INPUT-OUTPUT ORGANIZATION

- Peripheral Devices
- Input-Output Interface
- Asynchronous Data Transfer
- Modes of Transfer
- Priority Interrupt
- Direct Memory Access
- Input-Output Processor
- Serial Communication

PERIPHERAL DEVICES

Input Devices

- Keyboard
- Optical input devices
  - Card Reader
  - Paper Tape Reader
  - Bar code reader
  - Digitizer
  - Optical Mark Reader
- Magnetic Input Devices
  - Magnetic Stripe Reader
- Screen Input Devices
  - Touch Screen
  - Light Pen
  - Mouse
- Analog Input Devices

Output Devices

- Card Puncher, Paper Tape Puncher
- CRT
- Printer (Impact, Ink Jet, Laser, Dot Matrix)
- Plotter
- Analog
- Voice

INPUT/OUTPUT INTERFACE

- Provides a method for transferring information between internal storage (such as memory and CPU registers) and external I/O devices
- Resolves the *differences* between the computer and peripheral devices
  - Peripherals - Electromechanical Devices
  - CPU or Memory - Electronic Device
  - Data Transfer Rate
    - » Peripherals - Usually slower
    - » CPU or Memory - Usually faster than peripherals
      - Some kinds of Synchronization mechanism may be needed
  - Unit of Information
    - » Peripherals – Byte, Block, …
    - » CPU or Memory – Word
  - Data representations may differ

ISOLATED vs MEMORY MAPPED I/O

Isolated I/O

- Separate I/O read/write control lines in addition to memory read/write control lines
- Separate (isolated) memory and I/O address spaces
- Distinct input and output instructions

Memory-mapped I/O

- A single set of read/write control lines (no distinction between memory and I/O transfer)
- Memory and I/O addresses share the common address space
  -> reduces memory address range available
- No specific input or output instruction
  -> The same memory reference instructions can be used for I/O transfers
- Considerable flexibility in handling I/O operations

ASYNCHRONOUS DATA TRANSFER

Synchronous and Asynchronous Operations

Synchronous - All devices derive the timing information from common clock line

Asynchronous - No common clock

Asynchronous Data Transfer

Asynchronous data transfer between two independent units requires that *control signals* be transmitted between the communicating units *to indicate the time at which data is being transmitted*

Two Asynchronous Data Transfer Methods

Strobe pulse

- A strobe pulse is supplied by one unit to indicate the other unit when the transfer has to occur

Handshaking

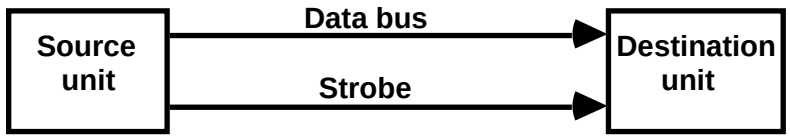
- A control signal is accompanied with each data being transmitted to indicate the presence of data
- The receiving unit responds with another control signal to acknowledge receipt of the data

STROBE CONTROL

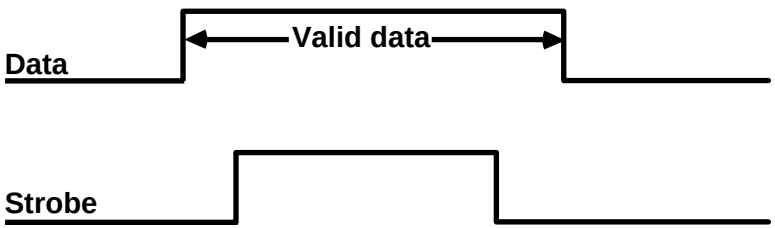
* Employs a single control line to time each transfer
* The strobe may be activated by either the source or the destination unit

Source-Initiated Strobe for Data Transfer

Block Diagram

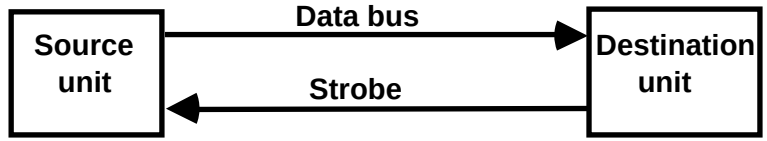


Timing Diagram

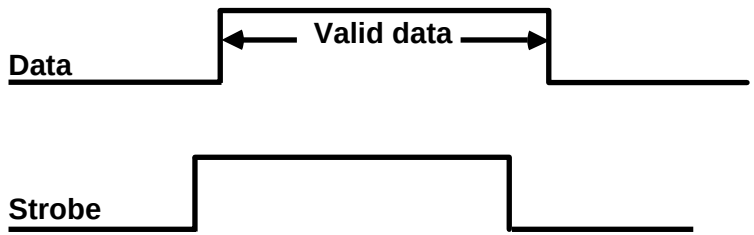


Destination-Initiated Strobe for Data Transfer

Block Diagram



Timing Diagram



HANDSHAKING

Strobe Methods

Source-Initiated

The source unit that initiates the transfer has no way of knowing whether the destination unit has actually received data

