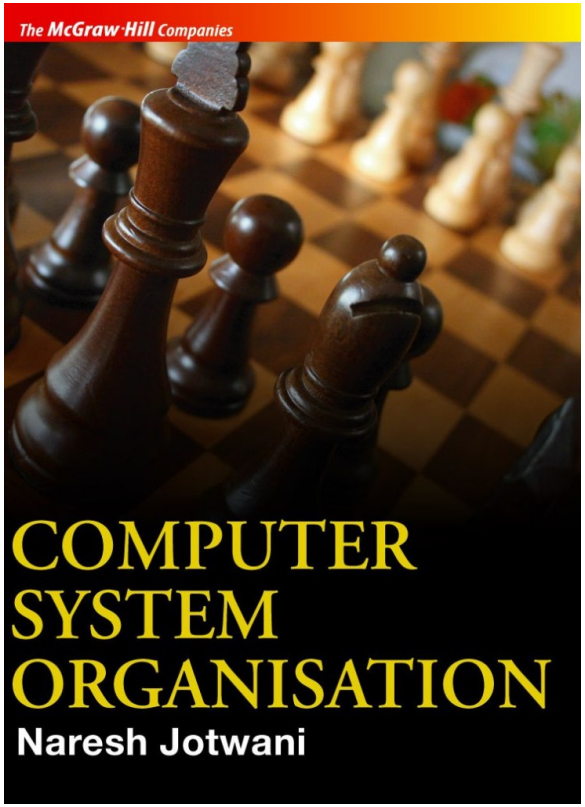
Mc Graw Hill Education

COMPUTER SYSTEM ORGANISATION

Naresh Jotwani

PowerPoint Slides



CHAPTER 10

INPUT AND OUTPUT ORGANISATION

Introduction

- I/O is an essential & integral part of computer system. Variety of I/O devices on a computer system - keyboard, mouse, display, magnetic disk, network adaptor, etc.
- I/O devices differ widely in terms of function, size, mode of operation, transfer speed, power consumption, etc.
- All devices must be connected to the processor and memory using the same basic architecture.
- Over the years, different mechanisms have been developed to connect I/O devices to systems, and to program I/O data transfers over the resulting connections.

I/O devices and controllers:

- An I/O device is connected to the computer system by using a *device controller*. On one side of the controller are the address, data, and control lines of the processor; on its other side is the device itself.

- I/O devices vary in of some or all of the following characteristics:
  - Representation of data: voltage, current, magnetic field, etc.
  - Speed of operation and data transfer
  - Timing and control requirements
  - Need to detect physical events – e.g. mouse clicks or key presses
  - Need for error detection and correction

- One device may need a fairly simple controller, while another may require a sophisticated one - e.g. mouse vs. magnetic disk.

- Depending on data transfer speed, usually a *data buffer* of appropriate size is also provided on the device controller.

- On the processor side, a device controller is connected to address, data, and control lines of the external bus.

- *A device controller is the device as seen by the programmer*. A running program performs input or output on a device only through the agency of the controller.

- An I/O operation can get quite complex, since timing constraints must also be taken into account.

Accessing devices

- IN and OUT instructions are used, respectively, to *read data in* from an input device, or to *write data out* to an output device.

- At the other end of the data transfer – in each case – is a processor register. The *I/O address* specified with the IN and OUT instruction selects a device.

- But we cannot assume that the device is always *ready* for the input or output operation. A program must test whether a device is *ready* for input or output, before performing the required operation.

- The next program reads in 100 bytes from an input device and stores them in successive memory locations. IN instruction is executed for data only after testing that the input device is *ready* for input.

```
//
// Program to read 100 bytes from an input device and
// store them in successive memory locations starting at
// location BASE.
//
      LOAD        #0, R0      // Initialize counter R0 to 0
//
LOOP: IN          11, R1      // Read device status from address 11
      AND         #1, R1      // Extract rightmost bit
      JZ          LOOP        // Loop if device not ready
//
      IN          10, R1      // Read data byte from device at
                              // address 10
      STOREB      BASE[R0], R1
                              // Store byte in memory location
      INC         R0          // Increment counter R0
      COMP        R0, #100    // Compare contents of R0 to 100
      JEQ         DONE        // If equal, jump to DONE
      JMP         LOOP        // Unconditional jump back to
                              // instruction with label LOOP
DONE: HALT                    // Halt the program
```

- The program tests for *device ready b*y checking the contents of the *device status register*.
  - Here the status register of the input device is available at address 11, and
  - the rightmost bit in device status register informs us whether the device is ready.

- When an input instruction IN specifies address 11, the processor places it on the address lines of the external bus.
  - Using the signal *memory-or-I/O*, the processor also indicates that an I/O operation is in progress, as opposed to a main memory operation.

- Several device controllers may be connected to the system at different I/O addresses.
  - On an I/O operation, address decoders on all the device controllers receive address bits in parallel.
  - In our program, *status register* of the input device is selected by the decoder output when address is 11. We can say that the status register is *located* at I/O address 11.
  - When the IN instruction specifies address 10, the processor puts out this address on the address lines. Then the *data register* of the device is selected, and we say that it is located at I/O address 10.

- Here the input device occupies two locations in the I/O address map of the processor – address 10 for its data register, and address 11 for its status register.

- This situation is depicted schematically in the next figure.

