8 Input–Output Design and Organization

Having considered the fundamental concepts related to instruction set design, assembly language programming, processor design, and memory design, we now turn our attention to the issues related to input output (I/O) design and organization. It should be emphasized at the outset that I/O plays a crucial role in any modern computer system. Therefore, a clear understanding and appreciation of the fundamentals of I/O operations, devices, and interfaces are of great importance.

Input output (I/O) devices vary substantially in their characteristics. One distinguishing factor among input devices (and also among output devices) is their data processing rate, defined as the average number of characters that can be processed by a device per second. For example, while the data processing rate of an input device such as the keyboard is about 10 characters (bytes)/second, a scanner can send data at a rate of about 200,000 characters/second. Similarly, while a laser printer can output data at a rate of about 100,000 characters/second, a graphic display can output data at a rate of about 30,000,000 characters/second.

Striking a character on the keyboard of a computer will cause a character (in the form of an ASCII code) to be sent to the computer. The amount of time passed before the next character is sent to the computer will depend on the skill of the user and even sometimes on his/her speed of thinking. It is often the case that the user knows what he/she wants to input, but sometimes they need to think before touching the next button on the keyboard. Therefore, input from a keyboard is slow and burst in nature and it will be a waste of time for the computer to spend its valuable time waiting for input from slow input devices. A mechanism is therefore needed whereby a device will have to interrupt the processor asking for attention whenever it is ready. This is called *interrupt-driven* communication between the computer and I/O devices (see Section 8.3).

Consider the case of a disk. A typical disk should be capable of transferring data at rates exceeding several million bytes/second. It would be a waste of time to transfer data byte by byte or even word by word. Therefore, it is always the case that data is transferred in the form of blocks, that is, entire programs. It is also necessary to provide a mechanism that allows a disk to transfer this huge volume of data without the intervention of the CPU. This will allow the CPU to perform other useful operation(s) while a huge amount of data is being transferred between the disk and the memory. This is the essence of the *direct memory access* (DMA) mechanism discussed in Section 8.4.

We begin our discussion by offering some basic concepts in Section 8.1.

8.1. BASIC CONCEPTS

Figure 8.1 shows a simple arrangement for connecting the processor and the memory in a given computer system to an input device, for example, a keyboard and an output device such as a graphic display. A single bus consisting of the required address, data, and control lines is used to connect the system's components in Figure 8.1.

The way in which the processor and the memory exchange data has been explained in Chapters 6 and 7. We are here concerned with the way the processor and the I/O devices exchange data. It has been indicated in the introduction part that there exists a big difference in the rate at which a processor can process information and those of input and output devices. One simple way to accommodate this speed difference is to have the input device, for example, a keyboard, deposit the character struck by the user in a register (*input register*), which indicates the availability of that character to the processor. When the input character has been taken by the processor, this will be indicated to the input device in order to proceed and input the next character, and so on. Similarly, when the processor has a character to output (display), it deposits it in a specific register dedicated for communication with the graphic display (*output register*). When the character has been taken by the graphic display, this will be indicated to the processor such that it can proceed and output the next character, and so on. This simple way of communication between the processor and I/O devices, called *I/O protocol*, requires the availability of the input and output registers. In a typical computer system, there is a number of input registers, each belonging to a specific input device. There is also a number of output registers,

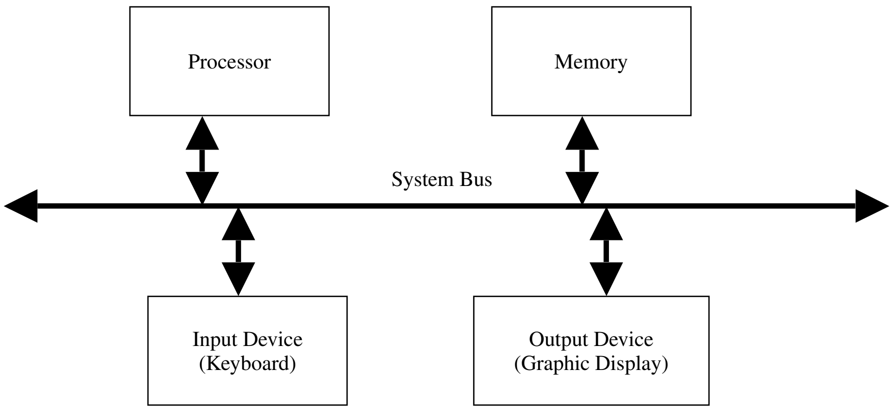
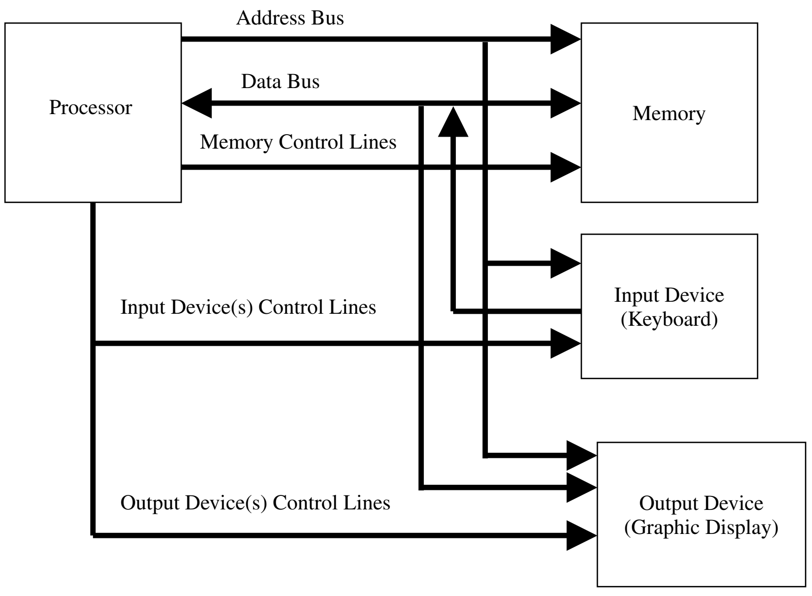


Figure 8.1 A single bus system

each belonging to a specific output device. In addition, a mechanism according to which the processor can address those input and output registers must be adopted. More than one arrangement exists to satisfy the abovementioned requirements. Among these, two particular methods are explained below.

In the first arrangement, I/O devices are assigned particular addresses, isolated from the address space assigned to the memory. The execution of an *input instruction* at an input device address will cause the character stored in the *input register* of that device to be transferred to a specific register in the CPU. Similarly, the execution of an *output instruction* at an output device address will cause the character stored in a specific register in the CPU to be transferred to the *output register* of that output device. This arrangement, called *shared I/O*, is shown schematically in Figure 8.2. In this case, the address and data lines from the CPU can be shared between the memory and the I/O devices. A separate control line will have to be used. This is because of the need for executing input and output instructions. In a typical computer system, there exists more than one input and more than one output device. Therefore, there is a need to have *address decoder circuitry* for device identification. There is also a need for *status registers* for each input and output device. The status of an input device, whether it is ready to send data to the processor, should be stored in the status register of that device. Similarly, the status of an output device, whether it is ready to receive data from the processor, should be stored in the status register of that device. Input (output) registers, status registers, and address decoder circuitry represent the main components of an I/O interface (module).



