Unit 11 Input / Output Basics

Structure:

11.1 Introduction

In addition to the CPU and a set of memory modules, another key element of a computer system is a set of I/O module. An I/O module is not just

mechanical connectors that wire a device into the system bus but it contains some intelligence, that is, contains a logic for performing a communication function between the peripheral and the bus. Owing to the following reason we need an I/O module.

- There are wide varieties of peripherals with a variety of operation methods. It is impractical to incorporate the necessary logic within the CPU to control a range of devices.
- The data transfer rate of peripherals is often much slower than that of the memory or CPU.
- Peripherals often use different data formats and word lengths than the computer system to which they are attached.

An I/O module has two major functions:

1) Interface to the CPU and memory via the system bus or central switch
2) Interface to one or more peripheral devices by tailored data links.

Objectives:

By the end of Unit 11, the learners should be able to:

1. Discuss the classification of external devices
2. Discuss I/O problems and functions of I/O modules
3. Explain the concept of programmed I/O.
4. Explain the various I/O instructions
5. Explain the concept of Interrupt driven I/O.

11.2 External Devices

I/O operations are accomplished through a wide assortment of external devices that provide a means of exchanging data between the external environment and the computer. An external device attaches to the computer by a link to an I/O module as shown in figure 11.1. The link is used to exchange control, status and data between the I/O module and the external

device. An external device connected to an I/O module is often referred to as a peripheral device or simply a peripheral.

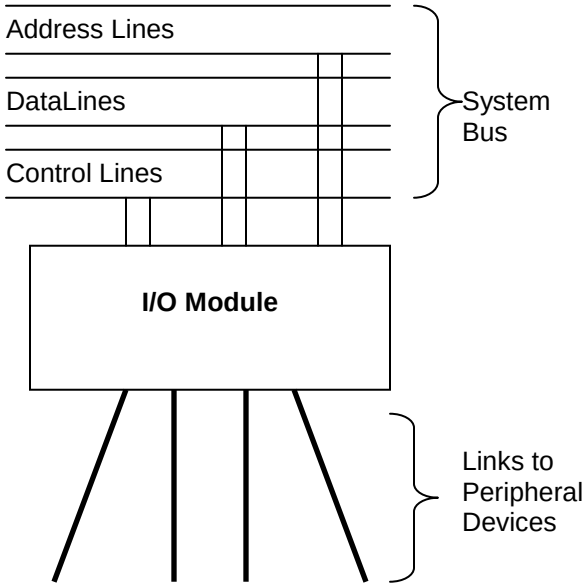

Figure 11.1: Generic Model of an I/O Module

Classification of External devices

External devices can be broadly classified into three categories:

1. Human readable: suitable for communicating with the computer user.
   Examples: Screen, keyboard, video display terminals (VDT) and printers.
2. machine readable: suitable for communicating with equipments
   Examples: magnetic disk & tape systems, Monitoring and control, sensors and actuators which are used in robotics.
   From functional point of view these devices are part of memory hierarchy. But from structural point of view these devices are controlled by I/O modules.
3. Communication: These devices allow a computer to exchange data with remote devices, which may be machine readable or human readable.
   Examples: Modem, Network Interface Card (NIC)

The nature of external devices and their interface is indicated in figure 11.2. The interface of I/O module is in the form of control, status and data signals. Data are in the form of a set of bits to be sent and received from the I/O module. Control signals determine the function of the device, like INPUT or READ to accept data from the I/O device, OUTPUT or WRITE to report status, or perform some control particular to that device. Some signals indicate the state of the device such as READY or NOTREADY to show whether the device is ready for data transfer.

Control logic associated with the device controls the device operation in response to the direction from the I/O module. The transducer converts the data forms from electrical signals into other forms of energy or vice versa. Typically a buffer is associated with the transducer to temporarily hold the data being transferred between the I/O module and external devices.

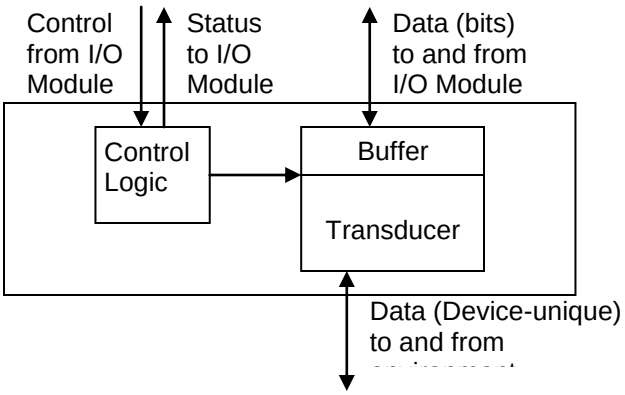

Figure 11.2: An External Device

Input / Output Problems

- Wide variety of peripherals
- Delivering different amounts of data
- At different speeds
- In different formats
- All slower than CPU and RAM

Hence Need I/O modules

11.3 Input / Output Module

It is the entity within a computer that is responsible for the control of one or more external devices and for the exchange of data between those devices and main memory and/or CPU.