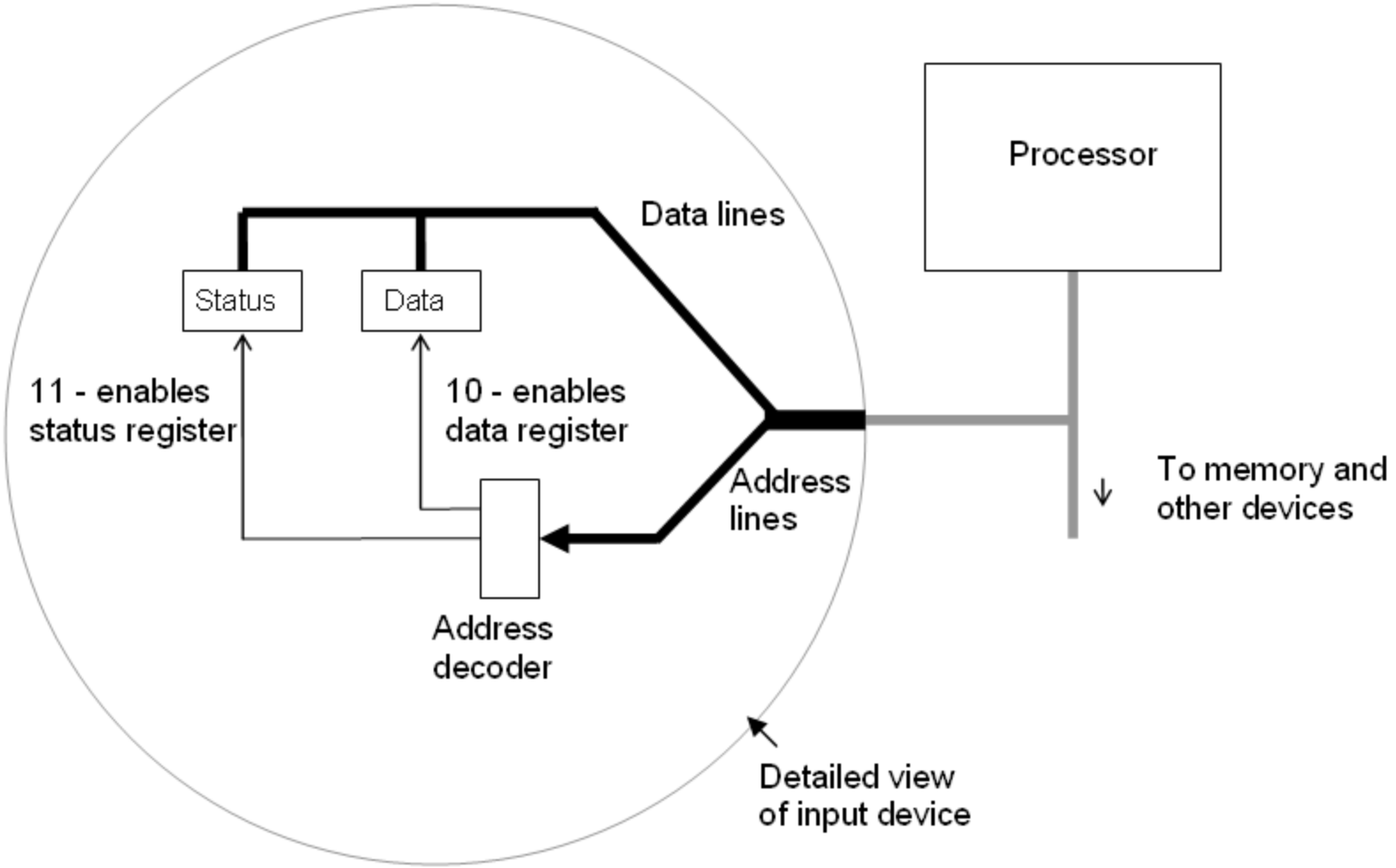
- Note that the physical input device – which may be a keypad or a mouse – is not seen in the figure.

- Instead, here the 'programmer's view' of the device is seen – which in this case consists only of:

  (i) the two registers at the indicated addresses, and

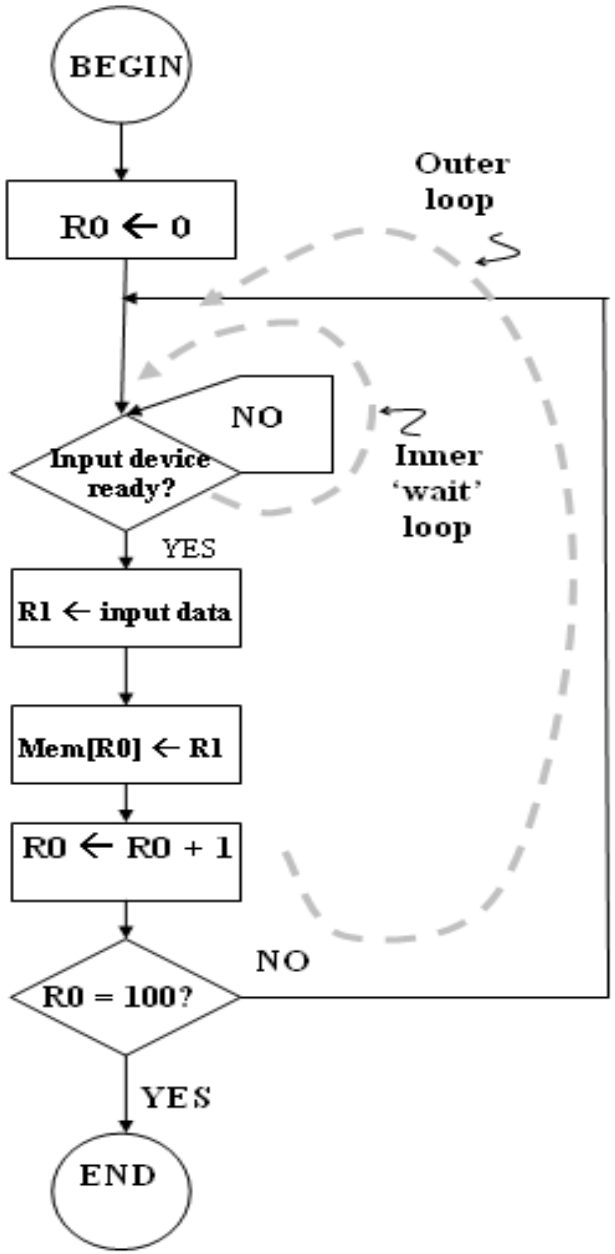
  (ii) the way in which these register contents are to be interpreted by the program.

Input device with data register and status register



- The flow chart in the next figure corresponds to the program seen earlier.
  - We see an *inner loop* within the outer *main loop* of the program.
  - The outer loop is executed once for every byte read in.
  - The inner loop has the function of forcing the processor to *do nothing* – i.e. simply *wait* – until the input device is ready for a byte to be read in.
- Therefore such a loop is usually called a *wait loop* or *idle loop*.

