Input-output

0.1 Introduction to IO

input output (IO) of computer makes it possible to have interaction with the outside world. IO are the devices outside the CPU and memory, to which CPU and memory communicate to get the information, data, and programs into computer as well as send these to outside world.

Various IO devices are - Keyboard, mouse, monitor(computer display), hard-disk, CD ROM, pen-drive, printer, network card, etc.

Due to the large variation in their speeds, IO devices cannot be connected to the bus directly. Typical speeds of these devices are are as follows: keyboard 5-10 characters per sec., printer 100 - 1000 characters per sec., hard disk - millions of characters (bytes per sec.), etc. If the CPU were designed to interact with the devices directly through the bus, its complexity would increases excessively.

Different IO devices also store the data in different formats. Consequently, they are connected to the bus via *IO controllers* (*IO modules*). One side of these controllers is interfaced to the bus and other to the device.

0.2 Simple Synchronous and Asynchronous I/O

The *Asynchronous* or *non-blocking* I/O is a form of I/O processing that permits other processing to continue before the transmission has finished.

I/O operations on a computer can be extremely slow compared to the processing of data. An I/O device can incorporate mechanical devices that must physically move, such as a hard drive seeking a track to read or write; this is often far slower than the switching of electric current in the CPU or memory. For example, during a hard-disk operation, that takes ten milliseconds to read/write, a processor that is clocked at 1 GHz could have performed ten million instruction-processing cycles. Thus an IO may communicate through *polling*, where the status of an IO is continuously monitored, whether it is ready for data transfer or not, or the IO device will *interrupt* the CPU when it requires the data transfer to take place, the process is called *interrupt driven* IO.

A simpler approach to I/O would be to start the IO and then wait for it to complete, called *synchronous or blocking I/O*). It would block the execution of instructions while the communication (i.e., data transfer) is in progress, leaving system resources idle. For example in the case of Direct memory Access*DMA*.

0.2.1 Asynchronous data transfer

If registers in the interface of two communicating devices share the common clock with CPU, the traffic between the two is *synchronous*. The *Asynchronous* data transfer takes place between two independent units, without sharing of a common clock. One way of achieving this is to indicate the intention of data transfer by means of *strobe pulse*. In figure 1 the CPU is *data source* and initiator of the strobe. In figure 2, the CPU is *data destination*, it initiates strobe and memory releases data. In both cases, source unit has no way to know that destination has received the data, similarly, the destination has no way to know that source has sent the data.

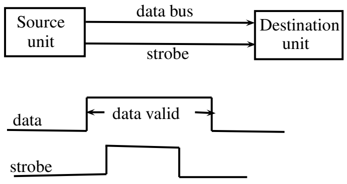

Figure 1: Timing diagram(TD) for memory write (source initiated strobe.)

0.3 Asynchronous data transfer with handshake

The data transfer between an interface and I/O device is commonly controlled by a set of handshaking lines. Hand shaking scheme provides a

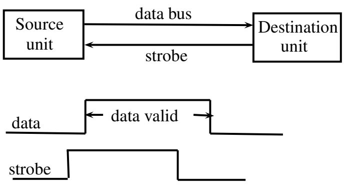

Figure 2: TD for memory read (destination initiated strobe.)

high degree of flexibility and reliability (the successful data transfer relies on the active participation of both parties). If one unit is faulty, data transfer cannot be initiated and completed. Such error can be detected by time out mechanism. The figure 3 demonstrates the source initiated data transfer.

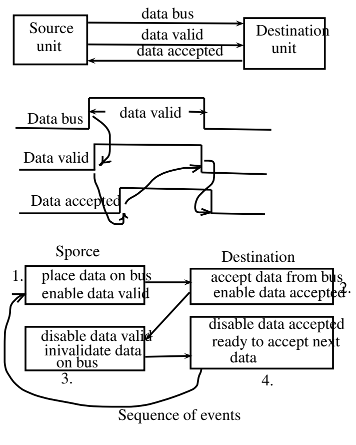
Source unit
data bus
data valid
data accepted
Destination unit
Data bus
data valid
Data valid
Data accepted
Sporce
Destination
1. place data on bus enable data valid
accept data from bus enable data accepted 2.
disable data valid inivalidate data on bus 3.
disable data accepted ready to accept next data 4.
Sequence of events

Figure 3: Source initiated transfer with handshake.

similarly figure 4 demonstrates the destination initiated data transfer.