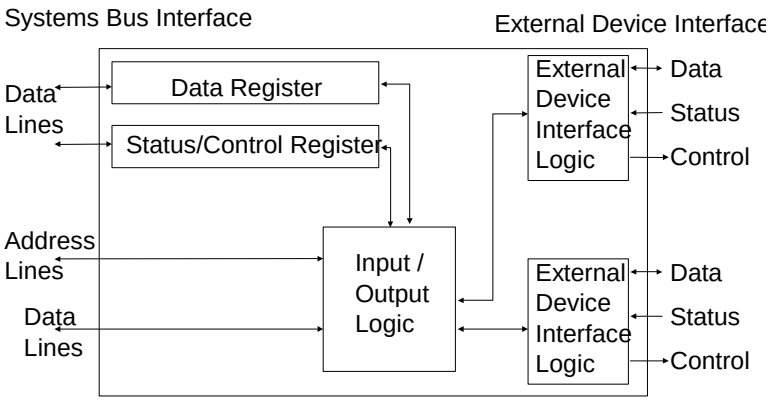

Figure 11.3: I/O Module Diagram

The block diagram of an I/O Module is as shown in figure 11.3. Thus I/O memory must have

- Interface to CPU and Memory
- Interface to one or more peripherals

I/O Module Function

The major functions or requirements for an I/O module fall into the following five categories.

- Control & Timing
- CPU Communication
- Device Communication
- Data Buffering
- Error Detection

During any period of time, the CPU may communicate with one or more external devices in unpredictable patterns on the program's need for I/O, the internal resources, main memory and the CPU must be shared among

number of activities, including handling data I/O. Thus the I/O device includes a control and timing requirement to coordinate the flow of traffic between internal resources and external devices to the CPU. Thus CPU might involve in sequence of operations like:

- CPU checks I/O module device status
- I/O module returns device status
- If ready, CPU requests data transfer
- I/O module gets data from device
- I/O module transfers data to CPU
- Variations for output, DMA, etc.

I/O module must have the capability to engage in communication with the CPU and external device. Thus CPU communication involves

- Command decoding: The I/O module accepts commands from the CPU carried on the control bus.
- Data: data are exchanged between the CPU and the I/O module over data bus
- Status reporting: Because peripherals are slow it is important to know the status of I/O module. I/O module can report with the status signals. Commonly used status signals are BUSY or READY. Various other status signals may be used to report various error conditions.
- Address recognition: just as each memory word has an address, there is address associated with every I/O device. Thus I/O module must be recognized with a unique address for each peripheral it controls.

The I/O module must also be able to perform device communication. This communication involves commands, status information, and data. Some of the essentials tasks are listed below:

- Data buffering: the transfer rate into and out of main memory or CPU is quite high, and the rate is much lower for most of the peripherals. The

table 11.1 gives the data rate of various devices. Data coming from main memory are sent to an I/O module in a rapid burst. The data is buffered in the I/O module and then sent to the peripheral device at its rate. In the opposite direction data are buffered so as not to tie up the memory in a slow transfer operation. Thus I/O module must be able to operate at both device and memory speeds.

Table 11.1: data rates of various devices

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Telephone channel	8 KB/sec
Dual ISDN lines	16 KB/sec
Laser printer	100 KB/sec
Scanner	400 KB/sec
Classic Ethernet	1.25 MB/sec
USB (Universal Serial Bus)	1.5 MB/sec
Digital camcorder	4 MB/sec
IDE disk	5 MB/sec
40x CD-ROM	6 MB/sec
Fast Ethernet	12.5 MB/sec
ISA bus	16.7 MB/sec
EIDE (ATA-2) disk	16.7 MB/sec
FireWire (IEEE 1394)	50 MB/sec
XGA Monitor	60 MB/sec
SONET OC-12 network	78 MB/sec
SCSI Ultra 2 disk	80 MB/sec
Gigabit Ethernet	125 MB/sec
Ultrium tape	320 MB/sec
PCI bus	528 MB/sec
Sun Gigaplane XB backplane	20 GB/sec

- Error detection: I/O module is often responsible for error detection and subsequently reporting errors to the CPU. We categorize here two classes of errors.
  1. Mechanical and electrical malfunctions: These types of errors are reported by the device such as paper jam, bad disk track, etc.

