10. Input/Output

Devices

Peripheral devices are all those devices which are attached externally to the fundamental components of a computer system (the CPU and Main Storage). A subset of these peripheral devices are those which allow the system to acquire data and to distribute data, both from external sources. These are referred to as Input devices and Output devices, respectively. Examples of Input devices are keyboards, pointing devices (mice, trackballs, etc), microphones and many others. Output devices include CRT displays (monitors), printers, and speakers. Some devices provide both input and output capability, such as modems and Network interfaces. External storage devices are sometimes considered in a class of their own.

It is also common to characterize I/O devices in terms of the amount of data they transfer on a single access. Byte-oriented (or perhaps, ‘word’) devices transfer a single byte between the system and the device on each access; keyboards and printers are examples of byte devices. Block-oriented devices transfer large blocks of bytes, say 512, on each access; Hard disk drives are block-oriented devices.

Device Components

An Input/Output (I/O) device consists of two main components, the device itself and the device controller. The devices themselves tend to be unique in many ways, both in the way data is stored and accessed, and in the way they are controlled to perform read and write operations. For instance, printers may be sent a byte at a time, over a parallel interface, and it may be necessary to wait for a response after each byte, while a modem is sent a continuous stream of bits over a serial interface. Even among devices of the same kind, say disk drives, internally they may require very different signals and command protocols to perform their function. Thus, while the primary job of the device is the transmission and/or storage of information, the primary job of the device controller is to handle the nuts and bolts of communication with the device while at the same time providing a consistent interface to the main system.

Within the device controller is the hardware necessary to communicate with the device as well as the hardware to communicate with the system. In support of the latter function the device controller generally has a set of registers, each of which is dedicated to a particular function. While device controllers vary greatly from device to device, some functions are common among all devices, and we can identify some common registers used in conjunction with these functions.
Data buffers
As a rule, the speed of communication between the controller and its device is much smaller than that between the controller and the system. When data is being received from the device it is generally stored in a storage device called a buffer until enough data has been received so that it is reasonable to forward the data to the system. For instance, if data is received from the device at the rate of a bit every microsecond, but data must be sent to the system at the rate of a byte every cycle, then the data is collected in the buffer at the rate of 1 bit/µs until, after 8µs there is enough data in the buffer to forward to the system on the following system cycle. Such a buffer is called a *speed-matching buffer*, and similar buffers will be found throughout a computer system but especially in I/O device controllers because of the great disparity generally found between device speeds and system speeds.

Speed matching buffers are used whenever data rates are different, even in cases where the clock speeds are the same. Take the case where a device can deliver a byte to the system on every cycle, but the system expects a word (4 bytes) every cycle. Again, a speed matching buffer would collect a byte on each of four cycles and, on the fifth cycle, deliver the entire word to the system. Note that this works in the other direction as well; the system can deliver a word to the device controller in one cycle, and the device controller will then deliver one byte to the device on each of the following four cycles.

Data Register
This register in the device controller is directly accessible by the system for transferring data between the system and the device controller.

Control Register
This register generally receives data (control information) from the system. It will govern, for instance whether the device is to do a read or a write, reset itself, or do some other device-specific function.

Status Register
This is a read-only register in the device controller which the system can access to determine the status of the device; for instance whether it is ready to receive data, or whether an error condition was detected on the device.

In addition to the above registers, the device controller may have other control signals which connect directly to the CPU. Of chief importance among these is the **interrupt** signal, to be discussed later.
Review Questions

1. Identify the purpose of each of the following
   a. Device Controller  d. Data Buffer
   b. Status Register  e. Control Register
   c. Data register

2. Give an example of a character-oriented device. A Block-oriented device.

3. Name three input devices. Name three output devices. (not including disk storage.)

4. If the word size of a CPU is 8 bytes, and a character-oriented device can deliver a single byte every two cycles
   a. How large must the data buffer for the device be?
   b. How often is data sent to the CPU?