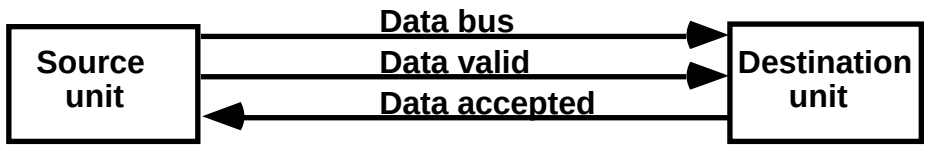
Destination-Initiated

The destination unit that initiates the transfer no way of knowing whether the source has actually placed the data on the bus

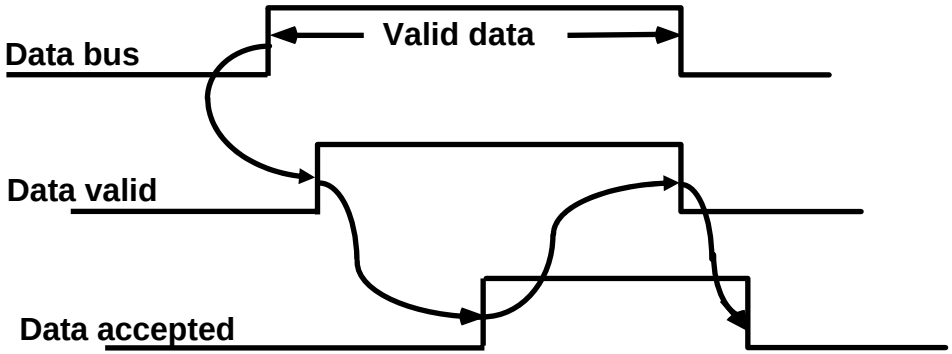
To solve this problem, the *HANDSHAKE* method introduces a second control signal to provide a *Reply* to the unit that initiates the transfer

SOURCE-INITIATED TRANSFER USING HANDSHAKE

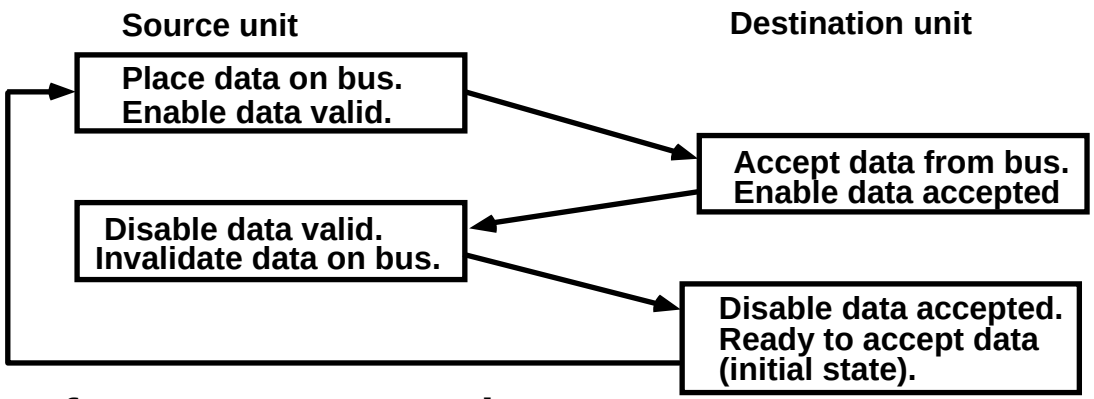
Block Diagram



Timing Diagram



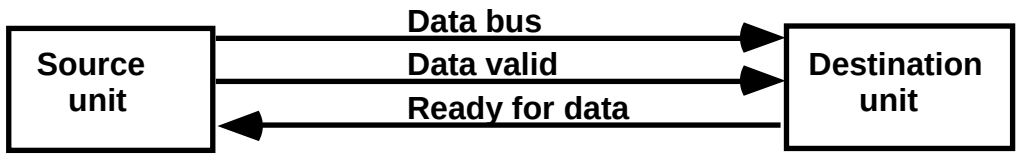
Sequence of Events



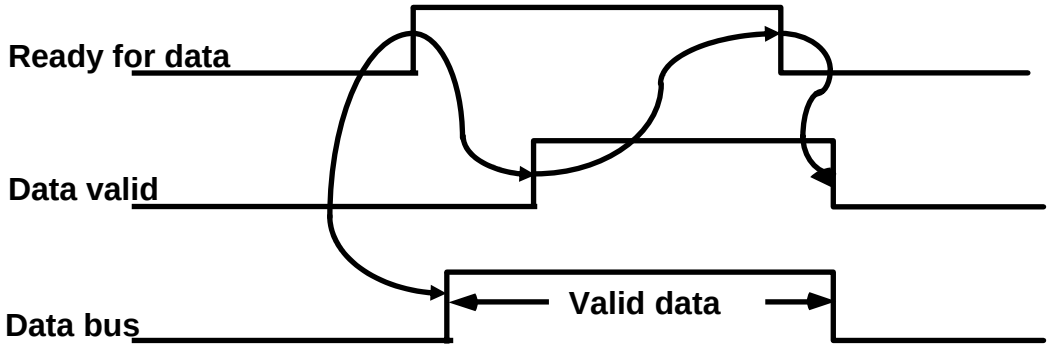
* Allows arbitrary delays from one state to the next
* Permits each unit to respond at its own data transfer rate
* The rate of transfer is determined by the slower unit