2. Unintentional change of bits pattern: As the data is transmitted from device to I/O module unintentionally, bit patterns might change. Some kind of error detecting code is often used to detect the transmission errors.

Common example is to use a parity bit on each character of data. For example ASCII character code occupies 7-bits of a byte. The eighth bit is set so that the total number of 1's in the byte is even for even parity or total number of 1's in the byte is odd for odd parity. Thus when a byte is received an I/O module checks the parity to find whether any error has occurred.

I/O Module Decisions

- Hide or reveal device properties like details of timing, formats to CPU.
- Support multiple or single device.
- Control device functions or leave it for CPU.
- Also OS decisions e.g. Unix treats everything it can as a file.

Consider the problem of moving a character code from the keyboard to the processor.

Striking a key, stores the corresponding character code in an 8-bit buffer register associated with the keyboard. Let us call this register DATAIN, as shown in Figure 11.4.

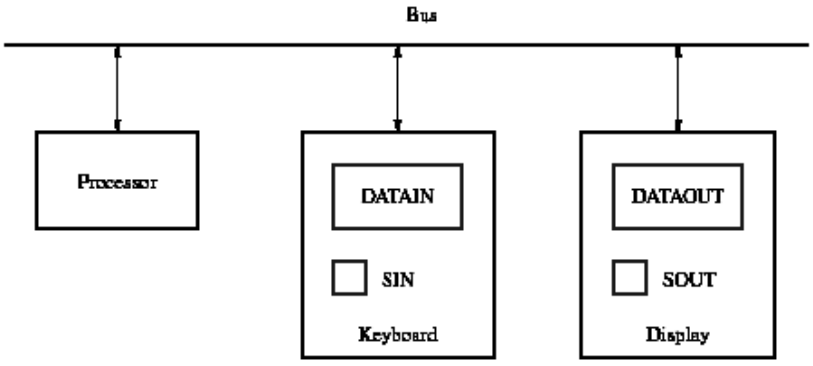

Figure 11.4: example of an I/O module

To inform the processor that a valid character is in DATAIN, a status control flag, SIN, is set to 1. A program monitors SIN and when SIN is set to 1, the processor reads the contents of DATAIN. When the character is transferred to the processor, SIN is automatically cleared to 0. If a second character is entered at the keyboard, SIN is again set to 1 and the process repeats.

An analogous process takes place when characters are transferred from the processor to the display. A buffer register, DATAOUT and a status control flag, SOUT are used for this transfer. When SOUT equals 1, the display is ready to receive a character. Under program control, the processor monitors SOUT, and when SOUT is set to 1, the processor transfers a character code to DATAOUT. The transfer of a character to DATAOUT clears SOUT to 0; when the display device is ready to receive a second character, SOUT is again set to 1. The buffer registers DATAIN and DATAOUT and the status flags SIN and SOUT are part of circuitry commonly known as a ***device interface***.

An I/O module that takes most of detailed processing burden, presenting to a high level CPU, is usually referred to as ***I/O channel or I/O processor***. An I/O module that is primitive and requires detailed control is usually referred to as an ***I/O controller or device controller***. I/O controller is seen on microcomputers, I/O channels on mainframes, with minicomputers employing the mixture.

Input Output Techniques

Three techniques are possible for I/O operations. They are:

- Programmed I/O
- Interrupt driven
- Direct Memory Access (DMA)

Table 11.2 indicates the relationship among the three techniques.

With ***Programmed I/O***, data are exchanged between the CPU and the I/O module. The CPU executes a program that gives it direct control of the I/O operation, including sensing device status, sending a read or write command and transferring data. When CPU issues a command to I/O module, it must wait until I/O operation is complete. If the CPU is faster than I/O module, there is wastage of CPU time.

With ***Interrupt driven I/O***, the CPU issues a command to I/O module and it does not wait until I/O operation is complete but instead continues to execute other instructions. When I/O module has completed its work it interrupts the CPU.

With both ***Programmed I/O*** and ***Interrupt driven I/O*** the CPU is responsible for extracting data from main memory for output and storing data in main memory for input.

Table 11.2: I/O Techniques

	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt driven I/O
Direct I/O-to-memory transfer		Direct Memory Access (DMA)

11.4 Programmed I/O

When the CPU is executing a program and encounters an instruction relating to I/O, it executes those instructions by issuing a command to the appropriate I/O module. With this technique or using programmed I/O, the I/O module will perform the requested action and then set appropriate bits in the status register. The I/O module does not take any further action to alert CPU that is it does not interrupt CPU. Hence it is the responsibility of the CPU to periodically check the status of the I/O module until it finds that the operation is complete.