CPU-I/O Interfaces

Handshaking

Communication between the system and the device controller is required to avoid conflicts in using common busses and resources, to ensure that no errors have occurred and to ensure that the CPU and the device are both ready to transfer data. In some cases this can be controlled by timing events in the CPU to ensure that whatever operations were previously initiated at the device have been completed. But it is also common to use a technique called handshaking to ensure a clean interface. A handshaking protocol can be implemented either using the registers described above or with discrete, special purpose, lines which directly connect the device controller with the CPU. Let’s suppose the latter case, as an example, with the following lines implemented between the controller and the CPU for an output only device:

58. Such an interface is called a ‘synchronous’ interface, since timing is critical.

59. These are termed ‘asynchronous’ interfaces since events can occur at times unrelated to specific machine cycles.
Then the handshaking protocol works as follows:

1. The CPU raises the signal ‘Ready to Send’, and holds it until the controller raises ‘Ready to Receive,’ or,
   The controller raises ‘Ready to Receive’ and holds it until the CPU raises ‘Ready to Send’.
2. In either case, when both ‘Ready to Receive’ and ‘Ready to Send’ are active the CPU puts data on the Data Out line and holds it there until the controller sends ‘Received OK’
3. The CPU drops ‘Data Out’. If ‘Received OK’ is not raised, the CPU will perform some error recovery procedure, perhaps retrying the operation.

The following figure shows a timing diagram of this operation:

System to Controller Interface

The device controller is the interface between the device and the main system. It’s interface to the device is unique to that device, but its interface to the system is one of very few common interfaces.
Serial Interface

With a serial interface, data is passed between the device controller and the system over a single data line, with bits sent sequentially over the line one at a time. There might be two data lines if the device is bidirectional; that is, it can both send and receive data. The most common serial interface is defined by an IEEE standard called RS232 (now called EIA-232). The RS232 protocol defines which physical lines are required, the pins they occupy on a plug, and the timing of the various lines to perform the necessary functions. This is an asynchronous handshaking protocol.

Two new high-speed serial interfaces are now in the process of replacing RS232 for many, if not most, applications. The first of these, originally designed for apple machines, is known as Firewire, which has been standardized by the IEEE as IEEE 1394. It can transmit data at speeds up to 400MBps. Up to 63 devices can be connected to a single Firewire port using daisy-chaining.

The second new high-speed serial interface is the Universal Serial Bus (USB). Version 1.0 of this standard, now becoming obsolete, transfers data at 1.5MBps. Version 2.0 transmits data at a rate slightly faster than Firewire: 460MBps, but a new version of Firewire will soon exceed this. USB devices can attach to a hub, so that up to 128 devices can share a single USB connection.

Parallel Interface

The parallel interface differs from the serial interface in that there are eight data lines instead of just one (in each direction.) In early implementations of parallel interfaces data could only be sent from the system to the device, usually a printer. The current parallel interface standard, IEEE 1284, provides for bidirectional communication. Newer printers use the data lines to the system to provide status information.

Bus Connected

Bus connected device controllers are tied directly to the system bus. This kind of connection is found in all desktop computer systems. The system bus is a set of lines which provide, data, control, and addressing information to all peripheral as well as to main memory. The bus consists of three groups of lines as already indicated: one group each for data, control and addressing. Depending on the kind of I/O addressing supported by the system architecture, there may be some additional addressing lines dedicated to I/O, or a subset of the systems address bus may be used by the I/O.
Channels

Channels are interconnection devices found in mainframe computer systems. They subsume most of the functions of the I/O device controller, but also contain a processor of their own, called the I/O processor, which can handle all of the programming functions required to communicate with devices. This means that the CPU can simply kickoff a channel program in the I/O processor and then go about its own business while the I/O processor completes the data transfers. Channels are high capacity, both in data transfer rate and also in the number of devices each channel can support.