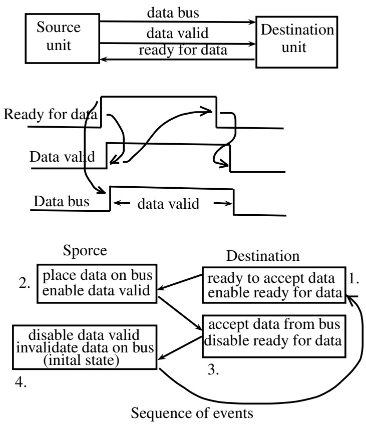

Figure 4: Destination initiated transfer with handshake.

0.4 I/O Controllers

Devices are connected to the bus through I/O controller, (also called I/O Module), as shown in the figure 5. One side of the IO controller is bus interface, while the other is connected to the device. It is necessary that both the sides of controller should be compatible to respective media as well signaling.

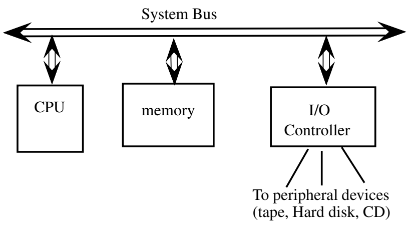

Figure 5: Connection of I/O devices and I/O controller.

The figure 6 shows a general IO device interface of the controller, and on the other side the bus interface. The Functions performed by an I/O controller are: Control and timing of data, read/write, Communication with processor and devices, Data buffering (to handle speed mismatch), and Error detection.

The data register holds the data-word sent/received by the bus, the status register tells about which devices are ready/busy. The data reg-

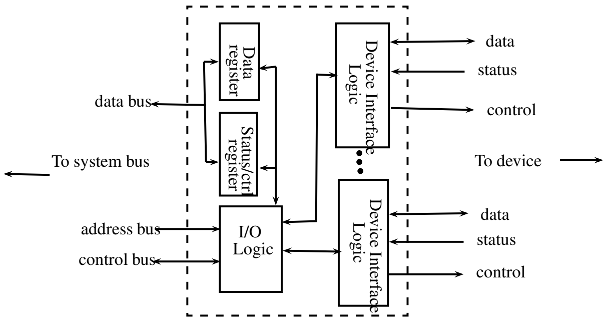

Figure 6: A general I/O device Interface.

ister acts as buffer to handle speed mismatch between the bus and IO device. The device interface provides the signal conditioning suitable to the device requirements, as well programs the device interface as input or output or input-output together.

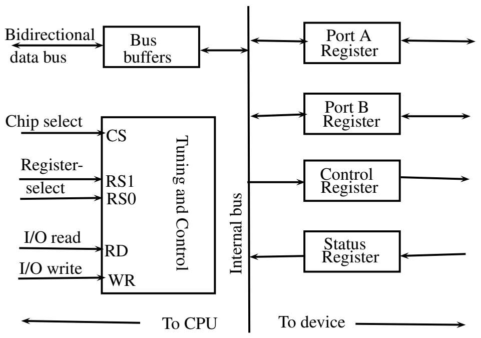

Figure 7: Typical IO device Interface.

The figure 7 shows a typical I/O Interface unit. This interface can be connected and can control and communicate with two IO devices, which can be connected to IO port register A and B. The control register controls the communication with the devices, and status register keeps the status of IO devices, whether they are busy, idle, or error, etc. The CPU can read the status register, as if it is a memory location, and can know the status of respective IO devices. To program the IO ports A and B, CPU write (program) the control register accordingly. Communication to port A, B, status, and control register is done by CPU by reading /

writing to these locations.

The table 1 demonstrates the configuration of an IO device.

Table 1: Configuring an IO device.

CS	RS1	RS0	Register selected
0	x	x	None: data bus in high impedance
1	0	0	Port A register
1	0	1	Port B register
1	1	0	Control register
1	1	1	Status register

0.4.1 Functions of I/O controller

Control and timing of data R/W coordinate the flow of traffic between internal resources (memory and CPU) v/s external devices.

The processor interrogates the I/O controller to check the status of the attached device. The I/O controller returns the device status. If device is ready to transmit, processor requests transfer of data by means of a command to I/O controller. The I/O controller obtains a unit of data (e.g., 8 or 16 bits) from the external device. The data are transferred from I/O controller to the processor.

The Processor communication involves *Command decoding*, i.e, I/O controller (say for disk) accepts commands from processor: READ SECTOR, WRITE SECTOR, SEEK track number, etc. Data are exchanged between processor and I/O controller.

The I/O controller signals are *Status reporting*, i.e., BUSY or READY; *Address recognition*, i.e. I/O controller must recognize address of each peripheral.

