What is the need of I/O interface?

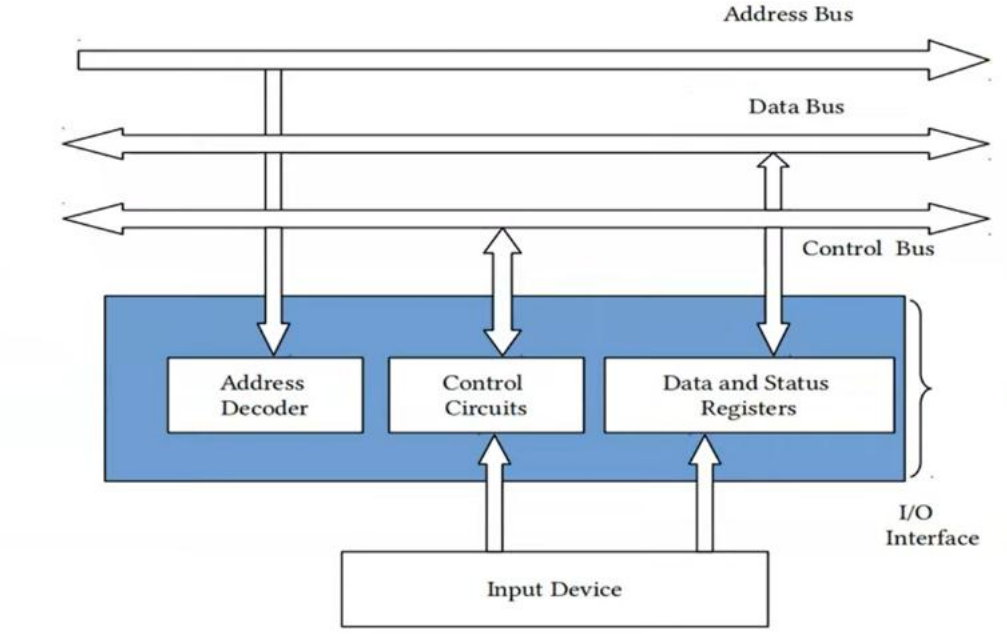
Peripherals are electromechanical or electromagnetic devices, and their manner of operation is different from the operation of the CPUand memory, which are electronic devices. So conversion of signal is required.

Data transfer rate of I/O is slower than CPU and memory.

Data codes and format in peripherals differ from the word format in the CPU and memory. So conversion of formats is required.

The operating modes of peripherals are different from each other and each must be controlled so a peripheral does not distrub the operation of other peripherals.

