragraph summarize the PCI signals. It is not intended to be a PCI design reference, only to illustrate very generally how PCI works. In order to design a fully compliant PCI card, please refer to the PCI standard [(PCISIG, 1995)].

5.1.1 Signal types

The PCI standard defines a few signal types:

- **t/s**
  Tri State. Signals of this type are shared bus signals and may be driven by only one driver at a time. The rest of the bus peripherals should tri-state signals of this type when not in use. When one driver stops driving a t/s signal, a new driver must wait at least one cycle before it can drive the same signal. This is called bus turnaround and is used to prevent any case where two or more drivers are trying to drive the same line.
  Example: **AD[31:0]**, **C/BE#[3:0]**, **PAR**, **REQ#**, **GNT#**, **AD[63:32]**, **C/BE#[7:4]**, **PAR64**.

- **s/t/s**
  Sustained Tri-State. Signals of this type are tri state signals. The difference between these signals and normal t/s signals is that any driver driving these signals must drive them high for at least one cycle before tri-stating them. By driving the signal high, the line is charged so when it is tri-stated one cycle later, it will stay high. This allows rapid switching from one driver to another, without undefined logic levels in between.
  Example: **FRAME#**, **TRDY#**, **IRDY#**, **STOP#**, **LOCK#**, **DEVSEL#**, **PERR#**, **REQ64#**, **ACK64#**.

- **o/d**
  Open Drain. Signals of this type are wired-ORed. Multiple drivers may drive this signal to a low state. When no driver is driving the signal, a pull-up resistor is used to keep it in the high state.
  Example: **SERR#**, **INTA#**, **INTB#**, **INTC#**, **INTD#**.

- **in**
  Input. Signals of this type are always input.
  Example: **CLK**, **RST#**, **IDSEL**, **TCK**, **TDI**, **TMS**, **TRST#**.

- **out**
  Output. Signals of this type are always output.
  Example: **TDO**.

5.1.2 System Signals

CLK
All the PCI Bus signals are fully synchronized to the PCI Clock signal, **CLK**.

RST#
This is the PCI Bus reset signal. It is asynchronous to the PCI clock signal.

5.1.3 Address/Data and Command

AD[31:0] t/s
This is the multiplexed address/data bus. All PCI transactions begins by a Master driving the address on the 1st cycle. When the Master is doing a read transaction, the 2nd cycle is a turnaround cycle in which the Master tri-states the bus and the target enables it's data buffers to drive the results on the bus. The actual data is transferred only beginning on the 3rd cycle onwards. When a Master is doing a

write transaction, it can begin driving the write data on the 2nd cycle, because no turnaround cycle is needed (the bus does not change direction).

C/BE#[3:0] **t/s**

These are dual function pins, and are always driven by the Master. During the 1st cycle, when **AD[31:0]** is driving the address, **C/BE#[3:0]** is driving the bus command (Memory read, Memory write, etc.). When **AD[31:0]** is driving the data, **C/BE#[3:0]** drive the byte enable for each byte lane on the data bus. During a write transaction, the Master drives **C/BE#[3:0]** with information specifying which byte lanes contain valid data. During a read transaction, the Master drives **C/BE#[3:0]** with information specifying which byte lanes are requested by the master.
The byte enable signals are active low and are summarized in the following table:

Byte Enable Signal	Maps To
C/BE#3	**AD[31:24]**
C/BE#2	**AD[23:16]**
C/BE#1	**AD[15:8]**
C/BE#0	**AD[7:0]**

Table 8 - PCI byte enable mappings

The PCI Bus commands are:

C/BE#3	C/BE#2	C/BE#1	C/BE#0	Command Type
0	0	0	0	**Interrupt Acknowledge**
0	0	0	1	**Special Cycle**
0	0	1	0	**I/O Read**
0	0	1	1	**I/O Write**
0	1	0	0	**Reserved**
0	1	0	1	**Reserved**
0	1	1	0	**Memory Read**
0	1	1	1	**Memory Write**
1	0	0	0	**Reserved**
1	0	0	1	**Reserved**
1	0	1	0	**Configuration Read**
1	0	1	1	**Configuration Write**
1	1	0	0	**Memory Read Multiple**
1	1	0	1	**Dual Address Cycle**
1	1	1	0	**Memory Read Line**
1	1	1	1	**Memory Write and Invalidate**

Table 9 - PCI commands

PAR **t/s**

The **PAR** signal used to ensures even parity across the **AD[31:0]** and **C/BE#[3:0]** signals, and is always valid one cycle after the information on **AD[31:0]** is valid, i.e.:

1. One cycle after an address phase (1st PCI cycle).
2. One cycle after a Master asserts IRDY on a write transaction (i.e. data ready to be written).
3. One cycle after a Target asserts TRDY on a read transaction (i.e. data is ready to be read).

5.1.4 Interface Control

FRAME# **s/t/s**

The **FRAME#** signal is used to start a PCI transaction. When the PCI bus is Idle, **FRAME#** is high. When a Master begins a new PCI transaction, **FRAME#** is driven low. **FRAME#** is kept low during all the transaction, until (but not including) the last data item transferred. When a Master read or writes the last data item in a burst access cycle, **FRAME#** will be driven high when **IRDY#** is driven low for the last time.

IRDY# **s/t/s**

IRDY# is used by the Master to indicate its readiness to perform data transfer on a word by word basis. When a Master is reading data, it will drive **IRDY#** high, until it can accept a data word. The Master will then drive **IRDY#** low on the same cycle it samples the data from the Target. When a Master is writing data, it will drive **IRDY#** high until **AD[31:0]** contains a valid data word to be written. It will then drive **IRDY#** low on the same cycle **AD[31:0]** is valid for write. The actual data transfer will take place only when both **IRDY#** and **TRDY#** are active. Once driven low, **IRDY#** cannot be driven high until the current bus transaction is done (i.e. **TRDY#** or **STOP#** also driven low).

TRDY# **s/t/s**

TRDY# is used by the Target to indicate its readiness to perform data transfer on a word by word basis. When a Target is accepting data (Master writing data), it will drive **TRDY#** high until it can accept the data word. The Target will then drive **TRDY#** low on the same cycle it samples the data item from **AD[31:0]**. When a Target is supplying data (Master reading data), it will drive **TRDY#** high until it can drive **AD[31:0]** with the requested data. The Target will then drive **TRDY#** low on the same cycle it drives the requested data item on **AD[31:0]**. The actual data transfer will take place only when both **IRDY#** and **TRDY#** are active. Once driven low, **TRDY#** cannot be driven high until data is transferred (i.e. **IRDY#** also driven low).

DEVSEL# **s/t/s**

DEVSEL# is used by the Target to acknowledge the Master it is handling the current bus cycle. When a Master begins a read/write transaction, it drives **FRAME#** low on the 1st cycle, together with the requested address on **AD[31:0]**. All the Targets on the bus recognize the beginning of a new transaction by **FRAME#** going low, and compare the address on **AD[31:0]** with their base address registers. The Master expects one Target, who's address range contains the address driven on **AD[31:0]** to respond with a low **DEVSEL#** within 4 clock cycles. If no target responds within 4 cycles, the Master assumes that no Target exists on the bus at the specified address, and the transaction is aborted by the Master.

IDSEL# **in**

IDSEL# is used by the Master to perform system configuration. A target will accept a configuration transaction only when **IDSEL#** is driven high on the 1st cycle. Since every PCI slot has a unique **IDSEL#** line, every slot can be uniquely accessed even before it's configured [3] . Empty PCI slots can be identified since no **DEVSEL#** would be driven as a response to a configuration access for that slot.

STOP# **s/t/s**

STOP# is used by the Target to end the current burst read or write transaction. If a Master samples **STOP#** low during a data word transfer, it must disconnect the bus and end the transaction. The state of **TRDY#** and **DEVSEL#** while **STOP#** was driven indicates the termination type: Disconnect with data, Disconnect without data, Target retry , or Target abort.

5.1.5 Arbitration

REQ# **t/s**[4]

REQ# is used **only** by bus Masters. When a Master wants to begin a transaction, it must first acquire the bus by driving **REQ#** low. All the **REQ#** lines from all the PCI slots supporting bus masters are connected to a central PCI Bus Arbiter, which will grant the bus only to one PCI Master at a time.

[3] Note: The PCI standard does **not** define a **standard** way in which a Master can drive a specific **IDSEL** line! On **most** PC based platforms, each slot has its **IDSEL** line connected through a resistor to one of the AD lines, usually beginning with **AD16** for the 1st slot. A Master will select the specific slot configuration space by making sure it drives an address enabling only a single card with **IDSEL**. Driving more than one card with **IDSEL** will cause bus clashes when both cards drives **DEVSEL#**, trying to accept the cycle, and may even lead to hardware malfunctions.

[4] Notice that both **REQ#** and **GNT#** are marked as **t/s**. This is because the signals must be tri stated when the PCI bus is reset.

GNT# **t/s**

GNT# is used **only** by bus Masters. **GNT#** is driven low by the central PCI Bus Arbiter to the PCI Bus Master when the Master is granted the use the shared PCI bus. **GNT#** lines are unique to every PCI slot supporting bus Masters. When a Bus Master receives **GNT#** low, it is allowed to access the PCI bus only after the current cycle taking place is ended. This is indicated by both **FRAME#** and **IRDY#** being high.

5.1.6 Error Reporting

PERR# **s/t/s**

PERR# is used by a PCI card to report parity errors on the data phase. **PERR#** is driven by the Master during a data read, and by the Target during a data write. **PERR#** timing is one cycle after **PAR** is driven with a valid value. (**PAR** is driven by the Target during a data read, and by the Master during a data write). Reporting parity error via **PERR#** is optional, and can be turned off by clearing a control bit in the Target or Master configuration space. Also notice that address phase parity errors should **not** be reported with **PERR#**.

SERR# **o/d**

SERR# is used by the Target or the Master to signal a system error condition, such as parity error during a transaction address phase (The 1st cycle, when the address is transmitted). Unlike **PERR#**, **SERR#** can be driven by any PCI target at any time since it is open drain. It is pulled high by a resistor located on the motherboard, or on another central resource. **SERR#** should be used with care, since on most system today it is meant to indicate a fatal system error which might imply a system reset!

5.1.7 64 Bit Extension

The following signals exists only on 64 bit PCI cards or slots. The signals are available on a secondary connector, similar to the ISA 16 bit extension of the XT bus. Using 64 bit cards on 32 bit slots (in 32 bit mode of course) is permitted, as well as using 32 bit cards in 64 bit slots.

AD[63:32] **t/s**

AD[63:32] are used to hold the extra 32 data bits during 64 bit data transfer, and to hold the extra 32 address bits when accessing a resource located in the 64 bit memory space.

C/BE#[7:4] **t/s**

During the address phase of a 64 bit PCI transaction, **C/BE#[7:4]** are unused. During the data phase, the **C/BE#[7:4]** lines are used just like the **C/BE#[3:0]** lines, indicating which byte lanes are valid.

PAR64 **t/s**

PAR64 is used in the same was as **PAR**, but contains the parity only for the **AD[63:32]** and **C/BE#[7:4]** lines. It obeys the same timing rules as **PAR**.

REQ64# **s/t/s**

REQ64# is used by a 64 bit master to request a 64 bit cycle. It obeys the same timing rules as the **FRAME#** signal.

ACK64# **s/t/s**

ACK64# is used by the target to accept a request for a 64 bit cycle. It obeys the same timing rules as the **DEVSEL#** signal.

5.1.8 Bus Snooping

The bus snooping lines allows a PCI bus master containing a write back cache to maintain the data coherency on systems with multiple bus masters. For example, Bus master "A" contains a modified copy of memory location "X" on his write back cache. In the same time, other bus masters may independently try to modify memory location "Y", sharing the same cache line as memory location "X". Since the cache logic writes back to memory whole cache lines, Bus master "A" must write back it's modified copy of the cache line **before** the other master can write location "Y", because otherwise any new value for "Y" would be overwritten when Bus master "A" would write back its modified cache line containing "X".

SDONE **in/out**[5]

SDONE is used by a bus master containing a write back cache to signal the currently addresses target not to acknowledge the current cycle until the master finishes searching for the specified address in its write back cache. **SDONE** Will be high during the search, and low on the 1st cycle the search is completed.

SBO# **in/out**

SBO# is driven high during the snoop address search (see above). If the address requested is cached in the local write back cache and is dirty, **SBO#** will go low on the same cycle **SDONE** goes high, to signal a back-off command to the target, so the modified cache line can be flushed. If **SBO#** goes high when **SDONE** goes high, the requested address is not in the master's cache, and the cycle can continue normally.

5.1.9 JTAG (IEEE 1149.1)

JTAG is an IEEE standard defining a serial bus for system testing. JTAG is used by many chip vendors for:

1. Board testing
   A PCB connectivity netlist can be verified by bypassing the output and input logic on the chip's I/O pads and drive out test values. By chaining all the chips on a PCB with JTAG, it is possible to test the PCB connectivity after manufacturing or during maintenance by driving values through an I/O pin using the JTAG port of the source chip, and reading those values on another I/O pin using the JTAG port on the target chip.
2. Software debugging of Microprocessors
   Most new microprocessors supports JTAG for the use of a background debug mode. This mode allows reading and writing internal CPU registers, inserting breakpoints, single stepping the CPU and even get a limited back trace. (For example, some members of the Motorola 683XX family). This gives designers most of the capabilities of an expensive ICE at fraction of the price.
3. In-System-Programming of programmable logic devices
   Most of the large pin count FPGAs and CPLDs today use PQFP packages which are soldered directly to the board during manufacturing. When using multiple chips it is very expensive to program these chips on a programmer. It is also impossible to reprogram the devices once they are soldered. Instead, CPLDs and FPGA chips are soldered to the board while they are still blank, and are chained together on the board by a JTAG chain. A single JTAG connector on the board can then be used to program all the chips without using an expensive and unreliable PQFP adapter for off-board chip programming.

Unfortunately, the PCI committee never defied a standard way to access the JTAG port on the PCI motherboard, nor did it define the JTAG bus topology. Because of this, only a few PCI designs supported the JTAG pins on the PCI connector. Some motherboard manufacturers has even went as far as violating the PCI spec and using the JTAG lines for an entirely different purpose such as Video sideband signals!

TDI

TDI is a Test Input pin, and it is used to shift data and instructions into the JTAG port.

TDO

TDO is used to shift data out from the JTAG port.

[5] **SDONE** and **SBO#** are **in** for PCI targets and **out** for PCI masters.

TCK
TCK is the JTAG clock. It is used to control the data shifting in and out of the JTAG port.

TMS
TMS is used to select the JTAG mode.

TRST#
TRST# is used to reset the JTAG port.

5.2 The PCI Bus Commands

As we have seen in section 5.1.3, the PCI Bus has defined 12 different commands and reserved 4 commands. The following paragraph contains a very short description of all the PCI Bus commands. We will take a closer look at the commands later, in section 5.5.

Memory Read
A Memory Read command is used by a PCI bus master to read one or more memory locations from a PCI bus target. It is recommended that Masters use this command when reading less than one cache line.

Memory Read Line
This command is the same as the Memory Read command, with the exception that the bus master is intending to read at least a cache line. This is a hint to the Target, which may use this hint to prefetch more data in advance, or it may choose to ignore it and treat this command in the same way as the Memory Read command. It is recommended that Masters use this command when reading between 1 to 2 cache lines.

Memory Read Multiple
This command is the same as the Memory Read command, with the exception that the bus master is guaranteed to read one or more cache lines. This is a hint to the target, which may use this hint to prefetch more data in advance, or it may choose to ignore it and treat this command in the same way as the Memory Read command. It is recommended that Masters use this command when reading 2 or more cache lines.

Memory Write
A Memory Write command is used by a PCI Bus master to write one or more memory locations to a PCI target.

Memory Write and Invalidate
This command is the same as the Memory Write command, with the exception that the Master is **guaranteed** to write one or more whole cache lines. This command is used to disable the snooping mechanism, because writing an entire cache line means that any dirty data located in any cache is now invalid and shouldn't be written back. When using this command, the master must **never** write incomplete cache lines.

I/O Read
An I/O Read command is used by a PCI bus master to read the one or more I/O locations from a PCI target.

I/O Write
An I/O Write command is used by a PCI bus master to write one or more I/O locations to a PCI target.

Configuration Read
A Configuration Read command is used by a PCI bus master to read one or more Configuration registers from a PCI target.

Configuration Write
A Configuration Write command is used by a PCI bus master to write one or more Configuration registers to a PCI target.

Interrupt Acknowledge
This command is used by X86 platforms to pass the interrupt acknowledge cycles needed when a CPU communicates with a 8259 Programmable Interrupt Controller. The X86 family chips has only one INTR line, and the PIC is using the Interrupt Acknowledge cycle to notify the PCU which IRQ line has been activated. In most PCI based PC motherboards, the PIC chips are NOT on the PCI Bus, and all the Interrupt Acknowledge cycles appear on the CPU local bus and not on the PCI Bus.

Special Cycle
The Special Cycle command is used to send an event to all the PCI Bus targets. The event is identified by an event code, which is allocated by the PCI SIG. Predefined events includes the X86 shutdown and halt events. Unlike other cycles, Special cycles are **not** acknowledged by the PCI bus targets, i.e. **DEVSEL#** is not asserted for this cycle.

Dual Address Cycle
A Dual Address Cycle is used to access a 64 bit address on 32 bit PCI slots. A Dual Address Cycle command is sent together with the upper 32 address bits on **AD[31:0]** followed by the required command (Memory or I/O R/W).

5.3 The PCI Address structure

The PCI bus recognizes 3 types of address spaces: Memory, I/O and Configuration. In the following paragraph we will discuss these addresses spaces and describe the way an address is constructed for the different PCI commands.

5.3.1 Memory address space

The Memory address space is the main address space used by PCI cards. A PCI memory address is 32 bit wide, or optionally, 64 bit wide. 64 bit addresses may be generated either by a 64 bit wide PCI bus master and 64 bit wide PCI target (plugged on a 64 bit wide PCI backplane!), but can also be generated in a 32 bit PCI slot when the Dual Address Cycle command is used. The PCI SIG recommends all the resources occupied by a device to be mapped into the PCI memory space.

During memory read/write commands, the address is interpreted according to the following rules:

AD[31:2] Longword address in the 4GB Memory address space. Since only longword addresses are allowed, target address are always aligned on a longword boundary, i.e. it is impossible to have one target on address 1000h, and a different target on address 1001h.

AD[1:0] Address increment mode, which defines how the next address in a burst sequence is calculated. The address increment modes are indicated the following table:

AD[1:0]	Mode	Description
00	Linear increment mode	The next word's address in the burst will be the next sequential address in memory after the current address.
01	Reserved	
10	Cacheline wrap mode	Same as Linear increment mode, but when the cacheline boundary is crossed, the next address wraps around to the beginning of the cacheline. When the burst length reaches the cacheline size, the next address is incremented by the cache line size into the same cacheline position in the next cacheline.
11	Reserved	

Table 10 - PCI addressing mode for memory read/write commands

5.3.2 I/O address space

The PCI I/O address space is 32 bit wide, but most system does not support more than 16 bits of I/O addresses. The PCI I/O address space is used mostly for backward compatibility with legacy hardware on X86 based systems, and is not recommended for new designs. On most systems, I/O access is slower than memory access, and I/O space is severely limited in the PC architecture. Note: On PCI implementations where no I/O instruction exists, I/O space cycles may be generated by a special mechanism, such as mapping part of the I/O space into a predefined memory area.

During I/O read/write commands, **AD[31:0]** contains the byte address in the 4GB I/O space. It is possible to have I/O targets on a byte boundary. This is really important when more than one peripheral may share addresses on a word boundary. Since data is still transferred by longwords, a target **must** check the individual byte enables and disconnect when a Master tries to access bytes outside the Target's address space.

For Example:
Master drives an I/O write transaction with **AD[31:0]** = 379h. If the next data word is written with **C/BE#[3:0]** = 0000b, it means bytes 378h to 37bh are meant to be written. Since 378h is below the requested Target I/O address, it must disconnect without accepting the data!. This can be summarized in the next table:

AD[1:0]	C/BE#[3]	C/BE#[2]	C/BE#[1]	C/BE#[0]
00b	X	X	X	X
01b	X	X	X	1
10b	X	X	1	1
11b	X	1	1	1

Table 11 - Legal CBE#[3:0] and AD[1:0] combinations for I/O read/write commands

5.3.3 Configuration address space

The configuration address space is a very small memory area, 256 bytes long, that is unique for every PCI bus/device/function triplet in the system. The PCI standard defines a standard header that uses the first 64 bytes of the header, and keeps the remaining 192 bytes user defined. Configuration space is accessed when a configuration command is sent to a card while its **IDSEL** line is active. Unlike other bussed PCI signals, **IDSEL** is uniquely driven into each card (a star topology), so when a card is selected by an active IDSEL signal, it is the only active card in that cycle.

During configuration read/write commands, the address field can be interpreted in one of two ways. Configuration Type 0 is issued from a master which is on the same bus as the target. Configuration Type 1 is issued by a PCI Master trying to access configuration space on a PCI device hidden behind one or more PCI to PCI bridges. Type 1 requests are intercepted by bridge devices, which may turn it into a Type 0 request on the secondary bus, or forward it if the secondary bus is not the final destination bus. We will take a closer look at PCI to PCI bridge devices in section 5.6.

Configuration Type 0:

AD[31:11] These lines are ignored. As we mentioned above, AD[31:16] are used in some designs to generate **IDSEL**.

AD[10:8] Function Number. A single PCI card may incorporate multiple functions, each with it's own configuration space. Since each function has a separate configuration space, it can be configured by a separate driver. This is very important when multiple legacy cards are integrated on a new card, and it is desired to keep the old drivers (for software compatibility).

AD[7:2] Configuration register number. This specifies one of 64 Long words. Only the first 16 registers are defined by the PCI standard. The rest are user defined.

AD[1:0] Always 00b.

Configuration Type 1:

AD[31:11] Reserved.
AD[23: 16] Bus Number.
AD[15:11] Device Number.
AD[10:8] Function Number.
AD[7:2] Configuration register number.
AD[1:0] Always 01b.

5.4 The PCI configuration header

The PCI configuration contains information about the PCI card in a particular slot. Some of these registers are read only, and some are read/write.

31 … 16	15 … 0	
Device ID	Vendor ID	00h
Status	Command	04h
Class Code	Revision ID	08h
BIST / Header Type / Latency Timer	Cache Line Size	0Ch
Base Address Register 0		10h
Base Address Register 1		14h
Base Address Register 2		18h
Base Address Register 3		1Ch
Base Address Register 4		20h
Base Address Register 5		24h
CardBus CIS Pointer		28h
Subsystem ID	Subsystem Vendor ID	2Ch
Expansion ROM Base Address		30h
Reserved		34h
Reserved		38h
Max_Lat / Min_Gnt / Interrupt Pin	Interrupt Line	3ch

Figure 28 - PCI configuration header 0

Here is a short list of some of the fields in this header:

Vendor ID

This read-only field contains a unique Vendor ID identifying the designer/manufacturer of this PCI chip. PCI vendor ID codes are assigned by the PCI SIG (Special Interest Group) for SIG members.

Device ID

This read-only field contains a unique Device ID for this PCI device. This field is allocated by the specific vendor designing/manufacturing the device. An operating system could use the Device ID and Vendor ID fields to select a specific device driver for this chip.

Command

This read/write field contains a 16 bit command register. Some of the bits in this register are reserved, and some are optional. Bits which are not implemented, must be read-only, and return 0 on read. Here is a short explanation of all the different bits in this register.

Bit Location	Description
0	I/O space enable. Writing 1 enables the device response to I/O space access. Defaults to 0 after RST#. If device has no I/O, this bit must be hardwired to 0.
1	Memory space enable. Writing 1 enables the device response to Memory space access. Defaults to 0 after RST#. If the device is not memory mapped, this bit must be hardwired to 0.
2	Master Enable. Writing 1 enables bus master behavior. Defaults to 0 after RST#. If this is a target only device, this bit must be hardwired to 0.
3	Special Cycle Enable. Writing 1 enables response to special cycles. Defaults to 0 after RST#. If special cycles are not used, this bit must be hardwired to 0.
4	MWI Enable. Writing 1 allows a bus master to use the Memory Write and Invalidate command. Writing 0 forces the use of Memory Write command. Defaults to 0 after RST#. If the master does not support MWI, this bit must be hardwired to 0.
5	VGA Palette snooping enable. When 1, device must snoop VGA palette access (i.e. snoop data on bus, but do not respond). When 0, treat like normal addresses. If device is not VGA compatible, this bit must be hardwired to 0.
6	Parity error enable. When 0, Ignore parity errors. When 1, do normal parity error action. If parity checking not supported, this bit must be hardwired to 0. Devices that do not check for parity errors must still generate correct parity.
7	Address/Data stepping enable. If device does address/data stepping, this bit must be hardwires to 1. If device does not do address/data stepping, this bit must be hardwires to 0. If device can optionally do address/data stepping, this bit must be read/write, and default to 1 after RST#..
8	SERR Enable. When 1, SERR# generation is enabled. When 0, no SERR# is generated. Defaults to 0 after RST#. Cant be hardwired to 0 if SERR# not supported.
9	Fast back to back Enable. When 1, master is allowed to generate fast back to back cycles to different targets. When 0, master is allowed to generate fast back to back cycles only to the same targets. This bit is initialized by the BIOS to 1 if all targets are fast back-to-back capable. Defaults to 0 after RST#.
15:10	Reserved. Must return 0 on read.

Table 12 - PCI Command configuration register

Status

This read-only field contains a status register. Some of the bits in this register are reserved, and some are optional. Bits which are not implemented, must return 0 on read. Here is a short explanation of all the different bits in this register.

Bit Location	Description
4:0	Reserved. Must return 0 on read.
5	66MHz Capable. Return 1 if device is 66MHz Capable, 0 if only 33MHz supported.
6	UDF Supported. Return 1 if adapter requires UDF configuration file. Return 0 for normal devices.
7	Fast Back to Back Capable. Return 1 if target is able to accept fast back to back transactions when the transactions are not to the same target.
8	Data parity Error Detected. Implemented by PCI masters only. True if parity error detected, **PERR#** set, parity error response bit is set.

10:9	**DEVSEL#** Timing. 00 for Fast, 01 for Medium, 10 for Slow, 11 is reserved. These bits must represent the slowest timing for all commands except for configuration read and write.
11	Signaled Target Abort. Set by a target when it terminated a cycle with a target abort.
12	Received Target Abort. Set by a master when it detected a cycle ending with a target abort. All masters must implement this bit.
13	Received Master Abort. Set by a master when it detected a cycle ending with a master abort. All masters must implement this bit.
14	Signaled System Error. Set if device asserted SERR#. Must be implemented only by device capable of generating **SERR#**.
15	Detected parity Error. Set if parity error detected. Must be set on error even if parity error handling (bit 6) is disabled.

Table 13 - PCI Status configuration register

Revision ID

This read-only field contains a unique Revision ID for this PCI device. This field is used to identify different versions of the same chip. For example, if different revisions of the chip contained different bugs, a device driver software could use the Revision ID field to select different workarounds to bypass different chip bugs.

Class Code

This read-only field contains a field describing the device type. Some of the device types identify specific types of devices which can then be programmed by a generic device driver (for example, IDE interface, VGA card, PCI to PCI bridge). Some other codes describe a device type, but still require a specific device driver. (A SCSI controller, for example). For a list of class codes please see Appendix C.

Cache Line Size

This read/write field is filled by the BIOS or the operating system by the CPU cache line size. This information is used by master and target devices if they support burst transfers in cacheline wrap mode.

Latency Timer

This read/write field is used to initialize the PCI master latency timer. The latency timer is initialized each time the PCI master device is granted the bus, and begins a countdown when its **GNT#** line is deasserted. When the latency timer expires, the master must terminate its burst. This field can be read-only for masters that does not support burst longer than two words, but must be less than 16. Actually, not all the bits of this register must be implemented. One or more bits (beginning with the least significant bit) can be made to return 0 instead, reducing the timer's granularity.

Header Type

This read-only field contains a unique header type code in bits 6:0. Currently defined values are 0 for the standard PCI header, 1 for PCI to PCI bridge header, and 2 for PCI to CardBus bridge header. Header type codes of 3 and above are reserved. The header type controls the layout of the configuration addresses in the range 10h to 3Fh.

Bit 7 of the header type field specifies whether this device is a multi function device. A value of 1 denotes a multi function device.

BIST

This read/write field controls Built-In Self Test for devices supporting this feature. If not used, this field must contain 0.

Bit Location	Name	Description
7	BIST Capable	Return 1 if device supports BIST, 0 otherwise.
6	Start BIST	Write 1 to start BIST. Device will reset bit back to 0 when BIST complete.
5:4	Reserved	Must return 0
3:0	Completion Code	Return 0 when completes with no errors. Non-zero values are device specific errors.

Table 14 - PCI BIST configuration register

Base Address Register 0 *through* **Base Address Register 5**

These read/write fields are initialized by the BIOS or operating system by writing the desired base address where the PCI device is to be located in the PCI memory map. A PCI device may use one or more base address registers.

CardBus CIS Pointer

This read-only field is used by devices which are both PCI and CardBus compatible. It is used to point to the Card Information Structure for the CardBus card.