DESTINATION-INITIATED TRANSFER USING HANDSHAKE

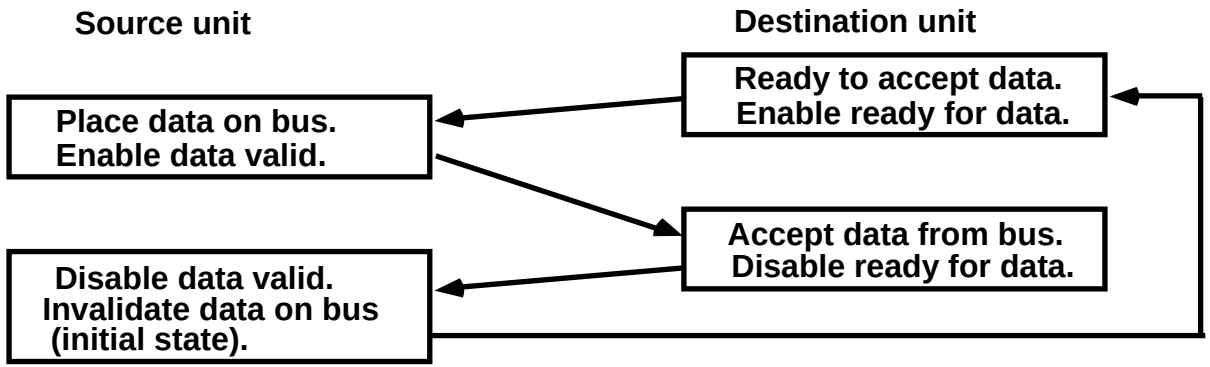
Block Diagram



Timing Diagram



Sequence of Events



* Handshaking provides a high degree of flexibility and reliability because the successful completion of a data transfer relies on active participation by both units
* If one unit is faulty, data transfer will not be completed
  -> Can be detected by means of a *timeout* mechanism

Asynchronous serial transfer

- In Serial data transfer each bit in the message is sent in sequence one at a time. Serial transmission is slower but less expensive.
- Serial transmission types:1)Synchronous 2)Asynchronous
- Syn: Two units share a common clock frequency and bits are transmitted continuously at the rate of clock pulse.
- Asy: binay infor is sent only when it is available and the line remain idle when there is no information to be transmitted.

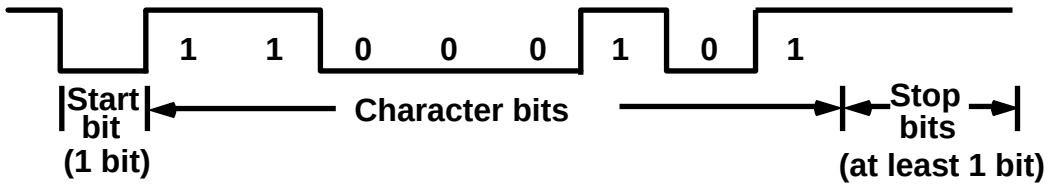
ASYNCHRONOUS SERIAL TRANSFER

Four Different Types of Transfer

- **Asynchronous serial transfer**
- **Synchronous serial transfer**
- **Asynchronous parallel transfer**
- **Synchronous parallel transfer**

Asynchronous Serial Transfer

- **Employs special bits which are inserted at both ends of the character code**
- **Each character consists of three parts; Start bit; Data bits; Stop bits.**



A character can be detected by the receiver from the knowledge of 4 rules;

- **When data are not being sent, the line is kept in the 1-state (idle state)**
- **The initiation of a character transmission is detected by a *Start Bit* , which is always a 0**
- **The character bits always follow the *Start Bit***
- **After the last character , a *Stop Bit* is detected when the line returns to the 1-state for at least 1 bit time**

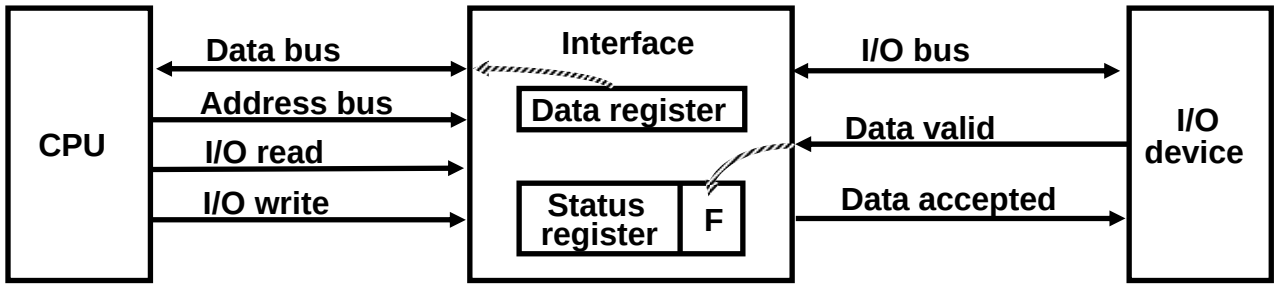
The receiver knows in advance the transfer rate of the bits and the number of information bits to expect