Address Bus
Data Bus
Memory Control Lines
Processor
Memory
Input Device(s) Control Lines
Input Device (Keyboard)
Output Device(s) Control Lines
Output Device (Graphic Display)

Figure 8.2 Shared I/O arrangement

The main advantage of the shared I/O arrangement is the separation between the memory address space and that of the I/O devices. Its main disadvantage is the need to have special *input* and *output instructions* in the processor instruction set. The shared I/O arrangement is mostly adopted by Intel.

The second possible I/O arrangement is to deal with *input* and *output registers* as if they are regular memory locations. In this case, a *read* operation from the address corresponding to the *input register* of an input device, for example, *Read Device 6*, is equivalent to performing an *input operation* from the input register in Device #6. Similarly, a *write* operation to the address corresponding to the *output register* of an output device, for example, *Write Device 9*, is equivalent to performing an *output operation* into the output register in Device #9. This arrangement is called *memory-mapped I/O*. It is shown in Figure 8.3.

The main advantage of the memory-mapped I/O is the use of the read and write instructions of the processor to perform the input and output operations, respectively. It eliminates the need for introducing special I/O instructions. The main disadvantage of the memory-mapped I/O is the need to reserve a certain part of the memory address space for addressing I/O devices, that is, a reduction in the available memory address space. The memory-mapped I/O has been mostly adopted by Motorola.

8.2. PROGRAMMED I/O

In this section, we present the main hardware components required for communications between the processor and I/O devices. The way according to which such

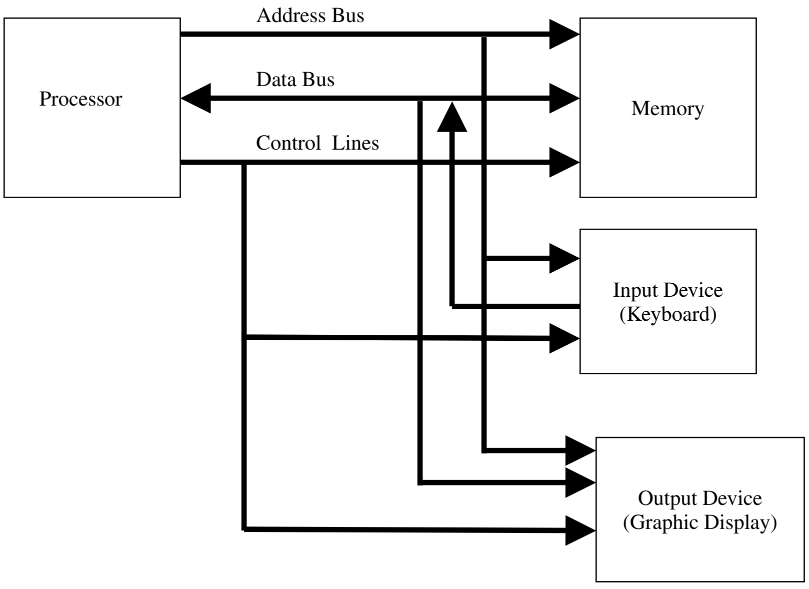


Figure 8.3 Memory mapped I/O arrangement

communications take place (protocol) is also indicated. This protocol has to be programmed in the form of routines that run under the control of the CPU. Consider, for example, an input operation from *Device 6* (could be the keyboard) in the case of *shared* I/O arrangement. Let us also assume that there are eight different I/O devices connected to the processor in this case (see Fig. 8.4).

The following protocol steps (program) have to be followed:

1. The processor executes an input instruction from device 6, for example, *INPUT 6*. The effect of executing this instruction is to send the device number to the address decoder circuitry in each input device in order to identify the specific input device to be involved. In this case, the output of the decoder in Device #6 will be enabled, while the outputs of all other decoders will be disabled.
2. The buffers (in the figure we assumed that there are eight such buffers) holding the data in the specified input device (Device #6) will be enabled by the output of the address decoder circuitry.
3. The data output of the enabled buffers will be available on the data bus.

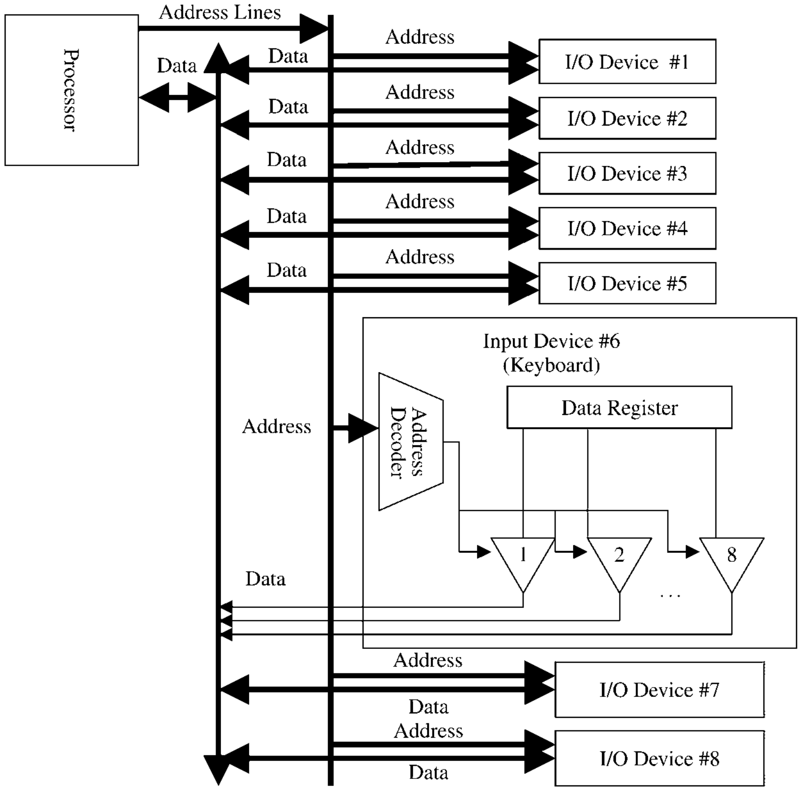


Figure 8.4 Example eight I/O device connection to a processor

4. The instruction decoding will gate the data available on the data bus into the input of a particular register in the CPU, normally the *accumulator*.