Subsystem Vendor ID, Subsystem ID

These read-only fields are used to identify a specific product using a generic PCI chip. This field can be used by device drivers to take advantage of specific features in some cards using generic PCI chips. Subsystem Vendor ID values are assigned by the PCI SIG. Subsystem ID values are unique for every Subsystem Vendor ID. This field is optional in PCI 2.1, but required according to Microsoft's PC97 and PCI 2.2 (to be released soon). It must contains 0 if not used. When implemented by a PCI chip, there is usually a way to set this field externally, most of the time through an external memory chip connected to the PCI device. This way PCI board vendors can set this without making their own PCI device.

Expansion ROM base Address

This is a special type of Base Address Register pointing to the card's expansion ROM. Expansion ROMs are optional, and may contain additional information about the card, boot code (in more than one executable format, as well as the architecture independent OpenBoot format).

Interrupt Line

This read-write field is initialized by the BIOS or Operating system, and will contain a system specific value identifying which hardware interrupt has been assigned to this card. This value is not used by the card itself, but rather by the driver and operating system which reads this field to determine which interrupt vector has been assigned for the device. This field is not used by devices not using interrupts.

Interrupt Pin

This read-only field defines which interrupt pin is used by this device. The values contained this field are 1 to 4, corresponding to **INTA#** to **INTD#**. Devices not using interrupts should put 0 in this field.

Min_Gnt, Max_Lat

These read-only fields are used by the BIOS to calculate the desired value for the latency timer of this device. The Min Gnt field measures, in 250us units, the minimum burst length this device requires. The Max_Lat field describes, in 250us units, how often the device requires access to the PCI Bus.

5.5 Basic PCI Cycles

5.5.1 PCI Memory or I/O Read Cycle

The following timing diagram demonstrates a typical PCI memory or I/O burst read transaction.

1. At Clock 0 the Master begins the transaction by driving **FRAME#** low, the requested address on **AD[31:0]**, and the requested command on **C/BE#[3:0]** (I/O Read, Memory Read, Memory Read Line, or Memory Read Multiple).
2. At Clock 1 the Target responds by asserting **DEVSEL#** low. Since clock 1 is used for bus turnaround on read transaction (Master stops driving **AD[31:0]**, Target starts driving **AD[31:0]**), actual data transfer must begin only on cycle 2, so **TRDY#** and **IRDY#** must be high, since no data is available yet.
3. At clock 2 both **IRDY#** and **TRDY#** are low, so the 1st data word is read (D0).
4. At clock 3 the Master is not ready to accept the new word, since **IRDY#** is high. The Target is ready to transfer the next word since **TRDY#** is low. Since not both are low, no data is transferred.
5. At clock 4 the master is able to receive data again, since **IRDY#** is low, and the 2nd data word is transferred. Since **FRAME#** was high during this last cycle, both the Master and the Target end the cycle.
6. At clock 5 the Target tri-states **AD[31:0]** (turnaround), and drives **TRDY#** high. The Master also drives **IRDY#** high. This is done since **IRDY#** and **TRDY#** are sustained-tri-state lines, and must be driven high for one cycle prior to tri-stating them. This is done in order to charge the line, which will cause it to stay at a high level even after it is tri stated.

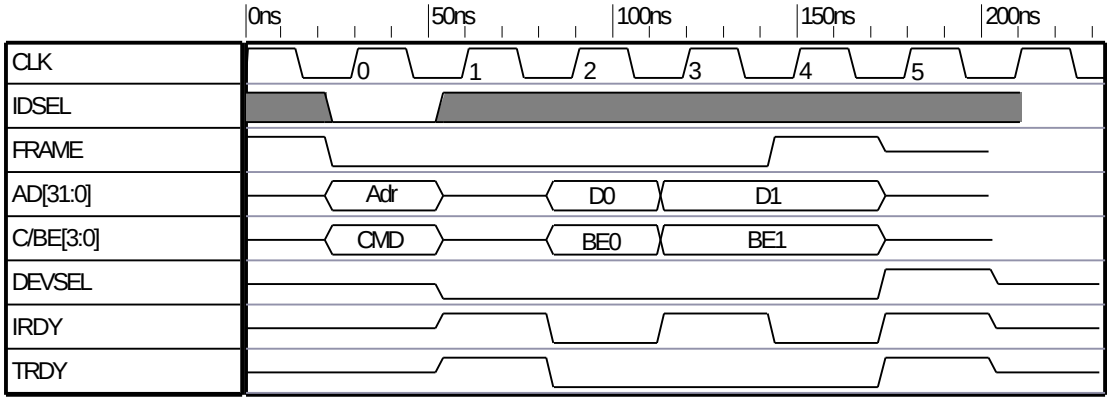


Figure 29 - PCI Memory and I/O read cycle

5.5.2 PCI Memory or I/O Write Cycle

The following timing diagram demonstrates a typical PCI memory or I/O burst read transaction.

1. At Clock 0 the Master begins the transaction by driving **FRAME#** low, the requested address on **AD[31:0]**, and the requested command on **C/BE#[3:0]** (I/O Write, Memory Write, or Memory Write Line).
2. At Clock 1 the Target responds by asserting **DEVSEL#** low. Since this is a write transaction, the master keeps driving **AD[31:0]** (this time with data), and no bus turnaround cycle is needed. **IRDY#** is driven low to indicate data is ready for write. Since the Target is driving **TRDY#** high, it is not ready to accept the data. **C/BE#[3:0]** are loaded with the appropriate byte enables.
3. At clock 2 both **IRDY#** and **TRDY#** are low, so the 1st data word is written (D0).
4. At clock 3 the Master is not ready to send a new word (as indicated by the invalid content of **AD[31:0]**), so **IRDY#** is driven high. The Target is ready to accept the next word since **TRDY#** is low, but since **IRDY#** is high, no data transfer takes place.
5. At clock 4 the master is able to write data again, since **IRDY#** is low, and data is transferred. Since **FRAME#** was high during this last cycle, both the Master and the Target end the cycle.

6. At clock 5 the Target drives **TRDY#** and **DEVSEL#** high. The Master also drives **IRDY#** high. It may even start a new transaction right now (this is called back to back transfers, when there are no turnaround cycles between multiple transactions. Back-to-back transfers can only happen after a write transaction, when no turnaround cycles are needed).

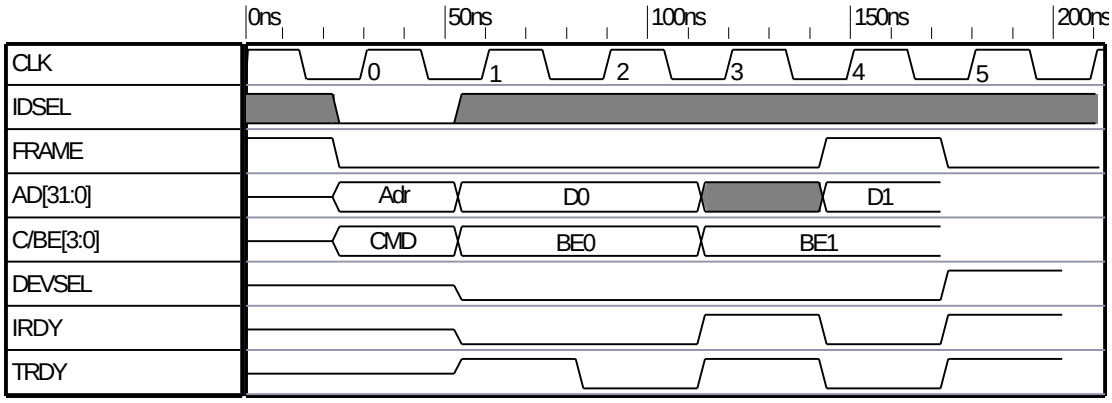


Figure 30 - PCI Memory and I/O write cycle

5.5.3 PCI Configuration Read Cycle

The following timing diagram demonstrates a typical PCI configuration type 0 read transaction. Since no address increment is defined for configuration cycles, burst behavior is undefined for configuration space, and therefore not used. Usually, the target will disconnect after the 1st word.

Configuration type 1 cycles do not use **IDSEL**, and are identical to ordinary read cycles, but do not support bursts, like type 0 configuration cycles.

1. At Clock 0 the Master begins the transaction by driving **FRAME#** low, **IDSEL** high, the requested address on **AD[10:0]**, and the requested command on **C/BE#[3:0]** (Configuration Read).
2. At Clock 1 the Target responds by asserting **DEVSEL#** low. Since clock 1 is used for bus turnaround on read transaction (Master stops driving **AD[31:0]**, Target starts driving **AD[31:0]**), actual data transfer must begin only on cycle 2, so **TRDY#** and **IRDY#** must be high, since no data is available yet.
3. At clock 2 both **IRDY#** and **TRDY#** are low, so the 1st data word is read (D0). Since **FRAME#** was high during this last cycle, both the Master and the Target end the cycle.
4. At clock 3 the Target tri-states **AD[31:0]** (turnaround), and drives **TRDY#** high. The Master also drives **IRDY#** high. This is done since **IRDY#** and **TRDY#** are sustained-tri-state lines, and must be driven high for one cycle prior to tri-stating them. This is done in order to charge the line, which will hold it at a high level even after it is tri stated.
5. At clock 4, all the sustained tri state lines driven high before (**TRDY#**, **IRDY#**, **DEVSEL#**, **FRAME#**) are now tri-stated.

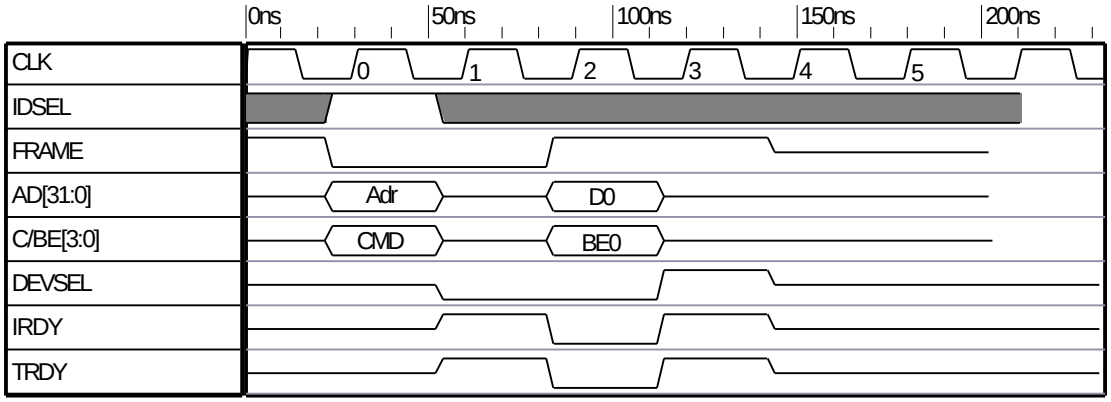


Figure 31 - PCI Configuration Read cycle

5.5.4 PCI Configuration Write Cycle

The following timing diagram demonstrates a typical PCI configuration write transaction. Since no address increment is defined for configuration cycles, burst behavior is undefined for configuration space, and therefore not used. Usually, the target will disconnect after the 1st word.

Configuration type 1 cycles do not use **IDSEL**, and are identical to ordinary write cycles, but do not support bursts, like type 0 configuration cycles.

1. At Clock 0 the Master begins the transaction by driving **FRAME#** low, **IDSEL** high, the requested address on **AD[10:0]**, and the requested command on **C/BE#[3:0]** (Configuration Write).
2. At Clock 1 the Target responds by asserting **DEVSEL#** low. **IRDY#** is driven low because the master is ready, but **TRDY#** is high, because the target is not ready yet. **FRAME#** is driven high since this is the last word for this burst.
3. At clock 2 both **IRDY#** and **TRDY#** are low, so the 1st data word is written (D0). Since **FRAME#** was high during this last cycle, both the Master and the Target end the cycle.
4. At clock 3 the Target tri-states **AD[31:0]** (turnaround), and drives **TRDY#** high. The Master also drives **IRDY#** high. This is done since **IRDY#** and **TRDY#** are sustained-tri-state lines, and must be driven high for one cycle prior to tri-stating them. This is done in order to charge the line, which will hold it at a high level even after it is tri stated.
5. At clock 4, all the sustained tri state lines driven high before (**TRDY#**, **IRDY#**, **DEVSEL#**, **FRAME#**) are now tri-stated.

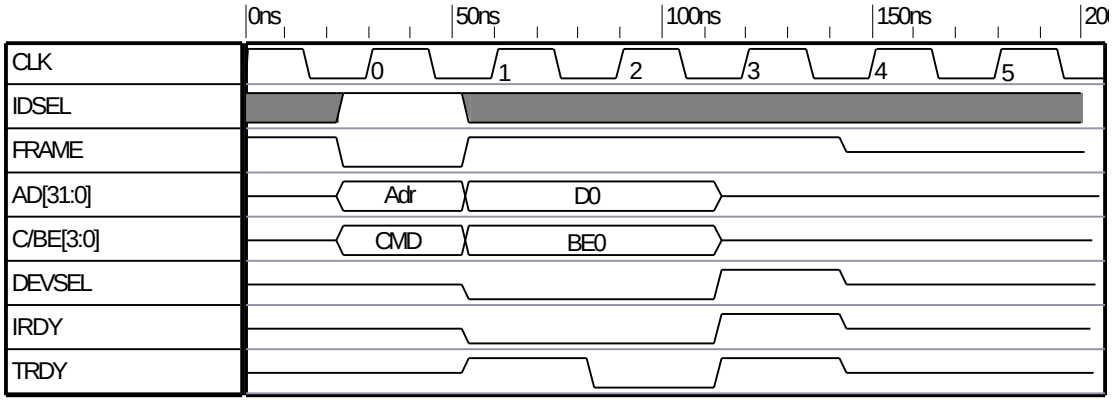


Figure 32 - PCI Configuration Write cycle

5.6 Abnormal cycle termination

The sample cycles we have shown so far were always terminated by the master by driving **FRAME#** high during the last data word transfer. In this section, we will show how PCI cycles can end in different ways.

All the figures in this section apply equally to all data transactions types. i.e. memory, I/O or configuration, as well as for read or write transactions. The data bus, on the other hand was drawn from the perspective of a read transaction (a new data word is valid after **TRDY#** is asserted, not when **IRDY#** is asserted), but all the information should equally apply to write transaction as well.

5.6.1 Target termination (Disconnect with data)

When a target can no longer sustain a burst, it can assert **STOP#** during the same cycle it asserts **TRDY#**. It must keep **STOP#** asserted until **FRAME#** is deasserted. **TRDY#** is deasserted after **IRDY#** is asserted, to prevent transferring another word. **FRAME#** will be deasserted in the same cycle **IRDY#** is asserted.

Target termination is used for a graceful transaction termination without any errors. The master may (or may not) continue the same burst by starting a new transaction at the subsequent address after the last one which was transferred.

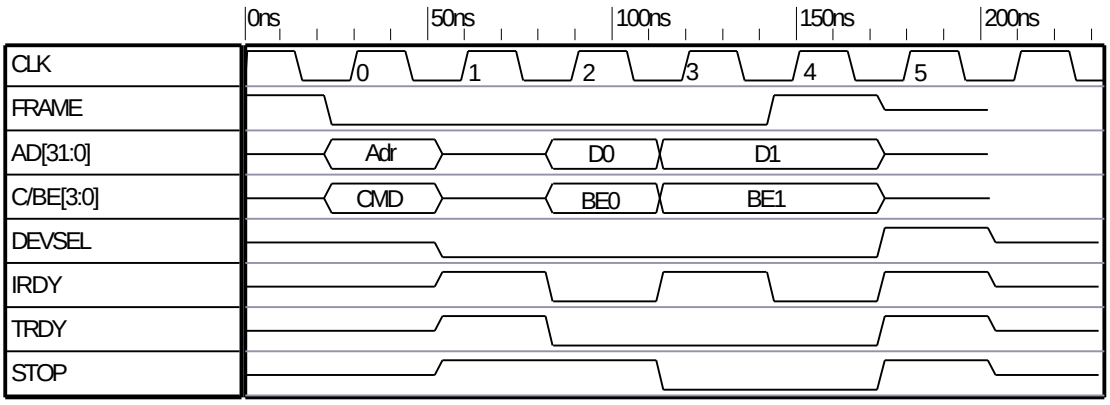


Figure 33 - Target Disconnect with data

5.6.2 Disconnect without data

When a target cannot supply the next word in a burst, it can assert **STOP#** and deasserts **TRDY#**. It keeps **STOP#** asserted until **FRAME#** is deasserted. **TRDY#** is kept deasserted until the end of the cycle, and can be tri-stated when **FRAME#** is high. If **FRAME#** was low, **IRDY#** is must be asserted at the same cycle **FRAME#** is deasserted.

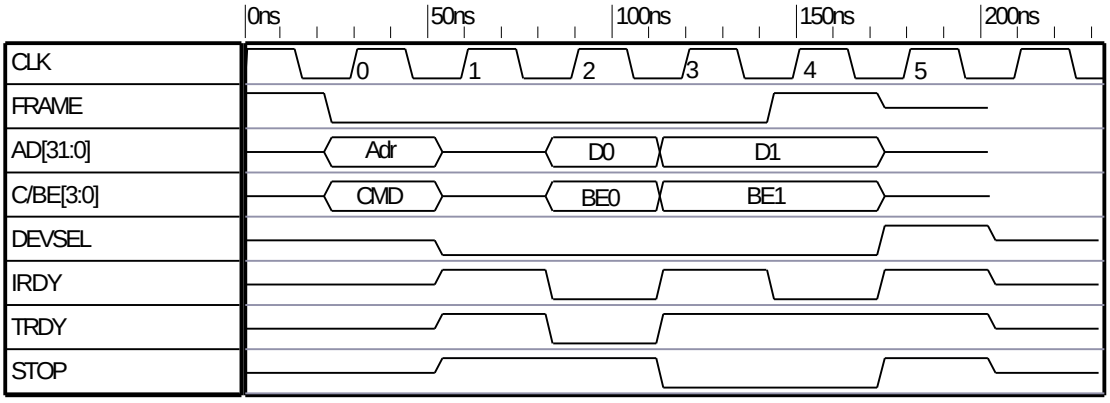


Figure 34 -Target disconnect without data

5.6.3 Target Retry (Disconnect without data)

Target retry is a special case of Disconnect without data occurring at the first word.

Target retry is used to signal the master that the target is not ready yet, but will be ready later. The master **must** retry the same transaction over and over until it succeeds. The master also must release the bus (by deasserting its **REQ#**) for at least two clock cycles between retry attempts to let other bus masters share the bus.

5.6.4 Target abort

When a serious error condition has occurred, the target can assert **STOP#** and deasserts **DEVSEL#** and **TRDY#**. It keeps **STOP#** asserted until **FRAME#** is deasserted. **TRDY#** and **DEVSEL#** are kept deasserted until the end of the cycle, and can be tri-stated when **FRAME#** is high. If **FRAME#** was low, **IRDY#** is must be asserted at the same cycle **FRAME#** is deasserted.

Target abort is used to stop a transaction only when the transaction has failed and will never succeed (for example, trying to write a read only memory). The master **will not** retry the transaction again.

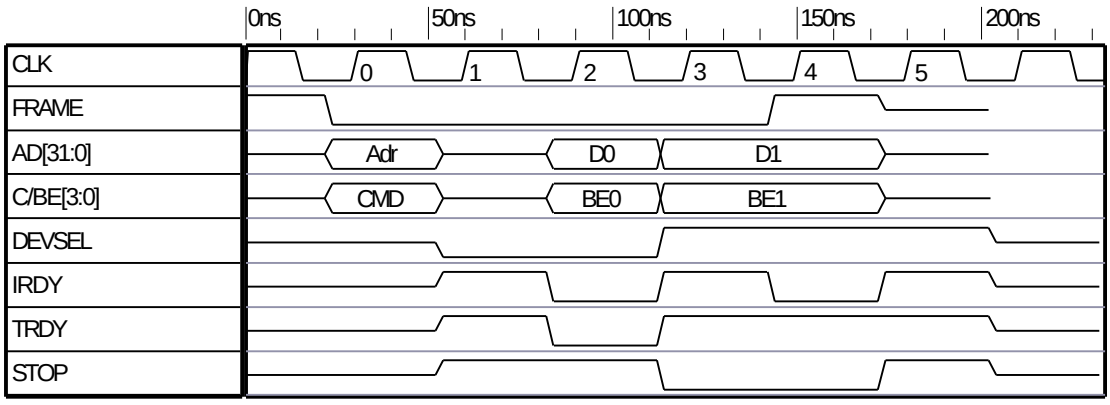


Figure 35 - target abort

5.6.5 Master abort

Another serious error condition occurs when the master begins a cycle, but none of the targets acknowledges it within 4 cycles. When this happens, the master will abort the cycle by deasserting **FRAME#** and ending the transaction.

5.7 The PCI to PCI Bridge

As we already noted, the PCI bus standard is limited to a small number of PCI slots. The main PCI mechanism to expand a PCI system beyond the limited number of slots is the PCI to PCI Bridge. A PCI bridge has two busses, a primary bus and a secondary bus. The two busses can be completely independent, as much as running on two different clock rates.

The PCI to PCI bridge appears as both a master and a target to both busses. When accessed as a target from the primary bus, any access falling within the bridge address space would be regenerated on the secondary bus. When accessed as a target on the secondary bus, any address falling outside the secondary bus address range would be regenerated on the primary bus. Notice that the secondary bus will not respond to configuration type 0 commands.

Very large PCI systems may have up to 255 busses arranged this way.

The PCI to PCI bridge has a slightly different configuration header, called header type 1. The header structure, as well as the register description is listed below.

PCI to PCI bridge are complex devices and we cannot cover all their functions. For a complete specifications of the PCI to PCI bridge devices, please refer to [(PCISIG, 1994)].

31	24	23	16	15	8	7	0	
Device ID				Vendor ID				00h
Status				Command				04h
Class Code							Revision ID	08h
BIST		Header Type		Latency Timer			Cache Line 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								34h
Expansion ROM Base Address								38h
Bridge Control				Interrupt Pin			Interrupt Line	3ch

Figure 36 - PCI configuration header 1

Here is a short list of some of the fields in this header. We mention only the fields which are different from a normal PCI device.

Primary Bus Number

This read/write field contains the bus number to which the primary PCI interface is connected. It is filled by the BIOS when the system boots.

Secondary Bus Number

This read/write field contains the bus number which is linked by the secondary PCI interface. It is filled by the BIOS when the system boots.

Subordinate Bus Number

This read/write field contains the highest bus number that is behind this bridge. If no more bridge chips are located on the secondary bus, this field will be initialized to the same value as the secondary bus number. It is filled by the BIOS when the system boots.

Secondary Latency Timer

This read/write field contains the latency timer value for the secondary PCI bus.

I/O Base register

This read/write field contains the lowest I/O address that must be forwarded by this bridge, if encountered on the primary PCI bus. The upper 4 bits of this register corresponds to **AD[15:12]**, while **AD[11:0]** are assumed to be 0. The lower 4 bits of this field are interpreted according to Table 15. If I/O cycles are not forwarded by the bridge, this register must be hardwired to 0. When set to 32 bit I/O addressing, the value of **AD[31:16]** is contained in the I/O Base Upper 16 bits register.

IO_BASE[3:0]	I/O Addressing Capability
0h	16 bit I/O addressing
1h	32 bit I/O addressing
2h-0fh	Reserved

Table 15 - PCI bridge IO_BASE decoding

I/O Limit Register

This read/write field contains the upper I/O address that is forwarded by this bridge downstream. This register is formatted in the same way was the I/O base register. When set to 32 bit I/O addressing, the value of **AD[31:16]** is contained in the I/O Limit Upper 16 bits register.

Secondary Status Register

This is the PCI status register for the secondary bus. It is identical to the standard status register except for bit 14, which has been redefined as "Received System Error". SERR# events are never generated by a bridge on the secondary bus, but are forwarded to the primary bus, if generated on the secondary bus.

Memory Base Register

This read/write field contains the base address of a memory mapped I/O range that is forwarded by this bridge downstream. The upper 12 bits correspond to **AD[31:20]**. The memory window must be 1MB aligned, and the lower 4 bits of this register must return 0 when read.

Memory Limit Register

This read/write field contains the upper address of a memory mapped I/O range that is forwarded by this bridge downstream. The upper 12 bits correspond to **AD[31:20]**. The memory window must be 1MB aligned, and the lower 4 bits of this register must return 0 when read.

Prefetchable Memory Base Register

This read/write field contains the base address of a prefetchable memory range that is forwarded by this bridge downstream. The upper 12 bits correspond to **AD[31:20]**. The memory window must be 1MB aligned, and the lower 4 bits of this register are read only. If they are 0, this base register supports only a 32 bit address range. If the lower 4 bits are 1, this base registers supports a 64 bit memory range, and the upper 32 bits are stored in the Prefetchable Base Upper 32 bits register.

Prefetchable Memory Limit Register

This read/write field contains the upper limit of a prefetchable memory range that is forwarded by this bridge downstream. The upper 12 bits correspond to **AD[31:20]**. The memory window must be 1MB aligned, and the lower 4 bits of this register are read only. If they are 0, this limit register supports only a 32 bit address range. If the lower 4 bits are 1, this limit registers supports a 64 bit memory range, and the upper 32 bits are stored in the Prefetchable Limit Upper 32 bits register.

Prefetchable Base Upper 32 bits Register

This read/write field contains the upper 32 bits of the base address of the prefetchable memory area when in 64 bit mode. See the Prefetchable Memory Base register.

Prefetchable Limit Upper 32 bits Register

This read/write field contains the upper 32 bits of the limit address of the prefetchable memory area when in 64 bit mode. See the Prefetchable Memory Limit register.

I/O Base Upper 16 bits Register

This read/write field contains the upper 16 bits of the base address of the I/O area when in 32 bit mode. See the I/O Base register.

I/O Limit Upper 16 bits Register

This read/write field contains the upper 16 bits of the limit address of the I/O area when in 32 bit mode. See the I/O Limit register.

Expansion ROM Base Address Register

This register is identical to the normal PCI register with the same name, but is located in a different offset.

Bridge Control Register

This register provides extensions to the command register that are specific to PCI to PCI bridges. The bridge control register provides many of the same controls for the secondary interface that are provided by the command register for the primary interface. There are also some bits that affect the operations of both interfaces of the PCI to PCI bridge. Here is a description of the bits in this register:

Bit Location	Description
0	Parity Error Response Enable. Same as the Parity Error enable bit in the command register, but for the secondary interface.
1	SERR# Enable. When set to 1, allows forwarding of SERR# events on the secondary bus to the primary bus.
2	ISA Enable. When 1, disable forwarding of I/O addresses when **AD[9:8]** are 0. These addresses are I/O addresses used by on board ISA devices.
3	VGA Enable. When 1, forward all the memory and I/O addresses of VGA compatible devices, regardless of the memory and I/O address ranges and the ISA Enable bit. VGA addresses are: Memory: `0A0000h` to `0BFFFFh` I/O: `3B0h` to `3BBh`, and `3D0h` to `3DFh`.
4	Reserved. Return 0 when read.
5	Master Abort Mode. When 0, do not report master aborts (return FFFFFFFFh on read, and discard data on write). When 1, report master aborts by signaling target abort on the requesting bus if possible, and **SERR#** if enabled.
6	Secondary Bus Reset. When 1, forces **RST#** on the secondary bus. When 0, releases **RST#** on the secondary bus. If primary bus **RST#** is asserted, secondary bus **RST#** is asserted as well.
7	Fast Back to Back Enable. Same as the Fast Back to Back enable bit in the PCI control register, but for the secondary bus.
15:8	Reserved. Return 0 when read.

Table 16 - PCI bridge control register

6. Implementation of PCI devices

6.1 PCI Implementation using CPLDs and FPGAs

PCI cards are usually implemented by one of three approaches:

1. Using an ASIC which includes a PCI interface core as well as other functionality.

   This approach is used for high volume applications where cost is the driving factor, and where a large gate count is needed to implement the design. In these applications, the PCI interface core is usually small compared with the custom logic.

2. Using a standard PCI interface chip.

   This approach is less cost effective than using an ASIC (for high volume applications), but it is practical when there is a need to interface existing chips to the PCI bus (for example, a DSP board for PCI), with small amounts of custom logic. These chips are available from a few companies with many models, some offering full master/target capabilities, and some offering target only. Due to their general-purpose nature, however, they may not offer the best performance for all applications.

3. Implementing a PCI interface using a CPLD or an FPGA.

   This approach is used when prototyping ASIC designs, or when there are special requirements from the PCI interface precluding the use of a standard PCI interface chip, or when low quantities are desired, precluding the use of an ASIC. for example, a design requiring a very low latency response may use a CPLD instead of a standard PCI interface chip.

Although the standard does not demands this explicitly, a PCI interface on an add-on card can only be implemented using a single chip. There are two main reasons for this:

- PCI signals can not drive more than electrical load on every PCI interface. This means that if multiple chips are used, the PCI signals must be divided among the chips.
- Some PCI signals such as TRDY and IRDY require the PCI interface to react to changes within one clock cycle. A TRDY/IRDY signal sampled at t-7ns must affect the output signals at t+11ns. Since the PCI clock is not necessarily running in a fixed frequency, IRDY and TRDY can only be sampled on the clock rising edge (so the guaranteed 7ns setup time can be used only for that). This leaves barely 11ns for the signal to be processed by two chips. This requirement is still impractical with the current generation of CPLDs and FPGAs.