The controller provides *Data buffering*, i.e., handles speed mismatch between CPU and the IO device, hence buffering makes possible the communication at the speed of CPU, memory, and device. The I/O controller is responsible for *error detection and reporting to processor*, like, paper jam, bad disk track, and changes to the bit pattern (parity check).

IO Channel/Processor: It is an I/O controller taking most of the detailed processing burden, presenting a high-level interface to the processor; it is used in mainframes. The various I/O Operations performed are: Control, test, read, write.

The *device interface* control provide communication between controller and device. The environment is magnetic surface in the case of hard disk, optical surface in case of CD-ROM. The figure 8 shows the IO device with interfaces. One side is connected to the IO-controller discussed above, and other side is interfaced to medium of data, like magnetic track in hard-disk and magnetic tape, and printer head in case of printer as the device.

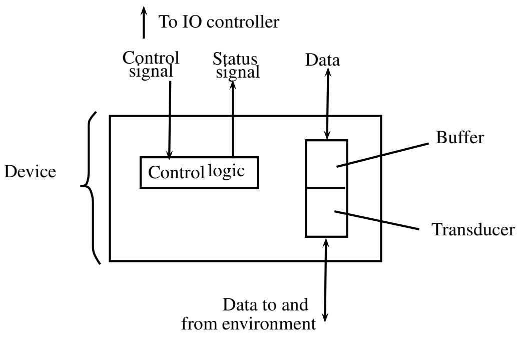

Figure 8: I/O device with Interfaces.

0.5 Memory mapped v/s Isolated IO

This classification of IO is not related to type of communication, but whether a IO can be treated as a memory location, or it is isolated from memory. Accordingly, an IO is classified as memory mapped and Isolated IO. The figure 9 shows the memory locations used as registers by DOS for memory-mapped IO.

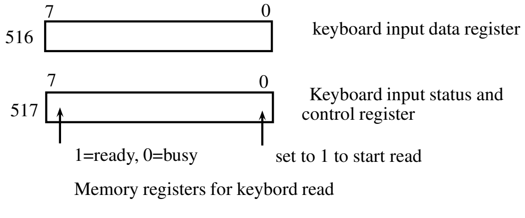
7
0
516
keyboard input data register
7
0
517
Keyboard input status and control register
1=ready, 0=busy
set to 1 to start read
Memory registers for keybord read

Figure 9: Register for Memory-mapped I/O.

The following program for DOS polls the device keyboard, whether it is ready or not to read. CPU repeatedly tests the status of I/O device, due to which it wastes lot of time for testing of availability of IO and much less for real work of performing the IO. However, the advantage is that its architecture is very simple.following are the code for memory mapped and Isolated IO, used by DOS (disk operating system).

```
Example 1: busy waiting then input for memory mapped IO
Address   Instructions      Oprand Comment
    200   Load  AC            "1"  Load accumulator with 1.
          Store AC            517  Store accumulator at 517
                                   to initiate keyboard read
    202   Load  AC            517  Get status byte of keyboard
          Brach if sign=0     202  loop until ready
          Load AC             516  load data byte

Example 2: busy waiting then input for Isolated IO

    200  Load IO               5   Initiate keyboard read
    201  Test I/O              5   Chekc for completion
         Branch if not ready  201
         IN                    5
```

In Memory Mapped I/O, the IO device is treated as memory a location.

The I/O devices are connected through I/O module or I/O controller to bus. A controller has number of ports, and each I/O port has an address. If that is address of device, independent of memory, it is *Isolated I/O.* Alternatively, address space of I/O can be treated as memory locations. Thus, address space of memory gets reduced by the magnitude I/O addresses counts. This is *Memory mapped I/O.*

In isolated I/O only "IN addr" or "OUT addr" instructions exist, where "addr" is address of the device. For 8-bit I/O address total 256 I/O addresses exists in 8085 micro-processor.

In memory mapped IO, all the memory reference instruction can be executed for IO ports, like, STA addr; LDA addr; MOV A, M; MOV B, M; MOV M, C; ADD M; ADI M; ORA M; ORI M; SUB M; SBI M for 8085 processor. Hence, memory mapped IO is more flexible to use compared to isolated IO. Now, at least in theory, entire memory can space can be used as IO address space.

0.6 Types of I/O

The types of IO (modes of transfer) are: *polled IO* or *programmed IO*, and *Interrupt driven IO.* We will take each type in detail.

0.6.1 Polled I/O

In this type, the CPU *polls* the device continuously with some interval, as whether the device wants to transfer the data or the device is ready to receive the data. If yes, the CPU transfers a byte to or receives a byte from it (see figure 10). Figure 11 shows the flow chart for write operation to the IO by polling it. For this the CPU reads the *status* bit, if it is set (indicating that device is ready), the byte is written by CPU to I/O, else it waits for some time and again tries. Though it is a simple approach, it is wasteful to CPU time as CPU needs to be busy most of the time testing the status of the device.