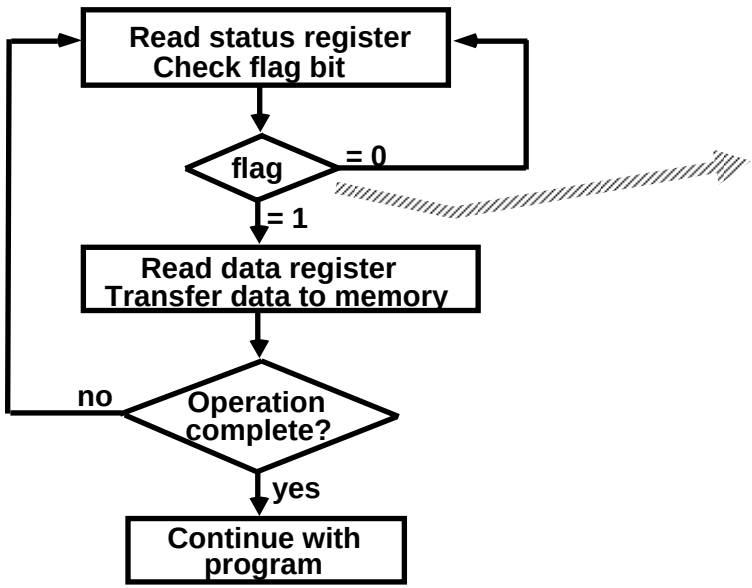
MODES OF TRANSFER - PROGRAM-CONTROLLED I/O -

3 different Data Transfer Modes between the central computer(CPU or Memory) and peripherals;

Program-Controlled I/O
Interrupt-Initiated I/O
Direct Memory Access (DMA)

Program-Controlled I/O(Input Dev to CPU)





Polling or Status Checking

- Continuous CPU involvement
- CPU slowed down to I/O speed
- Simple
- Least hardware

MODES OF TRANSFER - INTERRUPT INITIATED I/O & DMA

Interrupt Initiated I/O

- Polling takes valuable CPU time
- Open communication only when some data has to be passed -> *Interrupt*.
- I/O interface, instead of the CPU, monitors the I/O device
- When the interface determines that the I/O device is ready for data transfer, it generates an *Interrupt Request* to the CPU
- Upon detecting an interrupt, CPU stops momentarily the task it is doing, branches to the service routine to process the data transfer, and then returns to the task it was performing

DMA (Direct Memory Access)

- Large blocks of data transferred at a high speed to or from high speed devices, magnetic drums, disks, tapes, etc.
- DMA controller
  Interface that provides I/O transfer of data directly to and from the memory and the I/O device
- CPU initializes the DMA controller by sending a memory address and the number of words to be transferred
- Actual transfer of data is done directly between the device and memory through DMA controller
  -> Freeing CPU for other tasks

PRIORITY INTERRUPT

Priority

- Determines which interrupt is to be served first when two or more requests are made simultaneously
- Also determines which interrupts are permitted to interrupt the computer while another is being serviced
- Higher priority interrupts can make requests while servicing a lower priority interrupt

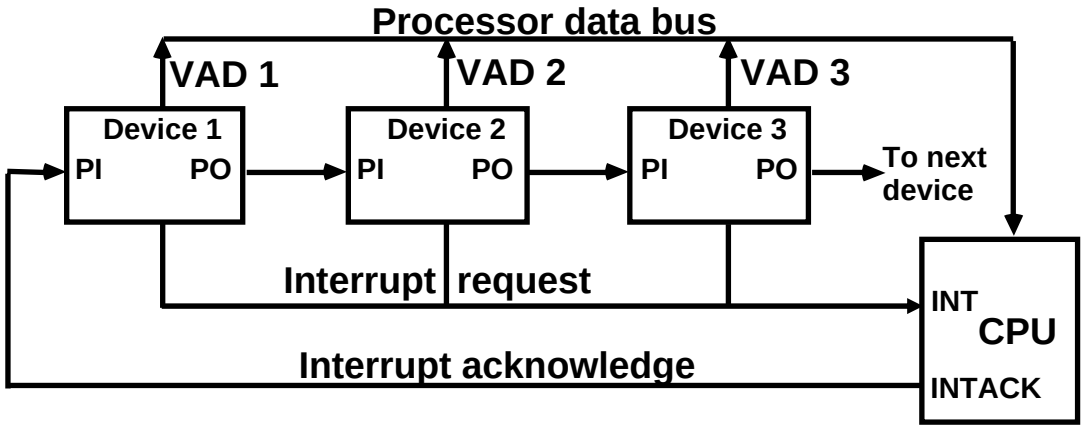
Priority Interrupt by Software(Polling)

- Priority is established by the order of polling the devices(interrupt sources)
- Flexible since it is established by software
- Low cost since it needs a very little hardware
- Very slow