These rules of thumb does not apply in the case of a PCI device on the motherboard, for a few reasons:

- Onboard PCI devices may use more than one device load (all Intel's PCI chipsets do) because the motherboard designer can take into account the extra load used by the onboard logic, and reduce the number of available PCI slots if required. For example, if 2 PCI loads are used by the onboard PCI logic, 8 loads are still available, which means that 4 PCI slots can be added to the motherboard.
- Onboard devices may assume a fixed frequency PCI clock, allowing 18ns for TRDY/IRDY processing instead of 11ns.

Implementing a custom PCI interface chip can be done with a CPLD or an FPGA. The following is a summary of the major differences between CPLDs and FPGAs and how these differences affect their use for implementing a PCI interface. Also, [(Brown & Rose, 1996)] is a good survey of popular CPLD and FPGA architectures.

6.1.1 General structure of a CPLD

A CPLD (which stands for Complex Programmable Logic Device) can be generally described as an architecture containing multiple PLD blocks (usually 2 to 16) which are fed from a central routing matrix. All the feedbacks from the PLD blocks are fed directly to the switch matrix instead of being fed directly to the PLD blocks. I/O pads are connected to some or all of the Macrocells on the PLD blocks. Macrocells which are not connected to I/O pads are called buried nodes. Buried nodes are distributed evenly among all the PLD blocks. Small CPLDs can be used to replace a few PLDs, while large CPLDs are suitable for very complex state machines that has to run at high frequencies.

A CPLD may have N PLD blocks, with M feedbacks (from I/O pins and buried nodes) going into the global routing matrix. In a 100% routable CPLD, every PLD block would have M*N inputs from the global routing matrix, making the CPLD effectively a single giant PLD with M*N nodes. This is impractical due to the size of the global routing matrix. Instead, only a subset of K < M*N signals are routed into each PLD block. Different CPLD architectures usually have different characteristics:

1. Different number of macrocells in each PLD block, different number of feedbacks, different number of input terms.
2. Different product term allocation methods between macrocells in the same PLD block. Every architecture has a basic number of product terms allocated to each macrocell, but it can use more product terms by borrowing them from neighboring macrocells or from another global resource (at the PLD block level).
3. Different macrocell structure features such as TFF/DFF/latch mode, input registers, dedicated XOR product term, asynchronous set/reset inputs, clock arrangement, etc.

Different members of the same CPLD family are usually differentiated by the number of PLD blocks in each chip (usually from 2 or 4 in the small chips to 8 or 16 in the large chips). The rest of the architecture usually stays the same.

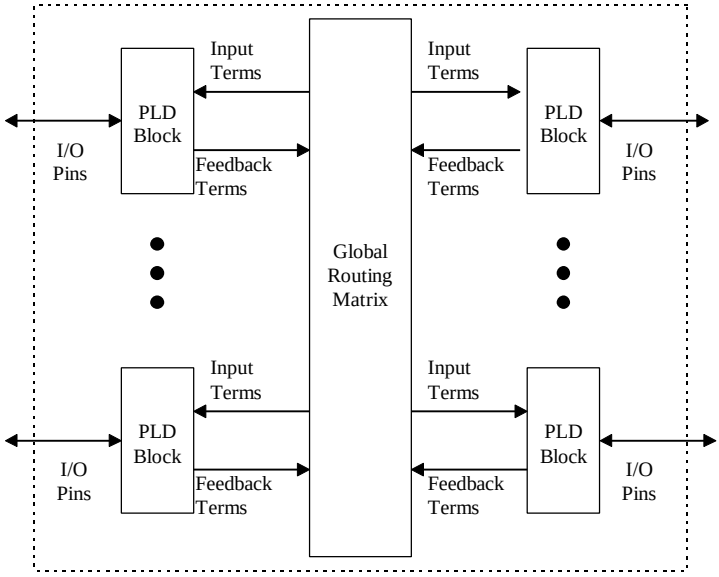


Figure 37 - A general CPLD model

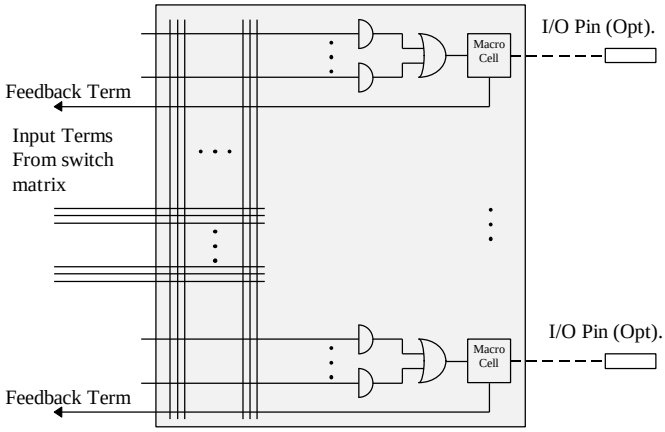


Figure 38 - PLD Block structure

One of the major drawbacks in most of the CPLD architectures is I/O pinout locking. When I/O pins are assigned for a specific CPLD design (when the design is sent for PCB layout or when the design is wire wrapped), successive place and route runs may not be able to fit the design using the same I/O pinout, and some I/O pins may need to be unlocked from their current I/O pin, and re-assigned new I/O pins. There are a few possible reasons for that phenomenon:

1. Modifying an equation which previously required L unique input terms from the global routing matrix may now require L+1 input terms. If the PLD block already used the maximum number of K input terms, some of the equations inside the block must be moved to a different PLD block, changing the CPLD I/O pin assignment.
2. CPLD macrocells do not use a fixed number of product terms per macrocell like PLDs. Instead, CPLD macrocells have a relatively low number of product terms (4 to 5), but they can "borrow" extra product terms from neighboring macrocells or from another common pool of product terms. If an equation is modified in such a way that it requires more product terms than before, it may no longer fit in the same PLD block, requiring I/O pinout changes.

6.1.2 General structure of an FPGA

FPGA, which is a acronym for Field Programmable Gate Arrays can be best described as a square array containing a large number of small logic blocks, each containing 2 to 12 inputs and 1 to 4 flip-flops. These logic blocks are usually locally connected in 4 directions, but some global routing resources are available in most of the FPGA architectures. I/O pads are usually connected to the logic blocks on the four edges of the array. FPGAs are suitable for register rich designs that are not required to run at very high speeds.

Due to the long delays caused by the FPGA routing resources, it is very common to use fanout control techniques, and duplicate a register driving a large number of inputs. Some tools offers automatic control over this feature, by specifying the maximum number of inputs to be driven by a single register output.

The timing constraints caused by FPGA routing makes it very hard to meet the PCI timing requirements. A careful analysis must be performed to choose the correct way to implement the synchronous logic depending on the specific timing characteristics of the specific FPGA family, considering the routing resources, the circuit floorplan design, and the macrocells timing parameters. In most cases, manual floor planning might be needed to lock critical registers in optimal positions.

A closer look at the general FPGA architecture shows that one of the big FPGA drawbacks, caused by it's lack of predictable route has a good side. Since each possible net can be routed through many different routes, it means that pins can still be connected to the same register, regardless of it's location in the FPGA.

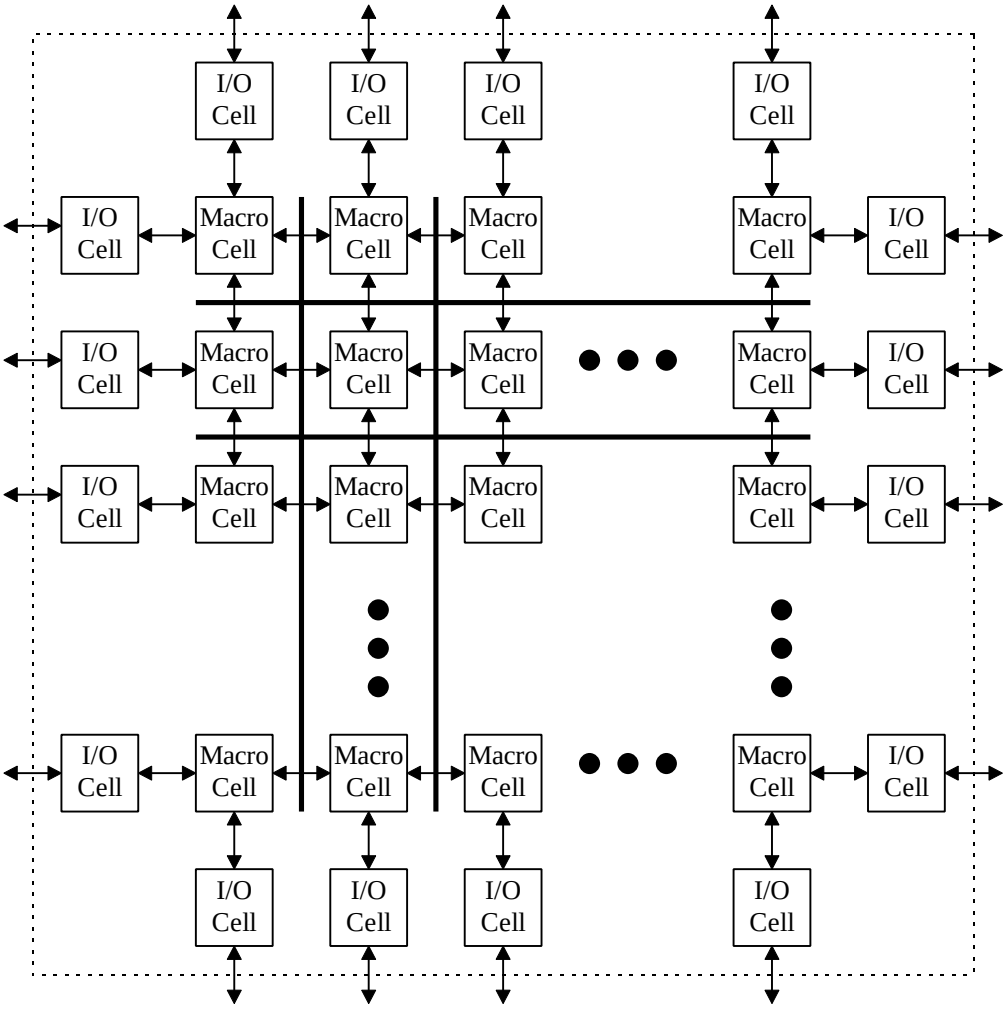


Figure 39 - A general FPGA model

6.1.3 Comparing FPGAs with CPLDs

The following table contains a summary of the main differences between CPLDs and FPGAs. The numbers represent typical architectures available at this time, and should only be used to show the relative difference between competing CPLD and FPGA architectures.

Feature	CPLDs	FPGAs
Registers (typical)	**32 to 512**	**256 to 6144**
Timing	**Fixed, predictable**	**Unpredictable**
I/O pin locking	**Hard to maintain**	**Easier to maintain**
Routing time	**up to 30 minutes**	**5 minutes to 15 hours**
Macrocell fan-in	**4-16 product terms, 20-36 inputs**	**2-6 inputs**
Price/Register	**Higher**	**Lower**

Table 17 - CPLD vs. FPGA features

6.1.4 PCI Bus considerations

When designing a PCI Interface, there are a few considerations that must be taken into account:

Working at high clock rates (33MHz).

Working at high frequencies with an FPGA usually requires manual place and route techniques, since the automatic tools does not seem to product results good enough. The new generation of place and route tools with timing constrains may change this in the future. Another option is to use architectures that are tuned for high clock rates, such as anti-fuse based FPGAs from QuickLogic and Actel.

Working at high frequencies using CPLDs is much simpler. The different timing parameters that are associated with the specific CPLD can be analyzed, and a flow graph can be drawn showing all the possible paths in the chip which do not violate the PCI 33MHz timing requirement. Once these paths are established, The PCI timing requirements will be kept as long as these paths are followed.

Some of the PCI features deserves special consideration. We will address these features in the following sections.

Generating and checking for parity

PAR is the parity I/O pin. During data transfer it contains the parity of the data bus **AD[31..0]** and the byte enable lines **C/BE#[3..0]**. It is transmitted one cycle later. The simplest way to generate a parity function requires a XOR tree of 36 inputs. The parity has to be generated during a data write in less than 30ns, and has to be checked during data read in less than 60ns. CPLDs can make a 36 input XOR tree in 2 or 3 levels of logic. FPGAs has a smaller number of inputs for every logic block, and may require up to 5 or 6 levels of logic to compute the same 36 input XOR tree. Here are a few tips than can make parity generation easier:

1. If the data written by the PCI device is pipelined, parity generation can be done during the pipeline, generating partial results using smaller XOR gates and combining them in later pipeline stages.[6]

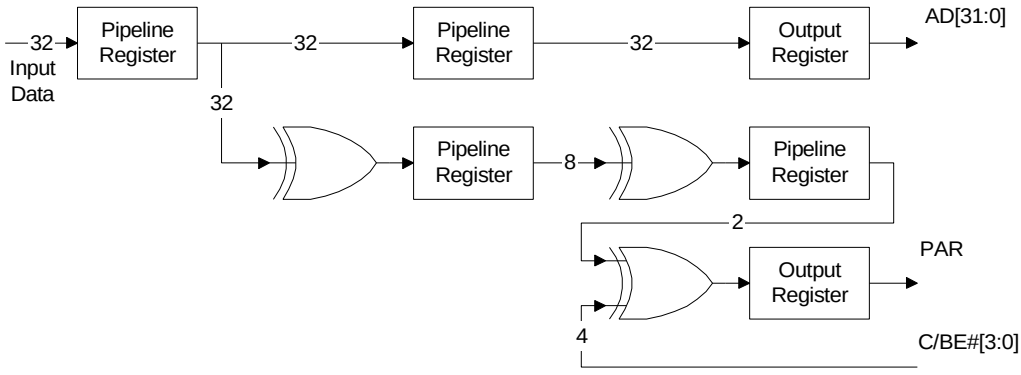


Figure 40 - Pipelined read parity implementation

2. The above method applies only to parity generation but not to parity checking, when parity must be generated to be compared with the incoming **PAR** line. In this case the **PERR** line which is used to inform the system of a parity error must act on data latched 2 cycles earlier, so no pipeline tricks can be done. If the parity checking logic cannot be made to fit, parity error checking may be ignored altogether. This means that the **PAR** line will always report no parity errors, regardless of the true state of the bus. Care must be taken to drive **PERR** correctly, enabling it only when required, pre-charging it at the end of the cycle, and tri-stating it immediately afterwards. This type of solution does not imply PCI incompatibility, but recommended only as a last resort.

[6] In Figure 40, an N input XOR gate is defined as N-1, 2 input XOR gates connected in series. This is effectively an N bit parity function. An N input, M output XOR array is defined as M parallel blocks of N/M input XOR gates.

3. When using deeper XOR trees and deep pipeline stages to generate PCI parity during data read, it is still required to generate parity cycles for configuration data read, which is usually not passed through the input data pipeline. Since most of the configuration information is constant, read only data, parity can be pre-calculated in these configuration words, so when a word is read from configuration space only the R/W bits must be XOR-ed with the pre-calculated parity bit for this word.

   For example:

   If the only writable bits in configuration space are bit 1 at the command register, and bits 24 to 31 at the BAR0 register, The parity function can be built from a 10 input XOR gate connected to 9 R/W bits and one fixed bit representing the parity of all data bits except bits 1 and 24 to 31. This fixed bit depends only on **AD[5:2]**, selecting which configuration register (0 to 15) is read now, and have a maximum of 8 product terms. This is much smaller than a 32 input XOR gate.

4. In the worst case, configuration data can be delayed until the parity calculation is ready, allowing bit serial implementation of parity calculation for configuration data.

Handling the IRDY/TRDY signals

IRDY and **TRDY** signals have critical timing characteristics. When **IRDY** is sampled by a target (or **TRDY** by a Master), it directly affects the contents of the data bus one cycle later. In order to provide optimal timing for **IRDY**, it should control the data path as close to the output as possible. FPGA implementations should give this part of the data path the highest priority in order to make the 33MHz timing requirement. Placing a register on the output, we can design the PCI bus output stage like this:

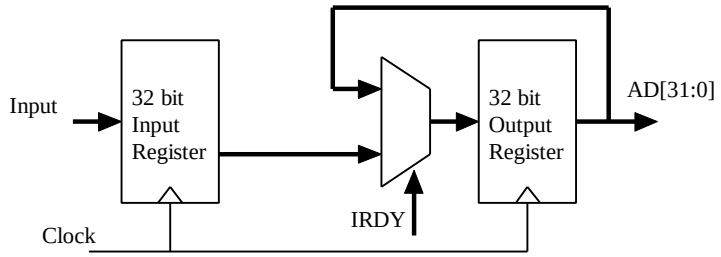


Figure 41 - IRDY signal path in PCI target designs

This is a D flip-flop with a synchronous enable. If the basic FPGA cell has a dedicated synchronous enable input, it should be used instead of the DFF + mux combination shown here. By using retiming techniques and dropping a redundant register, we can modify the previous drawing and design the PCI bus output stage like this:

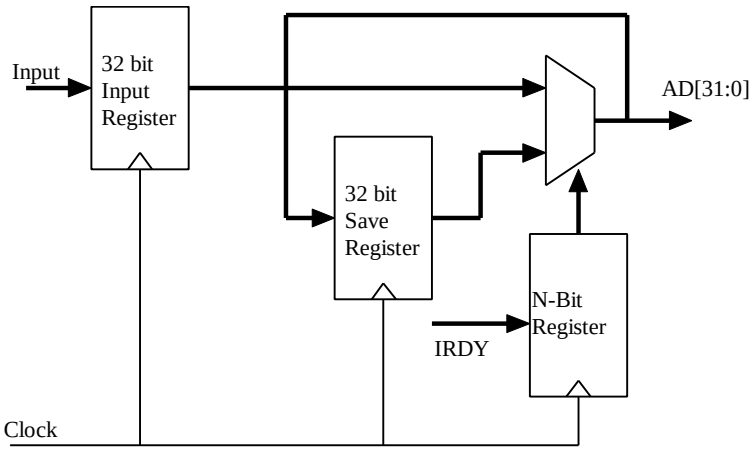


Figure 42 - Alternate IRDY signal path design

Some FPGA families may make better use of this version. The N-Bit register shown is IRDY driving multiple registers for optimal fanout control.

Static Timing Analysis for PCI applications using CPLDs

As we mentioned earlier, the PCI bus runs at a maximum frequency of 33MHz or 66Mhz, depending on the implementation. All the existing implementations are currently 33Mhz, since the 66MHz standard was added only in PCI 2.1, and supports only a single PCI slot. The AGP bus is a super set of the 66MHz PCI implementation. Assuming a 33MHz bus, we can summarize the PCI timing characteristics in the following table:

Name	Min (ns)	Max (ns)	Description
T_cyc	**30**		**Minimum clock cycle time**
T_high	**11**		**Minimum clock high time**
T_low	**11**		**Minimum clock low time**
T_skew		**2**	**Maximum clock skew allowed**
T_su	**7**		**A PCI input signal is guaranteed to be valid 7ns before the rising edge of the clock.**
T_h	**0**		**A PCI input signal is guaranteed to be valid 0ns after the rising edge of the clock.**
T_val	**2**	**11**	**The time delay from the rising edge of the clock to the point where the new signals are reaching their new state.**

Table 18 - PCI timing parameters

If we sum up T_skew + T_su + T_val we get 20ns, which means that PCI signals must propagate from any point on the bus, to the unterminated bus edge, and reflect backwards to all the PCI devices on the bus, at 10ns max.

PCI timing budget example

The following example demonstrates the calculation of the permitted data paths inside a CPLD when faced with the timing restrictions dictated by the PCI bus. As an example, the Mach 465 CPLD [(AMD, 1995)] was chosen. The following timing diagram includes all the relevant timing parameters:

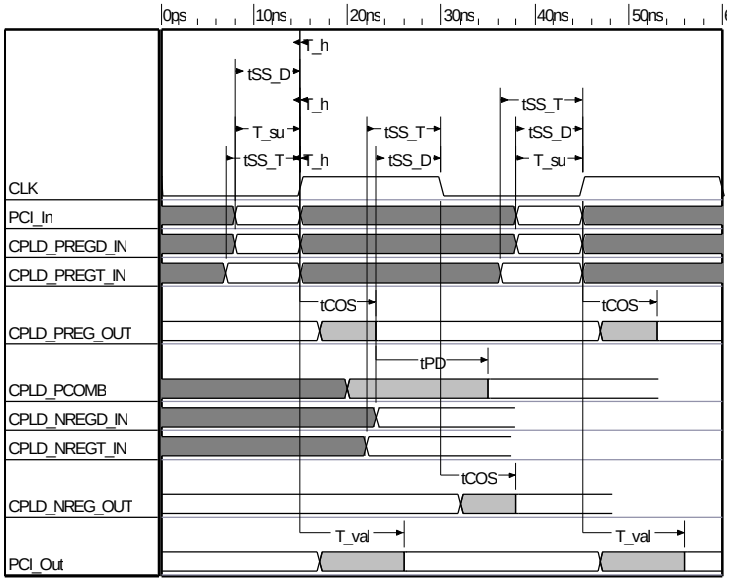


Figure 43 - PCI compliant static timing analysis for Mach 465-12

The diagram describes all the possible combinations of registers (T or D, clocked by the positive or negative clock edge), their setup/hold requirements, and their clock to output timing. It also describes

the timing of combinatorial nodes (without registers), which are driven by registers. It is assumed that the cycle time is 30ns, which corresponds to the maximum PCI clock, i.e. 33MHz. The diagram includes the following signals:

Signal	Description
PCI_In	PCI input signals setup/hold requirements
CPLD_PREGD_IN	CPLD positive edge D Register setup/hold requirements
CPLD_PREGT_IN	CPLD positive edge T Register setup/hold requirements
CPLD_PREG_OUT	CPLD positive edge register clock to output timing
CPLD_PCOMB	CPLD combinatorial node timing when driven by a positive edge register
CPLD_NREGD_IN	CPLD negative edge D register setup/hold requirements
CPLD_NREGT_IN	CPLD negative edge T register setup/hold requirements
CPLD_NREG_OUT	CPLD negative edge register clock to output timing
PCI_Out	PCI output signals clock to output requirements

Table 19 - Mach465-12 PCI static timing analysis parameters

The following timing parameters were taken from the Mach 465 data sheet:

Name	Description	Min Value	Max Value
tSS_D	Setup for D flip flop	7ns	
tSS_T	Setup for D flip flop	8ns	
tCOS	Clock to output for D or T flip flops		8ns
tPD	propagation delay for combinatorial nodes		12ns

Table 20 - Mach465-12 timing characteristic parameters

Notes:

1. PCI signals cannot be sampled using T registers, since they require a 8ns setup time, while the PCI standard guarantees only 7ns.
2. A positive edge D or T register output cannot be sampled by a negative D register input.
3. A negative edge D or T register output cannot be sampled by a positive D register input.
4. A positive edge D or T register output can be driven through a combinatorial node, and be sampled by a positive edge D or T register.

The above timing diagram can be translated to the following block diagram, which contains all the legal paths for moving data inside a Mach 465 while not violating the PCI timing requirements. The muxes in this diagram represents and AND-OR PLD structure with up to 20 product terms, and up to 34 input terms.

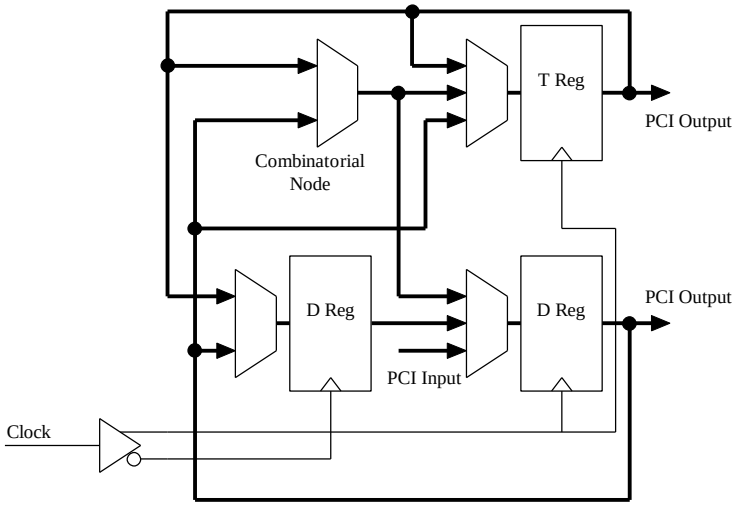


Figure 44 - PCI compliant data flow in a Mach 465 CPLD

PCI clock distribution

A PCI peripheral is usually implemented as a fully synchronous design. When designing a fully synchronous design, multiple chips may require copies of the main PCI clock. The problem lies in the fact that the PCI clock, like any other PCI signals, is allowed to drive only one bus load. If multiple components require the PCI clock, a clock regeneration circuit is needed. To make things harder, the PCI standard places almost no restrictions for motherboard designers when implementing the PCI system clock.

1. PCI clocks can run from 0Hz (DC) to 33MHz.

   PCI cards are expected to behave correctly under any legal condition of the PCI clock allowed by the PCI specifications. Cards are not expected to function correctly below a certain frequency threshold (for example, Ethernet cards are NOT expected to communicate with other Ethernet cards while the PCI bus runs at 1Hz), but are expected not to block the bus for other cards, and to respond to configuration access cycles.

2. The PCI clock frequency can change dynamically

   Notebooks are allowed to slow down the clock, or even to stop it completely, in order to save battery life. Desktop machines can use spread spectrum clocks (Each PCI cycle has a random cycle duration, but not below 30ns). Spread spectrum clocks are used in some designs to limit the amount of RFI energy emitted from the computer at the PCI bus frequency (and integer multiples of this frequency). By varying the clock duration, the energy dissipation is spread around the average frequency so there is no sharp peak at this frequency. This simplifies FCC certification, saving money for PC manufacturers.

There are a few ways to implement these clock regeneration circuits:

1. PLL and other circuits

   PLL based clock regeneration circuits take the PCI clock and regenerate an independent clock which is synchronized to the original PCI clock by having one of the clock outputs to act as a feedback for an internal phase comparator. By using the feedback, the PLL circuit can compensate for any phase difference between the original PCI clock and the new regenerated clocks. PLL circuits usually work in a limited frequency range, and do not respond well to frequency changes. PLL based designs are **not** PCI compliant for add-on boards because they will NOT work with PCI motherboards which change the bus speed dynamically, since a PLL may takes a few clock cycles to re-synchronize. Also, PLL circuits which can adapt to all the frequencies in the 0Hz to 33Mhz do not exist. Most PLL circuits will tend to oscillate even when the input frequency is DC, producing spurious clock cycles.

   Some Digital clock regeneration chips have solved some of the problems associated with analog PLLs such as a guarantee to adapt to a different clock after two clock cycles, and a guaranteed DC output given a DC input. Designs using these chips are still not PCI compliant for add-on cards, but are less likely to cause incompatibility problems compared with PLL based designs.

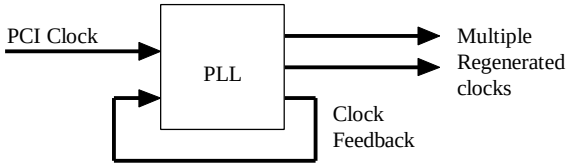


Figure 45 - A PLL based clock regenerator circuit

2. Clock buffers

   Circuits can based on a high speed clock buffer. Clock buffers can have propagation delays as low as 1.5ns. Still, this propagation delay must be compensated for. The net effect of a delayed clock is that the guaranteed signal setup time (T_su) goes from 7ns to 8.5ns, but the guaranteed hold time goes down from 0ns to -1.5ns! This means that the data may become invalid 1.5ns before the clock edge. The only way to compensate for a clock delay is to delay the data path as well, and to rebalance the T_su vs. T_h relationship. By delaying all signals by 3ns, the parameters are changed to T_su = 5.5ns, and T_h = 1.5ns. When driving output signals, the

opposite calculation must be performed. Since the clock is 1.5ns late, the T_val maximum value is reduced from 11ns to 9.5ns. This solution is fully PCI compliant, but it is hard to implement, since it requires adding an delay to bi-directional signals when in input mode, but no delay in output mode, yet having only a single electrical load.

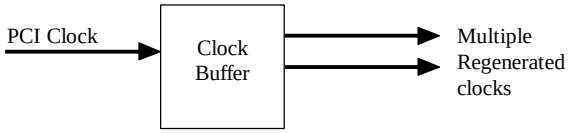


Figure 46 - Using a clock buffer

3. Clock buffers for the design back end

   Another possibility is to drive only one component, (the PCI interface itself chip) with the original clock, and to regenerate the clock from that component. The pipeline can then be constructed from multiple sections, as described above (see section 4.3.4). The clock must have a fairly long delay, larger than $t_{SU} + t_{CO}$, because each pipeline stage is sampled while the other pipeline is stable.
   Even with a 30ns cycle time (derived from the PCI 33MHz maximum clock rate), it is easy to achieve the timing requirements by delaying the clock by 15ns. This ensures that each pipeline stage is sampling it's inputs from stable output from the other pipeline stage. This is perhaps the easiest solution which is also fully PCI compliant, since no external clock buffers are needed (use the PCI interface chip itself as a clock regenerator), and no delay has to be added to all the PCI signals.