I/O Interface



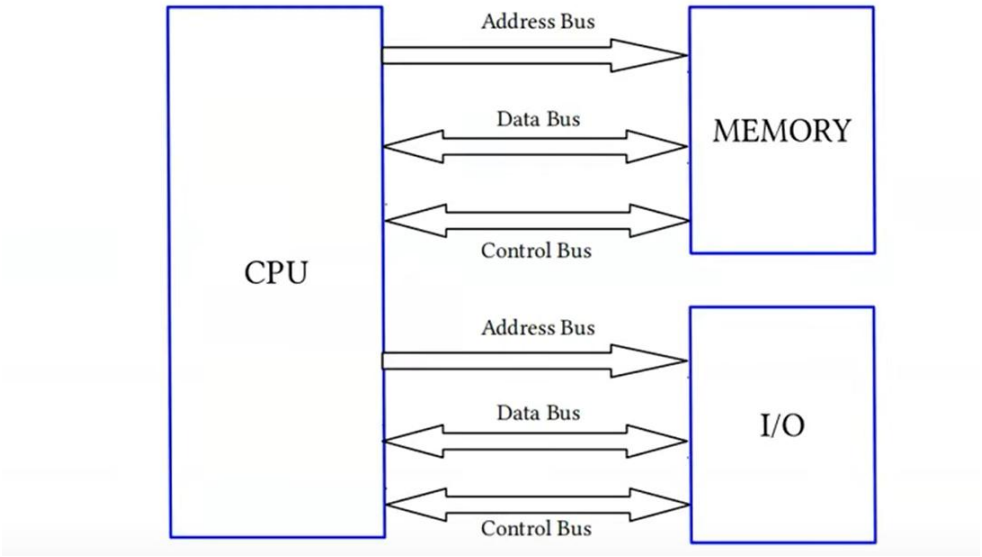
CPU's communication with memory and I/O

Separate Bus

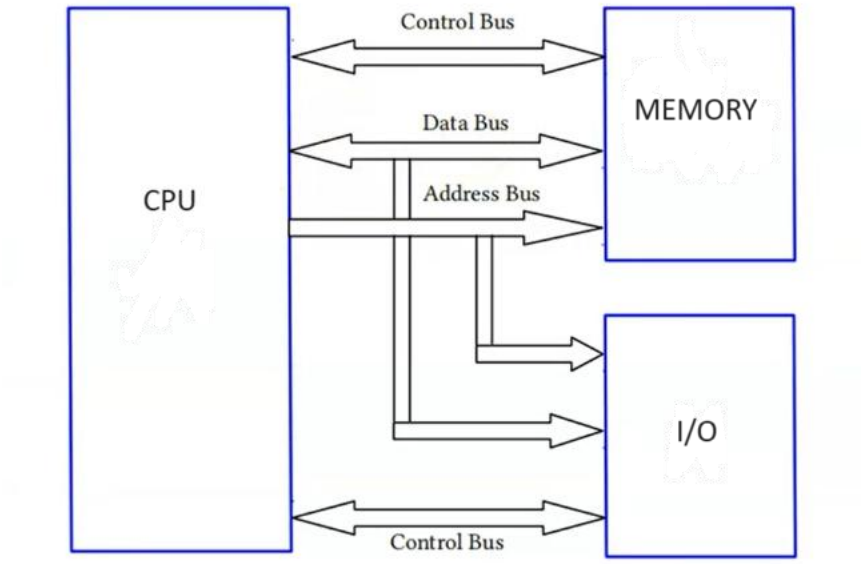
Common address bus and data bus

Common address bus data bus and control bus

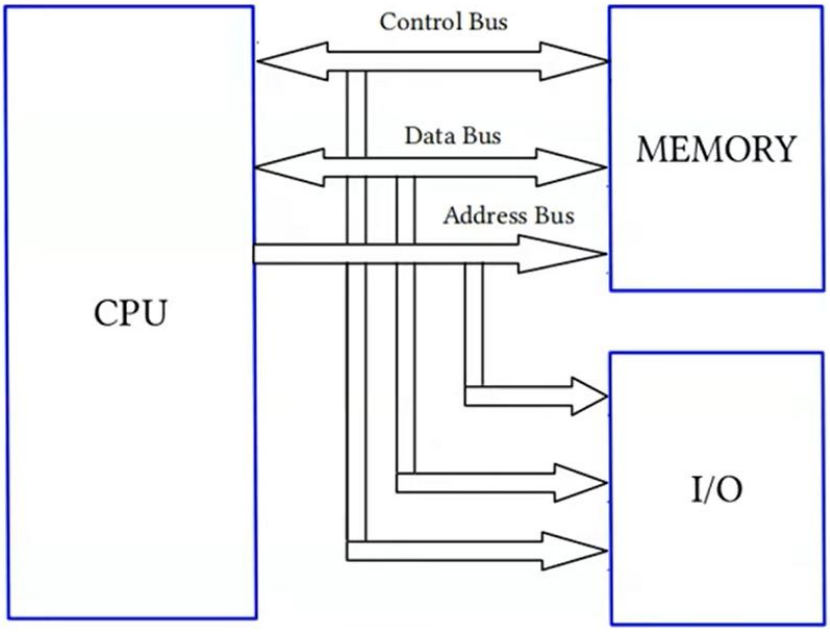
Separate Buses for both memory and I/O



Common Address and Data Bus and Separate Control Bus



Common Address, Data Bus and Control Bus



Memory Mapped I/O vs I/O Mapped I/O

Memory Mapped I/O	I/O Mapped I/O
I/O devices do not have separate address space.	I/O and memory both have their own separate address space.
Some memory space remains untilized.	Memory is fully utilized.
Same instructions can be used for both memory and I/O devices. e.g., MOV	I/O access and memory access instructions are distinct. e.g., IN, OUT for I/O.