Output operations can be performed in a way similar to the input operation explained above. The only difference will be the direction of data transfer, which will be from a specific CPU register to the output register in the specified output device. I/O operations performed in this manner are called *programmed I/O*. They are performed under the CPU control. A complete instruction fetch, decode, and execute cycle will have to be executed for every input and every output operation. Programmed I/O is useful in cases whereby one character at a time is to be transferred, for example, keyboard and character mode printers. Although simple, programmed I/O is slow.

One point that was overlooked in the above description of the programmed I/O is how to handle the substantial speed difference between I/O devices and the processor. A mechanism should be adopted in order to ensure that a character sent to the output register of an output device, such as a screen, is not overwritten by the processor (due to the processor's high speed) before it is displayed and that a character available in the input register of a keyboard is read only once by the processor. This brings up the issue of the status of the input and output devices. A mechanism that can be implemented requires the availability of a *Status Bit* (B_{in}) in the interface of each input device and *Status Bit* (B_{in}) in the interface of each output device. Whenever an input device such as a keyboard has a character available in its input register, it indicates that by setting B_{in} 1. A program in the processor can be used to continuously monitor B_{in}. When the program sees that B_{in} 1, it will interpret that to mean a character is available in the input register of that device. Reading such character will require executing the protocol explained above. Whenever the character is read, then the program can reset B_{in} 0, thus avoiding multiple read of the same character. In a similar manner, the processor can deposit a character in the output register of an output device such as a screen only when B_{out} 0. It is only after the screen has displayed the character that it sets B_{out} 1, indicating to the program that monitors B_{out} that the screen is ready to receive the next character. The process of checking the status of I/O devices in order to determine their readiness for receiving and/or sending characters, is called *software I/O polling*. A *hardware I/O polling* scheme is shown in Figure 8.5.

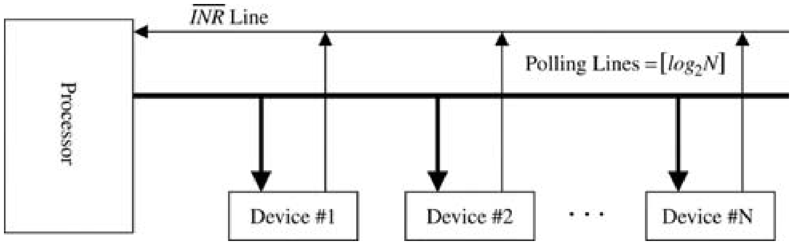


Figure 8.5 Hardware polling scheme

In the figure, each of the N I/O devices has access to the interrupt line $\overline{INR}$. Upon recognizing the arrival of a request (called Interrupt Request) on $\overline{INR}$, the processor polls the devices to determine the requesting device. This is done through the $\lceil \text{Log}_2 N \rceil$ polling lines. The priority of the requesting device will determine the order in which addresses are put on the polling lines. The address of the highest priority device is put first, followed by the next priority, and so on until the least priority device.

In addition to the I/O polling, two other mechanisms can be used to carry out I/O operations. These are *interrupt-driven I/O* and *direct memory access* (DMA). These are discussed in the next two sections.

8.3. INTERRUPT-DRIVEN I/O

It is often necessary to have the normal flow of a program interrupted, for example, to react to abnormal events, such as power failure. An interrupt can also be used to acknowledge the completion of a particular course of action, such as a printer indicating to the computer that it has completed printing the character(s) in its input register and that it is ready to receive other character(s). An interrupt can also be used in time-sharing systems to allocate CPU time among different programs. The instruction sets of modern CPUs often include instruction(s) that mimic the actions of the hardware interrupts.

When the CPU is interrupted, it is required to discontinue its current activity, attend to the interrupting condition (serve the interrupt), and then resume its activity from wherever it stopped. Discontinuity of the processor's current activity requires finishing executing the current instruction, saving the processor status (mostly in the form of pushing register values onto a stack), and transferring control (jump) to what is called the *interrupt service routine* (ISR). The service offered to an interrupt will depend on the source of the interrupt. For example, if the interrupt is due to power failure, then the action taken will be to save the values of all processor registers and pointers such that resumption of correct operation can be guaranteed upon power return. In the case of an I/O interrupt, serving an interrupt means to perform the required data transfer. Upon finishing serving an interrupt, the processor should restore the original status by popping the relevant values from the stack. Once the processor returns to the normal state, it can enable sources of interrupt again.

One important point that was overlooked in the above scenario is the issue of serving *multiple interrupts*, for example, the occurrence of yet another interrupt while the processor is currently serving an interrupt. Response to the new interrupt will depend upon the priority of the newly arrived interrupt with respect to that of the interrupt being currently served. If the newly arrived interrupt has priority less than or equal to that of the currently served one, then it can wait until the processor finishes serving the current interrupt. If, on the other hand, the newly arrived interrupt has priority higher than that of the currently served interrupt, for example, power failure interrupt occurring while serving an I/O interrupt, then the processor will have to push its status onto the stack and serve the higher priority interrupt. Correct handling of multiple interrupts in terms of storing and restoring the correct processor status is guaranteed due to the way the push and pop operations are performed. For example, to serve the first interrupt, STATUS 1 will be pushed onto the stack. Upon receiving the second interrupt, STATUS 2 will be pushed onto the stack. Upon serving the second interrupt, STATUS 2 will be popped out of the stack and upon serving the first interrupt, STATUS 1 will be popped out of the stack.

It is possible to have the interrupting device identify itself to the processor by sending a code following the interrupt request. The code sent by a given I/O device can represent its I/O address or the memory address location of the start of the ISR for that device. This scheme is called *vectored interrupt.*

8.3.1. Interrupt Hardware

In the above discussion, we have assumed that the processor has recognized the occurrence of an interrupt before proceeding to serve it. Computers are provided with *interrupt hardware* capability in the form of specialized *interrupt lines* to the processor. These lines are used to send interrupt signals to the processor. In the case of I/O, there exists more than one I/O device. The processor should be provided with a mechanism that enables it to handle simultaneous interrupt requests and to recognize the interrupting device. Two basic schemes can be implemented to achieve this task. The first scheme is called *daisy chain bus arbitration* (DCBA) and the second is called *independent source bus arbitration* (ISBA).

According to the DCBA (see Fig. 8.6*a*), I/O devices present their interrupt requests to the interrupt request line $\overline{INR}$ (similar to the polling arrangement). Upon recognizing the arrival of an interrupt request, the processor, through a daisy chained *grant line* (GL), sends its grant to the requesting device to start communication with the processor. The GL goes through all devices starting from the first device nearer to the processor and going to the next device and so on until it reaches the last device (Device #N). If Device #1 has put a request, then it will hold the grant signal and start communication with the processor. If, on the other hand, Device #1 has no interrupt request, it will pass the grant signal to device #2, which will repeat the same procedure, and so on. In the case of multiple requests, the DCBA arrangement gives highest priority to the device physically nearer to the processor. The furthest device from the processor has the lowest priority.