Priority Interrupt by Hardware

- Require a priority interrupt manager which accepts all the interrupt requests to determine the highest priority request
- Fast since identification of the highest priority interrupt request is identified by the hardware
- Fast since each interrupt source has its own interrupt vector to access directly to its own service routine

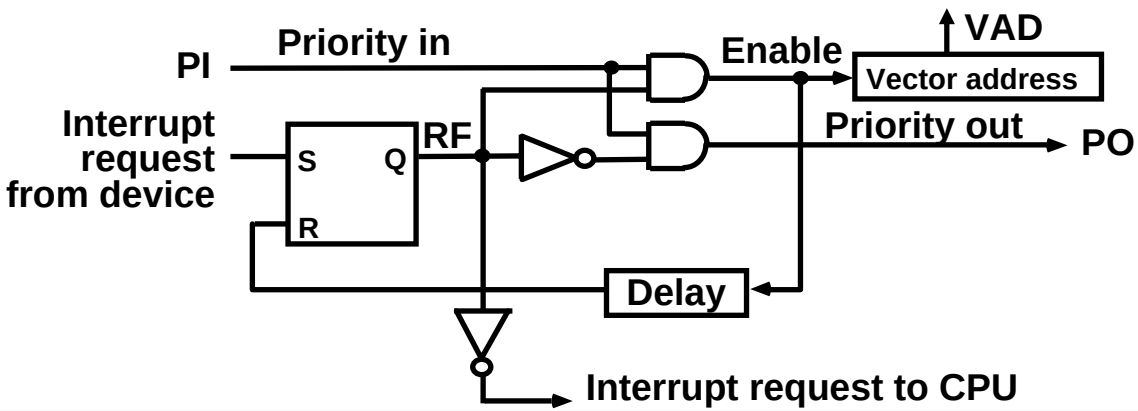
HARDWARE PRIORITY INTERRUPT - DAISY-CHAIN -



* Serial hardware priority function
* Interrupt Request Line
  - Single common line
* Interrupt Acknowledge Line
  - Daisy-Chain

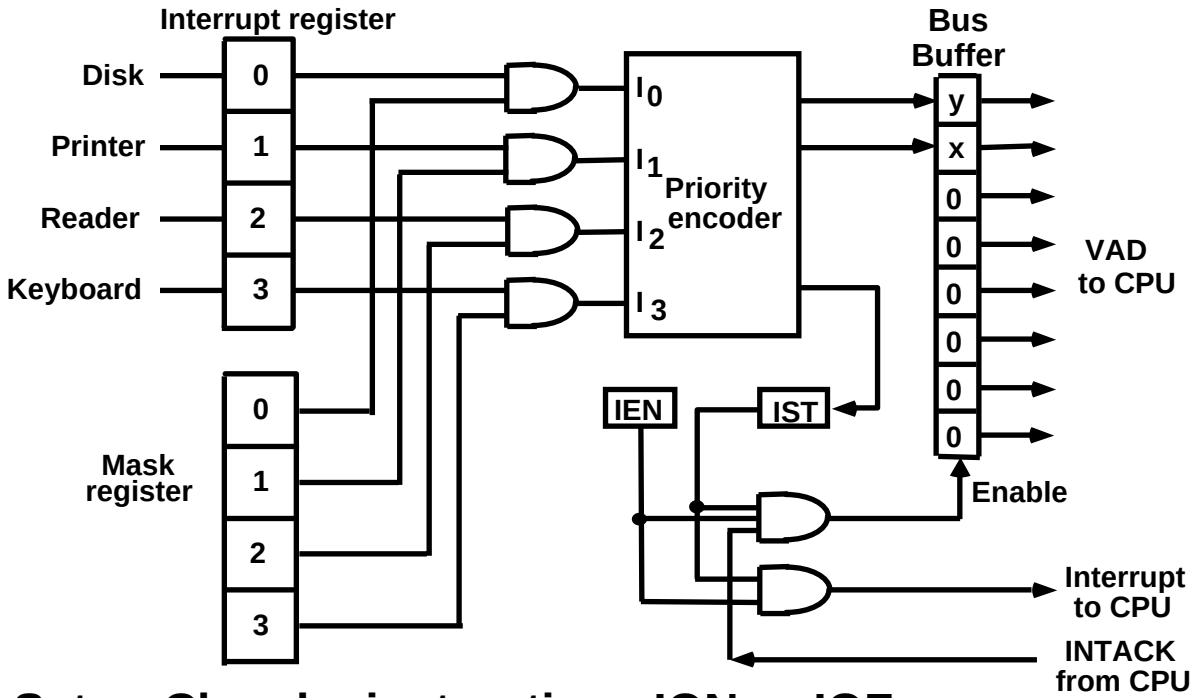
Interrupt Request from any device(>=1)
-> CPU responds by INTACK <- 1
-> Any device receives signal(INTACK) 1 at PI puts the VAD on the bus
Among interrupt requesting devices the only device which is physically closest to CPU gets INTACK=1, and it blocks INTACK to propagate to the next device