Modes of I/O Transfer

Program Controlled I/O

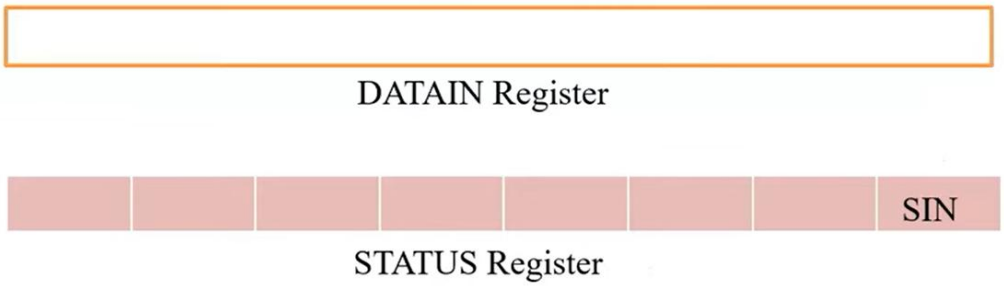
Interrupt Driven I/O

DMA

Program Controlled I/O

Processor repeatedly monitors a status flag to achieve the necessary synchronization.

Processor polls the I/O device.



Input/output Operations

Programmable I/O is the most simple type of I/O technique for the exchange of data between the processor and the I/O devices

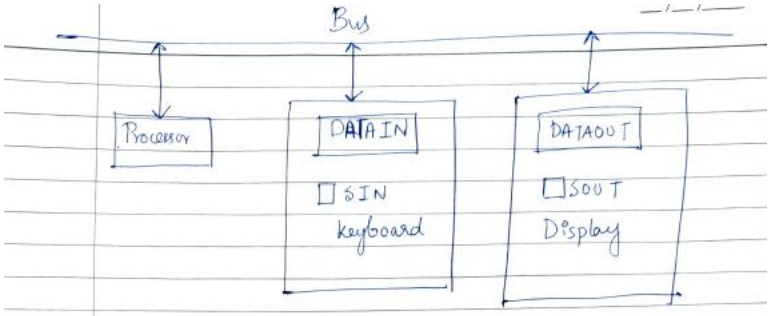
Ex: Consider a task that reads a character input from a keyboard and produces character output on a display screen.

Solution Background
(*) The rate of data transfer from the keyboard to a computer is (limited by the typing speed of the user) very slow. such as few characters per second.
(*) The rate of output transfers from the computer to the display is much higher (several thousand characters per sec).

(*) The speed of a processor is highest. (execute many millions of instructions per second).

(*) The difference in speed between the processor and I/O device should be synchronized.

Solution (1) Whenever you strike a key in the keyboard, the character code of this key will be stored in an 8-bit buffer register (DATAIN). To inform the processor that a valid character is in DATAIN, the value of status control flag SIN will be set to 1. A program monitors SIN, and when SIN is set to 1, the CPU reads the content of DATAIN. When the character is transferred to the CPU, SIN is automatically cleared to 0.



(2) A buffer register, DATAOUT and a status control flag SOUT are used to transfer the character from CPU to the display. When SOUT = 1, the display is ready to receive a character.

A program monitors SOUT and when SOUT is set to 1, the CPU 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 processor can monitor the keyboard status flag SIN and transfers a character from DATAIN to register R1 as:

```
READWAIT  Branch to READWAIT if SIN = 0
          Input from DATAIN to R1
```

(*) A similar sequence of operations to transfer output to the display as:

```
WRITEWAIT  Branch to WRITEWAIT if SOUT = 0
           Output from R1 to DATAOUT
```

Move Byte DATAIN, R1

Move Byte R1, DATAOUT

The "MoveByte" opcode indicates that the operand size is a byte to distinguish it from the opcode "Move" that has been used for word operands.

* It is more common to include SIN and SOUT in device status registers, one for each of the two devices. Let us assume bit b3 in registers INSTATUS and OUTSTATUS corresponds to SIN and SOUT respectively.

∴ The read operation can be described as

```
READWAIT   Testbit   #3, INSTATUS
           Branch=0  READWATT
           MoveByte  DATAIN, R1
```