The sequences of actions that take place with programmed I/O are:

- CPU requests I/O operation
- I/O module performs operation
- I/O module sets status bits
- CPU checks status bits periodically
- I/O module does not inform CPU directly
- I/O module does not interrupt CPU
- CPU may wait or come back later

I/O commands

To execute an I/O related instruction, the CPU issues an address, specifying the particular I/O module and external device and an I/O command. Four types of I/O commands can be received by the I/O module when it is addressed by the CPU. They are

- A control command: is used to activate a peripheral and tell what to do.

  Example: a magnetic tape may be directed to rewind or move forward a record.

- A test command: is used to test various status conditions associated with an I/O module and its peripherals. The CPU wants to know the interested peripheral for use. It also wants to know the most recent I/O operation is completed and if any errors have occurred.
- A read command: it causes the I/O module to obtain an item of data from the peripheral and place it in an internal buffer. The CPU then gets the data items by requesting I/O module to place it on the data bus.
- A write command: it causes the I/O module to take an item of data from the data bus and subsequently transmit the data item to the peripheral.

A flow chart for input of a block of data using programmed I/O is as shown in figure 11.5. It reads in a block of data from a peripheral device into memory.

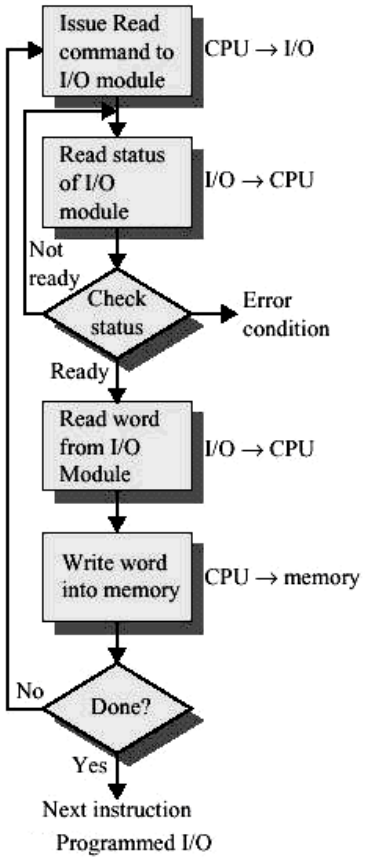

Fig. 11.5: input of block of data using programmed I/O

Data is read in one word at a time. For each word read in, the CPU keeps checking the status until the word is available in I/O module's data register.

That is the possible sequence of I/O commands and actions are:

- CPU issues address
- Identifies module (& device if >1 per module)
- CPU issues command
- Control - telling module what to do
- e.g. spin up disk
- Test - check status
- e.g. power? Error?

- Read/Write
- Module transfers data via buffer from/to device

From the flow chart it is clear that the main disadvantage of this technique is that it is a time consuming process that keeps the processor busy unnecessarily.

I/O instructions

Addressing I/O Devices

- Under programmed I/O data transfer is very like memory access (CPU viewpoint)
- Each device given unique identifier
- CPU commands contain identifier (address)

I/O Mapping

When the CPU, main memory, and I/O module share a common bus, two modes of addressing are possible.

1. *Memory mapped I/O*
    - Devices and memory share an address space
    - I/O looks just like memory read/write
    - No special commands for I/O
    - Large selection of memory access commands available
2. *Isolated I/O*
    - Separate address spaces
    - Need I/O or memory select lines
    - Special commands for I/O
    - Limited set

Consider an example: The processor can monitor the keyboard status flag SIN and transfer a character from DATAIN to register R1 using the following sequence of operations:

READWAIT Branch to READWAIT if SIN = 0

Input from DATAIN to R1

The Branch operation is usually implemented by two machine instructions. The first instruction tests the status flag and the second performs the branch. Although the details vary from computer to computer, the main idea is that the processor monitors the status flag by executing a short *wait loop* and proceeds to transfer the input data when SIN is set to 1 as a result of a key being struck. The Input operation resets SIN to 0.

Another example is transferring output to the display. The sequence of instructions is:

WRITEWAIT Branch to WRITEWAIT if SOUT = 0.

Output from R1 to DATAOUT

Again, the Branch operation is normally implemented by two machine instructions. The wait loop is executed repeatedly until the status flag SOUT is set to 1 by the display when it is free to receive a character. The Output operation transfers a character from R1 to DATAOUT to be displayed, and it clears SOUT to 0.