According to the ISBA (see Fig. 8.6*b*), each I/O device has its own interrupt request line, through which it can send its interrupt request, independent of the other devices. Similarly, each I/O device has its own grant line, through which it receives the grant signal for its request such that it can start communicating with the processor. I/O device priority in the ISBA does not depend on the device location. A priority arbitration circuitry is needed in order to deal with simultaneous interrupt requests.

8.3.2. Interrupt in Operating Systems

When an interrupt occurs, the operating system gains control. The operating system saves the state of the interrupted process, analyzes the interrupt, and passes control to the appropriate routine to handle the interrupt. There are several

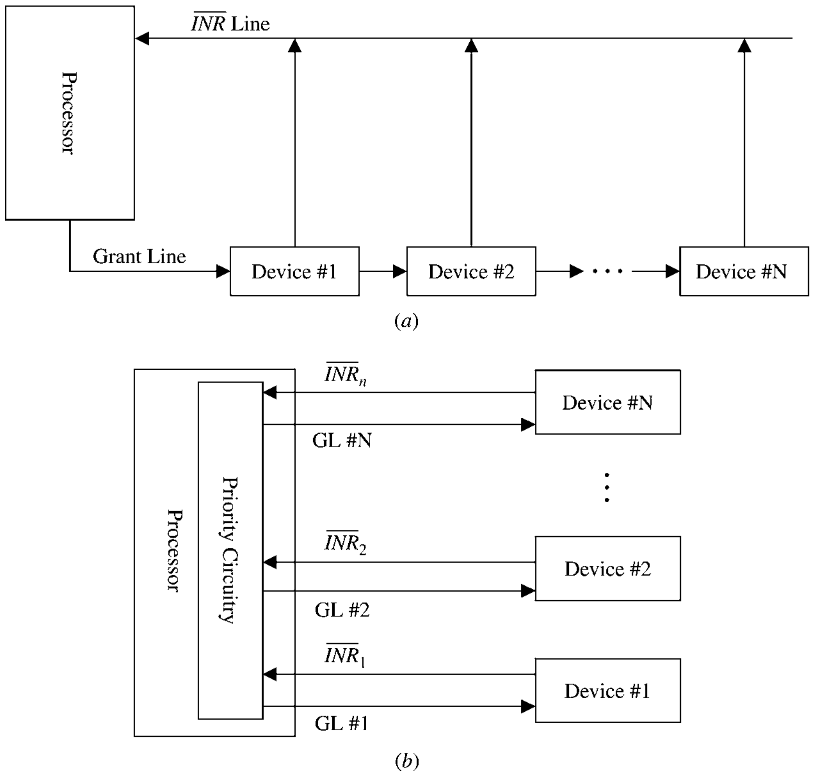


Figure 8.6 Interrupt hardware schemes. (*a*) Daisy chain interrupt arrangement (*b*) Independent interrupt arrangement

types of interrupts, including I/O interrupts. An I/O interrupt notifies the operating system that an I/O device has completed or suspended its operation and needs some service from the CPU. To process an interrupt, the context of the current process must be saved and the interrupt handling routine must be invoked. This process is called *context switching*. A process context has two parts: processor context and memory context. The processor context is the state of the CPU's registers including program counter (PC), program status words (PSWs), and other registers. The memory context is the state of the program's memory including the program and data. The interrupt handler is a routine that processes each different type of interrupt.

The operating system must provide programs with save area for their contexts. It also must provide an organized way for allocating and deallocating memory for the interrupted process. When the interrupt handling routine finishes processing the interrupt, the CPU is dispatched to either the interrupted process, or to the highest priority ready process. This will depend on whether the interrupted process is preemptive or nonpreemptive. If the process is nonpreemptive, it gets the CPU again. First the context must be restored, then control is returned to the interrupts process.

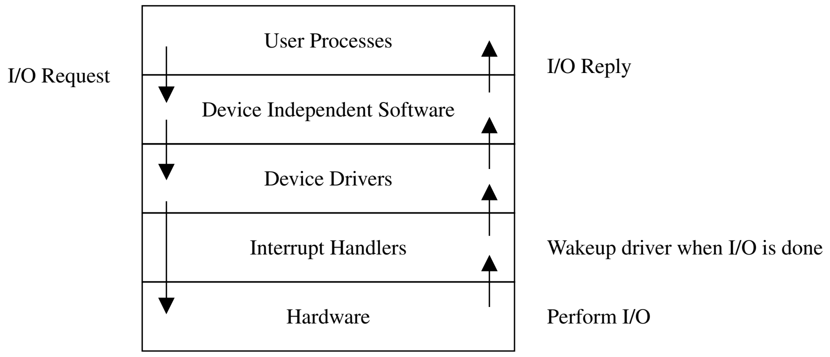


Figure 8.7 Layered I/O software

Figure 8.7 shows the layers of software involved in I/O operations. First, the program issues an I/O request via an I/O call. The request is passed through to the I/O device. When the device completes the I/O, an interrupt is sent and the interrupt handler is invoked. Eventually, control is relinquished back to the process that initiated the I/O.

Example 1: 80×86 Interrupt Architecture The 80×86 processors have just two hardware interrupt pins. These are labeled INTR and NMI. NMI is a nonmaskable interrupt, which means it cannot be blocked and the processor must respond to it. The NMI input is usually reserved for critical system functions. The INTR input is a maskable interrupt request line between the CPU and the programmable interrupt controller (8259A PIC). Interrupts on INTR can be enabled and disabled using the instructions STI (set interrupt flag) and CLI (clear interrupt flag), respectively.

Interrupt handlers are called interrupt service routines (ISR). The address of each interrupt service routine is stored in four consecutive memory locations in the interrupt vector table (IVT). The IVT stores pointers to ISR for each type of interrupt. When an interrupt occurs, an 8-bit type number is supplied to the processor, which identifies the appropriate entry in this table.

When an interrupt is generated by a device, it goes to the PIC. Multiple interrupts may be generated simultaneously. However, they are all buffered by the PIC. The PIC decides which one of these interrupts should be forwarded to the CPU. To inform the CPU that an outstanding interrupt is waiting to be processed, the PIC sends an interrupt request (INTR) to the CPU, which then, at the appropriate time, responds with an interrupt acknowledgment (INTA). At this time, PIC will put an 8-bit interrupt type number associated with the device on the bus so that the CPU can identify which interrupt handler to invoke. In the case when several interrupts are pending, PIC will send next interrupt request to the CPU only after it receives an end of interrupt command from the current ISR. Figure 8.8 shows the simple protocol that is used to determine which ISR is to be invoked.