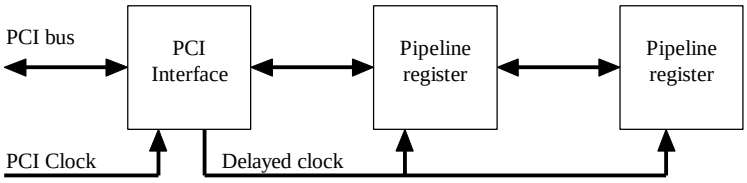


Figure 47 - A synchronous multi phase based PCI system

Optimizing PCI designs for CPLD

In this section the specific characteristics of CPLD devices are discussed, as well as their limitations. Some techniques to make optimum use of CPLDs in terms of density will also be discussed.

1. Use multiple levels of logic and common nodes as much as possible.

   Most synthesizers, if not all, cannot automatically generate nodes for common logic sub-expressions. For example, if 10 nodes are using the sub expression `A*B+C*D`, assigning it to a temporary node may not only save product terms, but may also save input terms to one or more PLD blocks. Nodes can be added as long as the path delays are not too long, as demonstrated in section 0.

2. Use grey code state machines.

   CPLD devices are not register rich. Using one-hot encoding as common in FPGA based state machines may waste more macrocells than needed. In order to make sure that the minimum number of product terms are assigned for the state machine variables, every pair of states which are linked by a state transition, should be assigned state values with a minimal hamming distance, preferably 1. A practical way it to follow all the possible state transitions and assign consecutive states in the state machine using grey code. This tends to minimize the number of flip flop changes in each state, and to reduce the state machine equations. This is a heuristic method, but it's producing excellent results.

3. Use T registers where appropriate.

   Most CPLD architectures support T registers which can greatly reduce the number of product terms required for registers which are enabled by a complex event (and unchanged otherwise). T registers are also very useful for counters, in which they can help building counters with a fixed number of product terms per register, regardless of the counter length. Some synthesis tools will automatically take advantage of these registers, but if not, it is important to assign T registers where appropriate. T registers are not very useful in data path registers which change every clock, and may cause even larger expressions than D registers.

4. Configuration cycles are least important. Optimize them for space, not speed.

   Configuration cycles are usually performed only when the system boots, and are not important for high performance. Configuration cycles must be made to work, but no more. Any extra gate invested in making configuration cycles work faster is a waste. It is not uncommon to find some designs using bit serial methods shifting in configuration data from an external slower memory, causing configuration cycles to take dozens of clock cycles.

5. Design large data paths considering product term limitations of the CPLD architecture.

   Most CPLD architectures allocate product terms for registers in small groups. This means that as long as a group is allocated, it does not matter how many product terms are used. For example, the Mach devices use 5 product terms for every register, and allocate more product terms in groups of 4. A 6 product term node takes the same amount of resources as a 9 product term node.

6. Use medium or slow decode if possible.

   Responding to PCI cycles using fast decode requires more logic than responding to PCI cycles using medium or slow decoding, because fast decode requires a single level of logic. Decoding N BAR bits requires 2*N product terms. Splitting these product terms across multiple nodes may help fitting those nodes. A CPLD architecture such as the MACH 4 has a limit of 34 input terms to the PLD block. This means that BAR decoding is limited to an absolute maximum number of 17 bits in a single node. Breaking BAR decoding into smaller sections can help fitting them into multiple PLD blocks, but slows the decoding speed, and may require the use of medium decode.

7. BAR registers should be merged when possible.

   Having more than one BAR register potentially frees system memory space, but wastes CPLD registers. A 16MB Memory BAR and a 256 byte memory BAR takes 32 registers bits. Combining these two BARs together into a single 32MB BAR requires only 7 register bits.

8. I/O BARs should be avoided as much as possible.

   I/O BARs are used for x86 backward compatibility only. Decoding I/O address space requires much more BAR bits because available I/O space is much smaller. I/O space should not be allocated unless it is absolutely necessary for backward compatibility.

9. Save BAR bits if unavoidable.

   It is always desirable not to “waste” address space by requesting more memory space than what is required for a device, but requesting 16 bytes vs. 64K bytes on a 4GB address space doesn’t save much of the address space. On the other hand, this saves 12 BAR bits and the associated decode logic. Even decoding 16MB as a minimum area isn’t that of a problem, considering that it takes just 1/256 of the address space. These BAR bits can make the difference between a design that can fit in a CPLD vs. a design that can’t be fit in a CPLD.

10. Pay attention to s/t/s signals.

    Most CPLD architectures offers registers where the clock-to-output (t_{CO}) response is quicker than the tri-state enable/disable control (t_{EA}). This means that when sustained tri state signals are turned off, it is important to keep them high even during the cycle they are turned off, otherwise, a small negative pulse is sent on the bus. This pulse will not appear in simple unit-delay simulators, and is very hard to detect without full timing simulation models.

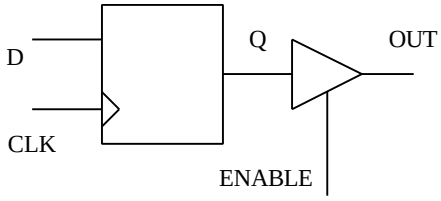


Figure 48 - Typical register output stage

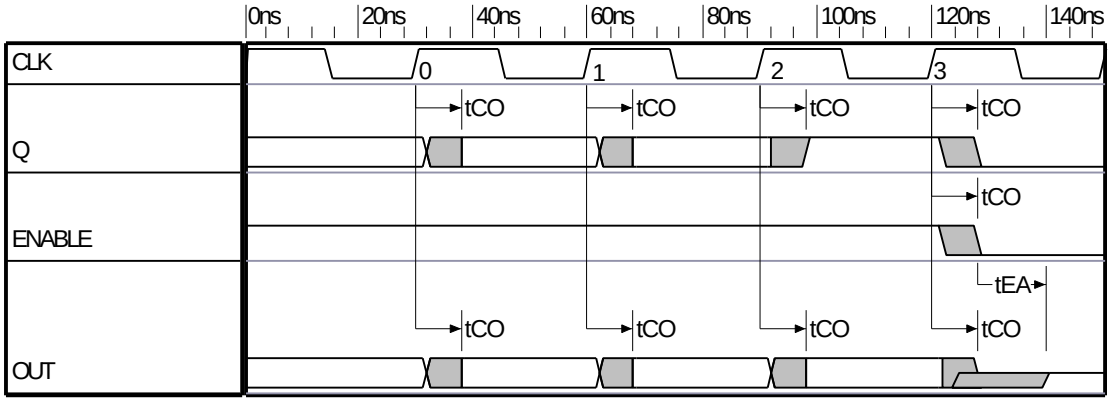


Figure 49 - Potential s/t/s spikes caused by simultaneous ENABLE and Q switching

It's easy to see in Figure 49 how the Q flip flop output, and the OUT pin may become 0 after tCO at clock 3, while it takes tCO+tEA for the output enable control pin to tri-state the OUT pin. This causes a short negative spike.

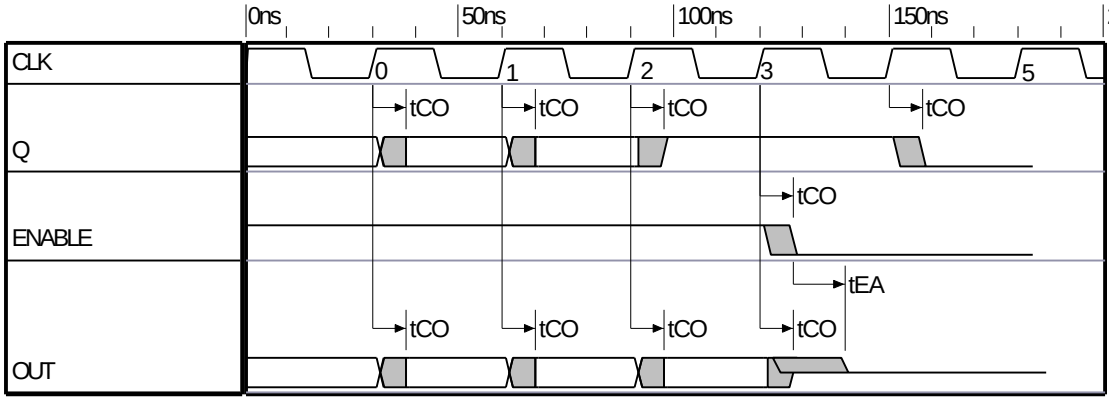


Figure 50 - Solving s/t/s spikes by keeping Q high for one extra cycle

In Figure 50 Q is held high for one extra clock, and so out is held high until tri-stated at tCO+tEA. Q will be 0 only one clock later, but by then OUT is already tri-stated, and Q wouldn't effect it.

6.2 The HDL design flow

The HDL design flow is a diagram showing the relation between all the tools used during the development, and the various files used to create the final design files. Our current design flow combines a CPLD synthesizer (MachXL, AMD's OEM version of Minc's PLDesigner-XL tool), an equation to verilog converter (written in Perl), a Verilog simulator (SILOS III), and a set of custom programs which allows the integration of test vectors gathered by running a PC based Logic analyzer into the simulation test bench. The next phase includes integrating a verilog synthesizer and Altera's Max2Plus software for designing 10Kxx devices, in order to prototype mode complex PCI devices (Masters, and Bridge chips). The diagram below shows the direct flow of information during the design process. It can be seen how:

1. Verilog source code can be simulated directly (running test bench code, and preliminary synthesizable design).
2. Verilog source code can be simulated after synthesis (during a synthesizable module design).
3. Verilog source code can be simulated with accurate timing information after synthesis, placement and routing (timing verification of a synthesizable design).
4. Other design sources (such as MachXL) can be simulated using XXX to verilog translators, where XXX is the desired input language.
5. Real world data (sampled by a logic analyzer) can be used to simulate real world loads by analyzing captured data and extracting simulation parameters.

An important part of the design flow is the feedback from the simulator back to the module source code. This is not a direct link since no source code files are generated by the simulator, but it is the simulator which gives the circuit designer the feedback information required to debug his source code model.

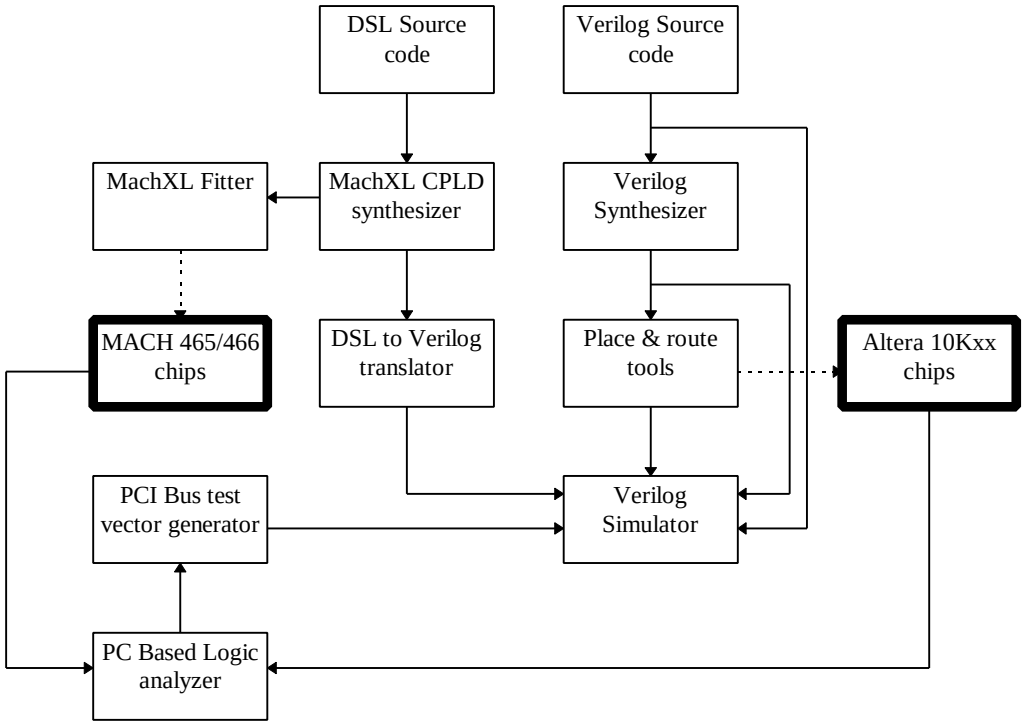
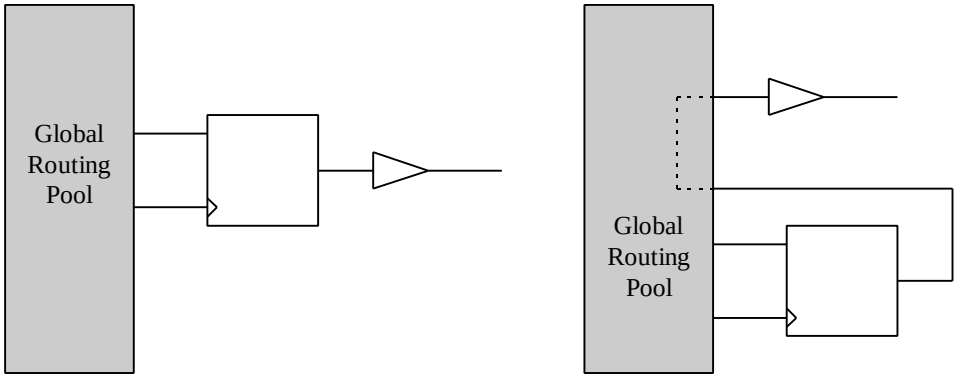


Figure 51 - Typical HDL design flow

6.2.1 A DSL to Verilog model translator

In order to simulate the design, a translator converting the final DSL (MachXL language) equations into Verilog was written in Perl. This translator takes the compiler documentation file and generates a verilog model from the equations. While writing the translator, a few decisions were taken:

1. Since building an accurate timing model requires having the fitter result file and having a successful fit, it was decided not to try building an accurate timing model which would have forced running a place and route run for every change to be simulated, which is counterproductive. Since CPLDs have fixed timings, accurate timing models are less important. The implications are mostly in the registered nodes, which have separate output equations for their tri-state buffer. The reason for requiring the fitter output is that it is impossible to distinguish a hidden node connected to a combinatorial output from a registered output node, since both have the same equations. Only the fitter table shows whether these two resources are mapped to the same macrocell or not. In order to make sure that the model will not fail where the real circuit doesn't, the simulated delay values were taken to be 4ns for each stage, which is just above twice the normal speed, compensating for any redundant node in the equations.



AMD Mach output register

AMD Mach buried node + combinatorial output

Figure 52 - Alternative methods for implementing registered outputs on AMD MACH devices

2. The Mach devices have selectable D/T type registers. It can be shown that T registers with synchronous reset will not be cleared during simulation, even though they are cleared fine in reality, if they were initialized with an "unknown" value. The solution was to produce output equations containing D type registers only, even though the real circuit may use T type registers.

   For example, the following equation is generated when a register is synthesized as a T type register:

   ```
   TF = CL*TF + !CL*A*TF + !CL*B*!TF
   ```

   TF is a T type register, CL is the synchronous clear input, and A, B are arbitrary terms. The T type equation will toggle the node value when the right side of the equation is true. When CL is set, the equation is reduced to TF=TF, which resets the T register. This can be translated into a D type node equation in the following form:

   ```
   DF = (CL*DF + !CL*A*DF + !CL*B*!DF)^DF
   ```

   or,

   ```
   DF = (CL*DF + !CL*A*DF + !CL*B*!DF)*!DF +
        !(CL*DF + !CL*A*DF + !CL*B*!DF)*DF
   ```

   The right hand side has been XOR-ed with the previous value, generating the true value to be latched into the D type register. When the simulation starts, the A, B, CL inputs are known, but the DF value is unknown. According to the verilog operator truth tables, unknowns can be reduced by these rules:

   ```
   0*X = 0
   1+X = 1
   ```

   If CL is asserted (CL =1), the DF equation above reduces to:

   ```
   DF = (DF)*!DF + !(DF)*DF
   ```

   This equation cannot be reduced further, even though assigning either 1 or 0 to DF will produce the same result, which is 0.

6.3 A PCI Bus simulation environment (test bench)

6.3.1 A PCI Verilog test bench

Here is a typical Verilog test bench for PCI, drawn as a structured diagram. This can be applied to VHDL as well.

The test bench contains a common, simulated bus into which all other modules are connected. The model must have at least one master and one target in order to do any meaningful bus traffic simulation. These models (there can be more than one Master or Target) are either synthesizable or non-synthesizable. In the case of a PCI Master model it is possible to write a simplified model which does not do any arbitration. This model is not good for anything other than testing bus targets, since it cannot coexist with other bus masters. A more sophisticated model will arbitrate the bus with other bus Master models. A Bus Arbiter model is used to arbitrate between multiple master models, if there are any. The clock generator module generates the PCI clock. It can be anything from a simple square wave generator to a complex model capable of generating a spread spectrum clock, or a variable frequency clock for power management simulation. A PCI to PCI bridge model may optionally be used to link more that one PCI bus during simulation.

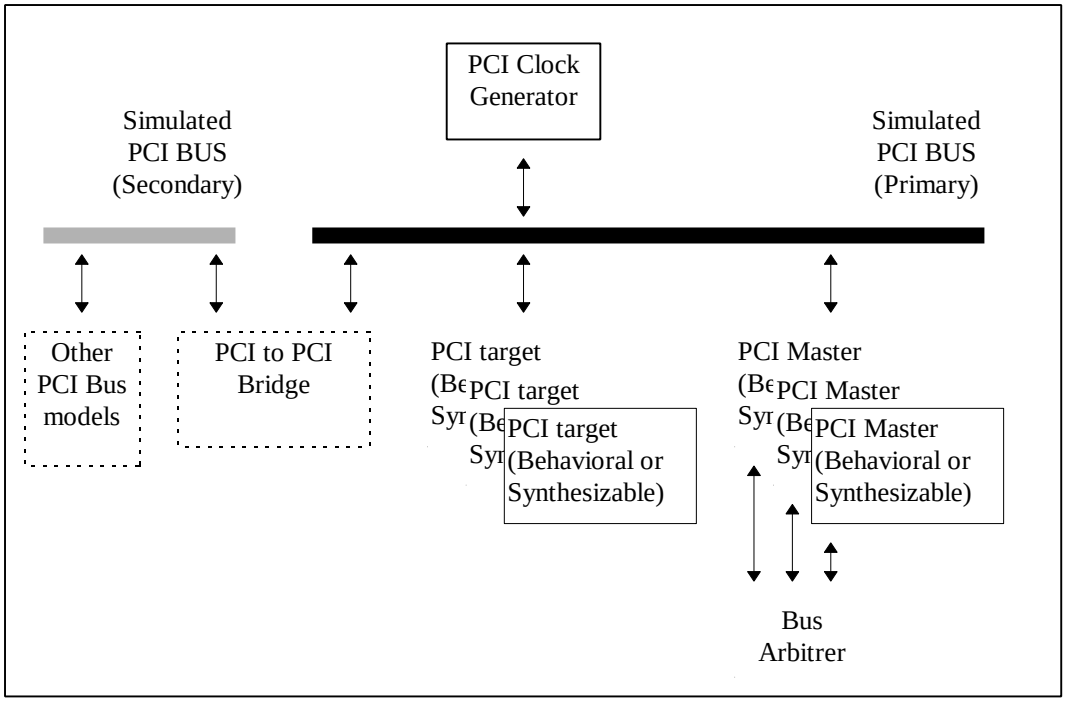


Figure 53 - A typical PCI test bench

6.3.2 Synthesizable vs. Non-Synthesizable models

Hardware models are usually divided into two groups: Synthesizable vs. Non-Synthesizable. Here is a short list of differences between the two groups:

Synthesizable models

Synthesizable models are written in a subset of Verilog called RTL, or Register Transfer Level. This subset is tool dependent, since every synthesizer supports different parts of the language constructs. As its name implies, a synthesizable model can be converted by a synthesizer into a netlist of gates. To be more precise, a synthesis result is a netlist of library components, selected from the chip vendor's library. This applies to both FPGA vendors and ASIC vendors. The netlist is then processed by the vendor's layout tools, producing ASIC masks, or a CPLD/FPGA configuration bit-stream.

Synthesizable models are usually simulated at three levels: Pre synthesis, Post synthesis, and post layout (back annotation). The top level is the original RTL code which is run under the test bench to verify its correctness. After synthesis, the resulting netlist is simulated again to make sure the synthesized output matches the original design. The lowest level is run after a placing and routing the model into an ASIC or FPGA. The list of net delays is fed back into the gate level netlist, and it is simulated again to make sure the additional delays are not affecting the design correctness.

Non-Synthesizable models

Non-synthesizable models are written using the full power of Verilog. These models cannot be synthesized into a netlist, for a few reasons:

1. The model uses a verilog construct not supported by the synthesis tool. For example, verilog tasks, functions and event control constructs are not supported by all synthesis tools.
2. The model relies on constructs that have no meaning in hardware. For example, checking a wire for a tri-state value can be used as part of a simulation test, but has no meaning in a chip, where all values are either 0 or 1.
3. The model relies on specific timing delays embedded in the design. While in theory a future synthesis tool/vendor library combination could support this, hard coding delay values in silicon is so hard, that this is not practical. Usually, synthesis tools simply ignore delay values.
4. The model relies on language construct combination that cannot be emulated by the vendor library. For example, it is impossible to code a register with an asynchronous reset if the vendor library does not support an asynchronous reset in any of its library components. Another good example is a register sensitive on both clock edges, which is a construct not widely supported by libraries.

Non-synthesizable models are used during simulation for two reasons:

Non-synthesizable models can sometimes be written more easily and quickly than synthesizable models. This allows prototyping an idea, and checking its correctness before committing time and effort into writing a synthesizable model.

Non-synthesizable models are also used for test benches, where they are used to generate stimulus for other, synthesizable models. The test bench is also used to verify output signal correctness, including:

1. Making sure output signals are what they should.
2. Making sure signals obey Setup, Hold, and Width specifications.
3. Making sure signals are actually tri-stated when they should be.
4. Making sure signals never carry unknown values, when they shouldn't.

Output checks allow designers to run very long test benches without having to manually look at the output waveforms, looking for anomalies.

6.4 Real world test vector integration

In order to perform real world data analysis, a PC based logic analyzer was used to gather information from a working PCI based system. The logic analyzer can record up to 128K clock cycles, covering all the PCI protocol lines. The data files produced by the logic analyzer were saved to a disk, and were analyzed by a separate program written in C. Many statistical parameters can be extracted from these samples, including:

1. Average data latencies (time for first data item in a burst), sorted by mode (read or write) and device (determined by the base address).
2. Average data throughput (number of wait states in a data transfer), sorted by mode (read or write) and device (determined by the base address).
3. Bus utilization (percentage of idle bus cycles).
4. Wasted retry time (number of cycles wasted by bus retries, after the first retry). We will mention this data later.

The essential parameters of each PCI transaction including base address, burst length, and wait state behavior can also be logged and fed back to the simulation environment.

6.5 Hardware-Software Co-simulation

Even when a behavioral software model of the PCI bus is written and is driving a synthesizable hardware module, a link to the software drivers must be provided. Without an interface to the driver software, another layer of simulation software must be written on top of the behavioral PCI bus model, to simulate the bus cycles generated by the driver software.

Verilog has a feature called PLI, which allows Verilog programs to call C subroutines when certain events occur (either a time event, or a signal change). We can use this feature to link the driver software with the Verilog simulator. We want all the PCI calls from the C software to be intercepted by the Verilog simulator, and the result to be returned back to the driver software. Unfortunately, once a Verilog program calls a PLI routine, Verilog execution is suspended until the C subroutine is terminated. This makes calling the software driver from within a PLI impossible, since the software driver generates multiple PCI bus access cycles, and each of them has to be handled by the Verilog bus model. Turning the software driver into a subroutine which returns a single PCI request at a time is a major design change.

The solution chosen was to run the software driver and the verilog simulation as two separate processes, communicating through a socket library. A PLI C subroutine is still needed, but this subroutine is called by the Verilog simulator every time the simulated PCI bus is free to accept another request. In this way, two separate execution threads are kept, the Verilog thread, and the software driver thread.

The PCI bus cycles requested by the program are trapped, and a short request is sent through the socket to the Verilog simulator indicating the PCI command, the address and the data (if write). In case of a read command, values are also returned from the Verilog simulator. The use of a socket library allows running the two programs on separate machines using TCP/IP.

Trapping of PCI requests in the driver code can be done either by wrapping the hardware PCI requests by a subroutines (or class member functions if C++ is used), using a one set of subroutines when compiling the driver for the actual hardware, and a different set of subroutines when compiling the driver for co-simulation. If a protected mode operating system is used, another possible approach is to write a device driver which traps the physical PCI requests performed by the driver and turns them into the socket library calls required. The drawback of the first method is that wrapping the PCI bus calls in subroutines reduces the driver performance, but this can be solved by a careful use of macros.

6.6 An actual PCI Target Implementation

An actual PCI target was developed as part of a PCI DRAM board by the author, as part of his work at FourFold Technologies Ltd. The board was sold to an OEM, and is being mass produced.

The interface was targeted for AMD Mach chips (the Mach 465/466 to be exact), and developed using the MachXL 5.3 software. This is a restricted version of Minc's PLDesigner-XL software, targeting AMD's PLD and CPLD chips only. Overall, the complete system (1 short PCI card and 3 memory modules) contains 18 CPLD chips, 128 DRAM chips, and clock generation circuits.

The raw equations files (after synthesis) of all the CPLD's were converted to Verilog by an internally-developed translator, as described in section 6.2.1.

A PCI test bench was written in Verilog, as well as behavioral models for non-logic components (DRAM chips and 2 clock oscillators). A complete system simulation was run, including all the logic chips on the card, and the entire DRAM array. In order to save simulation memory, each DRAM chip had only 4K bytes of RAM simulated, covering only a limited number of rows in each DRAM. The simulation was run using SILOS-III 97.100, under Windows NT 4.0 using a Pentium 133MHz machine w/64MB of system RAM. The simulation took roughly 20 minutes to run a 1.6ms simulation period (representing about 53,000 cycles), yielding a ratio of about 1:1,000,000 between simulation time and real time.

The PCI test bench was designed to send PCI transaction to the card only to addresses which are mapped to DRAM memory locations covered by the 4K memory area on each DRAM model instance. Since the card is 4 way interleaved, the 8 bit DRAM chips are 16-way interleaved, which means that even with only the first 4K of each DRAM model active, we had a 64K memory range in which we could freely burst.

In order to test the board design at all possible states, all the PCI master parameters were derived from a pseudo random sequence, including the staring address, the actual write data, the burst length, and the number of wait states for each word transferred. That insures that even rare bugs occurring only at very rare conditions, would eventually turn up. The use of a *pseudo* random sequence derived from a predetermined seed (as opposed to a true random sequence) allows the results of a simulation to be re-

createable if needed. We even went as far as implementing our own random function as a verilog function (taken from a C compiler library source code), to guarantee that we will get the same simulation results no matter what simulator we would run our test bench on.

PCI Timing compliance was guaranteed using static timing analysis methods, and not by using timing simulation. This was done by counting the number of gate levels.

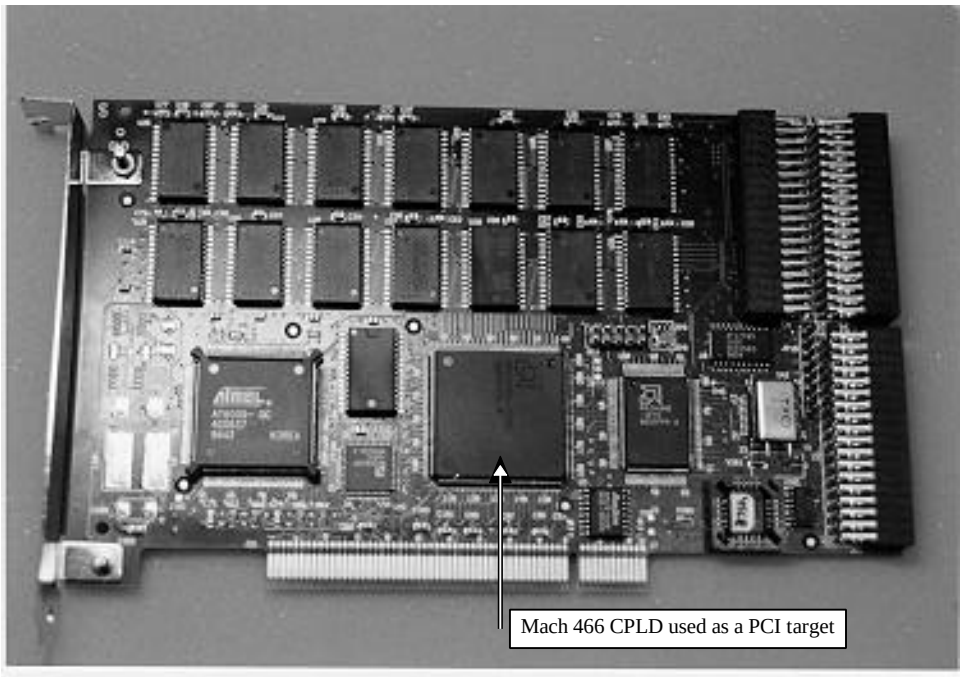


Figure 54 - GM256, FourFold's CPLD based PCI DRAM card

7. Improving the PCI bus

7.1 The Latency problem

As PCI systems grow to multiple bus masters, PCI to PCI bridge chips are used to link multiple PCI bus segments. As a result, the latency of memory and I/O read operations is getting longer, especially when a bus transaction has to go through one or more PCI to PCI bridge chips.