We assume that the initial state of SIN is 0 and the initial state of SOUT is 1. This initialization is normally performed by the device control circuits when the devices are placed under computer control before program execution begins. Until now, we have assumed that the addresses issued by the processor to access instructions and operands always refer to memory locations. Many computers use an arrangement called *memory-mapped I/O* in which some memory address values are used to refer to peripheral device buffer registers, such as DATAIN and DATAOUT.

Thus, no special instructions are needed to access the contents of these registers; data can be transferred between these registers and the

processor using instructions that we have already discussed, such as Move, Load or Store. For example, the contents of the keyboard character buffer DATAIN can be transferred to register R1 in the processor by the instruction

MoveByte DATAIN, R1

Similarly, the contents of register R1 can be transferred to DATAOUT by the instruction MoveByte R1,DATAOUT The status flags SIN and SOUT are automatically cleared when the buffer registers DATAIN and DATAOUT are referenced, respectively. The MoveByte operation code signifies that the operand size is a byte, to distinguish it from the operation code Move that has been used for word operands. We have established that the two data buffers may be addressed as if they were two memory locations. It is possible to deal with the status flags SIN and SOUT in the same way, by assigning them distinct addresses. However, it is more common to include SIN and SOUT in *device status* registers, one for each of the two devices.

Let us assume that bit *b*3 in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT, respectively. The read operation just described may now be implemented by the machine instruction sequence READWAIT Testbit #3, INSTATUS Branch=0 READWAIT MoveByte DATAIN, R1.

The write operation may be implemented as WRITEWAIT Testbit #3, OUTSTATUS.

Branch=0 WRITEWAIT

MoveByte R1, DATAOUT

The Testbit instruction tests the state of one bit in the destination location, where the bit position to be tested is indicated by the first operand. If the bit tested is equal to 0, then the condition of the branch instruction is true, and a branch is made to the beginning of the wait loop. When the device is

ready, that is, when the bit tested becomes equal to 1, the data are read from the input buffer or written into the output buffer.

The program shown in Figure 11.6 uses these two operations to read a line of characters typed at a keyboard and send them out to a display device. As the characters are read in, one by one, they are stored in a data area in the memory and then echoed back out to the display. The program finishes when the carriage return character, CR, is read, stored and sent to the display. The address of the first byte location of the memory data area where the line is to be stored is LOC. Register R0 is used to point to this area, and it is initially loaded with the address LOC by the first instruction in the program. R0 is incremented for each character read and displayed by the Autoincrement addressing mode used in the Compare instruction.

```
        Move        #LOC,R0          Initialize pointer register R0 to point to the
                                       address of the first location in memory
                                       where the characters are to be stored.
READ    TestBit     #3,INSTATUS      Wait for a character to be entered
        Branch=0    READ               in the keyboard buffer DATAIN.
        MoveByte    DATAIN,(R0)      Transfer the character from DATAIN into
                                       the memory (this clears SIN to 0).
ECHO    TestBit     #3,OUTSTATUS     Wait for the display to become ready.
        Branch=0    ECHO
        MoveByte    (R0),DATAOUT     Move the character just read to the display
                                       buffer register (this clears SOUT to 0).
        Compare     #CR,(R0)+        Check if the character just read is CR
                                       (carriage return). If it is not CR, then
        Branch≠0    READ               branch back and read another character.
                                       Also, increment the pointer to store the
                                       next character.
```

Figure 11.6: Program for read from a keyboard and write output to a display device

11.5 Interrupt Driven I/O

Using Program-controlled I/O requires continuous involvement of the processor in the I/O activities. It is desirable to avoid wasting processor

execution time. An alternative is for the CPU to issue an I/O command to a module and then go on other work. The I/O module will then interrupt the CPU requesting service when it is ready to exchange data with the CPU. The CPU will then execute the data transfer and then resumes its former processing. Based on the use of interrupts, this techniques improves the utilization of the processor.

An interrupt is more than a simple mechanism for coordinating I/O transfers. In a general sense, interrupts enable transfer of control from one program to another to be initiated by an event that is external to a computer. Execution of the interrupted program resumes after completion of execution of the interrupt service routine. The concept of interrupts is useful in operating systems and in many control applications where processing of certain routines has to be accurately timed relative to the external events. The latter type of application is generally referred to as real-time processing.

The operations listed can be better understood with an example to input a block of data. A flow chart using this technique for input of a block of data is as shown in figure 11.7.