One stage of the daisy chain priority arrangement



PI	RF	PO	Enable
0	0	0	0
0	1	0	0
1	0	1	0
1	1	1	1

PARALLEL PRIORITY INTERRUPT



IEN: Set or Clear by instructions ION or IOF

IST: Represents an unmasked interrupt has occurred. INTACK enables tristate Bus Buffer to load VAD generated by the Priority Logic

Interrupt Register:

- Each bit is associated with an Interrupt Request from different Interrupt Source - different priority level
- Each bit can be cleared by a program instruction

Mask Register:

- Mask Register is associated with Interrupt Register
- Each bit can be set or cleared by an Instruction

INTERRUPT PRIORITY ENCODER

Determines the highest priority interrupt when more than one interrupts take place

Priority Encoder Truth table

Inputs				Outputs			Boolean functions
I_0	I_1	I_2	I_3	x	y	IST	
1	d	d	d	0	0	1	
0	1	d	d	0	1	1	
0	0	1	d	1	0	1	$x = I_0' \; I_1'$
0	0	0	1	1	1	1	$y = I_0' \, I_1 + I_0' \, I_2'$
0	0	0	0	d	d	0	$(IST) = I_0 + I_1 + I_2 + I_3$

INTERRUPT CYCLE

At the end of each Instruction cycle

- CPU checks IEN and IST

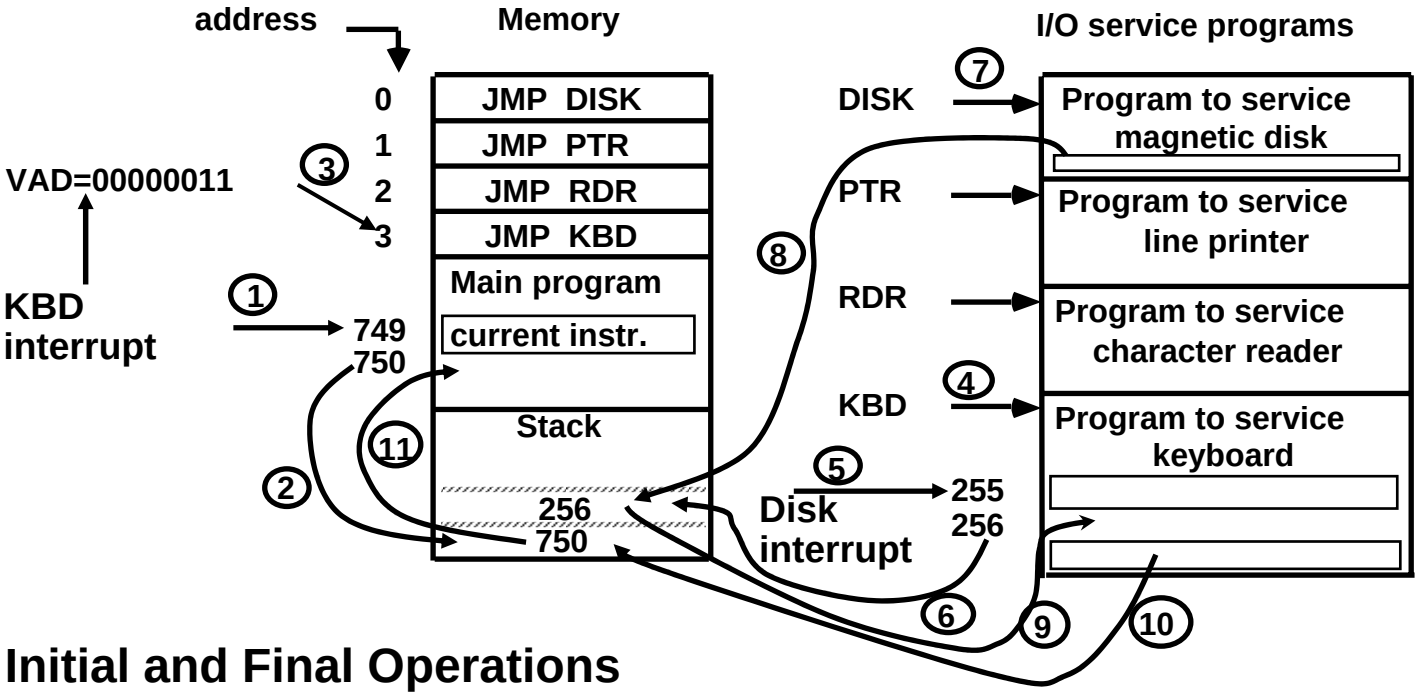
- If IEN • IST = 1, CPU -> Interrupt Cycle

SP ← SP - 1	Decrement stack pointer
M[SP] ← PC	Push PC into stack
INTACK ← 1	Enable interrupt acknowledge
PC ← VAD	Transfer vector address to PC
IEN ← 0	Disable further interrupts
Go To Fetch	to execute the first instruction in the interrupt service routine

Interrupt Service Routine

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in an operating system whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated.