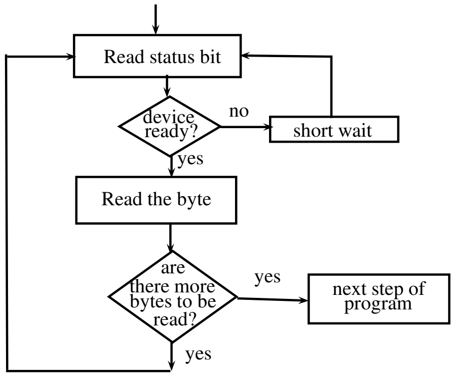

Figure 10: Read operation from device in polled I/O.

0.7 Interrupts driven I/O

The program controlled I/O degrades the system performance as the CPU gets tied down to the I/O. The interrupt mechanism greatly improves the performance of the CPU. In interrupt driven IO, timing of I/O is controlled by the device, and the CPU remains occupied in its own job for rest of the time. An IO device interrupts the CPU when IO is required. On this CPU saves its status, including the PC, and control is transferred to an ISR (interrupt service routine), which performs I/O. On return from

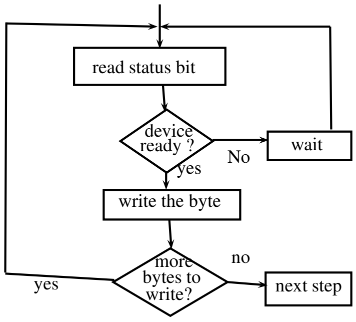

Figure 11: Write operation from device in polled I/O.

ISR, CPU resumes at its previous operation, from the next address where it suspended execution.

A CPU gets an interrupt when, for example:

- a character is entered on keyboard,
- monitor is ready for next refresh,
- a block transfer complete from memory to I/O or I/O to memory
- *hardware* interrupts are due to division by zero, or an attempt to execute a privileged instruction by user program.

Following are the sources of interrupts:

1. **Program:** The program generated interrupts are generated due to some conditions which occur as a result of executions of Instructions, like:

- arithmetic over flow
- divide by zero
- attempt to execute illegal machine instruction
- reference to outside user's allowed memory space

2. **Timer Interrupt:** These are generated by timer of processor; it allows to perform certain functions at regular intervals.

3. **IO:** An *IO* Interrupt is generated by IO controller to signal normal completion or start of an IO, or to send variety of error conditions.

4. **Hardware Failure:** These are due to power failure, or memory parity error.

Following are the advantages of interrupt mechanism:

- Improves processing efficiency due to slow IO and fast CPU,
- User program does not need any special code for interrupt, and
- Processor and operating system are responsible to suspend the program, and cause it to return back.

0.7.1 Interrupt Processing

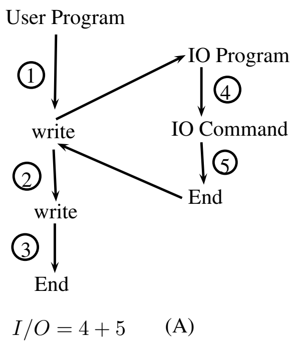

Figure 12: Steps for processing an interrupt.

The figure 12 shows the interrupt processing sequence. User program waits till the interrupt is processed and control returns back to it. The state 4 is "save context", 5 "Interrupt handler + context restore". The Control returns back immediately to user program.

The figure 13 shows another example of some interrupt. Here, the user program is called on the completion of processing of interrupt handler.

0.7.2 Handling Multiple Interrupts

Program may be receiving data from communication line, and send them to printer. So communication line will cause interrupt, as well as the printer. There are two approaches to handle multiple interrupts: 1) Disable interrupts, 2) Priority interrupts. Figure 14 shows the priority interrupts.

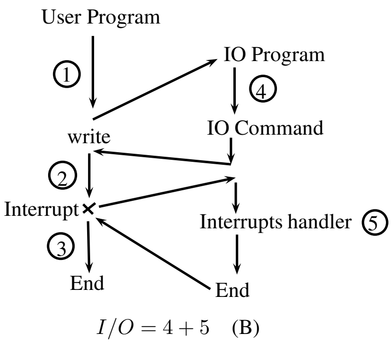

Figure 13: An interrupt initiates an operation, and again interrupted on completion of the event.