Flow chart of program to input data from device:



- The above program would not change much if we were writing out 100 bytes to an output device, rather than reading in 100 bytes.
  - Device would still have status and data registers, the outer loop and inner wait loop of the program would not change.
  - Within the outer loop, we would first load a memory byte into a register, and then send it out to the device using OUT instruction, with the appropriate address.
- But we also prefer that, as far as possible, the processor should not have to remain idle. Therefore a mechanism is needed for a device to inform the processor that it is *ready* for a data transfer or another operation.
- Such a mechanism - to avoid the use of wait loops - is provided on all processors, and it is known as the *interrupt mechanism*.

Device Status and Control

- Suppose the keyboard device is *ready* with the next byte of data. After testing for *device status*, the program reads in the next byte. *Device ready* bit should then be reset by the controller – so that the cycle can be repeated for the next byte transfer from the keyboard

- Thus the *ready* bit is a part of *device status*.
  - In the keyboard controller, when the user presses a key, the *ready* bit is set. When the program reads in the corresponding byte, the *ready* bit is reset.

- Generally, *device status* will also include information about errors or fault conditions which might have occurred in device operation.

- Example: When a user tries to write to a write-protected storage device, the controller must record this as an error condition, which must be reported to the user.

- In addition to *device status*, many I/O devices have functions, modes of operation, or parameters which need to be selected – i.e. 'set' – by the programmer.

- This is done by sending relevant parameters or commands to the *device control register(s)* in the controller, using OUT instructions.

- Thus, depending on its functions and complexity, a device controller would have some or all of the following registers and data buffers associated with it:
  - Input data register or buffer,
  - Output data register or buffer,
  - Status register(s), and
  - Control register(s).

- These registers and buffers take up an appropriate number of I/O addresses.

Interrupt Mechanism and Handling

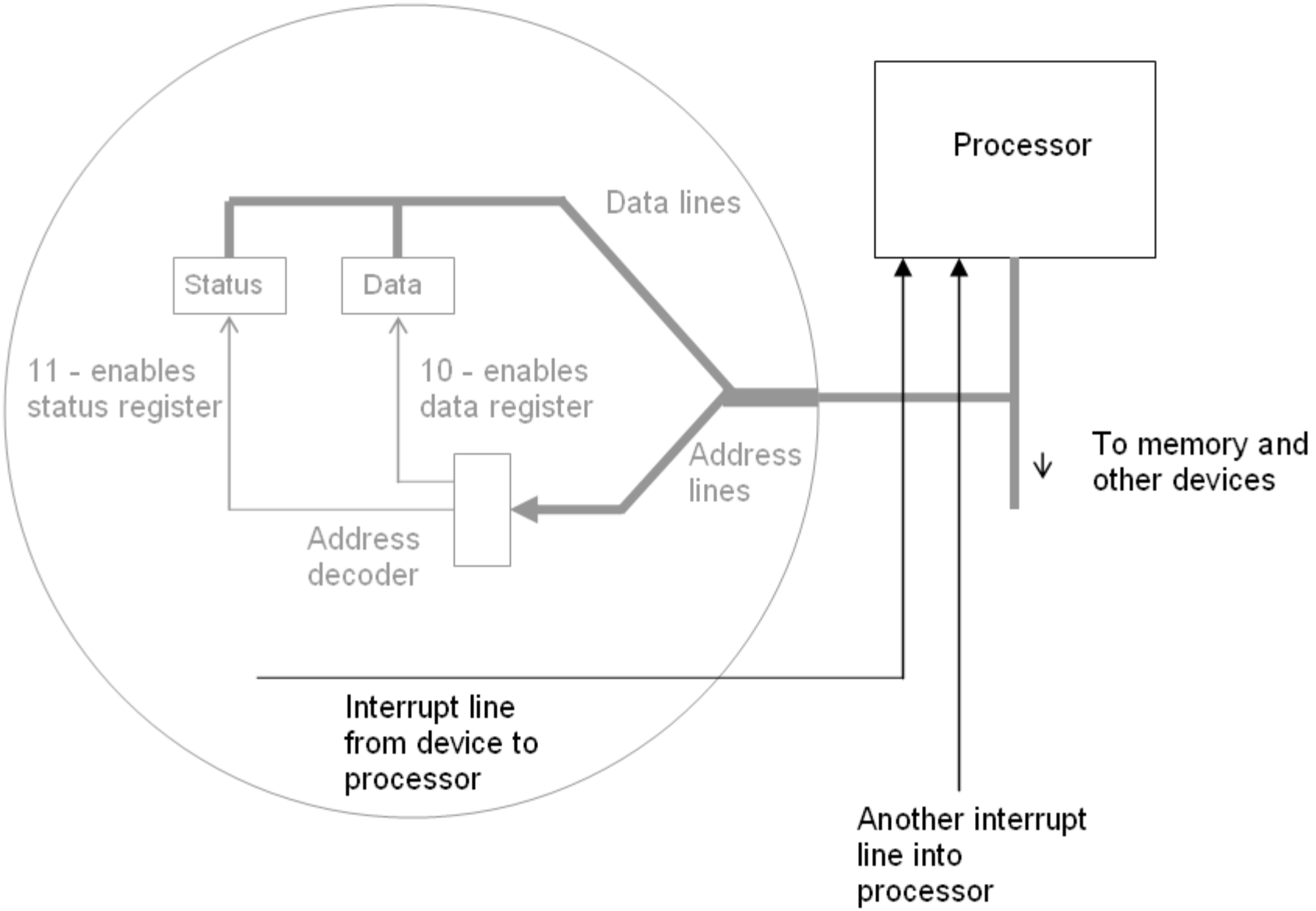
- A processor in a *wait loop* is in effect doing nothing. To avoid wasting processor cycles, a mechanism is required for an I/O device to *signal* to the processor that it is *ready*, or that the previous operation is completed.

- Such a mechanism, provided on all processors, is known as the *interrupt mechanism*.

- An *interrupt* is a signal which a device controller sends to the processor.

- At a minimum, a single-bit input line from the device controller to the processor is needed, as depicted in the next figure.
  - Value ‘0’ on this line indicates that the device is not signaling an interrupt to the processor, while value ‘1’ indicates that it is signaling an interrupt.