Review Questions

1. Identify the type of interface associated with each of the following characteristics:
   a. All I/O share the same address and data busses.
   b. Uses I/O programs running on a processor separate from the CPU.
   c. EIA-232 is an asynchronous standard for this interface.
   d. Transmits multiple data signals simultaneously.
   e. IEEE 1284 is a standard for this interface.
   f. Can use hubs to connect up to 128 devices
   g. Uses daisy-chaining and can transmit at 400MBps

I/O addressing

Typically, there are many I/O and peripheral devices attached to a computer system and, of course, the system needs a way to uniquely identify each one. In addition, each of the registers in the I/O controller we described above needs to be identified. Both of these goals are accomplished by assigning every I/O register in the system its own address. From a more global perspective, there are two distinct mechanisms used to provide these addresses.

Direct I/O

Also called Port Addressing, in this implementation every I/O register in the system is assigned an address, called a port, in an address space reserved for I/O. This scheme requires architecture support in the form of two special instructions, usually called IN and OUT. Each instruction has, as an operand, a
port address to or from which data will be transferred. The address itself is placed either on the memory address bus or on a reserved set of I/O address lines. If the memory address bus is used, the IN and OUT instructions tell the I/O controllers that they should observe the address bus while, at the same time, telling memory to ignore it. Since the role of the address bus is specified by the kind of instruction being used, all of the available CPU address space is available for I/O use. In Intel machines, there is a separate 16-bit I/O address bus connected only to the I/O devices.

**Memory Mapped**

With this scheme, the I/O registers are assigned memory addresses rather than port addresses. That is, a region of memory address space, usually 64KB at the top of the memory address space, are reserved for device controller registers. In this scheme, no special I/O instructions are architected; normal memory instructions like LOAD and STORE are used to access the desired resources. Memory is designed so that the I/O mapped addresses are ignored (treated as if no memory exists there, which it probably doesn’t) and the I/O controllers recognize addresses in the designated range as being intended for them. Display adapters usually use this kind of addressing for data to be displayed on a monitor.

<table>
<thead>
<tr>
<th>Review Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Which kind of addressing uses IN and OUT instructions to access the registers in a device controller?</td>
</tr>
<tr>
<td>2. Which kind of addressing uses Load and Store instructions to access the registers in a device controller?</td>
</tr>
<tr>
<td>3. If a (hypothetical) device controller had a very large number of control registers in it, which type of addressing would be appropriate?</td>
</tr>
<tr>
<td>4. In memory mapped I/O, how is data prevented from going to memory if it is intended for the I/O device?</td>
</tr>
<tr>
<td>5. In Port I/O, is a separate I/O address bus required?</td>
</tr>
</tbody>
</table>

**CPU - I/O Control**

**Programmed I/O**

The most straightforward implementation of I/O control is very much like the handshaking protocol described above, but controlled by software rather than hardware. A typical fragment of code might look like this:
Introduction to Computer Architecture

<table>
<thead>
<tr>
<th>OUT data</th>
<th>//Send data to the I/O device</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN status</td>
<td>//Read I/O status register</td>
</tr>
<tr>
<td>JZ loop:</td>
<td>//If no status, jump back to loop, read status again.</td>
</tr>
<tr>
<td>continue</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this code, after the data is sent to the device, the CPU waits in a very tight loop, continually sampling the status register of the device controller until the status becomes nonzero, indicating that the data transfer was completed. The CPU will now test the status to make sure the operation was error-free, and then proceed to send the next data, if any. As a rule, I/O operations are very slow compared with CPU speeds, so the CPU could spend quite a bit of time in this loop. No other useful work is being done by the CPU while waiting for the I/O status, so this mechanism is not too useful in a multitasking system, or any system where CPU utilization is an important performance parameter.