PCI 2.0 compliant target devices had to hold the bus until they could deliver read data, no matter how long it took. No means were available to make use of the bus during this latency time, nor was the target latency time limited. In theory, the target could retry the master, but since the master was not required to retry the transaction, the only way to guarantee data delivery from a long latency target devices was to hold the bus as long as necessary.

In order to solve the problem, the PCI 2.1 standard was modified in a few subtle ways:

1. Target latency was limited to 16 cycles on the first word, and 8 cycles on any subsequent word in a burst. Any target requiring a longer latency, was required to retry the master, by terminating the current cycle without data (**STOP#** active, **TRDY#** inactive).
2. A PCI 2.1 Master is now **required** to re-issue a request if a target is disconnected without delivering data. The master must retry the request until the target is ready.
3. After disconnecting, the target is required to refuse (also by disconnecting without data) any other request other than the previous word. This ensures that the read order is preserved.

This solution allows other masters to compete for the bus between the original request and the target reply, but it has two problems:

1. The Master has to poll the target until it receives the data, wasting bus bandwidth which can be used by other bus masters.
2. The master will always poll the target, and since the polling may happen every N cycles, it will never get the data from the target as soon as the data is ready, but as soon as the master retries after the data has been ready. This implies an average waste of N/2 cycles on every delayed transaction.

7.2 The Latency Hint Solution

A possible solution involves transferring a hint about the expected latency on the data bus while **STOP#** is asserted. The latency hint which is measured in cycles, will be used to hint the master on the expected latency of the current transfer. The master will need to retry the transfer only after a cycle counter initialized by the latency hint will expire.

Targets supporting this extension can be identified by a few methods:

1. Add a new read command that only targets supporting the extension would recognize. This has the drawback of requiring the master to know in advance whether the target supports the latency hint extension. This can be done by checking a new configuration register status bit in advance.
2. Use some of the data bits not used for the hint as magic number. For example:

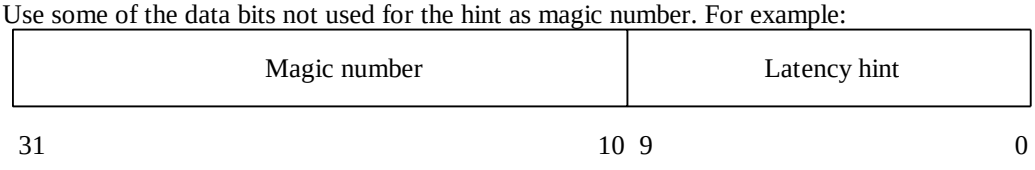


Figure 55 - A possible arrangement of a Latency hint word

If we choose a random number as a Magic number there are very low chances that any old target will return this value on the data bus by a chance. Even if it does, the master will wait a maximum number of 2^{10} cycles since the longest latency hint has been defined to be 10 bits. In this rare case, performance will be hurt, but system integrity would not be compromised.

We must never allow a latency hint which is more than 15 bits, since not only would the magic number be smaller, raising the chance of a random magic number generation by an old target, but also any target retry longer than 2^{15} cycles, is considered an error by the PCI standard, and can be dropped entirely, risking the system integrity!

7.3 The Early Read Solution

Another possible solution involves sending an advanced request to the target by using a PCI Special Cycle command. After a predetermined number of cycles, a normal data read request would be sent, which would receive the requested data immediately. The master can check if the target supports this mode by examining its configuration registers, and checking for this special mode which would be defined by a new configuration space flag. It would also indicate how many cycles after the advanced request the data itself can be received. The advantage of this approach is:

1. New targets can work with old masters. When a memory read request is received with no advanced request, the data would be fetched according to the normal PCI rules (either wait states, or target retry).
2. New masters can be programmed not to issue advanced requests to old targets. This would be done by software probing the PCI bus during system startup, examining configuration space, looking for the required flags on every target in the system. Even if an advanced request is sent by mistake, it is simply ignored (but time is lost since the target would start fetching the data only when the real memory read command is sent).
3. When both the target and the master supports the method, the time saved is the target decoding time, plus the bus turnaround time.

The drawbacks of this solution are:

1. The number of cycles between an advanced request and a memory read must be fixed since it is determined by reading the configuration space only once, during system startup.
2. PCI Special cycle commands use the low 16 bits to signal the special cycle message type. If a new special cycle message type is used for advanced read request, only the upper 16 bits are left for user data. This implies that PCI targets that use this service must allocate at least 64K of the PCI space.

7.4 The Split Transaction Solution

We are proposing an extension to the PCI protocol which solves the above problems by adding a new read command that allows queuing a read command, specifying not only the base address but also the length, and an ID tag. The ID tag is used by the target to identify the resulting data when it is ready. The master sending the request will tag the reply by an ID code. For example, we may assign the following new **C/BE#** commands:

C/BE#3	C/BE#2	C/BE#1	C/BE#0	Command Type
1	0	0	0	**Split Transaction Request**
1	0	0	1	**Split Transaction Reply**

Table 21 - Proposed encoding for new Split transaction PCI commands

A possible arrangement for the split transaction request might look like this:

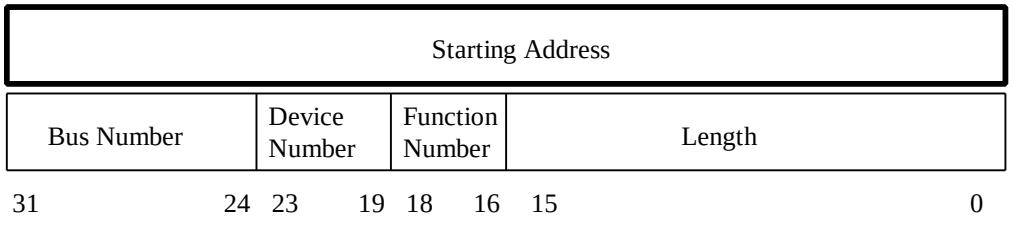


Figure 56 - Split Transaction request - method 1

The starting address is sent together with the Split Transaction Request command, while the length and ID tag word is sent on the next cycle, with **C/BE#** is 0 (all byte lanes on). The device number, bus number and function number of the requesting device are used as an ID tag.

The split transaction reply might look like this:

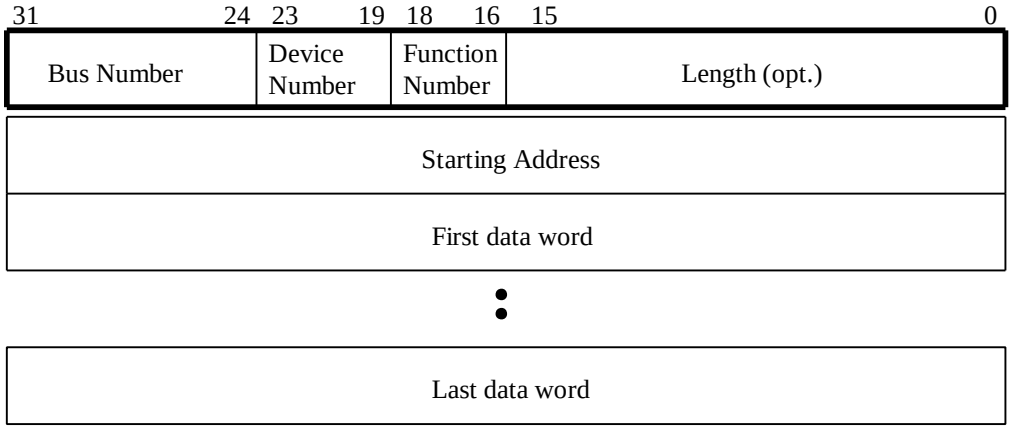


Figure 57 - Split Transaction reply - method 1

The Length field in the reply is optional since the reply is also terminated using the basic PCI protocol using the **STOP#** line. On the other hand, sending the length in advance might allow the master to optimize its internal buffer management, knowing in advance how many words are going to be returned. The reply length must be equal or less than the request length. In case it is less than the request length, more data will follow.

A slightly different organization for the split transaction request and reply might be:

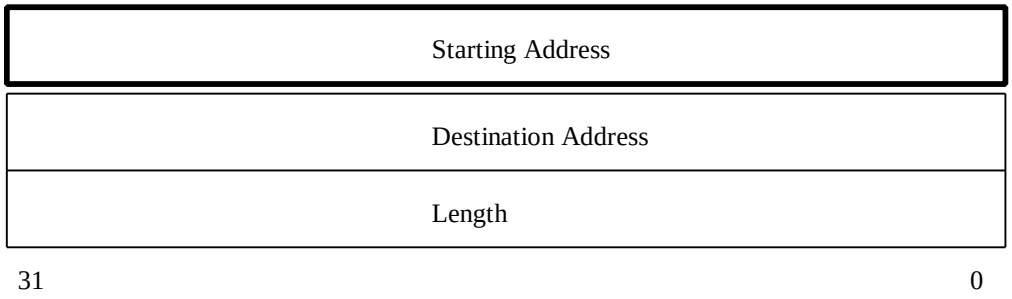


Figure 58 - Split Transaction request - method 2

The starting address is sent together with the Split Transaction Request command, while the destination address and length are sent on the next cycle, with **C/BE#** is 0 (all byte lanes on). The destination address of the requesting device are used as an ID tag.

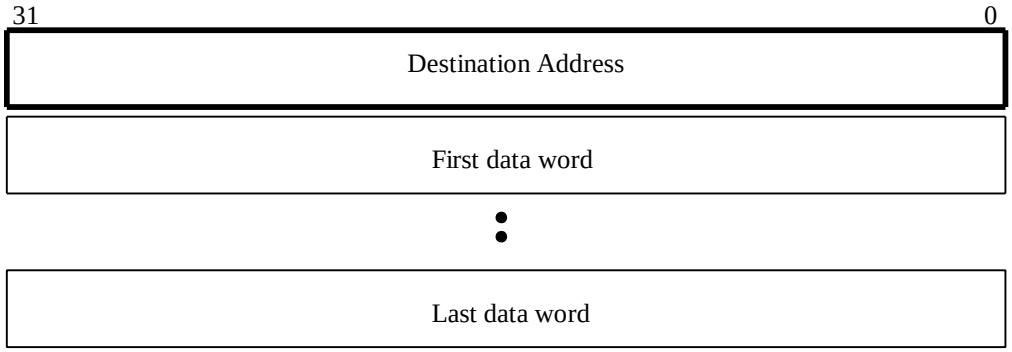


Figure 59 - Split Transaction reply - method 2

This method has two advantages over the previous method:

- By giving a Bus / Device / Function code, the original requester is identified. By giving the destination address instead, not only the original requester is identified, but also the original request is identified, since the same requester can send multiple request, each with a different destination address. Therefore the reply does not need to contain the starting address from the split transaction request.
- If we look at Figure 59, we can see that the reply uses exactly the same fields as a regular memory write transaction. Therefore, the reply can use normal memory write commands. This means that this method can be used by device #1 to request device #2 to transfer data to device #3, and not necessarily back to device #1. Not only that, device #3 doesn't need to support **any** protocol extensions. It is possible to look at this method as a way to standardize a bus mastering protocol. Current bus masters use a unique method to operate its bus mastering capabilities. It is not possible today to access a bus master device in a standard way.

It is possible to enhance this protocol even further:

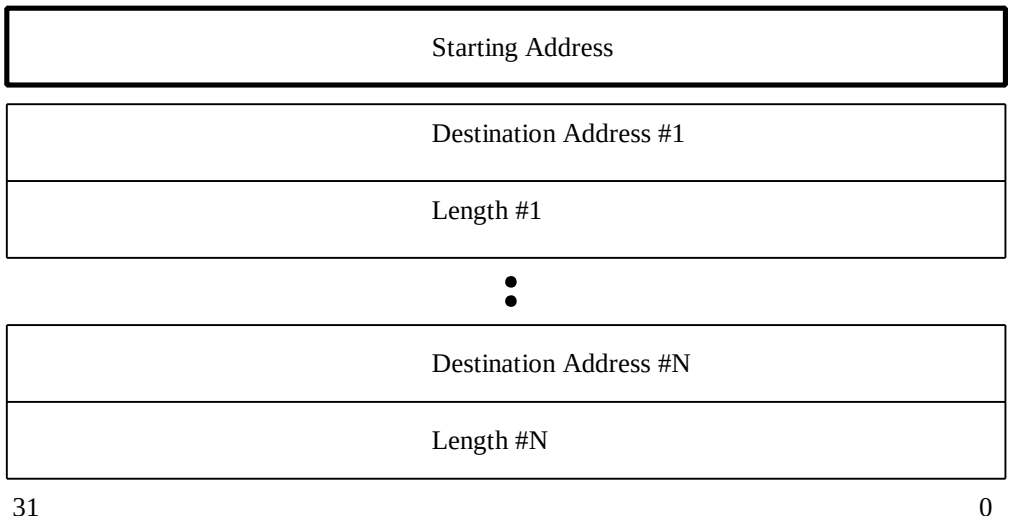


Figure 60 - Split Transaction request - method 3

This form specifies multiple requests in one transaction, and allows full use of scatter-gather Bus masters in a standard way. This is very important in systems using virtual memory, where a continuous memory buffer located in the logical address space might be discontinuous in the physical memory map due to virtual memory paging.

The master can try queuing as many request as it needs, but the target is allowed to terminate this cycle when its internal request queue is full. The cycle can be terminated using the **STOP#** signal and the standard PCI handshaking protocol.

The reply would come as multiple, standard, memory write transactions. There is no need to invent a new split transaction format for this request form, because multiple standard PCI write requests using fast back to back transactions can be as fast as possible (one clock for each data word, and one clock for each destination address).

7.4.1 The AGP protocol extensions

These protocol enhancements are inspired by similar enhancements made for the AGP bus. AGP is not replacing the PCI bus, but rather complements it. It trades the PCI generality for better system throughput. Both standards are used concurrently in the same machines, and it is expected that both standard would develop further. We have tried to use a different approach than AGP for three reasons:

1. AGP is a point-to-point bus (only one master and one target). This means there is no need to identify either side when sending requests and receiving reply. The PCI bus supports multiple masters and multiple targets, therefore each request and each reply must be identified.

2. AGP is using a different connector with more signals. We want to use the existing PCI slots, and therefore cannot assume that a new connector with more signals will be used. Even if we wanted to use unassigned PCI pins, it seems that the only unassigned pin on the PCI connector was recently assigned by the PCI SIG to be used for waking up devices in low power mode.
3. Since PCI transactions can be used over AGP, the AGP standard specifically avoids consuming any reserved PCI protocol resources (such as PCI reserved command codes) which may be used by a future PCI revision, making AGP incompatible with PCI. Since we consider our proposal a possible PCI bus extension, we can use the PCI reserved commands.

7.5 Performance analysis

After suggesting alternatives to the standard PCI protocol for long latency devices, it still left to prove that the new protocol modifications actually improve the bus utilization. The analysis is broken into a few tasks:

1. Collecting statistical information about the PCI traffic to be simulated.
2. Generating PCI traffic using the standard PCI protocols as well as the modified PCI protocols.
3. Analyzing the results.

7.5.1 Collecting PCI traffic information

In order to run a simulation of the PCI bus, we need information about the typical cycles on a PCI system. The best way to collect this type of information is to get it from a real system. As we noted, the latency problem is more apparent on systems with large busses and multiple bus masters, which may compete for the PCI bus. A typical system is a server with one of more SCSI cards, and one or more Fast Ethernet adapters.

There is more than one method for collecting this type of information:

Connecting a logic analyzer to a PCI bus

By connecting a logic analyzer to the PCI bus, it is possible to collect detailed cycle information. When this raw information is transferred to a PC, further analysis can be done to reveal higher level information such as typical latency of specific cards, target retry behavior, master retry behavior, total latency (from the first retry until a request is completed).

The drawback of this method is the limited buffer depth of most logic analyzer and the large volume of PCI information when sampled in its raw form. Even a logic analyzer with a 128K sample depth will store 10K-20K PCI transactions at most, which may not be enough to model a real system behavior.

Using a PCI bus monitor.

A PCI bus monitor can passively analyze PCI transaction information and save only the higher level information. For example, a single PCI clock cycle requires storing approximately 48 bits of information. A typical 10 cycle PCI transaction requires 60 bytes to be stored in its raw form. Since we only care about the base address, and the timing information, this information may take 6 to 8 bytes at most. A 10:1 reduction in the data rate allows a monitor card to output data directly to a mass storage device such as a fast SCSI disk. A 2GB SCSI disk may be enough to store the equivalent of 20GB raw PCI data. At 33MHz clock rate, with 6 bytes/clock, this amounts to 100 seconds of PCI activity, which is well beyond the capability of any logic analyzer.

Of course this solution requires designing an intelligent PCI monitor card with a separate output for a mass storage device or another PC for data collection.

Collecting statistical PCI information

A different approach would be to totally give up on any effort to collect real PCI transaction. Instead, a more intelligent PCI monitor card would listen to the bus and extract statistical information about the various cards plugged on the system. The card would be loaded with the PCI configuration header information of all the cards in the system, allowing it to distinguish between different PCI targets on the system just by looking at the destination address of every transaction. The card would collect statistical information about the PCI traffic in the system:

1. Average, minimum and maximum initial latency cycles, measured from asserting **FRAME#** until the first data word.
2. Average, minimum and maximum burst latency cycles, measured between consecutive words in a burst transfer.
3. Average, minimum and maximum transfer burst length.
4. Average, minimum and maximum wait states before retry cycle generation.
5. Average, minimum and maximum burst length before retry cycle generation.
6. The relative frequency of each base address access and the cycle type (read or write).

Since this type of information requires a fixed amount of memory on the monitor card, it can run indefinitely, collecting statistical information for as long as needed. As a result, we would have a statistical profile of the access patterns of every PCI card in the system.

7.5.2 Generating PCI bus traffic simulation

After we have collected the PCI cycle information, we can simulate the bus behavior, either by running the same PCI cycles sampled by the first two methods, or by generating random PCI cycles based on the statistical patterns calculated according to the third method.

Since simulating the PCI bus behavior using the modified protocol will necessarily generate different vectors on a cycle by cycle basis, we need to define a way to model the same PCI transactions using the modified protocol. The way we choose to use is to calculate for every PCI transaction the **total** latency for that transaction. For transaction requiring the use of target retries we would use the following formula:

$0.5*(N_i + N_{i-1}) - N_0$

Where: N_0 The cycle number of the initial target access cycle.

N_i The cycle number when the target request was completed.

N_{i-1} The cycle number of the last target retry before the request was completed.

We assume that the target latency is the average between the last known latency (when the request was still retried) and the known time it took the transaction to complete - when the request was finally completed.

When simulating PCI transactions using both the standard PCI protocol and the modified PCI protocols we evaluate, we have to modify the raw transactions sampled on the bus in a few ways:

1. We assume that when using the modified PCI protocols, we may begin simulating a new PCI cycle while the previous cycle is still retried, as long as both requests are not for the same target. Without this modification, no performance could be gained from utilizing the idle time on target retries. We cannot do this for multiple requests from the same target since we must keep all the PCI data ordering rules which forbids changing the read order.
2. We remove all the idle bus cycles, when simulating the standard PCI protocol, and the modified PCI protocols. This is necessary because we want to test the efficiency of the protocol, and not the efficiency of the PCI chipset that generated the original PCI requests, which may be lower.
3. We optimize all the transactions captured according to the PCI optimization rules (byte merging, consecutive word merging), again, this is necessary in order to test the bus efficiency and not the chipset efficiency.

7.5.3 Implementing the simulation framework

In order to simulate the PCI protocol and the proposed extensions, a simulation framework was written in C++. We have decided not to write the simulation in verilog for a few reasons:

- The complied, cycle based approach taken by the C++ class library yielded simulation times that are quicker by orders of magnitude compared with interpreted Verilog simulator (which we have access to).
- The clever use of classes and inheritance in C++ allows us to take a generic model and customize it very easily, without duplicating the identical functionality. If we would have written the models in Verilog, which has no language support for object oriented design, the result would have been less clear and much more complex.
- We did not want to be limited by the availability of a verilog software license for running the simulation.

This simulation framework can be divided into three parts:

- A generic cycle based simulation library, supporting synchronous modules, which can be instanciated in a hierarchy using wires. Each synchronous module can have one or more register objects, each storing a single bit using a 6 logic level system (low, high, high-Z, unknown, pull-high and pull-low). Module registers are connected by wire objects, also supporting the 6 logic level system.
- A complete set of generic, parametrized, PCI models including a PCI arbiter, PCI target, PCI master, a PCI backplane, and a PCI monitor/cycle checker. The modules support fine control over the behavior of all the modules including burst length (both master and target), wait states (with finer control over initial latency and burst latency), and retry generation by a target. The class hierarchy allows a user to create a derived class from one of the basic PCI models, and override only the necessary functions. Multiple types of arbiters are also available (fixed priority, round robin with optional multi-transaction-timer), and new ones can be created by the user.
- Modified versions of the generic PCI models supporting the new extensions we have defined above. These models include a target that can specify a retry hint, a PCI master that recognizes the target hint and uses it to request the bus only when the estimated latency expires, and more.

The class library makes heavy use of *virtual functions*. Virtual functions are defined in a base C++ class, but may be dynamically overloaded by a derived class. The PCI master and target classes uses virtual functions to supply a basic working model, which can be modified by overloading its functions, controlling the basic model parameters.

7.5.4 The simulation library classes

What follows here is a more in-depth review of the simulation framework C++ classes and their options.

The simulation library classes

The simulation library is made of 5 basic classes that are used to build general purpose models of synchronous logic devices.

`instance`

This base class contains a pointer to an instance name, and an owner pointer. The operations which can be performed on this class are:

`fullname` Traverses the owner link upwards and concatenates the instance names, building a global instance name, in the form *:instance1.instance2.instance..instanceN*

signal

This base class contains a single synchronous logic value. `logic_state` is an enumeration type containing 6 logic states: Low, High, High-Z, Unknown, Pull-up and Pull-Down. A synchronous logic value is stored in an array of two elements, one representing the current cell value, and the other one representing the new cell value currently being calculated.

When a cell is read, its value is taken from the current cell value, and when a cell is being written, the value is stored in the new cell value. The two element array is indexed by a variable which is toggled every clock cycle, thereby turning all the new cell values into current cell values, and freeing the previous current cell values for the calculation of the new cell values. The operations which can be performed on this class are:

`read`	Return current signal value
`write`	Write new value to signal
`readnew`	Return the new signal value currently being calculated
`ascii`	Return ASCII representation of signal value
`reset`	Writes a High-Z value to the signal
`ishigh`	Returns true if current signal value is High or Pull-Up
`islow`	Returns true if current signal value is Low or Pull-Down
`resolve`	Returns the resolution of the current signal value with a given logic state, according to Table 22.

Logic states	Low	High	High-Z	Unknown	Pull-Down	Pull-Up
Low	Low	Unknown	Low	Unknown	Low	Low
High	Unknown	High	High	Unknown	High	High
High-Z	Low	High	High-Z	Unknown	Pull-Down	Pull-Up
Unknown	Unknown	Unknown	Unknown	Unknown	Unknown	Unknown
Pull-Down	Low	High	Pull-Down	Unknown	Pull-Down	Unknown
Pull-Up	Low	High	Pull-Up	Unknown	Unknown	Pull-Up

Table 22 - logic state resolution table

reg

This class is derived from the `signal` class and the `instance` class. It stores one bit, and can be located inside a design hierarchy by using the `instance` base class. The `reg` class contains a link to a `wire` class. Registers are connected together by hooking them up to the same `wire` object. The `reg` class has also linked list pointers, which are used to link up all the registers in the system in a single list. This is used when all the `reg` objects are scanned at the end of a cycle, and any `reg` which was not modified in that cycle keeps its last value. The operations which can be performed on this class are:

`read`	Return own value if no wire attached, or return wire value if attached to a wire
`write`	Write value to `reg`, updating a wire if attached
`print`	Print the full register name with its value
`reset`	Writes a High-Z value to the register
`operator=`	An overloaded assignment operator (same as `write`).
`Operator<<`	An overloaded ostream operator, printing the register's value.

wire

This class is derived from the signal class and the instance class. It stores one bit, and can be located inside a design hierarchy by using the instance base class. The wire class is not written directly. Instead, it reflects the resolution value of all the registers attached to it. The operations which can be performed on this class are:

add_signal	Links a new register or signal to this wire.
write	Write value to a wire, (called automatically by reg::write).
print	Print the full wire name with its value
reset	Writes a High-Z value to the wire
Operator<<	An overloaded ostream operator, printing the wire's value.

sync_logic

This class is derived from the instance class. The sync_logic object represents a synchronous module, and almost always contains one or more aggregate reg members. The sync_logic class is actually an abstract class, defining a few global operations, and two functions that *must* be implemented by concrete classes derived from it. The sync_logic class has also linked list pointers, which are used to link up all the logic modules in the system in a single list. This list is used by the global_clock and global_reset functions to call all the clock and reset functions of all the modules in the system. The operations which can be performed on this class are:

clock	This pure virtual function must be defined by any concrete class derived from sync_logic. It is called by the simulator every clock cycle, and is supposed to update the object state.
reset	This pure virtual function must be defined by any concrete class derived from sync_logic. It is called by the simulator when the entire simulation is reset.
global_clock	This function is called by the user to advance the simulator state by one clock. It calls the clock functions of all the individual modules.
global_reset	This function is called by the user to reset the simulator state. It calls the reset functions of all the individual modules.

The PCI model classes

The PCI model library is made of a few basic classes that are used to build a complete PCI simulation.

PCI_reg

This class is derived from the sync_logic class. It is mostly a set of registers representing all the basic PCI signals. All the PCI models are derived from this class. The operations which can be performed on this class are:

do_par	Calculate the value of the **PAR#** signal based on the current **AD#** and **C/BE#** values.
do_perr	Calculate the value of the **PERR#** signal based on the current **PAR#** value and the previous cycle **AD#** and **C/BE#** values.
reset	Reset all the PCI registers.

PCI_bus

This class is derived from the `instance` class. It is mostly an aggregate collection of `wire` objects, mapped to all the basic PCI signals, representing a single PCI backplane. The operations which can be performed on this class are:

`print`	Print the entire state of all the PCI signals in a single line.
`add_pci_device`	Link a PCI model (derived from `PCI_reg`) to the bus.

PCI_config_space

This class is simply a data structure containing the standard PCI header. No operations can be performed on this class.

PCI_arbiter

This class is derived from the `PCI_reg` class. It is a PCI model of a central arbitration unit. This is an abstract class and cannot be used directly. It defines the extra registers for the **REQ#** and **GNT#** pairs. The operations which can be performed on this class are:

`reset`	Reset all the PCI arbiter registers, including all the **REQ#** and **GNT#** pairs. It is called automatically by the `sync_logic::global_reset` function.
`link_master`	Links a `PCI_master` object to this arbiter.

PCI_arbiter_fixed

This class is derived from the `PCI_arbiter` class. It is a PCI model of a central arbitration unit, implementing a fixed priority scheme (not really useful). The operations which can be performed on this class are:

`clock`	Implement the arbiter algorithm. It is called automatically by the `sync_logic::global_clock` function.

PCI_arbiter_round_robin

This class is derived from the `PCI_arbiter` class. It is a PCI model of a central arbitration unit, implementing a round robin scheduling algorithm, with an optional Multi-transaction-Timer register (Modeled after the Intel Triton VX, HX, TX arbiter). The operations which can be performed on this class are:

`clock`	Implement the arbiter algorithm. It is called automatically by the `sync_logic::global_clock` function.
`reset`	Reset all the PCI arbiter registers, including all the extra state required by the round robin scheduling algorithm. It is called automatically by the `sync_logic::global_reset` function.
`set_MTT`	Set the Multi-Transaction-Timer value. See page 99 for a description of this value.

PCI_monitor

This class is derived from the `PCI_reg` class. It is a PCI bus monitor, capable of monitoring a `PCI_bus` object, analyzing the traffic on it, extracting statistical parameters, and checking the traffic for PCI protocol violations. The operations which can be performed on this class are:

`reset`	Reset all the PCI monitor registers, including all the specific class information. It is called automatically by the `sync_logic::global_reset` function.
`clock`	perform the bus monitoring function. It is called automatically by the `sync_logic::global_clock` function.
`add_card`	Add a specific address range as a separate entity, for which the monitor will individually track statistical information.
`print_statistics`	Print the statistical results collected during the simulation. This function should be called when the simulation is terminated.

`PCI_profile`

This base class encapsulates a few common variables used by both PCI targets and PCI masters. The operations which can be performed on this class are:

`is_config`	Return true if the current bus transaction is configuration read or configuration write.
`is_IO`	Return true if the current bus transaction is I/O read or I/O write.
`is_mem`	Return true if the current bus transaction is one of the 5 memory read/write transactions.
`is_read`	Return true if the current bus transaction is a read transaction (memory, I/O, or configuration).
`is_write`	Return true if the current bus transaction is a write transaction (memory, I/O, or configuration).