In the computer designs that used a single PIC (PC and XT), eight different interrupt requests are allowed (IRQ0 IRQ7). Table 8.1 shows a list of standard interrupt type numbers for typical devices. When AT was designed, a second PIC was added,

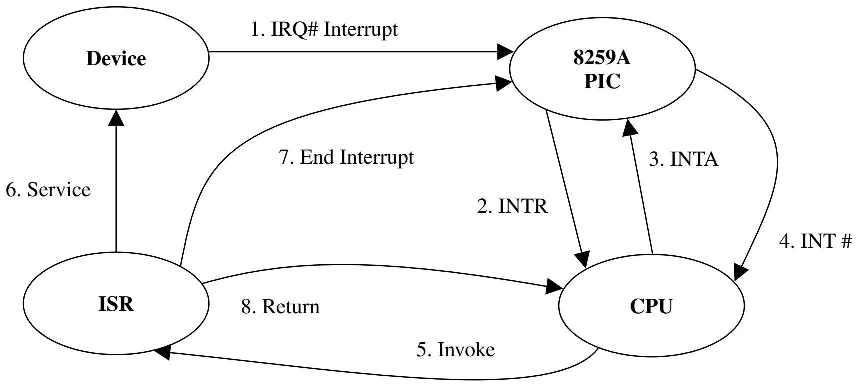


Figure 8.8 Interrupt handling in 80×86

increasing the number of interrupt inputs to 15. Figure 8.9 shows two PICS wired in cascade. One PIC is designated as master and the other becomes the slave. As shown in the figure, all slave interrupts are input via IRQ1 of the master. In general, eight different slaves can be accommodated by a single PIC.

Example 2: ARM Interrupt Architecture ARM stands for Advanced RISC Machines. ARM is a 16/32-bit architecture that is used for portable devices because of its low power consumption and reasonable performance. Interrupt requests to the ARM core are collected and controlled by the interrupt controller, which is called ATIC. The interrupt controller provides an interface to the core and can collect up to 64 interrupt requests.

The usual sequence of events for interrupts is as follows. Interrupts would be enabled at the source (such as a peripheral), then enabled in the interrupt controller, and finally, enabled to the core. When an interrupt occurs at the source, its signal is routed to the interrupt controller then to the ARM core. In the interrupt controller, the interrupt can be enabled or disabled to the core and can be assigned a priority

TABLE 8.1 Standard IBM-PC Interrupt Type Numbers for Typical Devices

Device	IRQ no.	Interrupt type number
Programmable interval timer	0	08H
Keyboard	1	09H
Cascading to the second PICs	2	Reserved
Serial communication port (COM2)	3	0BH
Serial communication port (COM1)	4	0CH
Fixed disk controller	5	0DH
Floppy disk controller	6	0EH
Parallel printer controller	7	0FH

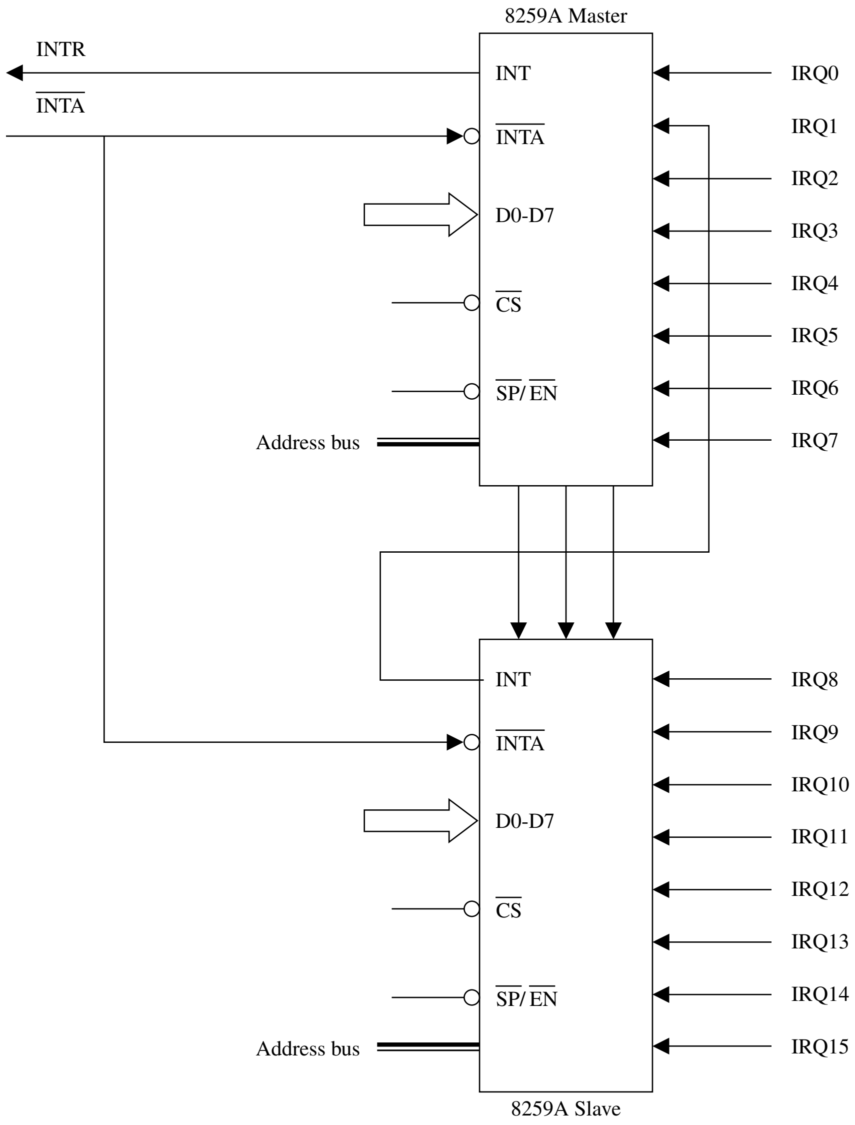


Figure 8.9 Fifteen different interrupts are supported by two PICs wired in cascade

level. Once the interrupt request reaches the core, it will halt the core from its normal processing routines to allow the interrupt request to be serviced.

Among the different interrupt requests that the ARM core can handle are IRQ and FIQ requests. The IRQ (normal interrupt request) is used for general-purpose interrupt handling. It has a lower priority than an FIQ (fast interrupt request) and is masked out when an FIQ sequence is entered. The FIQ is used to support high-speed data transfer or channel processes.

TABLE 8.2 Interrupt Vector Table

Exception type	Mode	Address
Reset	Supervisor	0 × 00000000
Undefined instructions	Undefined	0 × 00000004
Software interrupts (SWI)	Supervisor	0 × 00000008
Prefetch abort	Abort	0 × 0000000C
Data abort	Abort	0 × 00000010
IRQ (Normal interrupt)	IRQ	0 × 00000018
FIQ (Fast interrupt)	FIQ	0 × 0000001C

Similar to the 80 × 86, the addresses of the interrupt handlers are stored in a vector table, which is shown in Table 8.2. For example, when an IRQ is detected by the core, it accesses address 0 × 18 of the vector table and executes the instruction loaded in that address. Normally, the instruction found at 0 × 18 of the vector table is of the form: *LDR PC*, *IRQ Handler* (load the address of the IRQ interrupt handler in the PC). When an FIQ is detected by the core, it accesses address 0 × 1C of the vector table and executes the instruction loaded in that address. Normally, the instruction found at 0 × 1C of the vector table is of the form: *LDR PC*, *FIQ Handler*.