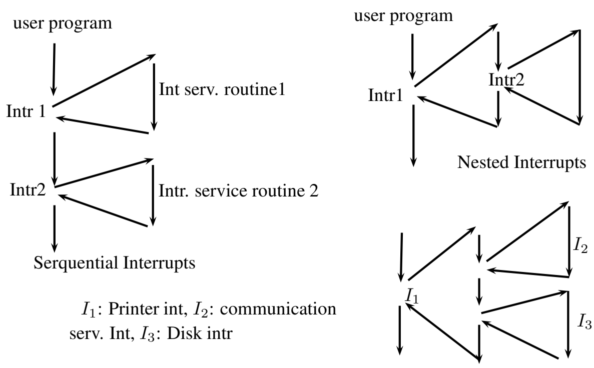

Figure 14: Priority Interrupts.

The working of Interrupt is as follows. If an Interrupt request pending is found true when it is tested by the CPU after execution of an instruction, it is first serviced, and CPU resumes normal execution for next instruction, else CPU continues with the next instruction. The figure 15 shows the interrupt servicing flow-chart.

The Steps for recognition and servicing of an interrupt as as follows:

i. CPU identifies the source of interrupt (may require polling of I/O device),

ii. CPU obtains address of ISR. (may be supplied by device along with Interrupt request),

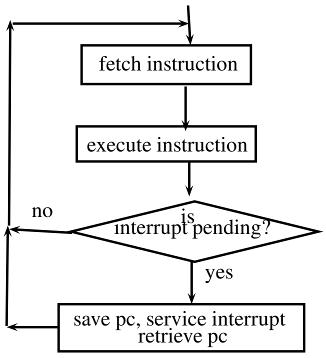

Figure 15: Interrupt servicing flow-chart.

iii. PC, CPU status saved,

iv. PC loaded with ISR address.

Usually, the ISR has a DI (disable Interrupt) instruction at its beginning and EI (Enable interrupt) at its end, followed by return instruction.

```
ISR: DI
     high priority ISR
     EI
     RET
```

In an interrupt system, one of the requirements is, when number of interrupts come, then to decide as which interrupt should be entertained. This solved by assigning priorities to the interrupts, for which the mechanism very. The other is to find out which device has interrupted. Having too many interrupt lines connection with the processor limit the maximum connections, due to the space available. Multiple interrupt lines also makes it difficult to find the source, as for each the status register need to be tested. In the case of vectored interrupts, the interrupting device supplies the interrupt vector, i.e, address for branching to a subroutine. However, which interrupt should be given the priority varies. It is by polling, using either daisy chaining or hardware polling. The order in which the polling takes place decides the priority of the interrupts. The priority may be fixed or may be rolling to give uniform treatment to the interrupting devices. Figure 16 shows the multi-line interrupts, i.e., each is connected to the CPU through a different line.

In a *single-line interrupt system* (figure 17) all the interrupts are ORed together. An interrupt generated from any device will set the interrupt flag true in the CPU. On knowing this, the CPU determines source of interrupt. The priority of the interrupting device is programmable.

0.8 Vectored Interrupts

In *Software poll*, on detecting interrupt, the processor branches to a service routine, which polls IO modules to determine who has interrupted, and then serves ISR of that. The disadvantage is time wasted in polling.

One Process of of is *Daisy Chaining*, where *hardware polled.* All the IO modules share a common interrupt line and Interrupt acknowledgement is daisy chained through these modules. The requesting module places a word on the data bus (address of IO module - called vector). And this vector is used a pointer to ISR (called vectored interrupt). The figure 18 shows the daisy-chained vectored interrupt.

The device sets INT=1 when it wants to cause the interrupt. Interrupt vector is an 8-bit signal for device to identify itself, which is used as an entry into a interrupt vector table to get the starting address of the ISR (Interrupt service routine).

The figure 19 shows a generalized vectored interrupt system with k number of devices interrupting the CPU. The interrupt controller received the interrupt, which sends the interrupt and interrupt to the CPU.

Most flexible and fastest response to interrupt is possible when interrupt request causes direct hardware implemented transition to current interrupt handling program. This requires that interrupting device should supply the staring address or the *transfer vector* of that program. The technique is called vectoring.

Figure 20 shows a variant of a vectored interrupt, where interrupt vector is supplied by the device itself via the data bus. Each I/O port may request the services of many different programs, and invoking these programs will require their starting address. The address supplied on data bus modifies PC, hence transferring control to that routine. Hence, this method takes control of data bus temporarily. An alternative way

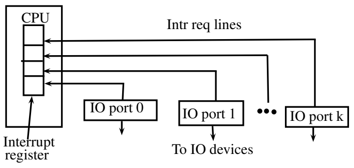

Figure 16: Multi-line interrupts.