Input device with interrupt signal:

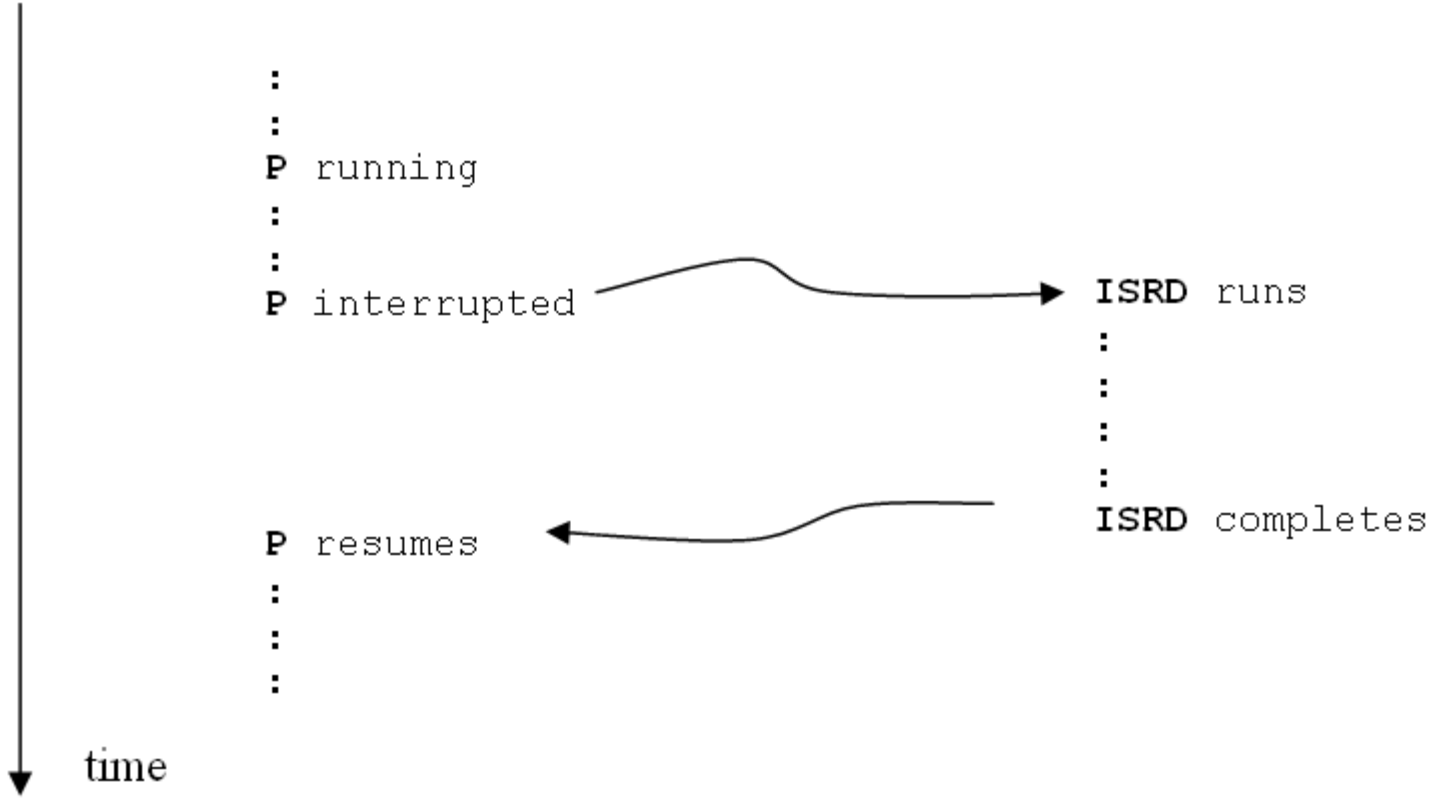


- What does the processor do if a device is signaling an interrupt?
  - Assume the processor is running program P when device D signals an interrupt. This means device D needs attention, and therefore the running of program P must be temporarily interrupted. It should be resumed after the interrupt from device D has been *serviced*.
- Another program in memory, say ISRD – *interrupt service routine* for device D – is provided to respond correctly to the interrupt by device D.
- Therefore the running of P should be interrupted in favour of ISRD, but P should resume after ISRD has done its work.
- This is indicated schematically in the next figure.

- Thus, an interrupt signal from an I/O device must have an associated *interrupt service routine* – or ISR – associated with it, which correctly responds to the interrupt.

- How does the processor know the memory address of the service routine of an interrupt?
  - Certain memory locations are pre-determined within the processor to store the required ISR addresses [often at the lower-most addresses in main memory].
  - On detecting an interrupt signal, the processor finds the address of the correct ISR from these locations.

Interrupt service routine invoked upon an interrupt:



- Note here that program P does not call ISRD.
- ISRD is *invoked* when device D signals an interrupt to the processor, and ISRD is the *service routine* for this interrupt.

- How does the processor detect that a device has signaled an interrupt?
  - Recall the instruction *fetch-execute* cycle which the processor performs repeatedly. The cycle is shown below, with a crucial extra test within the loop for any pending interrupt.

```
repeat forever
{
      test interrupt lines for any device interrupt
      if an interrupt is signaled
            save return address on stack and jump to ISR
      else
      {
            fetch the next instruction from main memory
            execute the instruction fetched
      }
}
```

- ISR is invoked upon the interrupt signal, and ends with RETURN. At this point, the saved return address is retrieved from the stack – and execution resumes exactly where it was interrupted.

- Processor may provide multiple interrupt signals for different device controllers. They are usually *prioritized* – i.e. a lower priority ISR can be interrupted by a higher priority one, but not vice versa.

- Interrupts can also be selectively *disabled* or *enabled* under program control.

- Within the processor, a register – often PSW – contains an *interrupt mask*, i.e bits which indicate the interrupts which are enabled at a given time.

- The ability to selectively disable/enable interrupts allows flexibility in deciding whether, how, and in what order the interrupting devices in a system will receive service.

- There are also techniques available for multiple devices to share a single interrupt line

  - On detecting an interrupt on a shared line, the corresponding ISR first *polls* the status registers of devices sharing the line – to determine which device had signaled the interrupt.
  - Then, the correct device is serviced by the ISR.