The following operations are declared as `virtual`, and may be overloaded by derived classes:

`read_next_data`	Return the next data item to be driven on the bus (used by the PCI master on writes and by the PCI target on reads).
`read_next_cbe`	Return the next byte enable value to be driven on the bus (used by the PCI master during reads and writes).
`write_next_data`	Write the data value obtained from the PCI bus using the byte enable value also obtained from the PCI bus (used by the PCI master on reads and by the PCI target on writes).
`undo_read_next_data`	Undo the effect of `read_next_data`, guaranteeing that a subsequent call to will return the last value. This is used by a PCI master in the case of a target disconnect without data during a write cycle.

`PCI_target`

This class is derived from the `PCI_reg` and `PCI_profile` classes. It implements a fully configurable PCI target device. The operations which can be performed on this class are:

`reset`	Reset all the PCI target registers. It is called automatically by the `sync_logic::global_reset` function.
`clock`	perform the PCI target function. It is called automatically by the `sync_logic::global_clock` function.
`set_decode_speed`	Set the decode speed for all the PCI commands the target responds to. Decode speed may be fast, medium, slow, or sub.
`set_command_decode_speed`	Set the decode speed for a specific PCI command. A PCI target may respond differently to different PCI commands.

`set_config_reg`	Write a specific value to a PCI target configuration register. This could optionally be done by using a PCI master and sending a configuration write command, but it is much too complex to be convenient.
`set_init_retry_threshold`	Set the maximum number of wait states the target may generate during the first word transfer. Any number of wait states above this number would result in a target retry condition instead. Any value over 16 is ignored, since the maximum number of legal wait states is 16.
`set_burst_retry_threshold`	Set the maximum number of wait states the target may generate during a burst word transfer. Any number of wait states above this number would result in a target retry condition instead. Any value over 8 is ignored, since the maximum number of legal wait states is 8.
`get_words_transferred`	Returns the total number of words transferred by the target so far.

The following operations are declared as `virtual`, and may be overloaded by derived classes:

`read_next_config`	This function reads the next word during configuration space read operation. It can be overloaded in order to implement target specific extended configuration space.
`write_next_config`	This function writes the next word to configuration space during configuration write operation. It can be overloaded in order to implement target specific extended configuration space.
`target_burst_length`	Called at the beginning of a transaction, this function must return the maximum number of words the target may burst. The default function allows an unlimited burst length.
`target_is_last`	Return true if the current word should end the current burst. This function should be overloaded only if the level of control offered by `target_burst_length` is not enough.
`target_initial_wait_states`	Called at the beginning of a transaction, this function must return the number of wait states for the current word, which is the first word in the transaction. The default value is zero wait states. The class automatically keeps at least one wait state for the turnaround cycle on fast decode targets during read.
`target_burst_wait_states`	This function must return the number of wait states for the current word in a burst (not the first word in the transaction). The default value is zero wait states.
`target_wait_states`	This function must return the number of wait states for the current word (This function is called for all words). This function must be overloaded only if the wait state calculation is identical for all words transferred. The default function calls `target_initial_wait_states` or `target_burst_wait_states` when appropriate.
`is_new_retry`	This function returns true if the current cycle must be terminated now with a target retry. This function must be overloaded only if the control level offered by `set_init_retry_threshold` and `set_burst_retry_threshold` is not enough.

`is_retry` This function returns true if the current cycle must be terminated now with a target retry. This function must check for previously retried cycles and retry if they were not completed. This is done automatically for `is_new_retry`, and must be overloaded only if absolutely necessary.

`retry_data` This function should return the value to be driven on the **AD#** lines on a target retry during a read operation. Current PCI masters ignore this value, but we use it as part of our PCI protocol extensions.

`PCI_master`

This abstract class is derived from the `PCI_reg` and `PCI_profile` classes. It implements a fully configurable PCI master device. A PCI master includes an integral, but totally separate PCI target, which must be used to hold configuration space registers for the PCI master. The `PCI_master` class cannot be instantiated directly, because control logic defining the master operation must be added. The operations which can be performed on this class are:

`reset` Reset all the PCI master registers. It is called automatically by the `sync_logic::global_reset` function.

`clock` perform the PCI master function. It is called automatically by the `sync_logic::global_clock` function.

The following operations are declared as `virtual`, and may be overloaded by derived classes:

`master_initial_wait_states` Called once at the beginning of a transaction, this function must return the number of wait states for the current word, which is the first word in the transaction. The default value is zero wait states.

`master_burst_wait_states` Called once for every word transferred (except for the first word), This function must return the number of wait states for the current word in a burst (not the first word in the transaction). The default value is zero wait states.

`master_wait_states` Called once for every word transferred, this function must return the number of wait states for the current word. This function must be overloaded only if the wait state calculation is identical for all words transferred. The default function calls `target_initial_wait_states` or `target_burst_wait_states` when appropriate.

`master_burst_length` Called once per transaction, this function must return the maximum number of words the master may burst. The default function allows an unlimited burst length.

`master_is_last` Called every clock cycle, this function returns true if the current word should end the current burst. This function should be overloaded only if the level of control offered by `master_burst_length` is not enough.

`get_address` Called once per transaction, this function must return the base address for a new transaction. There is no default function, and a derived class must be declared with this function.

`get_command` Called once per transaction, this function must return the 4 bit PCI command code required for a new transaction. There is no default function, and a derived class must be declared with this function.

`request_bus` — Called every clock cycle when the PCI master is in idle state, this function should return true if the bus needs to be requested. The default function always requests the bus to begin a new transaction.

The modified PCI model classes

These classes are derived from the basic PCI model library, and add the new protocol extensions, as described in sections 7.3 and 7.4. The basic PCI functions are modified by overloading the virtual functions defined by the `PCI_master` and `PCI_target` classes.

`PCI_hinting_target`

This class is derived from the `PCI_target` class. It implements the retry hint extension as described in section 7.3. This class behaves identically to `PCI_target`. The added functionality was achieved by overloading the default `retry_data` virtual function.

`PCI_hinted_master`

This class is derived from the `PCI_master` class. It honors the retry hint extension as described in section 7.3. This class behaves identically to `PCI_master`. The added functionality was achieved by overloading the default `request_bus` virtual function. The new operations which can be performed on this class are:

`set_retry_overhead` — This function sets the retry overhead factor, OV. If a target returns a retry hint of N cycles, `PCI_hinted_master` will request the bus after N-OV cycles, to compensate for the initial overhead. Default value is 0.

7.5.5 The simulation environment

We have gathered some statistical information from PCI cycle snapshots sampled by a logic analyzer on a running system, but the results were disappointing. It seems that the lack of bus masters on the PC system we had to use produced samples that contained only single threaded CPU activity. This means that the PCI sample snapshots contained almost exclusively one type of requests and not interleaved traffic generated by multiple masters on the bus.

More “interesting” bus traffic may be observed on systems with more than one bus masters, such as systems with a high performance PCI Ultra-wide SCSI controller, and a good Fast-Ethernet PCI based controller. An Ultra-wide SCSI card may peak at a transfer rate 40MB/s, and sustain this rate when used in a RAID configuration with multiple fast Ultra-wide SCSI disks. A Fast Ethernet card may sustain a transfer rate of up to 10MB/s. A heavy duty file server with more than one SCSI card and more than one Fast Ethernet controller, can have a very high bus utilization ratio, especially if a high performance operating system and application is keeping all the bus master cards busy.

Instead of using the exact traces of the PCI traffic we have sampled, we have extracted statistical information from some of the traffic, and complemented this information with information gathered from data sheets:

- The PC system memory is accessible as a target from the PCI bus through the Host PCI bridge chip. The PC market is dominated today by Intel’s Triton 430HX/VX/TX Pentium chipsets, and the following data applies to these chipsets:

  The PC system memory is accessible from the PCI bus with average initial latency of 8-12 cycles for read, and 3-4 cycles for write. The burst latency is very low, with no wait states inside cache lines (cache lines fall on a 32 byte boundary), and *sometime* a small delay between cache lines. The PC system memory can sustain long bursts (up to a 4K page boundary) if requested by the PCI master.

The Triton 440FX Pentium-Pro chipset will disconnect a PCI memory on a cache line boundary, but will almost always allow a PCI memory-read-line or memory-read-multiple command to continue up to a 4K page boundary.

- CPU writes to the PCI bus have a very low initial latency, 1-3 cycles, and burst length up to 7 words. Burst latency is also low, at about 0.5 cycles on the average.
- CPU reads from the PCI bus don't have a master initial latency (but the target will probably add its initial latency), but don't burst at all.

We have decided to simulate a hypothetical system with the following PCI devices:

- A round robin arbiter modeled after the PCI scheduler in the Intel Triton VX, HX and TX chipsets. This arbiter grants the bus to PCI master in a round robin fashion after the current master releases its **REQ#** line, or when the MTT (Multi-transaction-Timer) expires. This timer is reset each time a new master is granted the bus. Setting a high value for the MTT compensates a master performing many short bursts from losing the bus after every short transaction. Each master is still limited by its Latency-Timer, which releases a masters **REQ#** line when a single transaction is too long.
- A fast target representing system memory. The performance characteristics of this target appears on Table 24.
- A slow target representing a low quality VGA card, or an ISA VGA card behind an PCI-ISA bridge. The performance characteristics of this target is much worse than the system memory target. The performance characteristics of this target also appears on Table 24.
- Two masters representing bus masters (SCSI controller and an Ethernet card). These bus masters are high performance and only drive the system memory PCI target. The performance characteristics of these master appears on Table 23.
- A bus master representing CPU access to the VGA card. This bus master has lower performance, and supports no read bursts. This bus master does not need to access the system memory, because CPU to system memory access is handled by the PCI host bridge, and this traffic does not appear on the PCI bus. In fact, a PCI master and a PCI target may communicate concurrently with CPU traffic to/from the system memory. This is called *Peer Concurrency* in Intel's terminology.

Also, the following assumptions were made:

- Each bus master has a predetermined amount of traffic to transfer. Once this traffic is done, the master is idle until the end of the simulation, freeing the bus for other bus masters.
- Every PCI device has its own pseudo random number generator. This ensures that the exact transaction sequence of every master, and the exact behavior of every target is repeatable, even when the transaction order changes between masters due to protocol optimizations. Every random number generator is initialized with a unique seed, allowing simulation results to be regenerated by using the same seed values.
- The ratio between read and write has been determined to be the classical 80/20 for program read/write. Since disk read from programs causes SCSI or Ethernet *write* to system memory, these masters use the opposite ratio, 20/80.

7.5.6 Simulation results and conclusions

Latency Hint simulation base parameters

We have ran the system described above, with the parameters in Table 23 and Table 24. We have then ran additional runs, each time varying one or more parameters.

	CPU Host bridge	PCI bus master (Fast Ethernet)	PCI bus master (SCSI)
Name	PM1	PM2	PM3
Read/Write ratio	80/20 Random probability	20/80 Random probability	20/80 Random probability
Initial wait states	0	0	0
Burst wait states	0	0	0
Burst length	Read: 1 Write: Random (1 to 8)	Random: 8 to 384	128
Master Latency Timer	48	48	48
Retry overhead	0	N.A.	N.A.
Transaction count	200	100	100

Table 23 - Latency hint simulation master parameters

PM1 is the Host PCI bridge master, capable of write bursts but not read bursts. Most transactions are read.

PM2 is a Fast Ethernet PCI master. Burst length is random, representing an Ethernet frame size, up to 1536 bytes. Most transactions are PCI write, because most of the time the data is read by the CPU.

PM3 is an Ultra Wide SCSI PCI master. Burst length is fixed at 128, representing a 512 byte disk sector size. Most transactions are PCI write, because most of the time the data is read by the CPU.

Here is a brief description of all the master performance parameters:

Read/Write ratio

This number describes the percentage ratio between PCI read commands and write commands for this master.

Initial wait states

The number of wait states before the master is ready to send or receive the first word.

Burst wait states

The number of wait states before the master is ready to send or receive burst words, after the first word.

Burst length

The number of words the master is intending to send or accept.

Master Latency Timer

When a master is bursting and its **GNT#** line is deasserted during the burst, it has to release the bus when his MLT expires. MLT counts down cycles, beginning on each burst cycle.

Retry overhead (for latency hint aware masters only)

When a hinting target sends a latency hint, it can estimate when it will be ready, but it cannot estimate the master retry overhead. The master retry overhead is measured from **REQ#** going active until **FRAME#** goes active. When a master receives a latency hint, it subtracts the retry overhead parameter from the delay received by the target, to compensate for this extra time.

Transaction count

The total number of transaction the master runs before stopping. We can disable a master by setting its transaction count to 0.

Type	VGA target	System memory target
Name	PT1	PT2
Decode speed	Medium	Fast
Initial wait states	Random (0 to 40)	Read: Random (8 to 12) Write: Random (3 to 4)
Burst wait states	0	32 byte boundary: 1 Otherwise: 0
Burst length	Random (0 to 10)	Stop at 4K boundary
Initial retry threshold	16	16
Burst retry threshold	8	8

Table 24 - Latency hint simulation target parameters

Here is a brief description of all the target performance parameters:

Decode speed

A target has four possible speeds for asserting **DEVSEL#** after **FRAME#**. Fast, Medium, Slow, and Sub. Sub is used by subtractive decoders, which only decodes cycles ignored by other targets.

Initial wait states

The number of wait states before the target is ready to send or receive the first word.

Burst wait states

The number of wait states before the target is ready to send or receive burst words, after the first word.

Burst length

The maximum number of words the target is able to send or accept.

Initial retry threshold

A target that has to wait N wait states may do one of: Wait N cycles with **TRDY#** deasserted and then assert **TRDY#**, or retry the cycle (with or without a latency hint). If N is greater than the initial retry threshold, a retry will be generated. otherwise, wait states will be added. This parameter only applies to the first word in a burst, and is limited to 16 due to PCI 2.1 restrictions.

Burst retry threshold

This parameter is the same as the Initial retry threshold, but only applies to subsequent words in a burst (**not** the first word in a burst), and is limited to 8 due to PCI 2.1 restrictions.

Single master results

We have run a simulation of a single master device, as a function of two parameters, the target initial retry threshold, and the master retry overhead parameter.

As we have explained before, the target initial retry threshold parameter determines when will a target retry a cycle, and when it will insert wait states instead. To use this parameter, we must assume the target can estimate how many clock cycles it will take for a requested data word to be available. If the estimated delay is more than or equal the initial retry threshold, the target will retry. If the delay is less than the initial retry threshold, the target would insert wait states until the requested data will be available.

The master retry overhead parameter is used for the latency hint extension. When a master is retried with a target latency hint of N cycles, it should actually request the bus a few cycles before these N clock cycles would pass. The reason for that is that the retry hint measures the estimated retry time from the target's point of view, which is when the next request should be received. The master, however, has additional overhead involving requesting the bus and waiting for bus grant. Therefore, the master should request the bus a few cycles ahead.

The following graph shows the simulation results. 100 bus transactions, totaling 163 words, were run. The Y graph axis measures the number of clock cycles the simulation had to run. The lower the number, the less overhead, and the performance is better. The measurements are normalized and presented as percentage of the highest results (the slowest simulation).

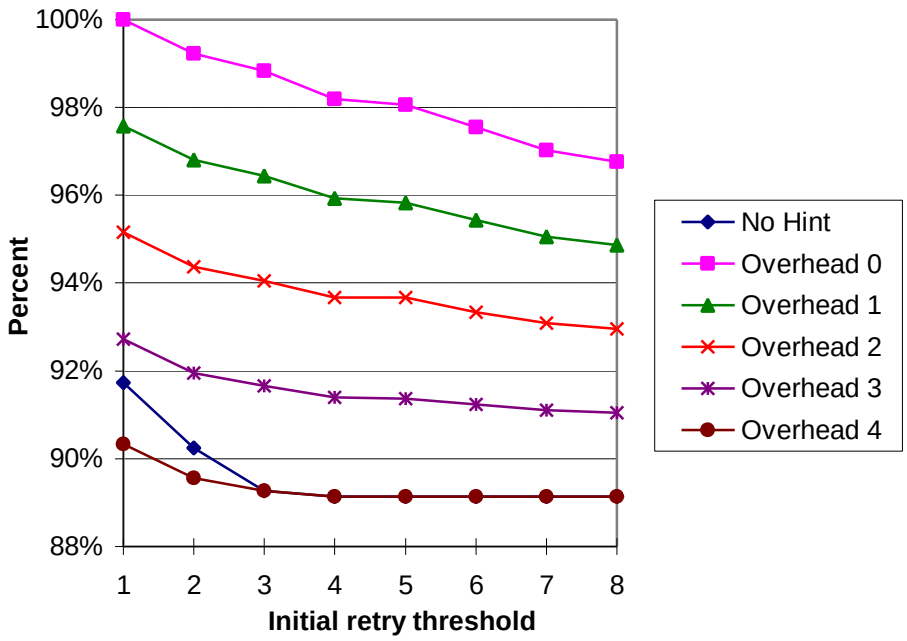


Figure 61 - Single Master performance as a function of the initial retry threshold and master retry overhead

The conclusions from this simulation are simple:

- Any initial retry threshold value below 4 is sub optimal, because it takes more time to retry the cycle, than to actually wait the requested number of wait states! with a single master simulation it takes 4 cycles to release the bus and request it again, therefore the performance flattens at any value above or equal to 4, because with 4 expected wait states the master can retry the cycle and get the data in time without causing additional delay.
- The optimum retry overhead is also 4 cycles. Any values above 4 means the bus would be requested too early, which would hurt performance with multiple masters.

Varying the arbiter Multi Transaction Timer

Now we want to test the system performance of multiple bus masters, using the standard PCI protocol. We try different combinations of two parameters, the target initial retry threshold (which we have discussed before), and the arbiter MTT (Multi Transaction Timer). The MTT determines the minimum time slice a master is guaranteed to be granted, even if it is split to more than one bus transaction. It is important to realize the difference between this parameter and the master latency timer, which determines the minimum time slice of a *single* burst transaction.

The following graph shows the simulation results. This time we have defined 3 masters, with 100 bus transactions for the host bridge, and 10 transactions each for the SCSI and Ethernet bus masters, which have much longer bursts. The host bridge has transferred 163 words to the VGA target, the SCSI master has transferred 1280 words to the system memory, and the Ethernet masters has transferred 1840 words to the system memory. We have varied the target initial retry threshold, with values of 1, 2, 3 4, 5, 6, 8, 12, and 16. The measurements in percent are normalized to the highest results (the slowest simulation). The Y graph axis measures the number of clock cycles the simulation had to run. The lower the number, the less overhead, and the performance is better.

The MLT (Master latency Timer) values for all the masters were taken to be 48 at this stage. Later, we will try different values of MLT when we cut down the number of combinations on other parameters.

Surprisingly enough, we find two points where the performance peaks, at MTT values of 20, and 40 to 44. We also find the best retry threshold value to be 2, with 3, 1, 6, and 5 all before 4, which was the best value for a single master. We also find that for threshold values over 4, there is usually only one performance peak, at around MTT = 20.

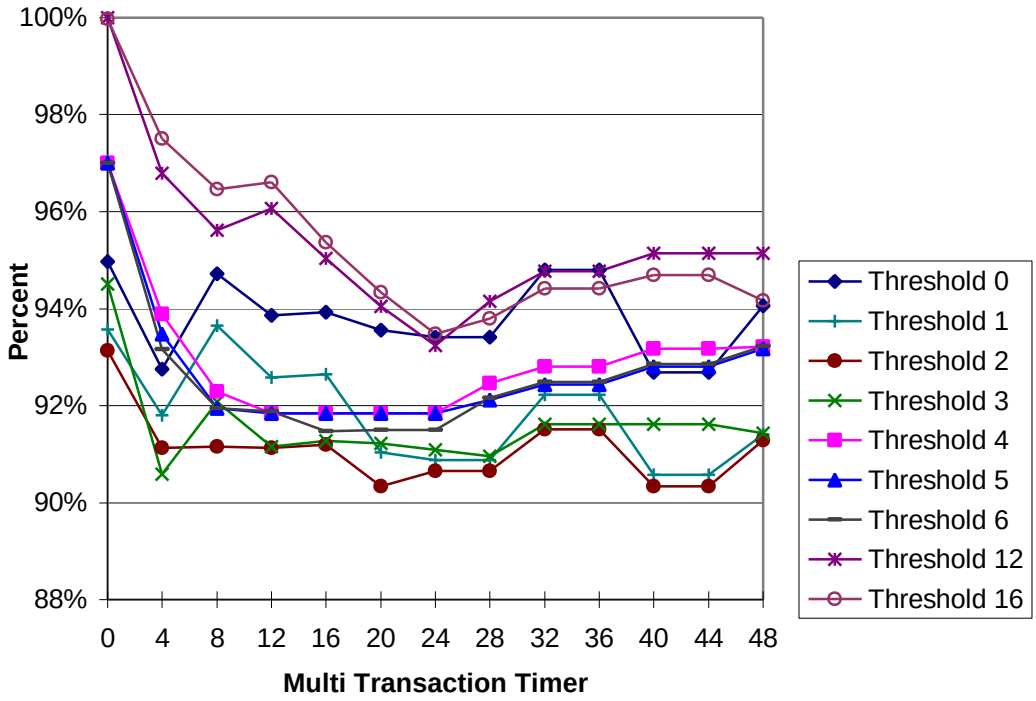


Figure 62 - Multiple Master performance as a function of the arbiter MTT and the Target initial retry threshold

Varying the Master Latency Timer

The next parameter we are testing is the Master Latency Timer, or MLT. The MLT determines the minimum time slice a master is guaranteed for a single burst. It is important to realize the difference between this parameter and the MTT, which guarantees a minimum time slice even for multiple bursts.

We have ran the same simulation conditions as before, but this time we are varying the MLT between 8, 16, 24, 32, 40 for the initial retry threshold of 2.

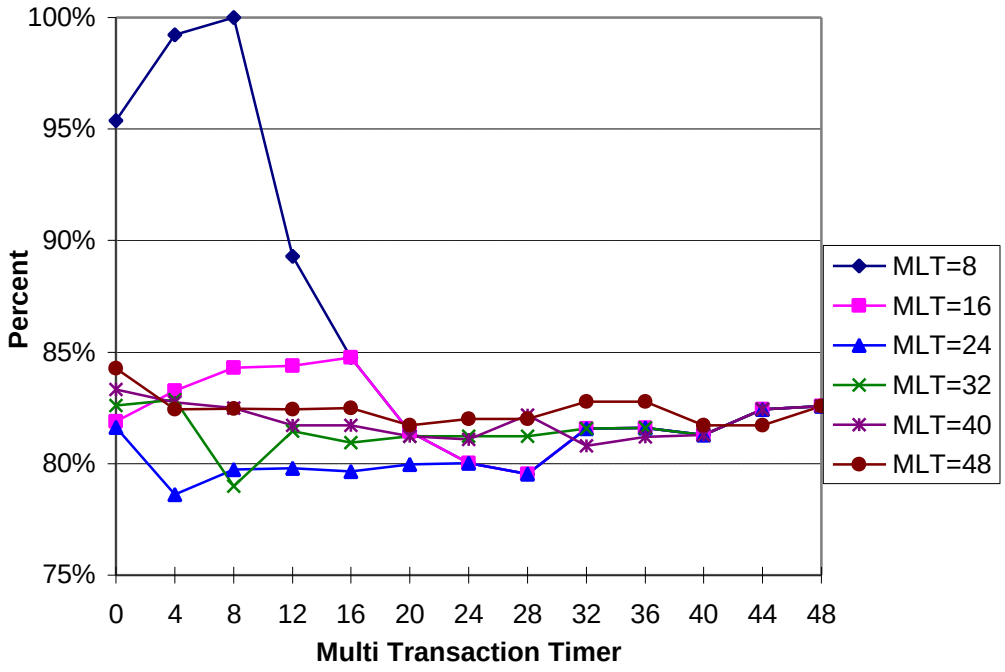


Figure 63 - Multiple Master performance as a function of the arbiter MTT and the MLT

One observation we can make, is that for all the simulations for which MLT>MTT, we get the same results for MTT regardless of MLT (the graphs merge at the MLT=MTT).

Simulating the retry hint extension

After measuring the system performance by varying the various PCI parameters, we now want to test the retry hint extension we have defined. We have picked up the best set of parameters, which are MLT=24 and MLT=32, and ran them with master retry overhead values of 1 through 16, as well as with no retry hint extensions (for reference). The best results are summarized in these graphs:

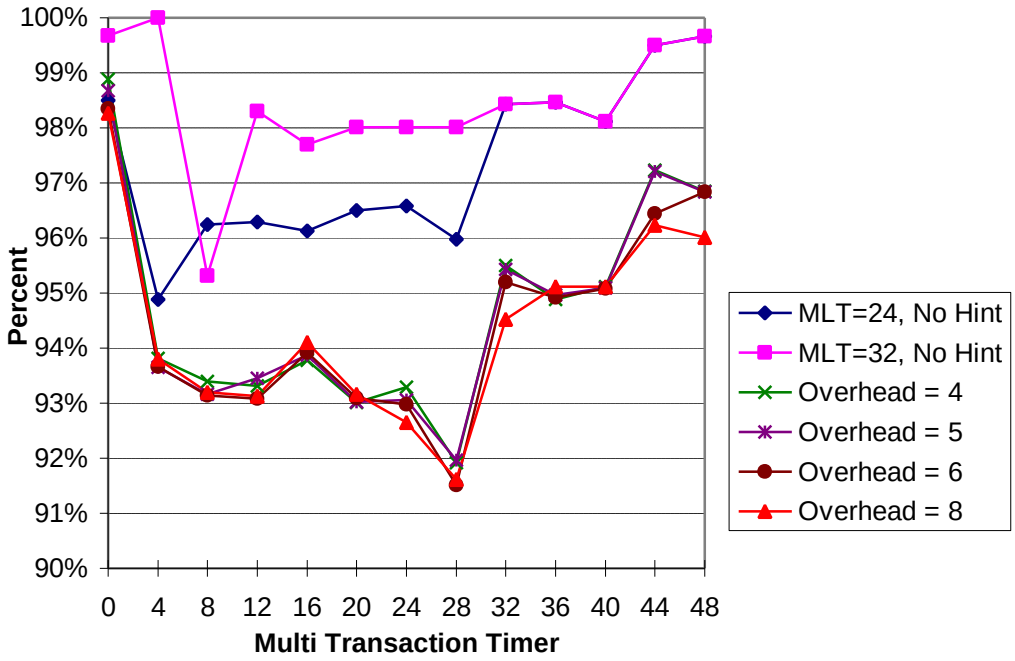


Figure 64 - Multiple Master performance with target retry hint, MLT=24

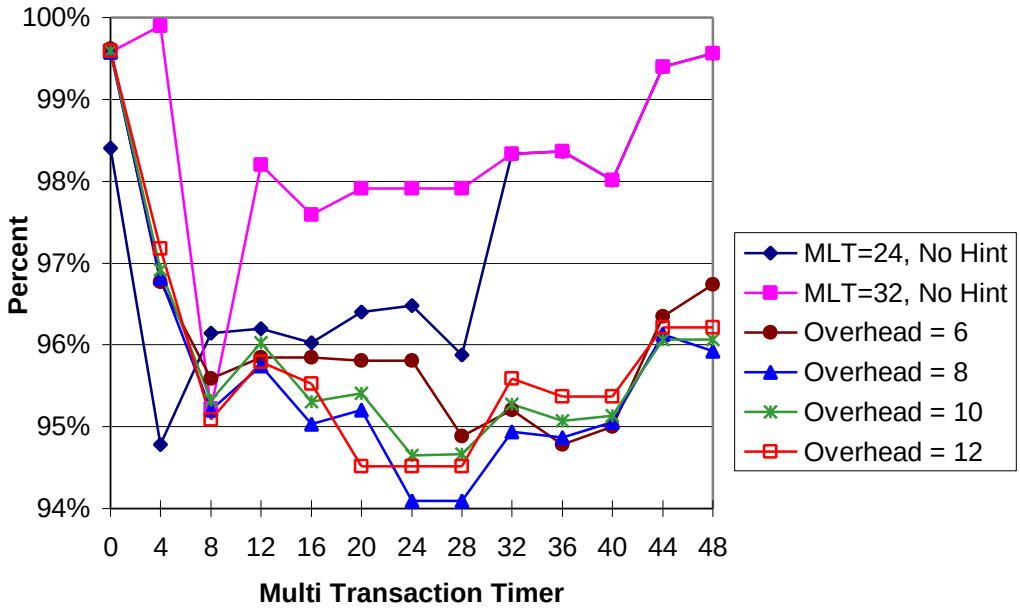


Figure 65 - Multiple Master performance with target retry hint, MLT=32

The performance improvement is obvious for both MLT=24 and MLT=32, even though it seems that we can get better results with MLT=24. We also see that the best results are for master retry overhead values of 6 or 8.

Latency Hint simulation explanations

Before running the simulation, we would have expected larger MTT and MLT values to give better results (at the cost of the individual latency of each master), but the results shows that making MTT and MLT larger does not necessarily increase system throughput.

After closely examining the bus cycle dumps of the simulations above, we have came up with a few explanations for the lack of positive correlation between MTT and any performance increase of the simulations:

1. By changing the MTT, we change the transaction order between different masters. Since our simulation stops only after all masters have done their work, there is always one master which finishes last, and whose bursts are never broken by other masters requesting the bus, since they already finished their work.

   Now, if MTT is made longer, a master which may have finished earlier before might now finish later, and cause the last master's transaction to be broken, adding extra latency.