INTERRUPT SERVICE ROUTINE



Initial and Final Operations

Each interrupt service routine must have an initial and final set of operations for controlling the registers in the hardware interrupt system

Initial Sequence

[1] Clear lower level Mask reg. bits
[2] IST <- 0
[3] Save contents of CPU registers
[4] IEN <- 1
[5] Go to Interrupt Service Routine

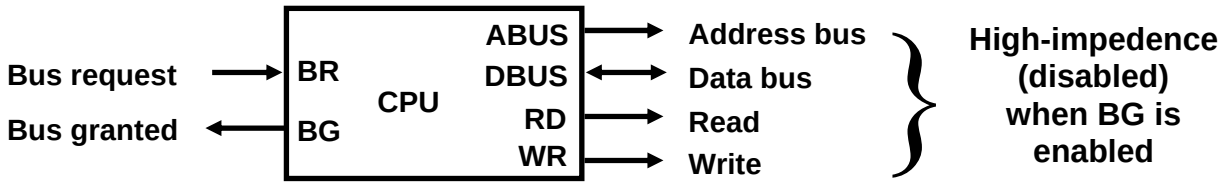
Final Sequence

[1] IEN <- 0
[2] Restore CPU registers
[3] Clear the bit in the Interrupt Reg
[4] Set lower level Mask reg. bits
[5] Restore return address, IEN <- 1

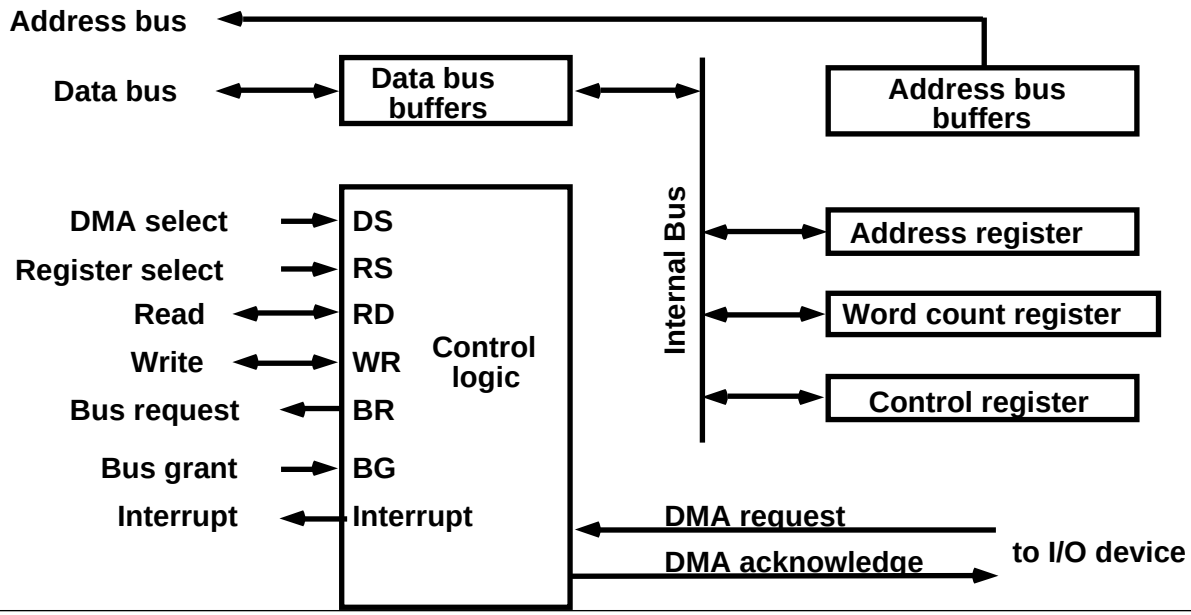
DIRECT MEMORY ACCESS

* Block of data transfer from high speed devices, Drum, Disk, Tape
* DMA controller - Interface which allows I/O transfer directly between Memory and Device, freeing CPU for other tasks
* CPU initializes DMA Controller by sending memory address and the block size(number of words)

CPU bus signals for DMA transfer



Block diagram of DMA controller



- When BG =1, the CPU can communicate with the DMA registers.
- When BG=1, CPU will give up the buses and DMA can directly communicate with the memory.
- DMA controller has 3 registers like address register, word count register and control register.
- The address register contains an address to specify the desired location in memory.
- The word count register holds the number of words to be transferred.
- The control register specifies the modes of transfer.

DMA I/O OPERATION

Starting an I/O

- CPU executes instruction to

Load Memory Address Register
Load Word Counter
Load Function(Read or Write) to be performed
Issue a GO command

Upon receiving a GO Command DMA performs I/O operation as follows independently from CPU

Input

[1] Input Device <- R (Read control signal)
[2] Buffer(DMA Controller) <- Input Byte; and assembles the byte into a word until word is full
[4] M <- memory address, W(Write control signal)
[5] Address Reg <- Address Reg +1; WC(Word Counter) <- WC - 1
[6] If WC = 0, then Interrupt to acknowledge done, else go to [1]

Output

[1] M <- M Address, R
M Address R <- M Address R + 1, WC <- WC - 1
[2] Disassemble the word
[3] Buffer <- One byte; Output Device <- W, for all disassembled bytes
[4] If WC = 0, then Interrupt to acknowledge done, else go to [1]