∴ The write operation

```
WRITEWAIT  Testbit   #3, OUTSTATUS
           Branch =0 WRITEWATT
           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.

Example

	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 A program that reads a line of characters and displays it.

Interrupt Driven I/O

Polling method – Processor waits for response from I/O device. During wait period processor not able perform useful computation.

Come out from these problem using Interrupt

The I/O devices can alert the processor when it becomes ready. It can do so by sending a hardware signal called an ***interrupt request*** to the processor.

Interrupt Example 1

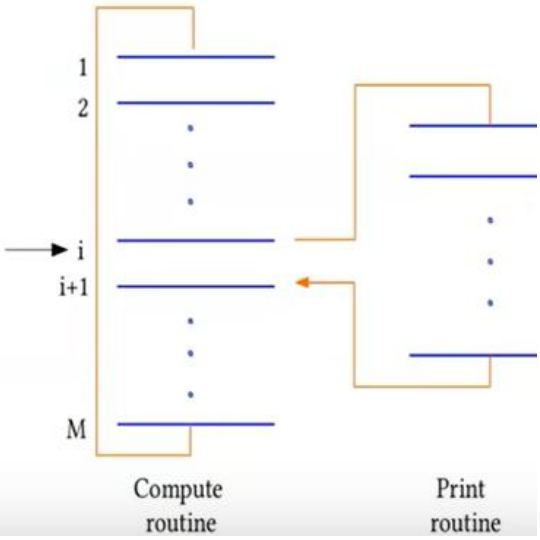
Consider a task that requires continuous extensive computations to be performed and the results to be printed on a printer.

COMPUTE produces a set of n lines of output.

PRINT routine sends one line of text to the printer for printing.

After printing one line printer sends Interrupt request signal to the processor. (At i th line.)

So processor interrupt the execution of COMPUTE routine and transfers the control to PRINT routine.



The routine executed in response to an interrupt request is called the ***interrupt-service routine***.

E.g. PRINT routine.

Processor responds or interrupt request responds to interrupted device by sending the control signal called **Interrupt Acknowledge**.

When interrupt occurs during execution of program processor register used, flag status information must saved in stack before execution of the interrupted program is resumed.

In this way, the original program can continue execution without being affected in any way by the interruption, **except for the time delay**.

The task of saving and restoring information can be done automatically by the processor **or by program instructions.**

The process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead.

Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine.

The process of saving and restoring registers involves memory transfers that increase the total execution time, and hence represent execution overhead.

Saving registers also increases the delay between the time an interrupt request is received and the start of execution of the interrupt-service routine.

This delay is called ***interrupt latency***.

This kind of delay not acceptable in **Real-time processing**.

Interrupt sequence

1. The device raises an interrupt request(INTR)

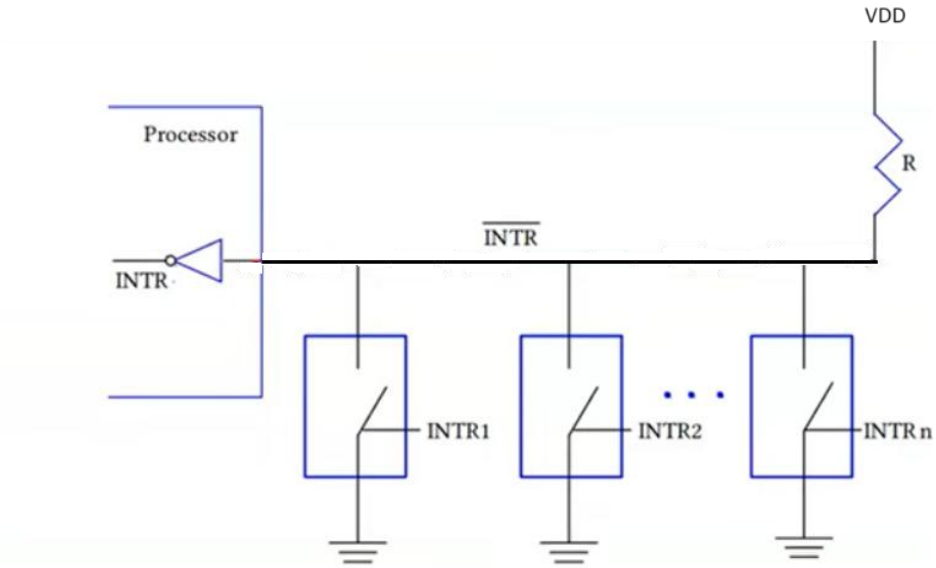
2. The processor COMPLETES the execution of the current instruction and the program currently being executed is interrupted and saves the contents of the PC and status/flag register.

[3. Interrupts are disabled by clearing the IF bit in the status/flag register to 0.]

4. The action requested by the interrupt is performed by the interrupt-service routine, during which time the device is informed that its request has been recognized, and in response, it deactivates the interrupt-request signal.

5. Upon completion of the interrupt-service routine, the saved contents of the PC and Status registers are restored (enabling interrupts by setting the IF bit to 1), and execution of the interrupted program is resumed.

Interrupt Hardware



$INTR = INTR_1 + INTR_2 + \ldots + INTR_n$

INTR is active low signal.

Enabling and Disabling The Interrupt

A single interrupt request from one device by activating the interrupt-request signal, remains activated until it learns that the processor has accepted its request. This means that the interrupt-request signal will be active during execution of the interrupt-service routine.

It is essential to ensure that this active request signal does not lead to successive interruptions, causing the system to enter an infinite loop from which it cannot recover.

In some situations interrupt have to be ignored e.g. PRINT interrupt from the printer cannot be serviced by the the processor if COMPUTE is not ready with text to print.

First Possibility

Ignore the Interrupt Request until the completion of the current ISR

The processor hardware ignores the interrupt request line until the execution of the first instruction of the ISR has been completed.

Ignore the Interrupt Request until the completion of the current ISR
i.e. first instruction of ISR is Interrupt Disable and last instruction is Interrupt Enable