When an interrupt occurs, the following happens inside the core:

1. The CPSR (current program state register) is copied to the SPSR (saved program status register) of the mode being entered.
2. The CPSR bits are set as appropriate to the mode being entered, the core is set to ARM state, and the relevant interrupt disable flags are set.
3. The appropriate set of banked registers are banked in.
4. The return address is stored in the link register (of the relevant mode).
5. The PC is set to the relevant vector address.

For example, when an IRQ interrupt is detected, the ARM core enables SPSR irq as the CPSR, enters the IRQ mode by setting the mode bits in the CSPR to 10010, disables Normal interrupts by setting the I bit in the CPSR, saves the address of the next instruction R14 irq, and loads 0 × 18 into the PC. At address 0 × 18, an instruction will load the address of the interrupt handler into the PC. Similarly, when an FIQ interrupt is detected, the ARM core enables SPSR fiq as the CPSR, enters the FIQ mode by setting the mode bits in the CSPR to 10001, disables Normal and Fast interrupts by setting the F and I bits in the CPSR, saves the address of the next instruction R14 fiq, and loads 0 × 1C into the PC. At address 0 × 1C, an instruction will load the address of the interrupt handler into the PC.

8.4. DIRECT MEMORY ACCESS (DMA)

The main idea of direct memory access (DMA) is to enable peripheral devices to cut out the “middle man” role of the CPU in data transfer. It allows peripheral devices to transfer data directly from and to memory without the intervention of the CPU. Having peripheral devices access memory directly would allow the CPU to do other work, which would lead to improved performance, especially in the cases of large transfers.

The DMA controller is a piece of hardware that controls one or more peripheral devices. It allows devices to transfer data to or from the system’s memory without the help of the processor. In a typical DMA transfer, some event notifies the *DMA controller* that data needs to be transferred to or from memory. Both the DMA and CPU use memory bus and only one or the other can use the memory at the same time. The DMA controller then sends a request to the CPU asking its permission to use the bus. The CPU returns an acknowledgment to the DMA controller granting it bus access. The DMA can now take control of the bus to independently conduct memory transfer. When the transfer is complete the DMA relinquishes its control of the bus to the CPU. Processors that support DMA provide one or more input signals that the bus requester can assert to gain control of the bus and one or more output signals that the CPU asserts to indicate it has relinquished the bus. Figure 8.10 shows how the DMA controller shares the CPU’s memory bus.

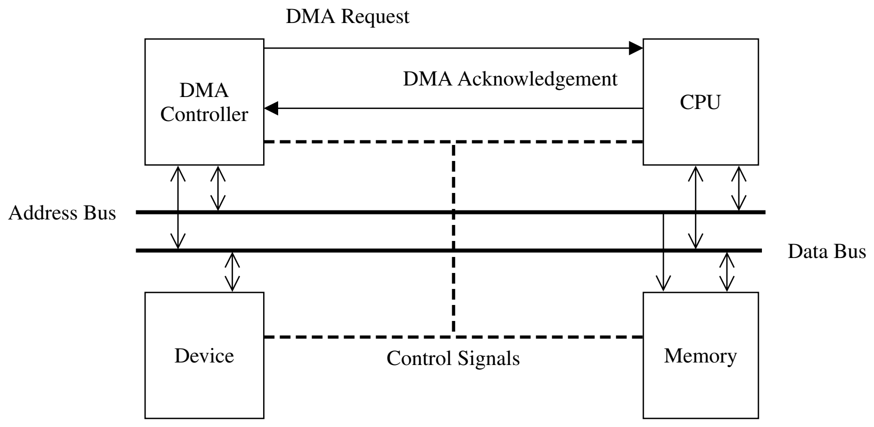


Figure 8.10 DMA controller shares the CPU's memory bus

Direct memory access controllers require initialization by the CPU. Typical setup parameters include the address of the source area, the address of the destination area, the length of the block, and whether the DMA controller should generate a processor interrupt once the block transfer is complete. A DMA controller has an address register, a word count register, and a control register. The address register contains an address that specifies the memory location of the data to be transferred. It is typically possible to have the DMA controller automatically increment the address register after each word transfer, so that the next transfer will be from the next memory location. The word count register holds the number of words to be transferred. The word count is decremented by one after each word transfer. The control register specifies the transfer mode.

Direct memory access data transfer can be performed in burst mode or single-cycle mode. In burst mode, the DMA controller keeps control of the bus until all the data has been transferred to (from) memory from (to) the peripheral device. This mode of transfer is needed for fast devices where data transfer cannot be stopped until the entire transfer is done. In single-cycle mode (cycle stealing), the DMA controller relinquishes the bus after each transfer of one data word. This minimizes the amount of time that the DMA controller keeps the CPU from controlling the bus, but it requires that the bus request/acknowledge sequence be performed for every single transfer. This overhead can result in a degradation of the performance. The single-cycle mode is preferred if the system cannot tolerate more than a few cycles of added interrupt latency or if the peripheral devices can buffer very large amounts of data, causing the DMA controller to tie up the bus for an excessive amount of time.

The following steps summarize the DMA operations:

1. DMA controller initiates data transfer.
2. Data is moved (increasing the address in memory, and reducing the count of words to be moved).

3. When word count reaches zero, the DMA informs the CPU of the termination by means of an interrupt.
4. The CPU regains access to the memory bus.

A DMA controller may have multiple channels. Each channel has associated with it an address register and a count register. To initiate a data transfer the device driver sets up the DMA channel's address and count registers together with the direction of the data transfer, read or write. While the transfer is taking place, the CPU is free to do other things. When the transfer is complete, the CPU is interrupted.

Direct memory access channels cannot be shared between device drivers. A device driver must be able to determine which DMA channel to use. Some devices have a fixed DMA channel, while others are more flexible, where the device driver can simply pick a free DMA channel to use.

Linux tracks the usage of the DMA channels using a vector of dma chan data structures (one per DMA channel). The dma chan data structure contains just two fields, a pointer to a string describing the owner of the DMA channel and a flag indicating if the DMA channel is allocated or not.