CYCLE STEALING

While DMA I/O takes place, CPU is also executing instructions

DMA Controller and CPU both access Memory -> Memory Access Conflict

Memory Bus Controller

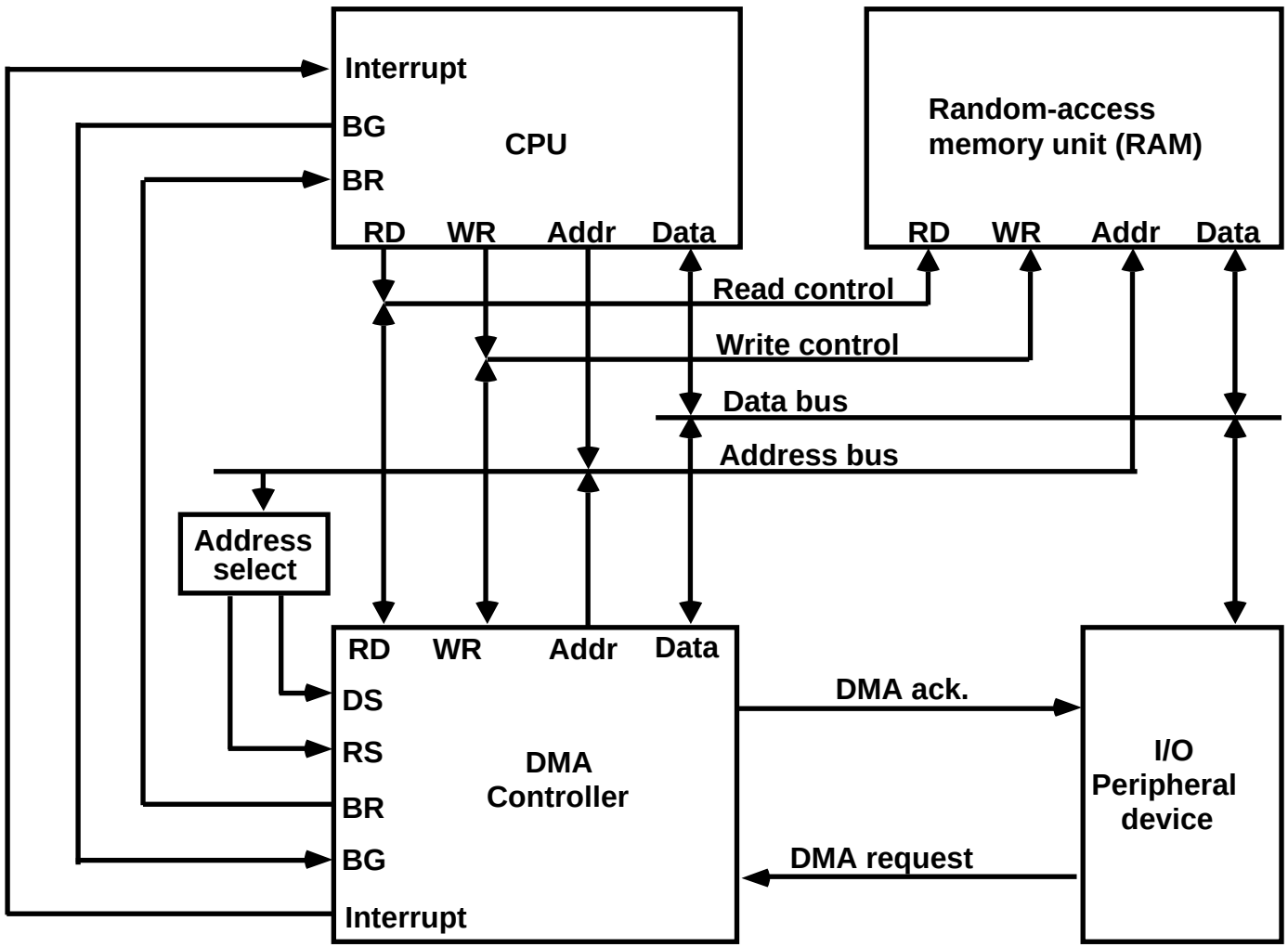
- Coordinating the activities of all devices requesting memory access
- Priority System

Memory accesses by CPU and DMA Controller are interwoven, with the top priority given to DMA Controller
-> Cycle Stealing

Cycle Steal

- CPU is usually much faster than I/O(DMA), thus CPU uses the most of the memory cycles
- DMA Controller steals the memory cycles from CPU
- For those stolen cycles, CPU remains idle
- For those slow CPU, DMA Controller may steal most of the memory cycles which may cause CPU remain idle long time

DMA TRANSFER



Interrupt Service Routine

An interrupt handler, also known as an interrupt service routine (ISR), is a callback subroutine in an operating system whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated.

Interrupt Overhead

The interrupt overhead is caused by context switching (storing and restoring the state of CPU)

On interrupt handler entry, the context of the current process and its thread must be saved. On exit, it must be restored.

On handler entry, memory locations different from the memory locations in the cache are used, and therefore cache updates are required.

-In a similar way, the interrupted process suffers from the interrupt:

-the cache it was using is disturbed, and cache updates are required