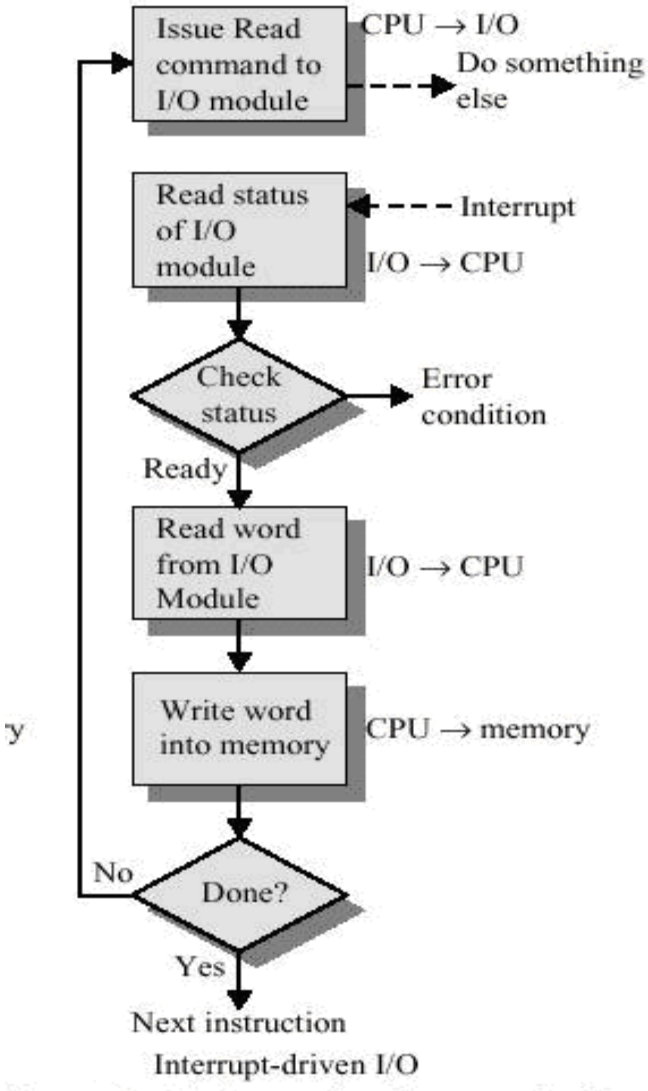

Figure 11.7: input block of data using interrupt driven I/O

Using Interrupt Driven I/O technique CPU issues read command. I/O module gets data from peripheral while CPU does other work and I/O module interrupts CPU, checks the status if no error, that is, the device is ready, then CPU requests data and I/O module transfers data. Thus CPU reads the data and stores it in the main memory.

Thus ***Interrupt Driven I/O***

- Overcomes CPU waiting.
- No repeated CPU checking of device.
- I/O module interrupts when ready.

Basic concepts of an Interrupt

An interrupt is an exception condition in a computer system caused by an event external to the CPU. Interrupts are commonly used in I/O operations by a device interface (or controller) to notify the CPU that it has completed an I/O operation.

An interrupt is indicated by a signal sent by the device interface to the CPU via an interrupt request line (on an external bus). This signal notifies the CPU that the signaling interface needs to be serviced. The signal is held until the CPU acknowledges or otherwise services the interface from which the interrupt originated.

Note: interrupts may be generated by interfaces or devices which are not involved in I/O. For example, if a computer system contains a clock (typically called a "timer") outside of the CPU, that clock may signal its "ticks" by interrupts. However, exception conditions or signals internal to the CPU, such as occur when there is an attempt to execute a non-existent instruction, are typically called traps, not interrupts.

Response of CPU to an Interrupt

The CPU checks periodically to determine if an interrupt signal is pending. This check is usually done at the end of each instruction, although some modern machines allow for interrupts to be checked for several times during the execution of very long instructions.

When the CPU detects an interrupt, it then saves its current state (at least the PC and the Processor Status Register containing condition codes); this

state information is usually saved in memory. After the interrupt has been serviced, this state information is restored in the CPU and the previously executing software resumes execution as if nothing had happened.

How is an Interrupt Serviced?

The CPU runs a program called an interrupt handler. Interrupt handler software may be general and be able to service all device interfaces, but typically an interrupt handler is designed to service an interface for exactly one type of device (e.g., a terminal or a disk).

The interrupt handler must make it possible for the software which was executing in the CPU and which was “interrupted" to resume its execution after the interrupt is serviced as if nothing had happened. Since the CPU hardware saves only a minimal amount of information when it saves the state of the CPU, the interrupt handler is responsible for saving and restoring data such as the contents of the general purpose registers which it uses.

Design Issues

- How do you identify the module issuing the interrupt?
- How do you deal with multiple interrupts?
- i.e. an interrupt handler being interrupted

Identifying Interrupting Module

When the CPU determines that an interrupt has occurred, it must be determined what device interface signaled the interrupt. Two primary approaches are used to do this. In some systems, a general interrupt handler queries each device interface. When one acknowledges positively that it originated the interrupt, then that device is serviced.