8.5. BUSES

A bus in computer terminology represents a physical connection used to carry a signal from one point to another. The signal carried by a bus may represent address, data, control signal, or power. Typically, a bus consists of a number of connections running together. Each connection is called a *bus line*. A bus line is normally identified by a number. Related groups of bus lines are usually identified by a name. For example, the group of bus lines 1 to 16 in a given computer system may be used to carry the address of memory locations, and therefore are identified as *address lines*. Depending on the signal carried, there exist at least four types of buses: *address*, *data*, *control*, and *power* buses. Data buses carry data, control buses carry control signals, and power buses carry the power-supply/ground voltage. The *size* (number of lines) of the address, data, and control bus varies from one system to another. Consider, for example, the bus connecting a CPU and memory in a given system, called the *CPU bus*. The size of the memory in that system is 512M-word and each word is 32 bits. In such system, the size of the address bus should be $\log_2(512 \times 2^{20})$ 29 lines, the size of the data bus should be 32 lines, and at least one control line ($\bar{R}/W$) should exist in that system.

In addition to carrying control signals, a control bus can carry timing signals. These are signals used to determine the exact timing for data transfer to and from a bus; that is, they determine when a given computer system component, such as the processor, memory, or I/O devices, can place data on the bus and when they can receive data from the bus. A bus can be *synchronous* if data transfer over the bus is controlled by a *bus clock*. The clock acts as the timing reference for all bus signals. A bus is *asynchronous* if data transfer over the bus is based on the availability of the data and not on a clock signal. Data is transferred over an asynchronous bus using a technique called *handshaking*. The operations of synchronous and asynchronous buses are explained below.

To understand the difference between synchronous and asynchronous, let us consider the case when a master such as a CPU or DMA is the source of data to be transferred to a slave such as an I/O device. The following is a sequence of events involving the master and slave:

1. Master: send request to use the bus
2. Master: request is granted and bus is allocated to master
3. Master: place address/data on bus
4. Slave: slave is selected
5. Master: signal data transfer
6. Slave: take data
7. Master: free the bus

8.5.1. Synchronous Buses

In synchronous buses, the steps of data transfer take place at fixed clock cycles. Everything is synchronized to bus clock and clock signals are made available to both master and slave. The bus clock is a square wave signal. A cycle starts at one rising edge of the clock and ends at the next rising edge, which is the beginning of the next cycle. A transfer may take multiple bus cycles depending on the speed parameters of the bus and the two ends of the transfer.

One scenario would be that on the first clock cycle, the master puts an address on the address bus, puts data on the data bus, and asserts the appropriate control lines. Slave recognizes its address on the address bus on the first cycle and reads the new value from the bus in the second cycle.

Synchronous buses are simple and easily implemented. However, when connecting devices with varying speeds to a synchronous bus, the slowest device will determine the speed of the bus. Also, the synchronous bus length could be limited to avoid clock-skewing problems.

8.5.2. Asynchronous Buses

There are no fixed clock cycles in asynchronous buses. Handshaking is used instead. Figure 8.11 shows the handshaking protocol. The master asserts the data-ready line

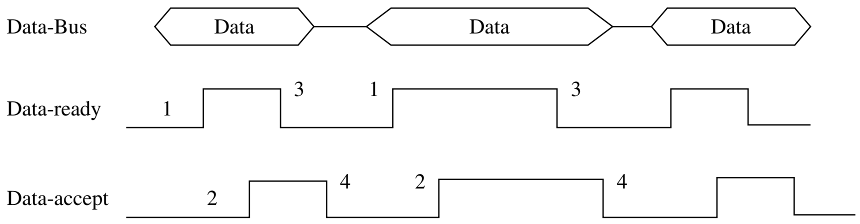


Figure 8.11 Asynchronous bus timing using handshaking protocol

(point 1 in the figure) until it sees a data-accept signal. When the slave sees a data-ready signal, it will assert the data-accept line (point 2 in the figure). The rising of the data-accept line will trigger the falling of the data-ready line and the removal of data from the bus. The falling of the data-ready line (point 3 in the figure) will trigger the falling of the data-accept line (point 4 in the figure). This handshaking, which is called fully interlocked, is repeated until the data is completely transferred. Asynchronous bus is appropriate for different speed devices.

8.5.3. Bus Arbitration

Bus arbitration is needed to resolve conflicts when two or more devices want to become the bus master at the same time. In short, arbitration is the process of selecting the next bus master from among multiple candidates. Conflicts can be resolved based on fairness or priority in a centralized or distributed mechanisms.

Centralized Arbitration In centralized arbitration schemes, a single arbiter is used to select the next master. A simple form of centralized arbitration uses a bus request line, a bus grant line, and a bus busy line. Each of these lines is shared by potential masters, which are daisy-chained in a cascade. Figure 8.12 shows this simple centralized arbitration scheme.

In the figure, each of the potential masters can submit a bus request at any time. A fixed priority is set among the masters from left to right. When a bus request is received at the central bus arbiter, it issues a bus grant by asserting the bus grant line. When the potential master that is closest to the arbiter (potential master 1) sees the bus grant signal, it checks to see if it had made a bus request. If yes, it takes over the bus and stops propagation of the bus grant signal any further. If it has not made a request, it will simple turn the bus grant signal to the next master to the right (potential master 2), and so on. When the transaction is complete, the busy line is deasserted.

Instead of using shared request and grant lines, multiple bus request and bus grant lines can be used. In one scheme, each master will have its own independent request and grant line as shown in Figure 8.13. The central arbiter can employ any priority-based or fairness-based tiebreaker. Another scheme allows the masters to have multiple priority levels. For each priority level, there is a bus request and a bus grant line. Within each priority level, daisy chain is used. In this scheme, each device is attached to the daisy chain of one priority level. If the arbiter receives multiple

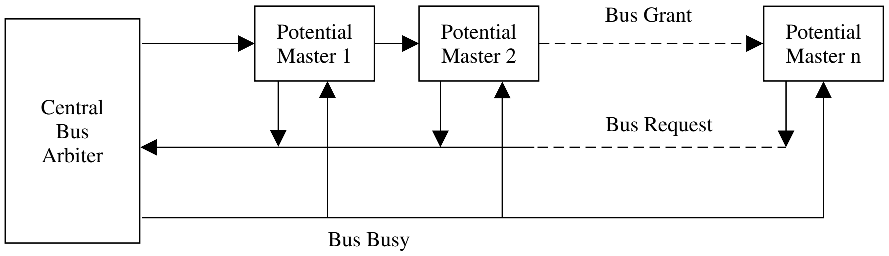


Figure 8.12 Centralized arbiter in a daisy chain scheme

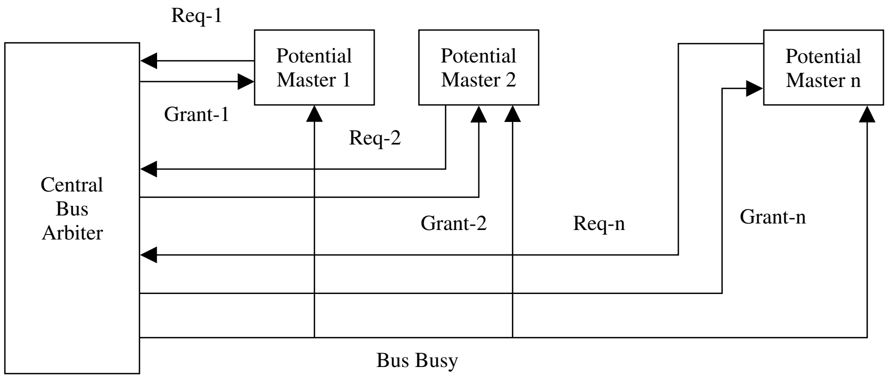


Figure 8.13 Centralized arbiter with independent request and grant lines

bus requests from different levels, it grants the bus to the level with the highest priority. Daisy chaining is used among the devices of that level. Figure 8.14 shows an example of four devices included in two priority levels. Potential master 1 and potential master 3 are daisy-chained in level 1 and potential master 2 and potential master 4 are daisy-chained in level 2.