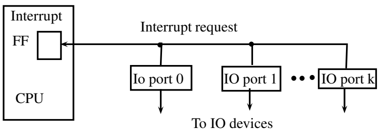

Figure 17: Single-line interrupt

.

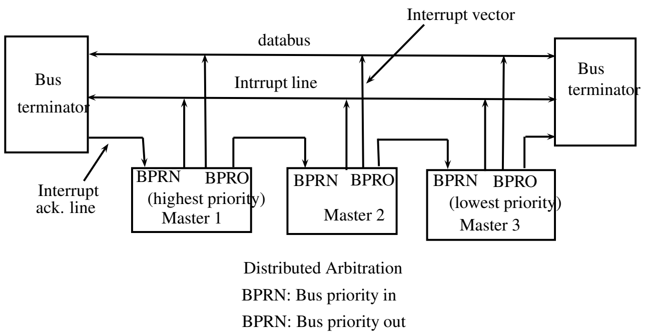

Figure 18: Daisy chained Vectored interrupt.

is to send the instruction *call x* to CPU for execution, which calls the subroutine to serve the interrupt.

0.8.1 Maskable Vectored Interrupt

Another approach for vectored interrupt is vectored *interrupt with masking*. The k masked interrupt signals are fed into a *priority encoder* that produces a $\lceil \log_2 k \rceil$-bit address, which is then inserted into program counter as a sub-field, see figure 21, to create an interrupt vector. When interrupt request from port i is received, priority encoder generates a 2-bit address, which is inserted into PC. Rest of the bits of PC are zeroed. Thus, 2 bit will generate addresses: 0-3. This is multiplied by 4 to get (0, 4, 8, 12) as the position of interrupt vectors. Fig. 22 shows that first four locations (words, each 32 bits) are assigned to interrupt vectors.

0.9 IO controller

The Intel 8255 (figure 23) is called IO controller/IO module. There are three IO ports: A, B, C. The port C can be used as two 4-bit ports

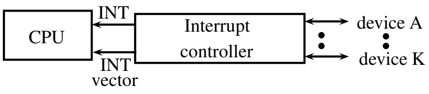

Figure 19: Vectored interrupt.

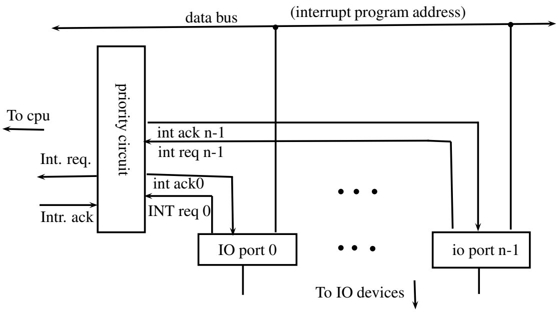

Figure 20: An implementation for vectored interrupt

or one 8-bit ports. A, B are always 8-bit ports. All these ports can be programmed as Input, output, bidirectional in polled mode or as interrupt mode. Programming is done by writing a 8-bit control word at port address 11_2. In polled mode certain lines can be used for handshaking.

0.9.1 IO controller-8255

Following tables show the direction of ports:

A_0	A_1	$\overline{RD}$	$\overline{WR}$	$\overline{CS}$	*Direction*
0	0	0	1	0	$A \rightarrow databus$
0	1	0	1	0	$B \rightarrow databus$
1	0	0	1	0	$C \rightarrow databus$
0	0	1	0	0	$databus \rightarrow A$
0	1	1	0	0	$databus \rightarrow B$
1	0	1	0	0	$databus \rightarrow C$
1	1	1	0	0	$databus \rightarrow control$

Three are 3-modes of operation for PPI 8255. Mode 0 is for *Basic input/output*, mode 1 for *strobed I/O*, and mode 2 for *bidirectional bus.*

Various bits of control word are: $D_7 = 1 \Rightarrow$ mode set flag, $1 =$ active. $D_6D_5 = 00 \Rightarrow mode\ 0$, $01 \Rightarrow mode\ 1$, $1x \Rightarrow mode\ 2$, $D_4 = 1 \Rightarrow$

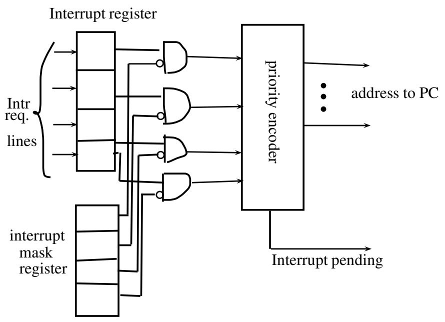

Figure 21: Interrupt vectoring with masking of interrupts.

port A I/P, 0: O/P, D_3 for port C upper, D_2 for mode 0 & 1, D_1 for port B, D_0 for port C lower. Any bit can be set or reset.