A more efficient alternative approach is to use vectored interrupts. When vectored interrupts are used, the interface signaling the interrupt will

“identify" itself by sending information (its "vector") to the CPU. That information will permit the CPU to select the correct interrupt handler. In actuality, that information may be a pointer to where the starting address of the interrupt handler is stored.

Multiple Interrupts

- Each interrupt line has a priority
- Higher priority lines can interrupt lower priority lines
- If bus mastering only current master can interrupt

Priorities

Because several interrupts may be found pending when the CPU checks for an interrupt, priorities may be used to determine in which order the interrupts are serviced. When several interrupts are pending, the one of highest priority is always serviced next. Priorities are assigned to device interfaces on the basis of any of several factors.

Generally faster devices have higher priorities assigned to their interfaces; disks transmit data at a faster rate than terminals so usually have higher priorities. Also, if a device interface is not serviced in time, incoming data may be written over by later data; thus devices such as sensors which do not have a means to recover lost data (without manual intervention) may be given fairly high priorities.

Priorities may be used to control the nesting of interrupts. In this case, the CPU is itself assigned a priority depending on what is executing; typical user programs usually run at the lowest priority while an interrupt handler may run at the same priority which is assigned the corresponding interface which it is servicing. Interrupts which carry a priority which is lower than (or equal to) the current CPU priority are masked out - they are simply not noticeable to the CPU. This allows "more important" device interfaces to interrupt the servicing of "less important" device interfaces, while "less important"

interrupts must await the completion of servicing "more important" ones. Furthermore, device interfaces can be prevented from interrupting their own interrupt handlers. In addition, all interrupts can be blocked out by raising the CPU priority level to its highest possible value; this would allow for a critical section of instructions to be executed without “interruption." However, this type of mechanism should only be used for very short segments of code and for very good reasons (e.g., during context switches).

Priorities may be implemented by providing several interrupt request lines in the external bus; each device is then attached to the line corresponding to the interrupt priority level to which it has been assigned. Furthermore, if several devices are attached to the same interrupt request line via daisy chaining, the one closest to be CPU will be serviced first.

Interrupt Handling

We have already introduced hardware interrupts, by which a hardware device can interrupt the normal operation of the CPU to signal that some significant or time-critical event has occurred (e.g., a character arriving over a serial line or an alarm event). The complete cycle of hardware and software operations involved in a typical interrupt, from signaling to resolution, are listed below:

Hardware Actions

- The device controller asserts an interrupt line on the system bus to start the interrupt sequence.
- As soon as the CPU is prepared to handle interrupt, it asserts an interrupt acknowledgement signal on the bus.
- When the device controller sees that its interrupt signal has been acknowledged it puts a small integer on the data lines to identify itself; this is the "interrupt vector".

- The CPU removes the interrupt vector from the bus and saves it temporarily.
- The CPU pushes the program counter and the Program Status Word (PSW, or FLAGS register) onto the stack.
- The CPU then locates a new program counter value using the interrupt vector as an index into a (operating system) table at a well-defined point in memory. This new program counter starts the interrupt service routine (normally part of the device driver) for the device causing the interrupt.

Software Actions

- The interrupt service routine saves all registers so that they can be restored later. This may be on the stack or in a system table.
- The exact device causing the interrupt is identified (e.g. by polling) - each interrupt vector may in general be shared by a number of devices.
- Any other information about the interrupt, e.g. status codes, can now be read in.
- If an I/O error has occurred it can be handed here.
- The interrupt is responded to in a device specific manner. **
- If necessary, a special code may be output to the device or interrupt controller to indicate that the interrupt has been handled.
- The saved register values are restored.
- Execute a "Return from Interrupt" instruction, which restores the PC and PSW from when the interrupt occurred.
- The program which was interrupt continues executing from exactly where it left off.

The step (**) above is where the "real" work will be done, e.g. reading in a character from a serial line into a local buffer or signaling to the rest of the program that some error or significant change in status has occurred.