  - This technique is known as *polling*.

Other types of interrupts:

- Conditions within the processor – e.g. *divide by zero* or *illegal memory access* – generate an internal signal which is handled in the same way as an external interrupt.

- Thus interrupt signals can be classified into *external* interrupts and *internal* interrupts (or *traps)*.

- A third category are *software interrupts*, which allow a running program to invoke operating system functions.
  - A software interrupt is a machine instruction – with a mnemonic such as SWI – with a single operand which identifies the specific interrupt.
  - Service routine for a software interrupt causes the processor to switch from *user mode* to *system mode*, and to execute an operating system function.

Example: A simple application

- A guinea pig enclosure with four sets of lights in its four corners, with one green light and one red light in each set. Two large switches in the centre of the enclosure, each of which may easily be turned ON or OFF by the guinea pig using its paw.

- Lights are controlled in this way →

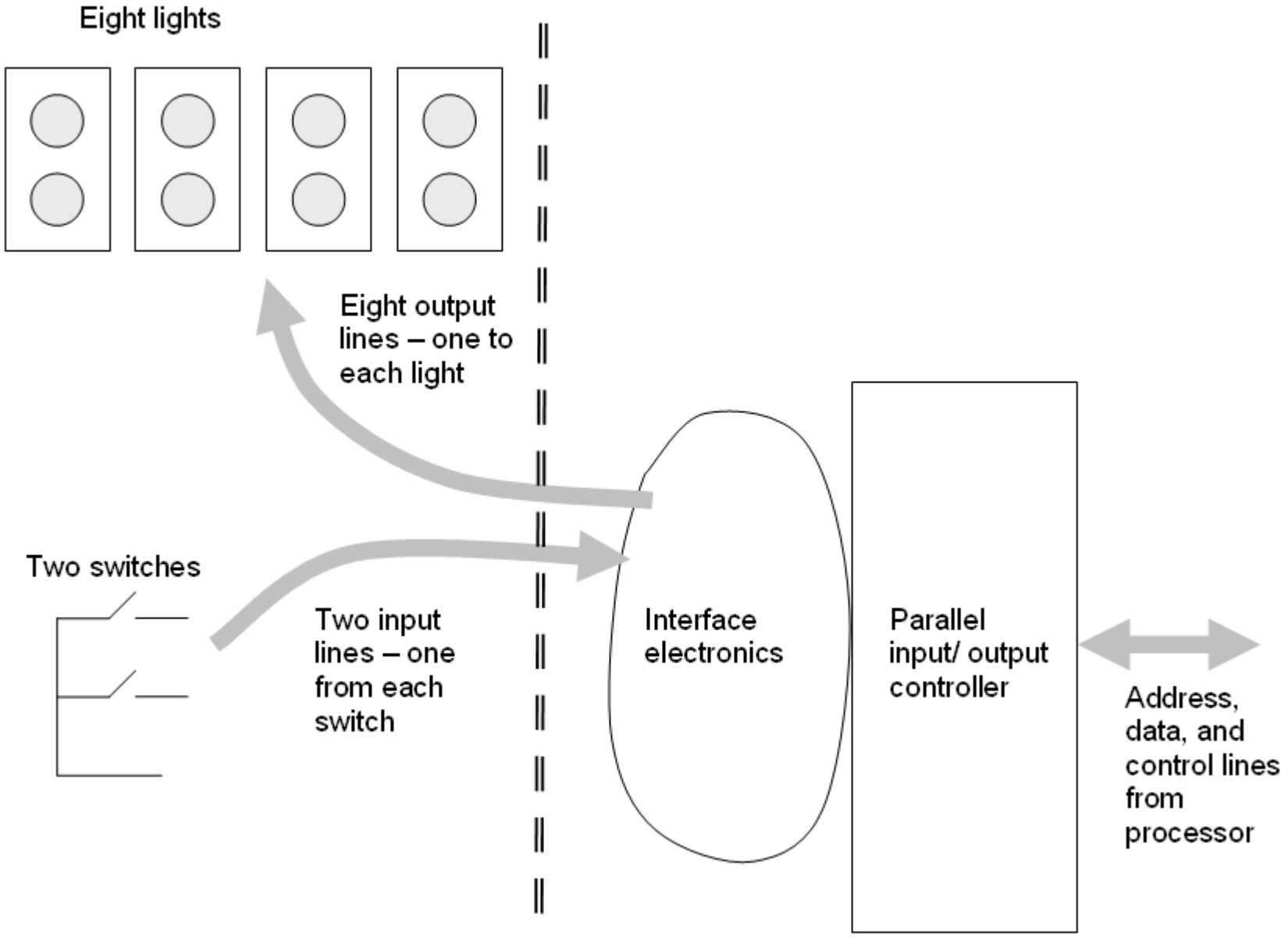
Switch ON	Light pattern
None	all sets of lights blink*
Any one	only two sets of lights blink*
Both	all lights turn off

- 'blink' means alternately turning the red and the green light ON for one second each, and repeating the process.

- Aim: To study if a guinea pig can associate specific switches with specific patterns of lights.

- The next figure shows a schematic diagram of this setup. Double dashed line separates the enclosure from the computer and the required *interface electronics*. The lines running from the switches to the interface electronics, and from the electronics to the lights, are shown using thick arrows

Experimental system of sample application



- An eight bit output data register in the device controller is provided to control each of the eight lights in the enclosure.
- By writing ‘0’ or ‘1’ in each of these eight bits, we can independently turn the respective light OFF or ON.
- We assume that the device controller has an eight bit wide input data register, even though only two of these eight bits are used, to read the status of the two switches.
- The ‘blink rate’ specifies that each light must be alternately turned ON and OFF once every second.
- Software running on the computer must sense the status of the switches once every second, and then turn the various lights ON or OFF as specified in the table.
- The required program in algorithmic notation is outlined in the next slide.

Program needed for sample application:

```
once every second, do
{
      for each of the four sets of lights, do
      {
            if the set is in blink mode,
                  the light which is ON is turned OFF,
                  and vice versa
      }
      read the status of the two switches
      if no switch is ON
            change all sets of lights to blink mode
      if any one switch is ON
            change two sets of lights to blink mode
      if both switches are ON
            turn off all lights
}
```

- The steps inside the outer loop involve *iteration*, *conditional execution* and the appropriate use of IN and OUT instructions.