Decentralized Arbitration In decentralized arbitration schemes, priority-based arbitration is usually used in a distributed fashion. Each potential master has a unique arbitration number, which is used in resolving conflicts when multiple requests are submitted. For example, a conflict can always be resolved in favor of the device with the highest arbitration number. The question now is how to determine which device has the highest arbitration number? One method is that a requesting device would make its unique arbitration number available to all other devices. Each device compares that number with its own arbitration number. The device with the smaller number is always dismissed. Eventually, the requester with the highest arbitration number will survive and be granted bus access.

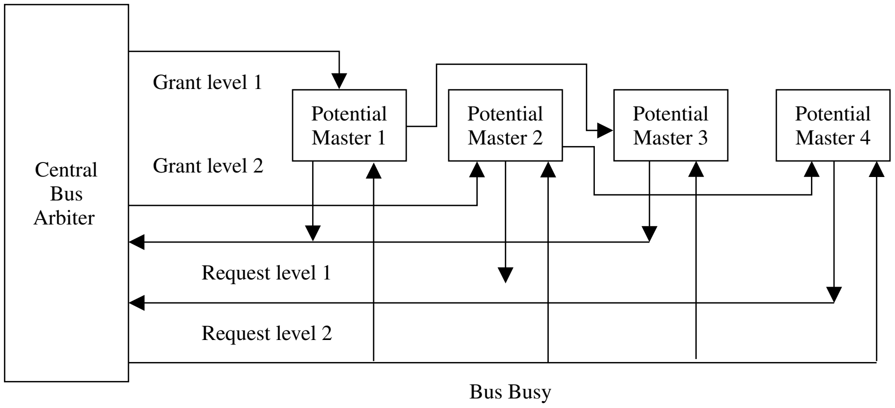


Figure 8.14 Centralized arbiter with two priority levels (four devices)

8.6. INPUT–OUTPUT INTERFACES

An interface is a data path between two separate devices in a computer system. Interface to buses can be classified based on the number of bits that are transmitted at a given time to serial versus parallel ports. In a serial port, only 1 bit of data is transferred at a time. Mice and modems are usually connected to serial ports. A parallel port allows more than 1 bit of data to be processed at once. Printers are the most common peripheral devices connected to parallel ports. Table 8.4 shows a summary of the variety of buses and interfaces used in personal computers.

TABLE 8.4 Descriptions of Buses and Interfaces Used in Personal Computers

Bus/Interface	Description
PS/2	A type of port (or interface) that can be used to connect mice and keyboards to the computer. The PS/2 port is sometimes called the mouse port.
Industry standard architecture (ISA)	ISA was originally an 8 bit bus and later expanded to a 16 bit bus in 1984. In 1993, Intel and Microsoft introduced a plug and play ISA bus that allowed the computer to automatically detect and set up computer ISA peripherals such as a modem or sound card.
Extended industry standard architecture (EISA)	EISA is an enhanced form of ISA, which allows for 32 bit data transfers, while maintaining support for 8 and 16 bit expansion boards. However, its bus speed, like ISA, is only 8 MHz. EISA is not widely used, due to its high cost and complicated nature.
Micro channel architecture (MCA)	MCA was introduced by IBM in 1987. It offered several additional features over the ISA such as a 32 bit bus, automatically configured cards and bus mastering for greater efficiency. It is slightly superior to EISA, but not many expansion boards were ever made to fit MCA specifications.
VESA (Video electronics standards association) local bus (VLB)	The VESA, a nonprofit organization founded by NEC, released the VLB in 1992. It is a 32 bit bus that had direct access to the system memory at the speed of the processor, commonly the 486 CPU (33/40 MHz). VLB 2.0 was later released in 1994 and had a 64 bit bus and a bus speed of 50 MHz.
Peripheral component interconnect (PCI)	PCI was introduced by Intel in 1992, revised in 1993 to version 2.0, and later revised in 1995 to PCI 2.1. It is a 32 bit bus that is also available as a 64 bit bus today. Many modern expansion boards are connected to PCI slots.
Advanced graphic port (AGP)	AGP was introduced by Intel in 1997. AGP is a 32 bit bus designed for the high demands of 3D graphics. AGP has a direct line to memory, which allows 3D elements to be stored in the system memory instead of the video memory. AGP is geared towards data intensive graphics cards, such as 3D accelerators; its design allows for data throughput at rates of 266 MB/s.

(*continued*)

TABLE 8.4 *Continued*

Bus/Interface	Description
Universal serial bus (USB)	USB is an external bus developed by Intel, Compaq, DEC, IBM, Microsoft, NEC and Northern Telcom. It was released in 1996 with the Intel 430HX Triton II Mother Board. USB has the capability of transferring 12 Mbps, supporting up to 127 devices. Many devices can be connected to USB ports, which support plug and play.
FireWire (IEEE 1394)	FireWire is a type of external bus, which supports very fast transfer rates: 400 Mbps. Because of this, FireWire is suitable for connecting video devices, such as VCRs, to the computer.
Small computer system interface (SCSI)	SCSI is a type of parallel interface that is commonly used for mass storage devices. SCSI can transfer data at rates of 4 MB/s; in addition, there are several varieties of SCSI that support higher speeds: Fast SCSI (10 MB/s), Ultra SCSI and Fast Wide SCSI (20 MB/s), as well as Ultra Wide SCSI (40 MB/s).
Integrated drive electronics (IDE)	IDE is a commonly used interface for hard disk drives and CD ROM drives. It is less expensive than SCSI, but offers slightly less in terms of performance.
Enhanced integrated drive electronics (EIDE)	EIDE is an improved version of IDE, which offers better performance than standard SCSI. It offers transfer rates between 4 and 16.6 MB/s.
PCI X	PCI X is a high performance bus that is designed to meet the increased I/O demands of technologies such as Fibre Channel, Gigabit Ethernet, and Ultra3 SCSI.
Communication and network riser (CNR)	CNR was introduced by Intel in 2000. It is a specification that supports audio, modem USB and local area networking interfaces of core logic chipsets.