2. A special condition occurs when masters are retried on write cycles. The masters release **REQ#** on the same cycle they release the bus, and immediately retry the transaction on the same cycle **GNT#** is deasserted. This is perfectly legal, since arbitration rules are that a master may begin a cycle on the same clock it loses **GNT**. (It has no way of knowing it has lost **GNT#** in the same cycle, only a clock later). The result is that with MTT or MLT long enough to hold **GNT#** low during all the retry, we actually end with two consecutive write retries, without other bus masters being able to access the bus between, which is inefficient. If MTT or MLT is short enough to remove GNT before the master is even retried for the first time, the master would not be able to run two consecutive retries, which may improve the bus utilization.

Latency Hint simulation conclusions

We have ran several simulations, comparing multiple master performance of the standard PCI protocol at various values.

Before running the simulation, we would have expected larger MTT and MLT values to give better results (at the cost of the individual latency of each master), but the results shows that increasing MTT and MLT does not necessarily increase system throughput.

We have then compared the best results achieved using the standard PCI protocol, and compared them with results achieved with an improved PCI protocol including a target retry hint extension, and for the same simulation setup we have found performance increase of up to 4% over the standard protocol.

Further work

The simulations we have run here had been running for an average of 6000 clock cycles each. With hundreds of simulations run, we could not run significantly longer simulations, due to time constraints. More accurate results could be achieved by running longer simulations, as well as with multiple data sets. We could test more combinations of the retry hint, target threshold, MLT and MTT parameters.

Also, the our simulation model assumed the VGA target had a relatively long latency, because this was the main problem the latency hint extension was designed to solve (reducing the impact of long latency devices on the whole system). We could run these simulations for more realistic scenarios where the VGA card has a lower latency and see if we still get any improvements.

Instead, we could also write simulation models for the other extensions we have suggested earlier, namely the split transaction enhancements. These extensions may achieve better speed improvements, but their effect on the PCI bus is more complex in terms of transaction ordering. The PCI standard specifies very specific transaction ordering rules, and we must inspect their implications on the split transaction enhancements.

8. Bibliography

(Aichinger, 1992) Aichinger, Barbara P. *FutureBus+ as an I/O Bus: Profile B,* 19th International Symposium on Computer Architecture, pp. 300-307, Gold Coast, Australia, May 1992.

(AMD, 1995) *MACH 1, 2, 3, and 4 Family Data Book.* AMD, 1995.

(Balakrishnan, 1984) Balakrishnan, R.V. *The proposed IEEE 896 Futurebus - A Solution to the Bus Driving Problem.* IEEE Micro, August 1984.

(Brown & Rose, 1996) Brown, Stephen. and Rose, Jonathan. *FPGA and CPLD Architectures: A Tutorial*. IEEE Design and Test of Computers, Summer 1996.

(Byte, 1987) *The Macintosh II.* Byte, April 1987.

(Force, 1997) *CompactPCI vs. Industrial PC's*, Force Computers, 1997.

(IBM, 1989) *Special Issue on the Micro Channel Architecture,* IBM Personal Systems Technical Journal, Issue 4, 1989.

(IEEE, 1990) *Draft Standard: FutureBus+ P896.1 Logical Layer Specification Draft 8.2 P896.1/D8.2*, IEEE Computer Society Press, 1990.

(IEEE, 1995) *Physical and Environmental Layers for PCI Mezzanine Cards: PMC,* P1386.1/D2.0, IEEE, April 1995.

(IEEE, 1996) *HiRelPCI Draft,* unapproved IEEE Standards Draft, November 1996.

(Intel, 1996) *Accelerated Graphics Port Interface Specification, Revision 1.0,* Intel Corporation, July 1996.

(Intel, 1997) *Low Pin Count Interface Specification, Revision 1.0 ,* Intel Corporation, 1997.

(Kendall, 1994) Kendall, Guy W. *Inside the PCI Local Bus.* Byte, Feb. 1994.

(Kunkel & Smith, 1986) Kunkel, Steven R. and Smith, James E., *Optimal pipelining in supercomputers.* IEEE, 1986.

(Messmer, 1995) Messmer, Hans-Peter, *The Indispensable PC Hardware Book*. 2nd. Ed., Addison Wesley, 1995.

(MR, 1987) *NuBUS Raised From Obscurity by Macintosh II.* Microprocessor Report, Oct. 1987.

(NI, 1997) *PXI Specification, Revision 1.0 ,* National Instruments, August 1997.

(PC104, 1997) *PC/104-Plus Standard*, PC/104 Consortium, February 1997.

(PCMCIA, 1997) *PC Card Standard, March 1997 Release,* Personal Computer Memory Card International Association (PCMCIA), March 1997.

(PICMG, 1995) *CompactPCI Specification, Revision 1.0*, PCI Industrial Computer Manufacturers Group (PICMG), November 1995.

(PCISIG, 1994) *PCI to PCI Bridge Architecture Specification, Revision 1.0* . PCI Special Interest Group, April 1994.

(PCISIG, 1995) *PCI Local Bus Specifications, Revision 2.1* . PCI Special Interest Group, June 1995.

(PCISIG, 1996) *SmallPCI Specification, Revision 1.0* . PCI Special Interest Group, 1996.

(Pescatore, 1989) Pescatore, Carmine John Jr. *The MicroChannel Architecture: An Adapter Designer's Perspective.* MS thesis, Department of Electrical Engineering, Duke University, 1989.

(Peterson, 1989) Peterson, Wade D. *The VMEbus Handbook.* VITA, 1989.

(Shanley, 1995) Shanley, Tom. *PCI System Architecture.* MindShare, Inc., 1995.

(Shanley & Anderson, 1995) Shanley, Tom and Anderson, Don. *ISA System Architecture*. MindShare, Inc., 3rd Edition, 1995.

(Shanley & Anderson, 1995b) Shanley, Tom and Anderson, Don. *CardBus System Architecture*. MindShare, Inc., 1st Edition, 1995.

(Slater & Thorson, 1992) Slater, M. and Thorson, M. *Local Buses Poised to Enter PC Mainstream.* Microprocessor Report, 6(9):7-13, July 8, 1992.

(Solari, 1992) Solari, Edward. *ISA & EISA Theory and Operation.* Annabooks, 1992.

(Theus, 1992) Theus, John. *FutureBus+ Coming of Age*. A three-part series on the FutureBus+ standard. MicroProcessor Report, 6(7-9), May-July 1992.

(Tving, 1993) Tving, Ivan. *Multiprocessor interconnections using SCI.* ID-DTH, 1993.

(VITA, 1994) *VME64 Draft Specification Revision 1.10*, VITA, 1994.

(Wang et. Al., 1995) Wang, Karl. et. al., *Designing the MPC105 PCI Bridge/Memory Controller*. IEEE Micro, April 1995.

Appendix A - ISA Protocol summary

The ISA bus was the workhorse of the PC industry for the last 15 years, and is found in every PC today, so it is worth a close examination.

The ISA bus was originally an 8 bit wide bus designed for the original IBM-PC, and later extended to 16 bit for the IBM-AT. As a result, the ISA bus has an 8 bit mode, using one connector, and a 16 bit mode with more signals on an extra connector which is in-line with the 8 bit connector.

XT Bus signal summary

These signals are on the 62 pin edge connector that appeared in the original IBM PC.

SD7..0 This is the 8 bit system data bus. During 8 bit cycles, these lines are used to hold 8 bit data for both odd and even addresses. During 16 bit cycles, these lines are used to hold data for even addresses.

SA19..0 This is the latched system address bus. It is driven by the platform, and contains the low 20 address bits. The **SA** bus is always updated at the positive edge of BALE, and latched at the negative edge of **BALE**.

BALE This is the Buffered Address Latch Enable line. The system address bus is updated while **BALE** is high, and latched on the falling edge of **BALE**.

AEN **AEN** is asserted when one of the onboard DMA controllers has been granted the bus, and instructs non-DMA I/O Resources to ignore the current cycle.

SMEMR* System Memory Read Command. This line is asserted to begin a memory read bus cycle. Since this signal originates on the 8 bit PC bus, it is active only while accessing the low 1MB memory area. Any memory read operation above 1M will not trigger this line to prevent 8 bit cards, which decode only 20 address bits, from responding to this operation.

SMEMW* System Memory Write Command. This line is asserted to begin a memory write bus cycle. Since this signal originates on the 8 bit PC bus, it is active only while accessing the low 1MB memory area. Any memory read operation above 1M will not trigger this line to prevent 8 bit cards, which decode only 20 address bits, from responding to this operation.

IOR* I/O Read Command. This line is asserted during an I/O read bus cycle.

IOW* I/O Write Command. This line is asserted during an I/O write bus cycle.

NOWS* No Wait States. This signals is asserted by a memory target when it wants to reduce the number of wait states in a cycle.

IOCHRDY I/O Channel Ready. This signal is asserted by a memory target when is need to extend a normal cycle with one or more wait states.

IOCHCHK I/O Channel Check. This signal generates a Non Maskable Interrupt, which unlike what its name implies, **can** be masked by either the channel check enable/disable gate (port 61 bit 3), or the NMI enable/disable gate (port 70 bit 7).

RESET This signal resets all the peripherals on the bus. It is usually generated only when the computer is reset.

BCLK This line is the system bus clock. All the bus signals are generated relative to this clock. On the original XT, it ran at 4.77Mhz. On modern PCs, it is usually running at 8MHz or 8.33MHz, but some system have the option to overclock this up to 16Mhz. Not all cards can work at this faster rate.

OSC A 14.31818MHz clock signal that is not synchronized to any bus signal. This signal is a multiple of the color burst frequency (3.57MHz) and was used by video adapter display cards in the past.

REFRESH* This signal is asserted when the system board refresh logic is executing a refresh cycle.

IRQ3-7,9 These 6 lines allow edge triggered interrupts to be generated by the interface cards.

DRQ1-3 These 3 lines are data request lines, used during DMA cycles. They form 3 pairs of lines (together with **DACK1-3**). When using a DMA channel, a card should request service by driving the **DRQ** line, and receive an acknowledge by the **DACK** line.

DACK1-3* See **DACK1-3** above.

TC Transfer Complete. This signal, generated by the internal DMA controllers, indicates the end of the DMA block transfer.

AT Bus Signal summary

These signals are on the 36 pin edge connector that was added for the IBM-AT.

SD8..15 System Data Bus. These are extra 8 bits which expand the system data bus to 16 bits. During 16 bit cycles, these lines are used to hold data for odd addresses.

LA17..23 These lines, driven by the platform, contain the upper 7 address bits. Unlike **SA19..0**, these signals are unlatched, and appear at least one BCLK cycle before **SA19..0**, which means **LA23..17** can be used for address decoding a cycle earlier. If these line are needed during the whole memory cycle, they must be latched by **BALE**, because they advance to the next address before the end of the current cycle.

SBHE* This signal, driven by the bus master during 16 bit cycles, indicates that the 8 upper data lines (SD8..15) contain valid data.

MEMCS16* This signal, driven by a memory target, indicates the card's ability to handle 16 bit memory transfers. This also makes ISA cycles shorter. This signal is ignored during I/O cycles.

IOCS16* This signal, driven by an I/O target, indicating the ability to handle 16 bit I/O transfers. This also makes ISA cycles shorter. This signal is ignored during memory cycles.

MASTER* This signal can be used by bus master cards to gain access to the system bus. In order to gain bus access, a bus master should assert **DRQ*** and wait for a **DACK*** signal from the DMA controller. The bus master can then drive **MASTER***, relinquishes the system bus from the DMA controller. This allows the bus master to drive the ISA bus directly. It also allows the bus master to drive more than one word before releasing the bus, yielding a higher transfer rate than what possible using the internal DMA controller, which requires a **DRQ*** / **DACK*** handshake for every word. As a side effect, **AEN*** is not asserted if **MASTER*** is asserted, allowing the bus master card full access to any other I/O card in the system.

MEMR* Memory Read Command. This line is asserted to begin a memory read bus cycle. This signal is used by targets wishing to decode any address in the 16 Mbyte range.

MEMW* Memory Write Command. This line is asserted to begin a memory write bus cycle. This signal is used by targets wishing to decode any address in the 16 Mbyte range.

IRQ10-12,14,15 These 5 additional lines allow more edge triggered interrupts to be generated by the interface cards.

Simple Memory and I/O write access cycles

ISA bus cycles are always synchronized to the BCLK signal, and the length of every bus cycle is measured by the number of BCLK cycles it takes to complete. It must be noted though, that the BCLK signal is regenerated from the local CPU bus control signals at the beginning of every ISA bus cycle, which means that the BCLK signal may show some jitter when viewed on an oscilloscope.

The length of ISA bus cycle is summarized in the following table:

Cycle type	8 bit	16 bit
Standard	4 wait states	1 wait state
Shortened (Zero wait state)	1, 2, or 3 wait states	0 wait states (memory only)
Ready	more than 4 wait states	more than 1 wait state

Table 25 - ISA wait states

Since each cycle requires at least 2 BCLK cycles, the shortest ISA bus cycle takes 2 BCLK cycles, and standard 16 bit memory ISA bus cycles takes 3 BCLK cycles.

Here is a typical memory or I/O transfer:

1. System drives the **LA[23:17]** bus.
2. A 16 bit memory card may perform chip select on **LA[23:17]** and drive **MEMCS16*** if the address is in the card's range. If no **MEMCS16*** is detected within the correct time window, this will be an 8 bit cycle.
3. System drives **BALE** high, enabling **SA[19:0]** and **SBHE*** on the ISA bus.
4. A 16 bit I/O card may perform chip select on **SA[19:0]** and drive **IOCS16*** if the address is in the card's range. If no **IOCS16*** is detected within the correct time window, this will be an 8 bit cycle.
5. System drives one of the command lines (**IOR***, **IOW***, **MEMR***, **MEMW***, **SMEMR***, **SMEMW***). In case of an address below 1Mbyte, both the **MEMx*** lines are driven, and **SMEMx*** lines are driven (about 60 ns later).
6. If this is a write operation, the card may latch the data now.
7. The card can extend the cycle by one or more wait states if it asserts the **IOCHRDY*** line. It can also shorten the cycle by one or more wait states if it drives the **NOWS*** line.
8. The cycle ends after a predetermined number of clock cycles (depending on the cycle type). If **IOCHRDY*** was asserted before, the cycle will end only when **IOCHRDY*** is deasserted.
9. If this is a read cycle, the system will latch the read data when the command line is deasserted.

The data bus content and the cycle type can be determined according to the following table:

SBHE*	**SA0**	**Data driven on write**	**Data returned on 16 bit read**	**Data returned on 8 bit read**
0	0	**SD[7:0]** Even **SD[15:8]** Odd	**SD[7:0]** Even **SD[15:8]** Odd	**SD[7:0]** Even
0	1	**SD[7:0]** Odd **SD[15:8]** Odd	**SD[15:8]** Odd	**SD[7:0]** Odd
1	0	**SD[7:0]** Even		**SD[7:0]** Even
1	1	**SD[7:0]** Odd		**SD[7:0]** Odd

Table 26 - ISA cycle width encoding

Notes:

1. "Odd" means data byte on odd byte address, while "Even" means data byte on an even byte address.
2. If **SBHE*** was asserted but no **MEMCS16*** or **IOCS16*** was driven, the cycle become an 8 bit cycle, as if **SBHE*** was not driven. This will split a 16 bit cycle into two 8 bit cycles.
3. When **SBHE*** = 0, **SA0** = 1, The same write data is driven on both **SD[7:0]** and **SD[15:8]**. 8 bit devices will use **SD[7:0]**, while 16 bit devices will use **SD[15:8]**.
4. It can also be observed that 16 bit memory targets requires no byte lane steering (a data path connecting **SD[7:0]** to the board's internal data bus D[15:8]), as long as they are plugged only in a 16 bit system. In this case Odd data is always on **SD[15:8]** and even data on **SD[7:0]**. If the card needs to be backward compatible with 8 bit systems, it must handle the case of **SBHE*** = **SA0** = 1, where Odd data is on **SD[7:0]**, and requires byte steering logic to get it to/from D[15:8] on the card's internal memory bus.

The following sections will illustrates the exact timing relationships between the bus signals during different types of memory or I/O cycles.

ISA 8 bit memory and I/O standard cycles

As it can be seen, the standard 8 bit cycle takes 6 clock cycles to complete. The diagram shows the timing for both read and write. The write data indicates when the data written is valid, while the read data indicates when the data is latched by the system.

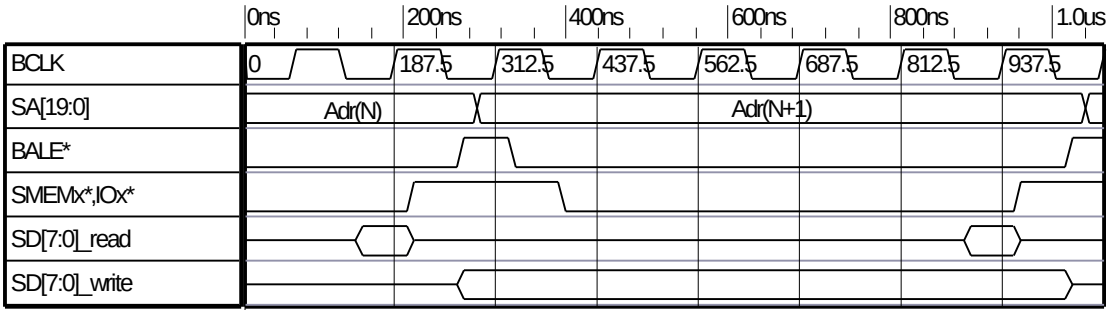


Figure 66 - ISA 8 bit standard memory and I/O cycle

ISA 8 bit memory and I/O no wait states cycles

As it can be seen, the 8 bit no wait states cycle takes 3 to 5 clock cycles to complete, depending how early **NOWS*** is asserted. The diagram below shows the timing for a 1 wait state, 3 clock cycles, memory or I/O cycle.

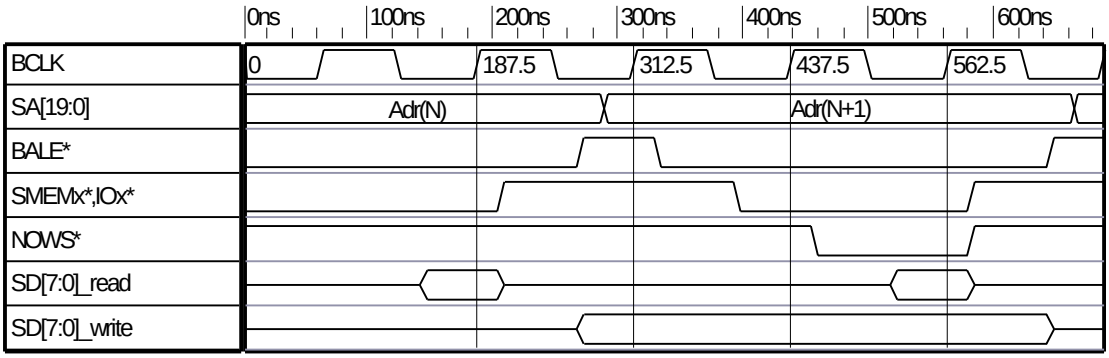


Figure 67 - ISA 8 bit no wait states memory and I/O cycle

ISA 8 bit memory and I/O "ready cycles"

The "ready cycle" is a cycle that is delayed by one or more wait states using the I/O channel ready signal, or **IOCHRDY***. (Hence the name "ready cycle"). The 8 bit ready cycle, takes 7 or more clock cycles to complete, depending on when **IOCHRDY*** is released. The diagram below shows the timing for a 5 wait state, 7 clock cycles, memory or I/O cycle. The shaded area in the **IOCHRDY*** timing indicates that since a minimum of 4 wait states is mandatory on 8 bit cycles (unless **NOWS*** is asserted before), it doesn't matter when IOCHRDY* is asserted, as long as its done before the end of the cycle.

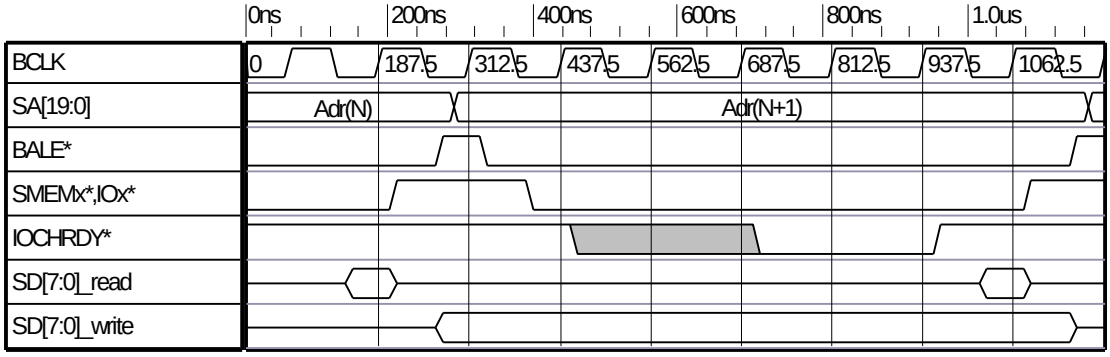


Figure 68 - ISA 8 bit, memory and I/O "ready cycle"

ISA 16 bit memory and I/O standard cycles

As it can be seen, the standard 16 bit cycle is significantly faster than the 8 bit cycle, and takes only 3 clock cycles to complete. The signals for I/O and memory cycles are slightly different in the 16 bit mode, as it can be seen in the two different diagrams. The main difference comes from the fact that while memory cycles can use **LA[23:17]** to generate **MEMCS16*** early enough, 16 bit I/O cycles must decode **SA[15:0]**, which are available only after **BALE** is high. Therefore, **IOCS16*** is sampled one clock later than when **MEMCS16*** would have been sampled.

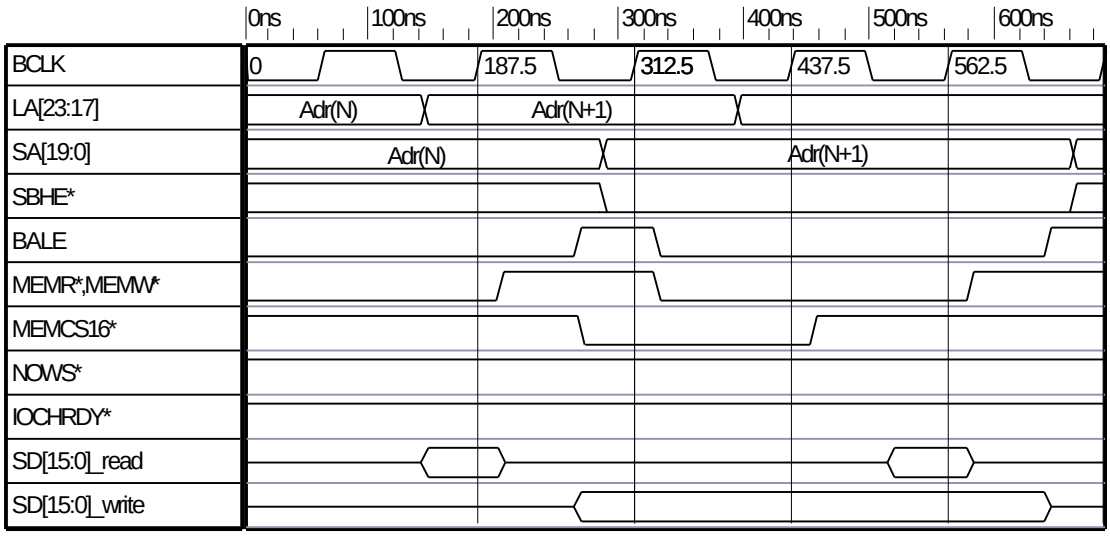


Figure 69 - ISA 16 bit standard memory cycle

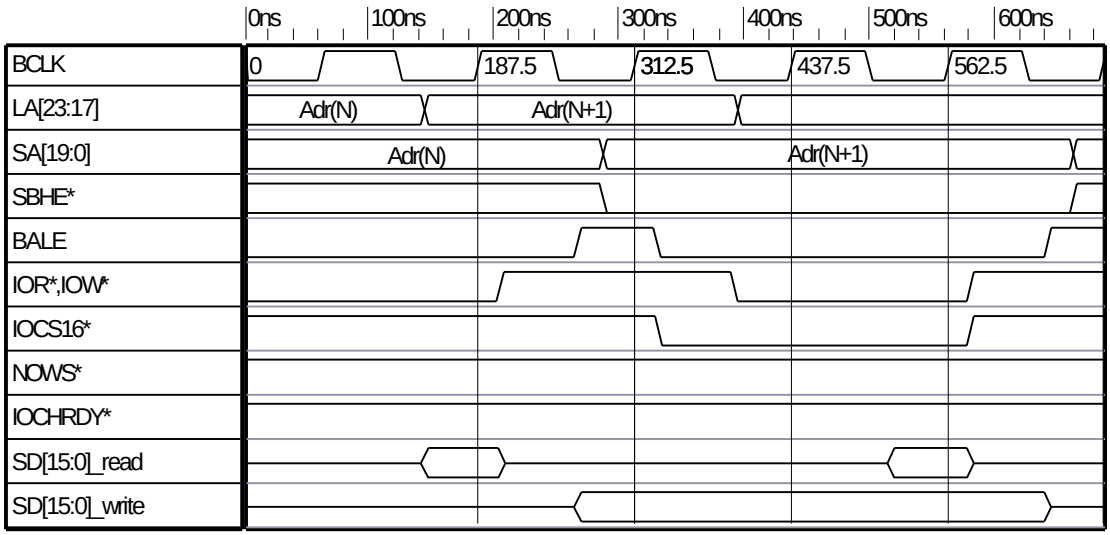


Figure 70 - ISA 16 bit standard I/O cycle

ISA 16 bit memory no wait states cycles

The 16 bit no wait states cycles are only available for memory operations. A 16 bit no wait states cycle can be as short as 2 cycles (true zero wait states).

ISA 16 bit memory and I/O "ready cycles"

The 16 bit "ready cycles" are available for both memory and I/O. Their internal timing is slightly different, but the overall cycle length is the same. The reason for that was explained before when we discussed the 16 bit standard cycles.

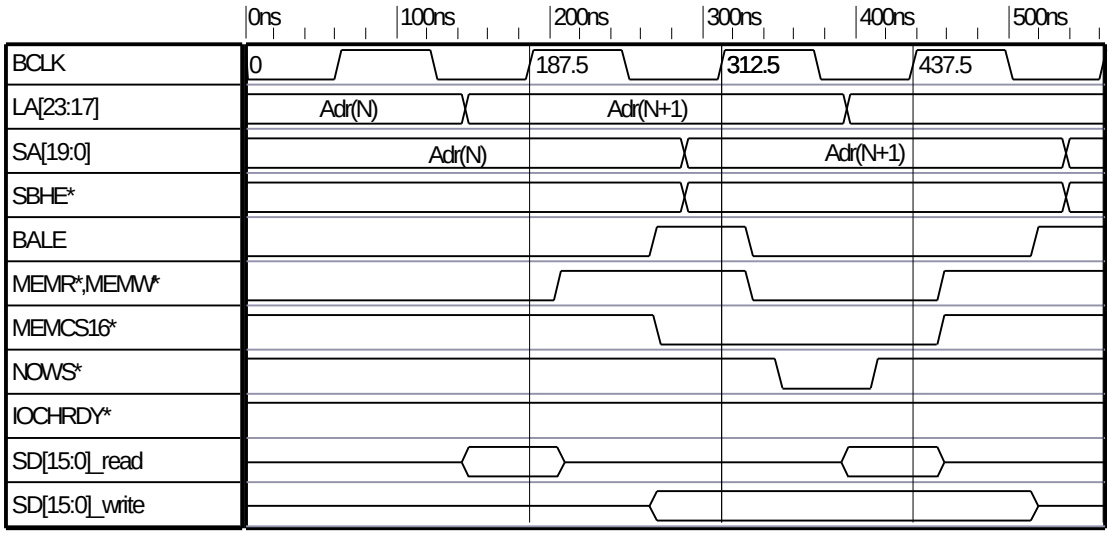


Figure 71 - ISA 16 bit no wait states memory cycle

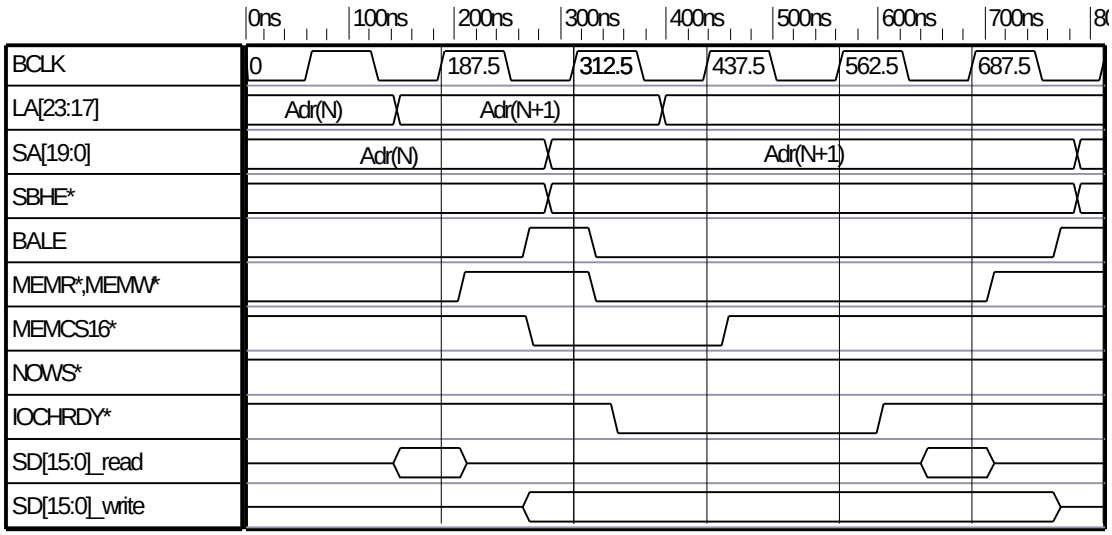


Figure 72 - ISA 16 bit memory "ready cycle"

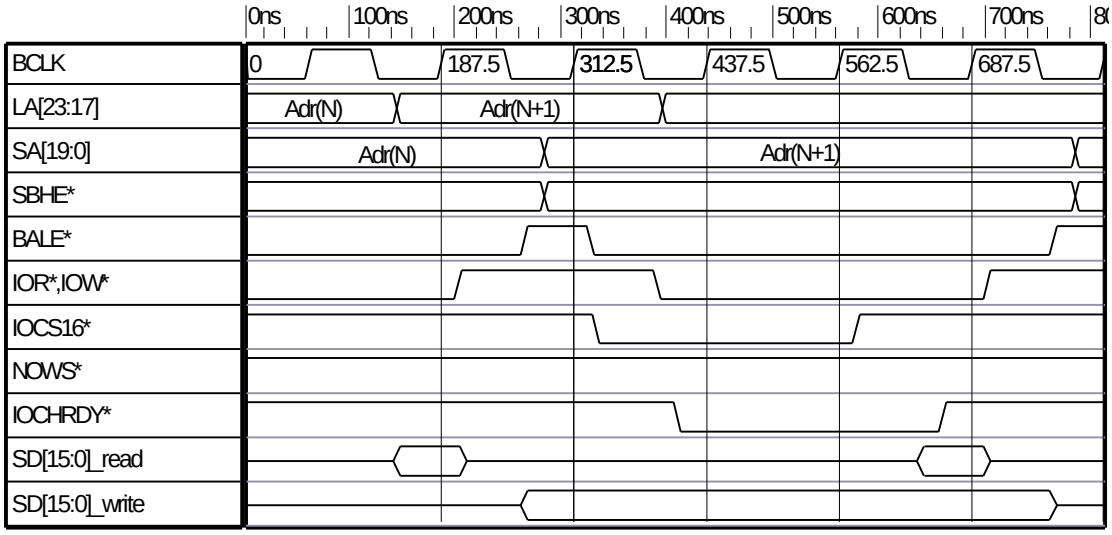


Figure 73 - ISA 16 bit I/O "ready cycle"

DMA cycles

The PC has two internal DMA controllers, each supporting 4 channels. One DMA controller supports 8 bit memory cycles, and the other supports 16 bit memory cycles. Since one channel is used to cascade both DMA controllers, we are left with 4 channels supporting 8 bit transfers, and 3 channels supporting 16 bit transfers.