- How do these actions take place once in every second?
  - By a timer-like device which signals an interrupt to the processor precisely once every second.
  - Some processors – including all microcontrollers – have *programmable timer-cum-counter* devices built into them.
  - Such a timer device can also be connected externally.

- With such a timer device, the actions within the outer loop make up the *service routine* for timer interrupts.

  - Once every second, the service routine is invoked and the required operations on switches and lights take place.

- Without a timer device the required program must take the form outlined in the next slide.

Program for sample application, without timer device:

```
repeat forever
{
      for each of the four sets of lights, do
      {
            if the set is in blink mode,
                  the light which is ON is turned OFF,
                  and vice versa
      }
      read the status of the two switches
      if no switch is ON
            change all sets of lights to blink mode
      if any one switch is ON
            change two sets of lights to blink mode
      if both switches are ON
            turn off all lights
      execute wait loop of one second
}
```

- If no timer device is available, the processor has to enter a *wait loop* for the required time. Programmer must take into account the time of execution of each instruction in the loop, and then select the appropriate *loop count*.

- We have assumed that the processor has only one task - monitoring the switches and controlling the lights.

  - Therefore, nothing is lost by 'doing nothing' between successive rounds of the outer loop.

  - In effect, the processor is doing useful work only once in every second, and that too for a fraction of a millisecond.

- In a dedicated, low-cost, and simple application, this may often be acceptable.

Modes of data transfer:

- I/O options available to a system designer can be arranged in the form of a set of choices which must be made, as shown in the next table.

- First column lists the I/O function or feature, against which the option listed in either the second or the third column may be selected.

- For multiple I/O devices of one kind, an interrupt line can be shared between devices.
  - When an interrupt is received on a shared line, the processor must poll the devices to determine which device needs attention.
  - This technique is listed as ‘Polling’ in the second row of the table.

Function or feature	*Option I*	*Option II*
Check for device status	Polling	Interrupt driven
Device identification	Polling – possibly with shared interrupt line	Each device has its own separate interrupt line
Unit of data transfer	Character	Block
Path of data transfer	Between device controller and programmable register	Between device controller and main memory
Width of data transfer	Single bit – serial transfer	Multiple bits – parallel transfer
Timing	Synchronous	Asynchronous

- Unit of data transfer between an I/O device and processor/ memory can be either a byte (or word), or a *block* of bytes. This depends on the type of device; e.g. a mouse and a magnetic disk have different units of data transfer.

- These two types of devices are usually known as *character devices* and *block devices*.

- During I/O, data may be transferred between the device and the system in one of two different ways:
  - Between the device controller and a register in the processor, using IN and OUT instructions (*Programmed I/O).*
  - Directly between the device controller and main memory (*Direct Memory Access* - DMA).

- Usually character devices are operated using programmed I/O, and block devices in DMA mode.

- Data path between the device controller and the device may allow *serial* (*single bit*) or in *parallel (multiple bit)* data transfers.

- A serial device interface offers several advantages: ease of use, compact size, and lower cost. Serial devices today offer reasonable speeds. Example: USB is a serial interface, while a PCMCIA slot is a parallel interface.

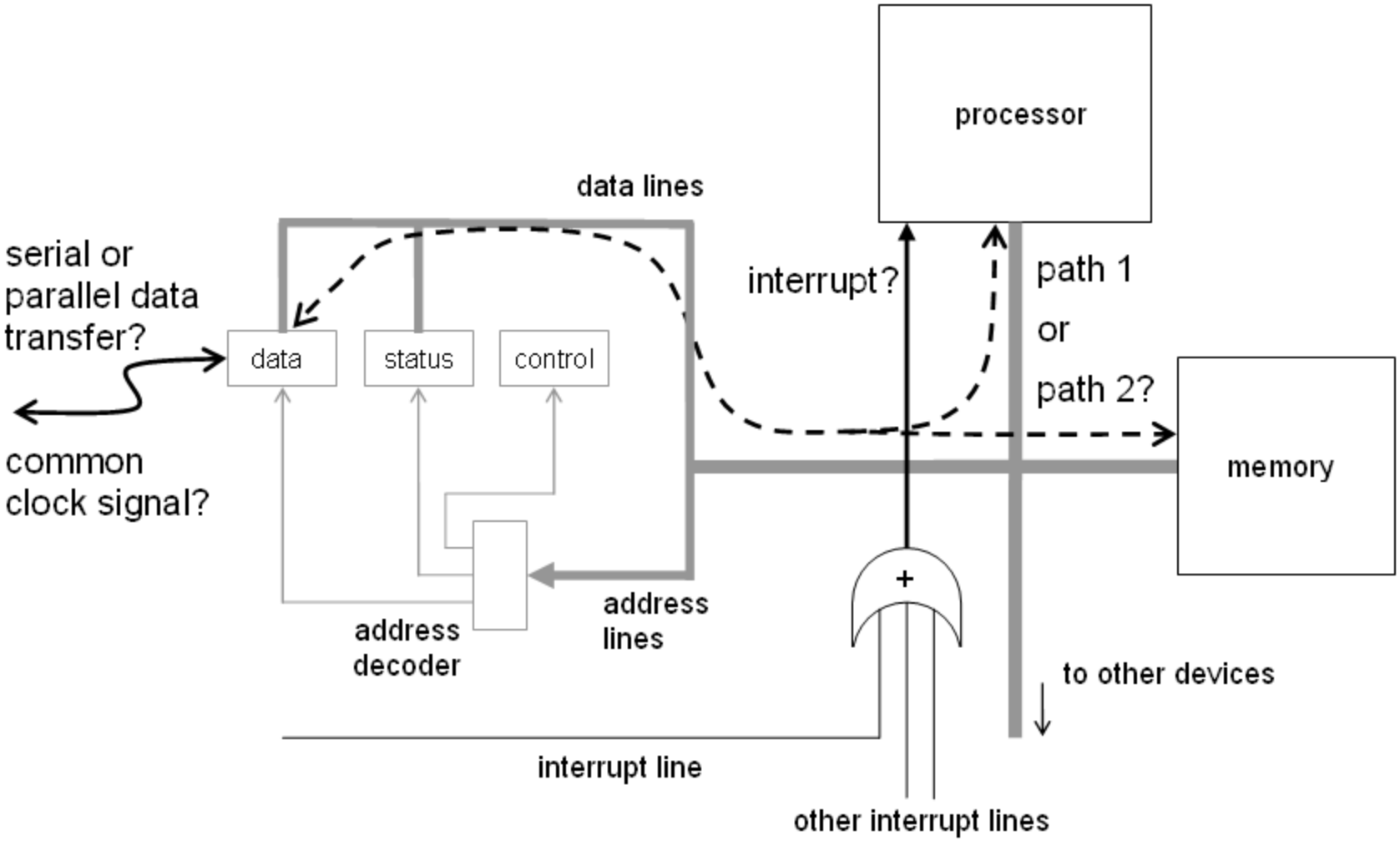
- When the sending and the receiving end share a common clock signal, the data transfers are called *synchronous*.