**Interrupt Driven I/O**

Notice, in the above example, that the CPU is tied up here doing absolutely nothing useful beyond waiting for the device to respond. In a multitasking system, in order to get CPU utilization to a high point, it is possible, and necessary, to have the CPU doing useful work as much a possible, and this situation is a prime candidate for improvement. Ideally, as soon as the OUT data instruction has been issued, this process should be removed from the CPU (by the operating system) and another process started up. To implement such a scheme, though, requires that the I/O device have some proactive way to notify the system that it has completed its task. This is the job of the **hardware interrupt**. The hardware interrupt is a line from the I/O controller to the CPU which, when activated, causes an interruption in whatever process is using the CPU. It causes the operating system to take over control and, eventually, ‘wakes up’ the original process which made the I/O request so that it may continue its task. The overall operation may look something like the following:

- Process A issues an I/O request
- The Operating System (OS) suspends Process A and starts Process B
- The I/O device is ready for service; it activates the Interrupt Signal to the CPU
- The OS (actually a portion of it called the I/O interrupt handler) examines the interrupt and determines which process is involved.
- Process B is suspended and Process A resumes.
Interrupts will be discussed in more detail shortly.

Direct Memory Access (DMA)

So far we have not discussed what happens to data when it is transferred from an I/O device to the system. In the system as we have described it so far, each byte of data transferred from an I/O device is sent to the CPU (over the system data bus) which then turns around and sends it back over the system data bus to memory. This involves the CPU in every byte of each data transfer, and busies up the system bus twice for each byte transferred. DMA, as the name implies, allows I/O controllers to communicate directly with memory, without interrupting the current CPU process. Not only is the CPU allowed to continue without interruption, but the system bus is only used once for each word transferred. This requires the addition of an additional piece of hardware called the DMA module.

The DMA module contains information received from the CPU regarding what kind of operation (Load, Store) is to be done, the addresses of the device and memory locations involved, and the number of words to transfer. When it is time to actually transfer data, the DMA module controls the system data bus. If only single byte transfers are done, then the DMA module can ‘steal cycles’ from the CPU’s use of the bus by raising a line to the CPU saying the bus is busy for one cycle. It is not uncommon that the system bus may be idle anyway, so that there is no impact on CPU performance while the DMA transfer takes place. Such transfers may involve just one byte at a time, or an entire block (say, 512 bytes) might be transferred to/from memory in a single operation.

Review Questions

1. Which type of I/O control consumes CPU cycles waiting for a response from an I/O device?
2. Which type of I/O can bypass the CPU when transferring data?
3. Which type of I/O is typically used in most modern computing systems, especially multitasking systems?
4. Which type of I/O control requires no special hardware to handle I/O responses?
5. Which type of I/O control relies on ‘cycle stealing’ to transfer data?
6. Which type of I/O control is the least efficient with respect to CPU usage? Which is the most?

Interrupts

We have already seen how interrupts are used to signal the CPU that an I/O device is requesting service. These interrupts are part of a set of interrupts called hardware
interrupts. In addition there are a set of interrupts called software interrupts. The mechanisms for handling these interrupts have much in common, as well as some significant differences.

**Hardware Interrupts**, as might be expected, are interrupts generated by hardware events. In addition to I/O, or external, interrupts, there are also interrupts generated within the CPU itself, such as error interrupts (usually called exceptions, such as parity errors, or overflows detected in the ALU), timer interrupts (signals from the hardware clocking circuitry that indicate a certain amount of time has elapsed), and page faults.

The CPU chip has input pins which are allocated to receiving the signals generated by hardware events. An Intel chip has ten such inputs; two of them are used for the interrupt signal itself, while the other eight are for the interrupt code, which indicates which kind of device, or service, is being requested.

The two interrupt signals have somewhat different uses. In general, when a device signals an interrupt the CPU process currently running is not immediately interrupted; the interrupt is recognized at the next convenient time, usually at the completion of the currently executing instruction. However, it sometimes happens that the CPU is performing some task, say an Operating System function, for instance that ought not be interrupted until the entire task, not just one instruction, is completed. To provide for this situation the CPU has the capability to ‘mask’ the interrupt signal; that is, by setting a mask bit in the CPU the hardware pin on the chip which receives the interrupt is ignored; the CPU will not even be aware of the existence of the interrupt until the mask is turned off. But now we have other situations of a catastrophic, or emergency, nature which we would like to be recognized regardless of the mask bit. An imminent power failure, say, would be an instance of such an ‘emergency’. Thus the second interrupt pin on the chip is provided. This pin is for a signal called a Non-maskable Interrupt. Connected to this pin are those devices whose interrupts must be serviced immediately, regardless of the CPU operation currently in process.