The original DMA controller used in a PC was the Intel 8237 which was originally a 8080/8085 peripheral chip. This CPU had a 16 bit address bus and an 8 bit data bus. The 8237 was designed for a maximum DMA block length of 64K words. When a single 8237 was used in the PC design, an external 4 bit page register was added for every DMA channel to set its upper address bits. This feature has allowed the DMA channels to access the full 1MB address map, but still at a maximum block length of 64K. Since the page register bits are not incremented during DMA, a DMA transfer cannot cross a 64K boundary. When the AT was designed, 4 more bits were added to the page registers, and an extra 8237 chip was added. This chip is configured for handling 16 bit words, by shifting its address bus by one bit. As a result, its DMA block size can be twice as big, a whole 128KB, but a DMA transfer still cannot cross a 128KB boundary.

Interrupts

The PC has two internal Interrupt controllers, each supporting 8 channels. Since one channel is used to cascade both Interrupt controllers, we are left with 15 interrupt lines.

The original DMA controller used in a PC was the Intel 8251 which was originally a 8080/8085 peripheral chip. The task of this chip is to accept 8 discrete interrupt inputs, and generate an interrupt bus sequence which includes CPU handshaking and driving an 8 bit interrupt vector on the CPU data bus. The 8251 can prioritize the interrupts, mask them, set the interrupt vector base address, and select level or edge triggering.

The original PC peripherals were designed to use edge triggered interrupts. The main problem with this arrangement is that it is close to impossible to share an interrupt line by more than one device on the ISA bus. The reason for that is that an edge triggered interrupt that is received while a higher priority interrupt is being served will be lost. A level triggered interrupt could keep the interrupt line active until the interrupt was handled.

Bus Master cycles

The DMA controllers used the PC are limited in several ways:

- They cannot transfer a block longer than 64K.
- They are limited to consecutive blocks, and do not support advanced scatter/gather DMA operations.
- They are relatively slow, due to the requirement of a **DRQ*** / **DACK*** handshake before every word.

Bus master cards solve all these problems. The main difference between a bus master card and a card using DMA, is that when using DMA, the control signals and the bus address are generated by the internal DMA controller. The card is only driving (or reading) the data bus. A bus master however, has complete control over the bus, once it is granted access.

Bus master cards still requires one of the 16 bit DMA channel. The software must program this DMA channel as slave DMA. This tells the DMA controller to grant the bus to when this DMA channel is active. The system DMA controller assumes that a slave DMA controller will drive the bus. In fact, the bus is driven by the bus master card.

Appendix B - NuBUS Protocol summary

The NuBUS is a synchronous bus with burst capabilities, which makes it very similar to the PCI bus. We will show how NuBUS works, which allows us to compare it to the PCI bus.

Signal Summary

CLK* This is the NuBUS 10MHz clock signal. All signals are synchronized to this clock. The clock is asymmetrical, and is high for 75ns, and low for 25ns. Signals are updated on the rising edge of the clock, and sampled on the falling edge of the clock. The asymmetrical waveform allows 75ns for driver enable, setup time, and decoding, and 25ns for hold time and clock skew.

RESET* An active low reset signal.

PFW* Power Fail Warning

ID0*-ID3* Card Slot ID. These 4 bits form a unique ID for each NuBUS slot.

AD0*-AD31* These are 32 multiplexed address/data signals.

TM0*-TM1* These are the Transfer Mode bits. When the NuBUS cycle begins, these 2 bits, together with AD0* and AD1* specify one of 16 possible memory read/write commands. When the cycle ends, these signals carry the cycle status code.

SP* Parity.

SPV* Parity Valid.

START* This signal will be active during the first clock of a new transfer cycle.

ACK* This is the Transfer Acknowledge signal. It is driven by the target, and will be active when the cycle ends.

REQ* Bus Request

ARB0*-ARB3* Arbitration Level

NuBUS data types

NuBUS, like PCI, has a 32 bit data bus. NuBUS data transfers, like PCI, are *unaligned*. This means that a byte is transferred on the same signal lines on which it appears in a 32 bit word. NuBUS allows transmission of word, halfwords, and bytes according to the following figure:

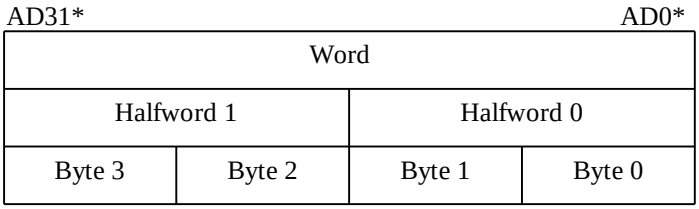


Figure 74 - NuBUS data types

Simple Memory read/write access cycles

The NuBUS read cycle begins when the master drives the **START*** signal low, and drives the address on the **AD0*-AD31*** lines and an operation code on the **TM0*-TM1*** lines. The transfer type is derived from the **TM0*-TM1*** lines and the **AD0*-AD1*** lines according to Table 27. As we can see, **TM1*** effectively determines whether the cycle is a read cycle or a write cycle. The rest of the lines determine which part of the word is being transferred (as we can see in Figure 74). The only special data type is the Block, which we will discuss later.

Read/Write	Data Type	TM1*	TM0*	AD1*	AD0*
Write	Byte 3	L	L	L	L
Write	Byte 2	L	L	L	H
Write	Byte 1	L	L	H	L
Write	Byte 0	L	L	H	H
Write	Halfword 1	L	H	L	L
Write	Block	L	H	L	H
Write	Halfword 0	L	H	H	L
Write	Word	L	H	H	H
Read	Byte 3	H	L	L	L
Read	Byte 2	H	L	L	H
Read	Byte 1	H	L	H	L
Read	Byte 0	H	L	H	H
Read	Halfword 1	H	H	L	L
Read	Block	H	H	L	H
Read	Halfword 0	H	H	H	L
Read	Word	H	H	H	H

Table 27 - NuBUS transfer type encoding

In the case of a read cycle, the master stops driving the **AD*** and **TM*** lines. The NuBUS cycle ends one or more cycle later, when the target responds with the data on the **AD*** lines, a status code on the **TM*** lines, and an active low signal on the **ACK*** line.

In the case of a write cycle, the master drives the data on the **AD*** lines during the next cycle, and stops driving the **TM*** lines. When the target is ready to accept the data, one or more cycle later, it drives the status code on the **TM*** lines, and an active low signal on the **ACK*** line.

The status code is interpreted according to the following table:

Response Status	TM1*	TM0*
Transfer Complete (OK)	L	L
Error	L	H
Bus Timeout Error	H	L
Try Again Later	H	H

Table 28 - NuBUS status codes

The *Transfer complete* code is returned when the transfer was finished successfully. The *Error* status is returned when the data returned may not be valid (for example, parity error was detected). *Bus Timeout Error* is generated when the target did not answer within 255 clock cycles, and the central logic had to terminate the cycle. The *Try Again Later* status is returned when no data is available now, but may be available later (for example, accessing a dual port memory which is currently busy).

The following figures show all the basic NuBUS read and write cycles.

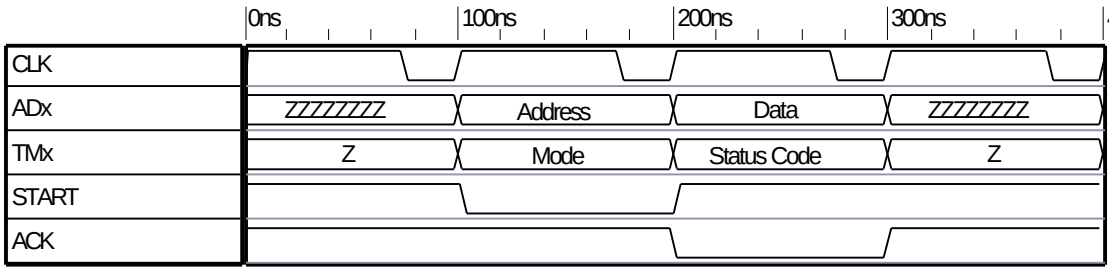


Figure 75 - NuBUS zero wait state read and write cycles

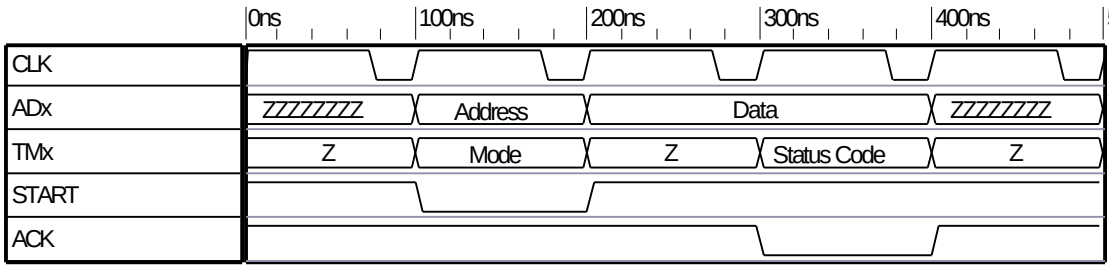


Figure 76 - NuBUS one wait state write cycle

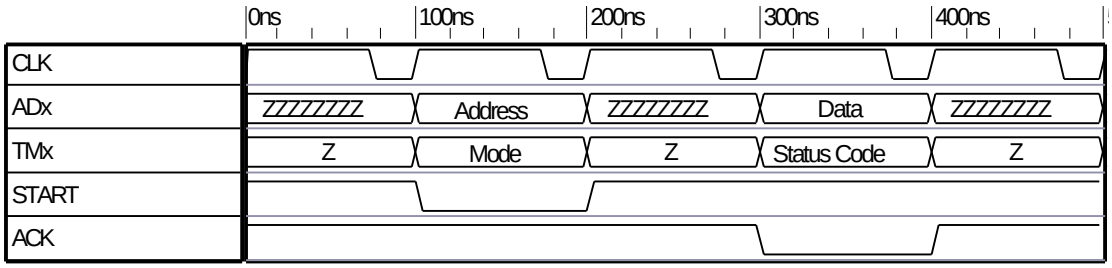


Figure 77 - NuBUS one wait state read cycle

Burst Memory read/write access cycles

The NuBUS burst transfers block length can be either 2, 4, 8, or 16 words. The block address must be naturally aligned, i.e. a 2^N word block must begin on a 2^N word boundary, with **AD[(N-1)..0]** = 0.

Block Size (Words)	Block Starting Address	AD5*	AD4*	AD3*	AD2*
2	(A31..A3)000	AD5*	AD4*	AD3*	H
4	(A31..A4)0000	AD5*	AD4*	H	L
8	(A31..A5)00000	AD5*	H	L	L
16	(A31..A6)000000	H	L	L	L

Table 29 - NuBUS block transfer

NuBUS burst transfers begin like single word transfers. The master selects the "Burst read" or "Burst Write" transfer type, drives the address and block size on the **AD*** bus, and asserts **START***.

For write bursts, the master then drives each word on the bus, and the target acknowledges them by asserting **TM0***. The last word is acknowledged using **ACK***, with a status code on **TM***.

For read bursts, the master then tri-states the **AD*** and the **TM*** bus. Each word transferred by the target on **AD*** is qualified by having **TM0*** asserted. The last word is acknowledged using **ACK***, with a status code on **TM***.

The following figures illustrate the burst mode:

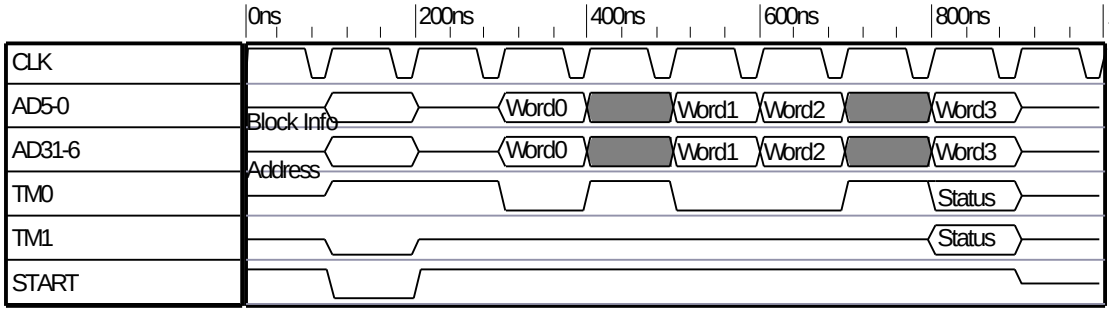


Figure 78 - NuBUS read burst cycle

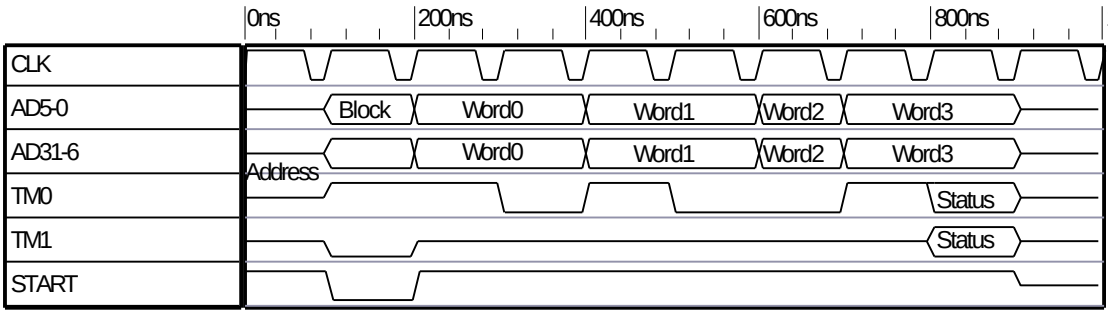


Figure 79 - NuBUS write burst cycle

NuBUS Interrupts

Unlike other busses, NuBUS does not support true interrupt signals. Instead, NuBUS systems use *Virtual Interrupts*. Virtual Interrupts are special memory locations that are watched by the system central logic. When any device writes a value to these addresses, the central logic generates an interrupt for the system CPU. Using this method interrupts can still be generated, but no special signals have to be dedicated for interrupts. Also, the number of unique interrupts is only limited by the system implementation, and not by the bus architecture.

Multiple Bus Masters

NuBUS supports multiple bus masters, allowing any expansion card to request the bus, perform one or more transactions, and release it. The NuBUS arbitration mechanism is distributed, very similar to MicroChannel (unlike the central arbitration logic used by PCI). Each card has four lines, **ID0*-ID3***, that are driven by the motherboard with a unique slot number. The slot number also represents the priority of this card relative to other cards in the system. An ID value of 0000 has the highest priority, and a value of 1111 has the lowest priority. A card may request the bus by driving **REQ*** (an open collector output) low, with the slot ID number on the **ARB0*-ARB3*** lines. The **ARB0*-ARB3*** buffers must be open collector, because multiple cards may request the bus at the same clock cycle. As a result, **ARB0*-ARB3*** will contain a value which is a bitwise AND of all the ID codes driven by all cards requesting the bus. Any card whose ID is higher than the **ARB0*-ARB3*** as resolved on the bus, is required to release its **ARB0*-ARB3*** lines by driving them with 1111. Since the arbitration logic is purely combinatorial, any device removing its ID code from the bus immediately affects other cards. This causes **ARB0*-ARB3*** to eventually settle down at the only stable state, which is the lowest ID code. NuBUS has allocated two bus clock cycles for this process.

Fairness is achieved by a requiring each card, releasing its **REQ*** line, not to assert it again as long as **REQ*** is still low (driven by another bus master). As a result, even when multiple cards request the bus simultaneously, all cards will eventually gain access to the bus because once a card has won the arbitration and performed its data transfer, it cannot request the bus again until all other cards has released their **REQ*** line. This is the same fairness algorithm as used by MicroChannel.

A card may stop driving **REQ*** as soon as it starts a cycle, and a new arbitration cycle may begin while the current transfer is still taking place. This way no cycles are wasted on arbitration, as it is pipelined with the data transfer. If a card requires multiple transactions, it has to drive **REQ*** low on all transactions, until the beginning of the last transaction. (This is very much like PCI).

A sample NuBUS arbitration logic may look like this:

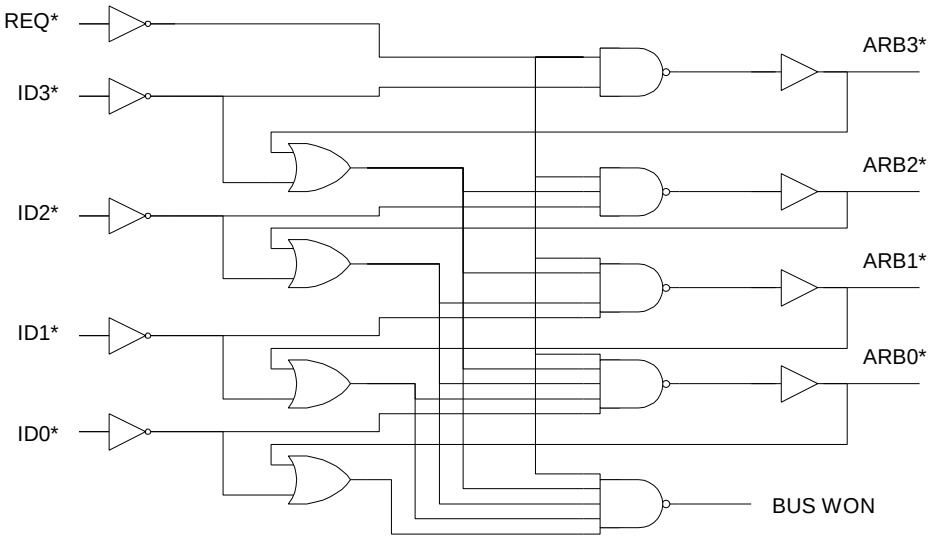


Figure 80 - NuBUS arbitration logic

Bus Locking

NuBUS supports two types of locks: bus locking and resource locking.

Bus locking is used by a master to ensure that two or more bus transactions are atomic. Bus locking is done by keeping **REQ*** active after the master has won the arbitration competition. It is not recommended to perform bus locking for long periods of time. Bus locking should be done only for the minimal amount of time required.

Resource locking is used to inform a slave card to lock out all local access routes on the card to a resource being addressed by NuBus. For example, a co-processor card might communicate with other processors over NuBus using a dual ported RAM addressed as a NuBus slave. A NuBus master might need to perform an atomic operation on this dual ported RAM, while making sure the local co-processor cannot access this RAM through the other RAM port on the card. If the NuBus master does a resource lock on the dual ported RAM, the co-processor is prevented from accessing this RAM while the lock is in place.

Resource locking is done by issuing an Attention-Bus-Lock cycle. A master can issue this cycle by asserting both **START*** and **ACK*** at the beginning of a bus transaction, while driving the Attention-Resource-Lock on the **TM*** lines (see Table 30). To end the locked transaction, the master issues an Attention-Null cycle (**START*** and **ACK*** asserted together with the appropriate code on the **TM*** lines). During this period, any card accessed by the master will lock its local resources.

Response Status	TM1*	TM0*
Attention-Null	L	L
Reserved	L	H
Attention-Resource-Lock	H	L
Reserved	H	H

Table 30 - NuBUS attention cycle codes

Appendix C - PCI Class Codes

Base Class	Sub Class	Interface	Meaning
00h	00h	00h	All old devices implemented before base Classes were defined, except for VGA compatible devices.
	01h	00h	All VGA compatible devices implemented before base Classes were defined.
01h	00h	00h	SCSI Bus controller
	01h		IDE controller
	02h	00h	Floppy Disk controller
	03h	00h	IPI Bus controller
	03h	00h	RAID Bus controller
	03h	00h	Other mass storage controller
02h	00h	00h	Ethernet controller
	01h	00h	Token Ring controller
	02h	00h	FDDI controller
	03h	00h	ATM controller
	80h	00h	Other network controller
03h	00h	00h	VGA compatible controller
	00h	01h	8514 compatible controller
	01h	00h	XGA compatible controller
	80h	00h	Other display controller
04h	00h	00h	Video device
	01h	00h	Audio device
	80h	00h	Other multimedia device
05h	00h	00h	RAM
	01h	00h	FLASH
	80h	00h	Other memory controller
06h	00h	00h	Host bridge
	01h	00h	ISA bridge
	02h	00h	EISA bridge
	03h	00h	MCA bridge
	04h	00h	PCI-to-PCI bridge
	05h	00h	PCMCIA bridge
	06h	00h	NuBUS bridge
	07h	00h	CardBus bridge
	80h	00h	Other bridge device

Base Class	Sub Class	Interface	Meaning
07h	00h	00h	Generic XT-compatible serial controller
		01h	16450-compatible serial controller
		02h	16550-compatible serial controller
	01h	00h	Parallel port
		01h	Bidirectional parallel port
		02h	ECP 1.X compliant parallel port
	80h	00h	Other communication device
08h	00h	00h	Generic 8259 PIC
		01h	ISA PIC
		02h	EISA PIC
	01h	00h	Generic 8237 DMA controller
		01h	ISA DMA controller
		02h	EISA DMA controller
	02h	00h	Generic 8254 system timer
		01h	ISA system timer
		02h	EISA system timers (two timers)
	03h	00h	Generic RTC controller
		01h	ISA RTC controller
	80h	00h	Other system peripheral
09h	00h	00h	Keyboard controller
	01h	00h	Digitizer (pen)
	02h	00h	Mouse controller
	80h	00h	Other input controller
0Ah	00h	00h	Generic docking station
	80h	00h	Other type of docking station
0Bh	00h	00h	386
	01h	00h	486
	02h	00h	Pentium
	10h	00h	Alpha
	20h	00h	PowerPC
	40h	00h	Co-processor
0Ch	00h	00h	FireWire (IEEE 1394)
	01h	00h	ACCESS.bus
	02h	00h	SSA
	03h	00h	Universal Serial Bus (USB)
	04h	00h	Fibre Channel

Appendix D - Glossary

ASIC	Application Specific IC. An IC manufactured for a company based on a unique design given by the company, but done in a standard chip process.
AGP	Accelerated Graphics Port. Standard that specifies a high bandwidth connection between the graphics controller and the main memory.
CardBus	Laptop version of the PCI bus.
CompactPCI	Passive backplane bus that attempts to replace VME. Cards have the same size as VME cards.
CMOS	Complementary MOS. A chip manufacturing technology widely used by almost all chip manufacturers today.
CPLD	Complex PLDs. A programmable logic chip based on linking multiple PLD blocks on the same chip.
DMA	Direct Memory Access. A method for transferring data from one bus agent to another without CPU intervention.
EISA	Enhanced ISA. An industry standard invented by Compaq to replace the aging ISA standard.
FPGA	Field Programmable gate Arrays. A programmable logic chip based on a matrix of basic logic cells.
GPIB	A standard interface for test and measurement equipment. Intended by HP, and adopted by IEEE as IEEE 488.
HDL	Hardware Description Language. A generic name for a computer language used to describe digital circuits for CPLD, FPGA and ASIC design.
HiRelPCI	Standard currently developed by IEEE. It supports SCI.
IDE	Integrated Drive Electronics. A mass storage interface standard, roughly based on integrating the original IBM PC hard disk controller into the drive itself.
ISA	Industry Standard Architecture. The original bus designed for the IBM PC and IBM AT.
JTAG	Joint Test Action Group. A group dedicated to setting industry standards related to electronic testing. Also a name of a computer interface for in-system testing of chips.
MESI	Modified/Exclusive/Shared/invalid. A bus coherency protocol used to guarantee cache coherency in multiprocessing systems.
MOS	Metal Oxide Semiconductor. A type of transistor used on almost all the chips manufactured today.
NuBUS	A computer bus widely in use by the Macintosh II series of computers.
PAL	Programmable Array Logic. A synonym for PLD.
PC/104	The embedded system version of the ISA bus. Also the name of the consortium that serves as a custodian of the PC/104 standard.
PC/104-Plus	Standard that specifies PC/104 size cards with both ISA and PCI bus.
PC Card	New name for the 1994 release of the PCMCIA standard.
PCI	Peripheral Component Interconnect. A local bus standard.
PCI-SIG	PCI Special Interest Group. An association of members of the microcomputer industry established to monitor and enhance the development of the PCI bus.

PCI-ISA	A passive backplane standard based on a full length CPU card containing both an ISA and a PCI connector.
PCMCIA	Personal Computer Memory Card International Association. Organization that developed the PCMCIA standard, a laptop oriented expansion bus.
PICMG	PCI Industrial Computer Manufacturers Group. A consortium of computer product vendors established to extend the PCI specification for use in industrial computing applications. Developed the PCI-ISA Passive Backplane and CompactPCI standards.
PISA	A passive backplane standard based on a half length CPU card containing an EISA like connector with ISA signals on the top row and PCI signals on the bottom row.
PLD	Programmable Logic Device. A logic chip based on a programmable AND-OR array linking a number of input and output pins.
PLI	Procedural Language Interface. A standard API for linking C modules with Verilog programs.
PMC	PCI Mezzanine Card. Defined as IEEE standard P1386.1, PMC cards use PCI chips and may be mounted on VME cards.
PXI	PCI eXtentions for Instrumentation. A VXI like bus based on CompactPCI.
RTL	Register Transfer Level. A logical abstraction level of a digital integrated circuit description, which can be easily translated to a real circuit.
SCI	Scalable Coherent Interface. IEEE standard 1596-1992. Specifies a method of interconnecting multiple processing nodes.
SCSI	Small Computer System Interface. A popular standard for linking several mass storage devices to a computer.
SmallPCI	Expansion card with form factor identical to PC Card and CardBus. Primarily intended for OEM products.
VESA	Video Electronics Standards Association. A technical forum setting PC computer graphics related standards.
VHDL	VHSIC Hardware Description Language. A popular HDL inspired by Ada, and designed to be a Military standard. It is now IEEE-1076
Verilog	A popular HDL inspired by C, and designed by Gateway corporation (now owned by Cadence). It is now IEEE-1364.
VGA	Video Gate Array. A graphics card designed by IBM, later adopted as a baseline standard. All modern graphic cards have a basic VGA compatible mode.
VITA	VME International Trade Association. The organization of VME manufacturers.
VME	Versa Module Eurocard. Passive backplane bus.
VXI	VME eXtension for Instrumentation. A test and measurement bus based on 9U VME cards.
X86	A generic name for the Intel 16/32 bit architecture implemented by the 8088 through Pentium II series of micro processors. X86 compatible microprocessors are also implemented by other companies such as IDT, Cyrix, IBM, AMD and SGS-Thomson.