- But sometimes this is not possible, e.g. in the connection between a telephone modem and a PC.

- In such cases, the receiving end relies on transitions in signal voltage (or current) to synchronize itself with the signal received. Such data transfer mechanisms are called *asynchronous*.

- The next figure depicts these possible modes of data transfer.

Modes of data transfer:



- Lower part of the figure shows interrupt lines from three different devices being combined using an *or* gate, and supplied as a single interrupt line to the processor.

- If such a shared interrupt line is available, then the processor is interrupted when one or more of devices need service, and devices must be polled.

- If each device in the system has its own interrupt line, then a signal on that line immediately identifies the device, and there is no need for any polling.

- If no interrupt line is available, then the software must poll the devices periodically to check whether any of them needs service.

```
in priority order
(or in the order specified by the designer)
{
    read status of device k

    if device k needs service
        call service routine of device k

    select next device
}
```

- The above figure shows a simplified view of the polling algorithm. The order in which devices are polled can be determined by (i) device address, or (ii) device *priority*, as assigned by the system designer

- In polling order, the respective device status register is read into the processor, and the relevant bits examined. If the polled device needs service at the time, the appropriate *service routine* is called, and then polling resumes from the next device in polling order.

- To transfer data to/ from the device controller, there are two possible paths shown in the previous figure.
  - 1. ‘Path 1’ is between the device controller and the processor. This path corresponds to programmed I/O. Each byte (or word) is transferred using an IN or OUT instruction, as seen in Program 10.1.
  - 2. ‘Path 2’ is between the device controller and main memory. Data transfers carried out along this path are in *Direct Memory Access* (DMA) mode.

- On the left of the figure, we see the choice between *serial* and *parallel* data transfers. For serial I/O – i.e. one bit at a time – only two lines are needed between the device and the device controller.

- When multiple bits – *say m bits* – are transferred at a time, the transfer is said to be in *m*-bit *parallel* mode. Here, the data path must consist of at least *m*+1 lines, with one of them serving as *signal ground*.

Character devices and block devices:

- A **character** I/O device transfers one 'character' at a time. A **block** I/O device transfers an entire block of bytes at a time – where a block may be of fixed or variable number of bytes.
- Example: **character** device: keyboard; **block** device - magnetic disk

- In case of a keyboard (or mouse), the speed at which data is entered into the computer is limited by the user's ability to operate the device.
- It is necessary that such a device sends data to the processor – i.e. to the application program – as soon as the user generates it, which means one character or one mouse-click at a time.

- The smallest unit of data which a magnetic disk can read or write is a *sector* (in the range of 512 to 4096 bytes).
  - The *disk controller* must have sufficient buffer memory to smoothen the movement of blocks of data between magnetic disk and main memory

- In general device controllers for block devices have their own memory buffers, complex logic and timing circuits, and an *embedded* processor (to monitor and control operations).
  - Such device controllers make use of DMA mode of data transfer, and have a fairly sophisticated set of commands.
  - The processor issues these commands to device controller using programmed I/O. Data transfer itself takes place in DMA mode.

Direct Memory Access (DMA):

- Programmed I/O: Data is transferred between device controller and a register. This method works fine for simpler and slower devices.

- In the operation of block devices, an entire block of data is transferred between the device and main memory, without the data passing through the processor.
  - The processor is only interrupted at the end of the block transfer of data.
  - This mode of I/O is more efficient than programmed I/O, because data does not pass through the processor on every byte or word.
  - i.e. The processor can continue to run another related or unrelated program.