Finally, we know that a typical computer system has many hardware devices which must have lines connected to the CPU, despite that fact that only one (maskable) pin is available for this. The solution is the addition of other chips between the I/O controllers and the CPU called Programmable (or Priority) Interface Controllers (PCIs). Each of these chips is basically a multiplexor which acts as a funnel, taking up to eight interrupts (and each of their associated eight bit interrupt code buses), prioritizing them, and sending one set of nine lines to the CPU. If more than eight devices are in use, than additional PCIs can be cascaded together to provide the desired I/O capacity. Note that in the DOS operating system, only sixteen such external hardware interrupts (called
IRQs, for interrupt requests) could be handled\(^60\).

**Software Interrupts** are interrupts explicitly generated by software. Most CPU architectures include an INT XX instruction for this purpose. The XX is a one-byte interrupt number which identifies the *interrupt routine* to be executed (more on interrupt routines shortly.)

**Interrupt Processing**

Interrupt handling is done by a part of the Operating system called the Interrupt Handler. When the interrupt is recognized, the Interrupt Handler is run. It in turn determines which Interrupt Routine is required to be run. Each distinct I/O device, error condition, etc. has its own Interrupt Routine, and the routine to be run is identified by a code which accompanies the interrupt. The routines themselves are stored in memory as part of the operating system, and may be located anywhere in main storage. However, located at the very beginning of main storage is a table, the **Interrupt Vector Table (IVT)**, which contains all the addresses of the beginning of each interrupt routine. Consider DOS. In a system running Microsoft OSs this table contains 256 addresses, each of which is four bytes in size. Thus, the Interrupt Vector Table occupies the first 1024 bytes of memory. The eight bit interrupt code that accompanies each interrupt points to one of these 256 addresses. When the interrupt starts being processed, the I/O handler accesses the table, and loads the appropriate address into the program counter, whereupon the interrupt routine begins to run.

The interrupt routine is basically a programming subroutine, frequently a *device driver*, and as such care has to be taken whenever it is called. First, of course, the status and important information regarding the currently running process must be saved, so that it can be restarted when the routine finishes. Second, the address of the next instruction of the current process must be saved so that it can be restored to the PC at the conclusion of handling the Interrupt. The routine itself must save any CPU register contents that it wants to use, and restore them when it is done so that the resumed process finds the CPU in the exact same condition it left it. It is usual to save all this information on a stack in main storage. The overall process is outlined here:

- Process A is in process; Instruction address x is in the Program Counter (PC).
- An interrupt is recognized; the accompanying 8-bit id is Y.
- The contents of the PC, x, is pushed onto the stack, and the interrupt handler starts.
- The interrupt handler fetches address z from IVT location Y and puts it in the PC.

---

\(^60\)Strictly speaking, only 15 can be used because one of the eight ports on the PCI chip which connects to the CPU, IRQ 2, is used to provide the cascade function for the second PCI chip.
• The Interrupt Routine at address z begins to run.
• The IR saves all necessary CPU registers on the stack.
• The IR performs the operations required by the interrupt.
• The IR restores all saved CPU registers from the stack.
• The IR pops address x off the stack into the PC.
• Process A resumes with the instruction at address x.

Review Questions

1. Name two uses for the stack in handling interrupts.
2. How are interrupt codes transmitted to the CPU?
3. How are interrupts prioritized?
5. When are interrupts masked by an I/O subroutine?
6. At what point in the execution of an instruction is an outstanding interrupt generally recognized?
7. If an IVT contains 8-byte addresses, what is the byte-address of the subroutine address for an interrupt with an interrupt code of 26?
8. If an Interrupt Vector Table has 512 2-byte addresses in it,
   a. How many bits does the interrupt number consist of?
   b. How big is Main Storage (RAM)?