```
ISR
DI
.....
......
.....
EI
IRET
```

Second Possibility:

The processor automatically disables interrupts before starting the ISR.

On entry to an ISR

1. Processor first saves the contents of the program counter (PC) and the processor status (PS) register with IF=1 on stack
2. Automatically disable interrupts before starting the execution of the interrupt-service routine.

On Exit from an ISR

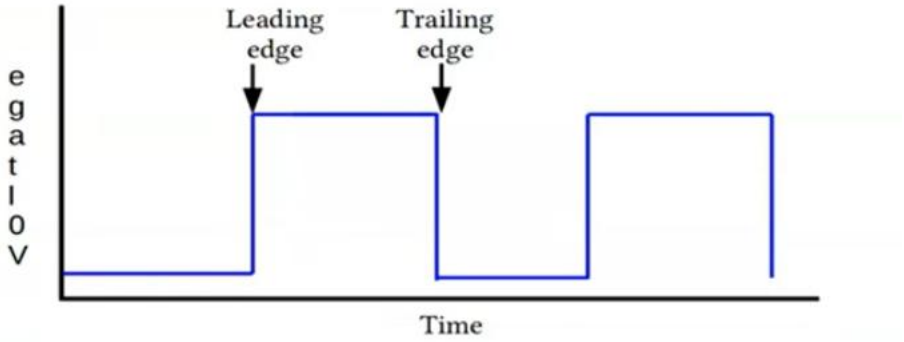
When return from interrupt instruction is executed the contents of PC and PS is popped with IF=1 from stack.

Third Possibility

INTR line must accept only at the leading edge of the signal (Edge triggered line)

Processor receives only one request regardless of how long the line is activated.

So there is no question of multiple interruption or no need of explicit instruction for enable/disable interrupt.



Handling Multiple Requests

When a number of devices can send interrupt requests on a common line INTR connected to the processor, How can the processor determine which device is requesting an interrupt?

Solution: POLLING

When an interrupt request is received it is necessary to identify the particular device that has raised the request.

The information needed to determine whether a device is requesting an interrupt is available in its status register.

When a device raises an interrupt request, it sets one of the bits in it's status register to 1(one).

The simplest way to identify the interrupting device is to have the ISR poll all the I/O devices connected to the bus.

The first device encountered with it's IRQ bit set is the device that should be serviced.

Given that different devices are likely to require different interrupt-services, how can the processor obtain the starting address of the appropriate routine in each case?

Solution: Vectored Interrupt

To reduce the time involved in the polling process, a device requesting an interrupt may identify itself directly to the processor. Then, the processor can immediately start executing the corresponding ISR.

A device requesting an interrupt can identify itself if it has its own interrupt-request signal, or if it can send a special code (like memory address of ISR) to the processor