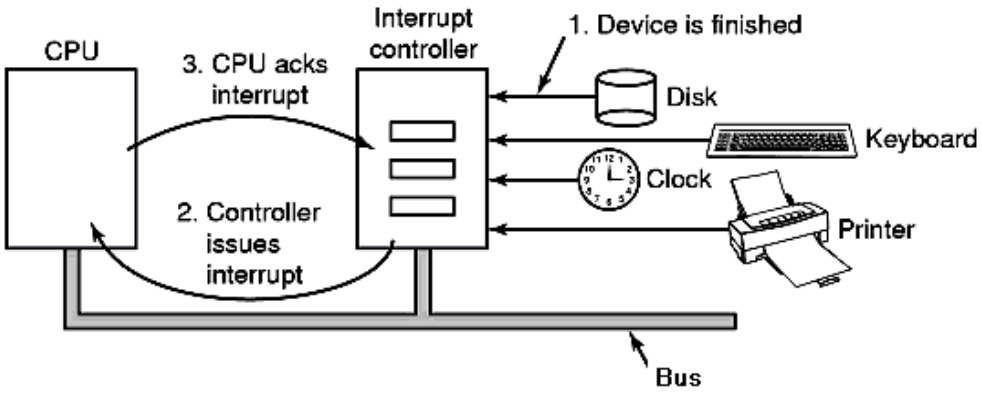

Figure 11.8: Connections between Devices and Interrupt Controller actually use the Bus Lines instead of dedicated Wires

Types of Interrupts

An interrupt is a event that causes the execution of one program to be suspended and another program to be executed. So far we have assumed that the occurrence of this event is caused by a request received by an I/O device in the form of a hardware signal over the computer bus. In fact, there are many uses for interrupts other than for controlling I/O transfers, some of which we will now describe.

Recovery from errors

Computers use a variety of techniques to ensure that all hardware components are operating properly. For example, many computers include a parity check code in the main memory, which allows detection of errors in the stored data. If an error occurs, the control hardware detects it and informs the CPU by raising an interrupt. The CPU may also interrupt a program if it detects an error or an unusual condition while executing the instructions of this program. For example, the OP-code field of an instruction may not correspond to any legal instruction, or some instruction may attempt a division by zero.

When an interrupt is initiated as a result of such causes, the CPU proceeds exactly in the same manner as in the case of an I/O interrupt request. It suspends the program being executed and starts an interrupt service routine. The interrupt service routine should take appropriate action to recover from the error, if possible or inform the user about it.

Debugging

Another important use of interrupts is as an aid in debugging programs. System software usually includes a program called a debugger, which helps the programmer to find errors in the program. The debugger uses interrupts to provide two important facilities: trace and break points. A trace facility causes an interrupt to occur after execution of every instruction in a program that is being debugged. The interrupt service routine for a trace interrupt starts the execution of the debugging routine, which allows the user to examine the contents of registers, memory locations and so on. On return from the debugging routine, the next instruction is executed, then the debugging routine is activated again. During the execution of the debugging routine trace interrupts are disabled.

Break points provide a similar facility, except that the program being debugged is interrupted at only specific points selected by the user. An instruction called trap or software interrupt is usually provided for this purpose. Execution of this instruction results in exactly the same actions as when a hardware interrupt request is received. Thus if the user wishes to interrupt a program after execution of the instruction i the debugging routine replaces instruction i+1 with a software interrupt instruction. When the program is executed and reaches that point, it is interrupted and the debugging routine is activated. Later, when the user is ready to continue executing the program being debugged, the debugging routine restores the instruction that was at location i+1 and executes a Return-from-interrupt instruction.

Communication between Programs

Software interrupt instructions are used by the operating system to communicate with and control the execution of other programs.

11.6 Summary

An I/O module is a key element of a computer system. An I/O module has two major functions: first one to interface with the CPU and memory via the system bus or central switch and second to interface with one or more peripheral devices by tailored data links. Basic I/O operations and Programmed I/O, Interrupt-driven I/O, Direct memory access (DMA) techniques are also discussed.

Programmed I/O does not interrupt the processor. The processor must periodically check the status of the I/O module until the task is completed. Showing how characters are transferred between the processor and keyboard and display devices. Interrupt-driven I/O allows the processor to do other tasks while waiting for the I/O device to send an interrupt. Interrupt-driven I/O is more efficient than Programmed I/O.

Self Assessment Questions

1. External devices can be broadly classified into _____ categories.
2. _____ is a device that allows a computer to exchange data with remote devices.
3. The data rate of a mouse is _____.
4. In _____ I/O, the I/O module doesn't interrupt the CPU.
5. In _____, devices and memory share an address space.

11.7 Terminal Questions

1. List the classification of external devices.
2. Explain I/O module in detail.
3. Explain the concept of programmed I/O.
4. Discuss the working principle of an interrupt.
5. Explain how CPU responses to an interrupt.

11.8 Answers

Self Assessment Questions

1. two
2. modem or NIC
3. 100 bytes / sec
4. programmed
5. memory mapped I/O

Terminal Questions

1. Refer Section 11.2
2. Refer Section 11.3
3. Refer Section 11.4
4. Refer Section 11.5
5. Refer Section 11.5