- This mode of data transfer is *Direct Memory Access.*

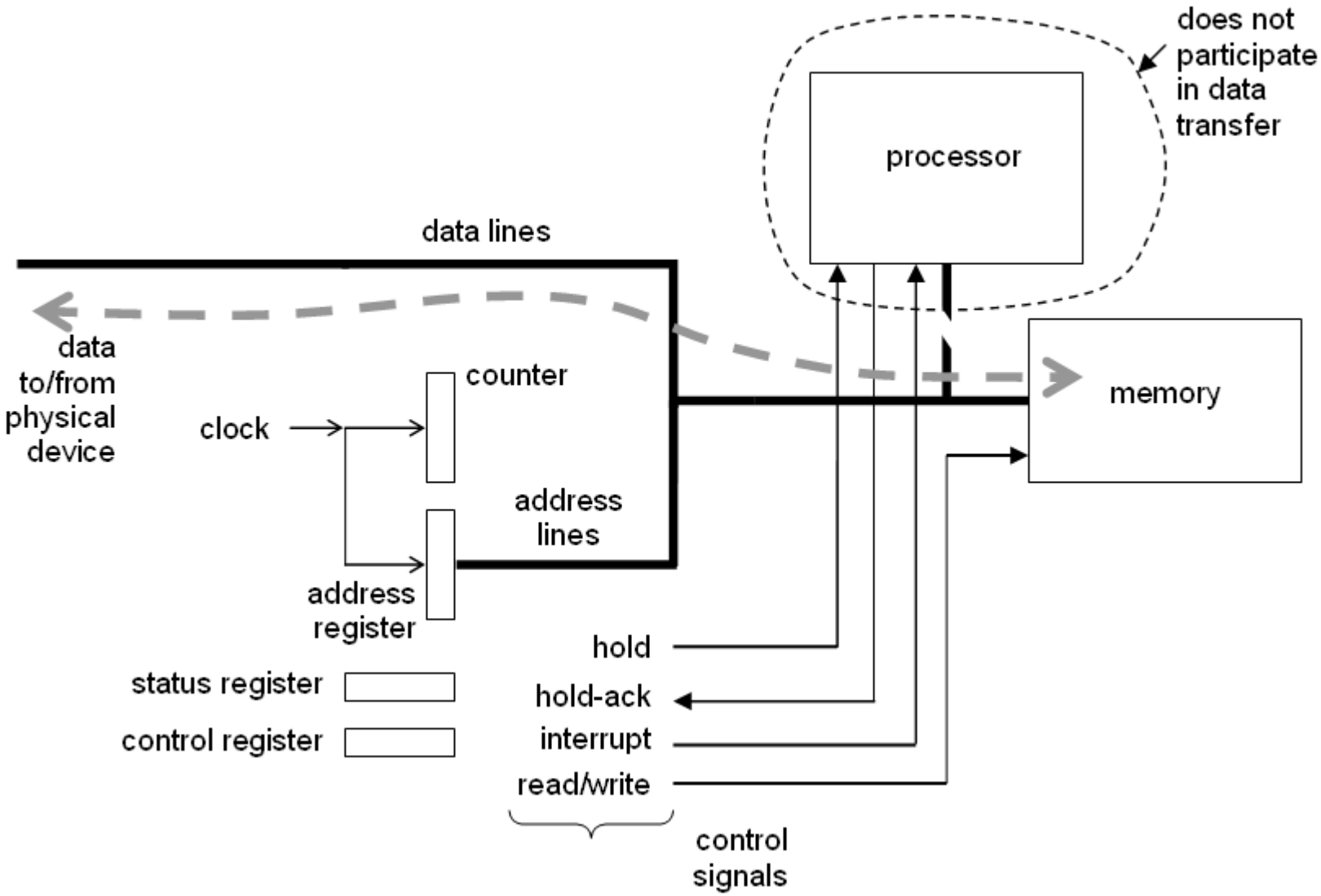
- Processor intervention is needed to set up a DMA operation, and also at its completion. The data transfer itself takes place without any processor intervention.

- The additional hardware provided on the system for such transfers is known as a *DMA controller*.

- While a DMA operation is in progress, both processor and DMA controller need to access main memory. The processor goes through its own fetch and execute cycles, even while a DMA operation is in progress to or from main memory.

- The next figure depicts such a data transfer taking place between an input device and main memory. Path of data is shown using a dashed, thick arrow, from the device on the left, via DMA controller, to main memory.

DMA data transfer between device and main memory



DMA controller:

- An *address register* contains successive memory addresses for read/write DMA operations, and a *counter* keeps track of the number of bytes transferred.

- Using *hold* signal, DMA controller signals the processor to 'hold' its memory accesses, to allow DMA operation to take place.

- DMA controller cannot access memory until confirmation is received from processor that its memory accesses are 'on hold'. Therefore a signal with the function of *hold acknowledgement* is also provided, which is sent by processor to DMA controller.

- DMA controller also has own its *status* & *control* registers. To signal completion of DMA operation, the controller can use an interrupt line to the processor.

- Thus a complete DMA I/O operation has three phases:

(i) Using programmed I/O, program running on the processor supplies memory address, byte count, & read/ write command to DMA controller. This is the DMA setup and initiation phase.

(ii) Based on the parameters & command supplied, DMA controller transfers data to/ from main memory, using the mechanism outlined. On the device side, the device controller performs the required functions.

(iii) At the end of data transfer, the controller sends an interrupt to the processor. Interrupt service routine completes remaining verification and signaling required.

I/O Interfaces:

- A modern PC has a vast range of *add-on* options and I/O devices which can be attached. Industry associations have developed interface standards, which manufacturers follow.

- Common examples: Serial and parallel ports, USB ports, PCMCIA slot, RJ-11 jack, jacks for external speakers, and the 15-pin connector for video output.

- Such an interface standard has several important parts:

  1) Mechanical specification – the precise mechanical shape and dimensions of the two opposite sides of a connection, type of fit between the two sides, number and layout of pins, and so on.

2) Electrical specification – this may include voltage, current or power levels, speed and timing constraints, electrical characteristics such as contact resistance and 'line impedance'.

3) Functional specification - the functional role of the electrical signal on each of the pins.

- Based on the functional specification, a *device driver* for a connected device can be written. The device driver enables a user program to send data to the device, or receive data from it.

- The device driver provides a '*software specification*' of the device. But this part of the specification is usually not included in the interface standard.

I/O Processors

- An *I/O Processor* on the internal system bus frees the main processor(s) from almost all work related to input and output of data.

- An I/O processor can perform functions such as the following:
  - Interface to external bus or busses
  - Direct memory access
  - Interaction with device controllers
  - Interaction with relevant parts of the operating system
  - Servicing of interrupts according to priorities
  - Buffering of data, etc.

- Advances in electronic technology have made devices and controllers faster & smarter.
  - Embedded processors provide higher levels of functionality in device controllers.
  - Standard interfaces between devices and device controllers, and speeds of internal system busses, have become much faster.
  - Higher-end systems make use of multiple processors for highly compute-intensive applications.

- Therefore today a separate I/O processor is not essential for achieving high aggregate data rates on I/O.
  - However, for specialized I/O requirements, a separate processor may be required in order to achieve desired performance. [Example on next slide].

Example: *Graphics processor*

- Rendering of animated 3D objects requires the processing of a large number of ‘vectors’ through a common sequence of operations. The stages of such processing make up a *rendering pipeline*.

- When these functions are implemented on a special graphics processor, the speed and quality of computer generated graphics is greatly enhanced. The graphics processor executes its own machine instructions to perform its specialized functions.

- Graphics processor has pipeline stages corresponding to stages of the graphics rendering pipeline.

Summary

- Input and output sub-system of a computer system
- Role of device controllers
- Concepts of programmed I/O and polling
- Mechanism of interrupts
- Various modes of I/O data transfers
- Direct memory access
- Standardized I/O interfaces
- I/O processors