0.9.2 Interrupt Controller

8255 chip is used as interrupt controller (see figure 24). The port C is used as interrupt port for strobing. Port A is used as data port for data input or output. Interrupt signal is stored in CPU register which is periodically tested by the CPU. Usually, the interrupts are assigned priorities based on the priorities of IO devices or services they are performing.

0.10 Direct Memory Access

In Direct Memory Access, CPU relinquishes the bus, and get itself isolated. The IO takes place between memory and IO device at clock speed. When I/O is complete, DMA controller removes the bus request line, CPU takes over the bus, and processing resumes at the point is was left. IO transfers are limited by the speed by which the CPU can test and service IO. The testing IO status and executing IO commands can be better used for processing tasks. DMA request by IO device is for demand of BUS and not CPU (in interrupt it is reverse). The DMA request can be granted BUS at the end of any *CPU Cycles*.

DMA v/s Interrupts: DMA break points are after each of following to opcode fetch, decode opcode, fetch operand (if any), execute instruction. However, the interrupt break point is after instruction is executed, and not in between (see figure 25).

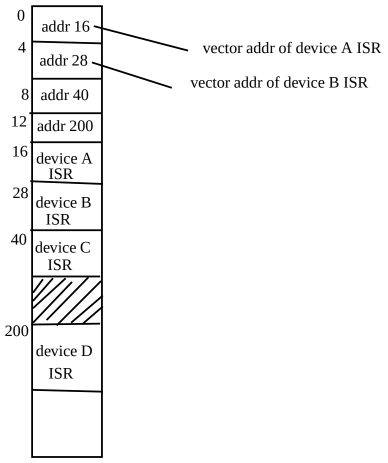

Figure 22: Interrupt vectors are stored in memory

The function blocks of DMA are shown in figure 26. Functional blocks: DC (data count) keeps initial count of number of words /bytes to be transferred, IOAR is memory address from that location onward data is to be transferred to/from memory, IODR is IO data reg., for holding word/byte while it is being transferred to IO/RAM.

DMA is used for bulk data transfer between memory and IO. The CPU relinquishes the control of the bus, and surrenders it to DMA for doing this data transfer. The transfer is initiated by CPU, and DMA controller interrupts the CPU to indicate that DMA is over. During the DMA transfer, the DMA is bus master.

DMA steps:

1. CPU executes two IO instructions which loads IOAR and DC. (IOAR is base address of main memory, DC is words count to be transferred)

2. DMA controller gives bus request when ready to transfer the data. On this DMA acknowledges (grants the bus to DMA controller) (bus priority control is used when requests are too many).

3. DMA controller transfers the data.

4. If IO device is not ready but $DC > 0$, the DMA controller deactivates the DMA request. On this CPU acknowledges bus grant low, and resumes normal operation.

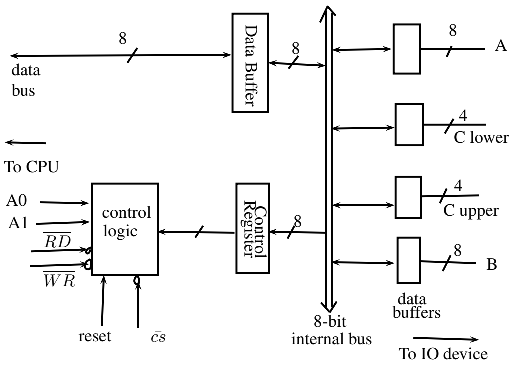

Figure 23: IO controller (intel 8255)/ PPI-programmable peripheral Interface.

Classification of DMAs The DMA types are -

1. Block transfer DMA
2. Cycle stealing DMA (bus cycles are stolen by DMA, during the time cpu is inactive, to carry out DMA)

In a block Transfer DMA Data of arbitrary length are transferred in a single continuous burst. DMA controller is bus master. It is used when secondary memory devices are mag. disk, and cannot be stopped or slowed down without loss of data. Supports maximum transfer rate. CPU has to remain inactive for long period.

Cycle stealing DMA

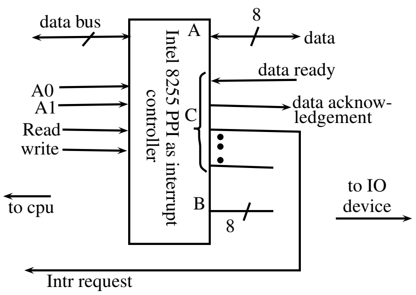

Figure 24: Programmable peripheral Interface as Interrupt Controller.

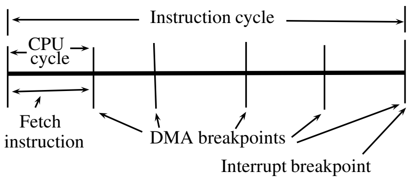

Figure 25: DMA and Instruction breakpoints during an instruction cycle.

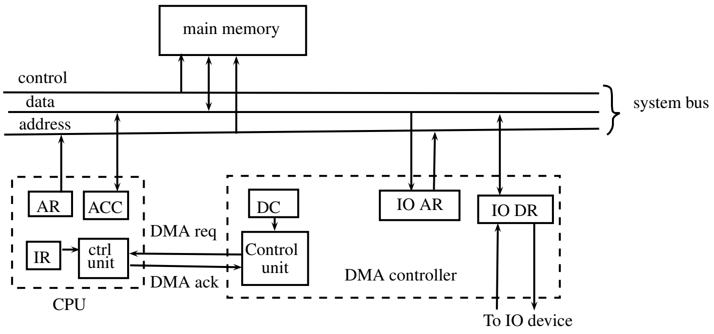

Figure 26: DMA controller block diagram.

It allows DMA controller to use system bus to transfer one or several words/bytes, and returns control back to CPU. The long strings of data are split. It reduces the DMA speed but also causes the interference due to DMA to CPU.

Exercises

1. Indicate whether the following constitutes a control, status, or data transfer commands:

   (a) Skip next instruction if flag is set

   (b) seek a given record from a magnetic disk

   (c) check if IO device is ready

   (d) Move printer paper to beginning of next page

   (e) Read interface status register

2. Why does the DMA has priority over CPU when both request a memory transfer?

3. How many 8-bit characters can be transmitted per second over a 9600 baud serial communication link using asynchronous mode of transmission with one start bit, eight data bits, two stop bits, and one parity bit?
   (a) 600 (b) *800* (c) 876 (d) 1200

4. A DMA module is transferring characters to memory using cycle stealing, from the device transmitting at 14400 bps. The processor is fetching instructions at the rate of 1 million instructions per seconds (1 MIPS). By how much magnitude the processor will be slowed down due to the DMA activity?

5. Consider the system in which bus cycles takes 500 nsec. Transfer of bus control in either direction from processor to I/O device or vice-versa, takes 250 nsec. One of the I/O devices has a data transfer rate of 75 KB/s and employs DMA. Data are transferred one byte at a time.

   (a) Suppose we employ DMA in a burst mode. That is, the DMA interface gains bus mastership prior to the start of a block transfer and maintains control of the bus until the whole block is transferred. How long would the device tie up the bus when transferred a block of 256 bytes?

   (b) Repeat the above for cycle stealing mode.

6. An asynchronous link between two computers uses the start-stop scheme, with one start bit and one stop bit, and transmission rate of 38.8 kilobits per sec. What is the effective transmission rate as seen by the two computers?

7. A DMA controller serves four receive only telecommunications links (one DMA per channel) having speed of 64 kbps each.

   (a) Would you operate the controller in burst-mode or cycle stealing mode?

   (b) What priority scheme would you employ for service of the DMA channel?

8. A processor and I/O device D are connected to main memory M via a shared bus having width of one word. CPU can execute 10^6 instructions per sec. An average instruction requires five machine cycles, three of which use the memory bus. A memory read or write operation uses one machine cycle. Suppose that processor is

continuously executing background program that requires 95% of its instruction execution rate but not any I/O instructions. Assume that one processor cycle equals one bus cycle. Now suppose that the I/O device is to be used to transfer very large blocks data between M and D.

(a) If programmed I/O is used and each one-word I/O transfer requires the processor to execute two instructions, estimate the maximum I/O data-transfer rate, in words per sec., possible through D.

(b) Estimate the same rate if DMA is used.

9. A typical CPU allows most interrupts to be enabled and disabled under software control. In contrast, no CPU provides facilitates to disable DMA request signals. Explain why it is so?

Bibliography

[1] John P. Hayes, "Computer Architecture and Organization", 2nd Edition, McGraw-Hill, 1988.

[2] William Stalling, "Computer Organization and Architecture", 8th Edition, Pearson, 2010.

[3] M. Morris Mano, “Computer System Architecture-3rd Edition”, Pearson, 2006 (chapter 11).

[4] http://krchowdhary.com/co/co.html