A commonly used scheme is to allocate permanently an area in the memory to hold the addresses of interrupt-service routines. These addresses are usually referred to as **interrupt vectors**, and they are said to constitute the **interrupt-vector table**.

When an interrupt request arrives, the information provided by the requesting device is used as a pointer into the interrupt-vector table, and the address in the corresponding interrupt vector is automatically loaded into the program counter.

Some Points

IO devices send interrupt vector code over the data bus.

When a device sends an interrupt request, the processor may not be ready to receive the interrupt-vector code.

May be interrupt is disabled by the processor at that moment.
or, the data bus may be used by the processor to complete the execution of the current instruction.

So, when the processor is ready to receive the interrupt vector code, it sends the INTA signal to the device.

Only after receiving the INTA signal, the device places the interrupt vector code on the data bus.

Interrupt Nesting
If request comes from more than one device

Sometimes some device need immediate response from processor e.g. system clock, real time system

Interrupt requests from higher-priority devices will be accepted.

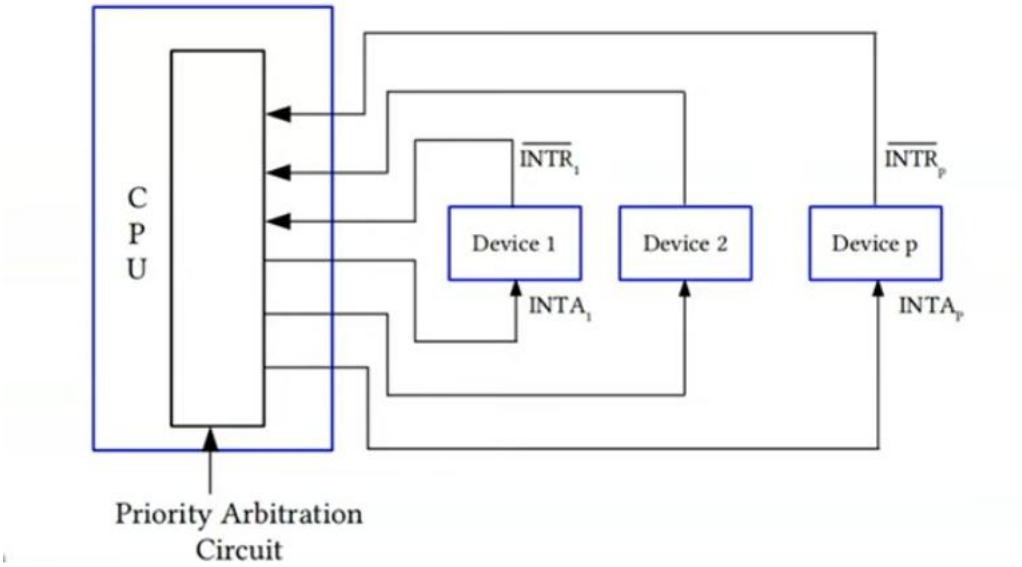
This can be resolved by using **Priority Based Interrupt**.
It uses Multiple-level priority

Interrupt requests will be accepted from some devices but not from others, depending upon the device's priority. To implement this scheme, we can assign a priority level to the processor that can be changed under program control.

The processor's priority can be encoded in a few bits of the processor status register.

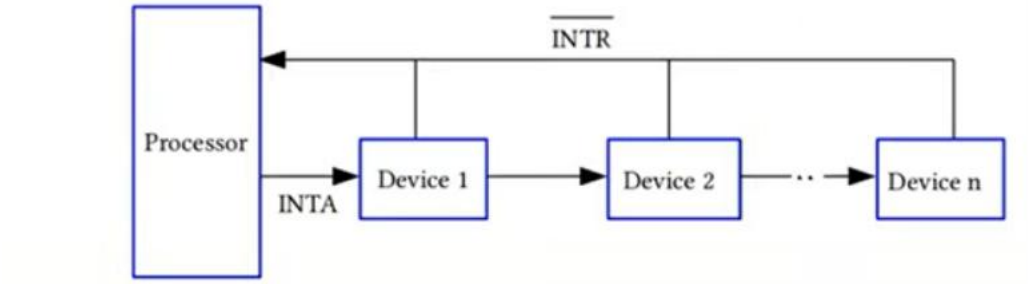
Multiple priority scheme implemented using individual INTR and INTA

Priority arbitration circuit: A logic circuit which combines all interrupts but allows only the highest-priority request.

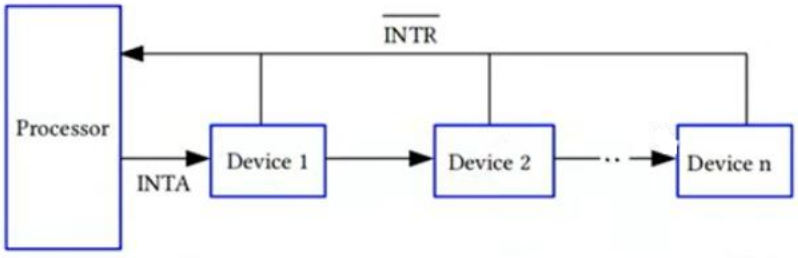


Two types of Priority Scheme: Fixed Priority: Lower number indicates higher priority. Rotating Priority: Once one device is serviced, that device will become the lowest priority device and the device next in sequence will become the highest priority device.

Daisy Chain



When IR is active on INTR line then INTA is sent to Device 1 and passes to device 2 if it does not require any service.



INTR line is common to all the devices.

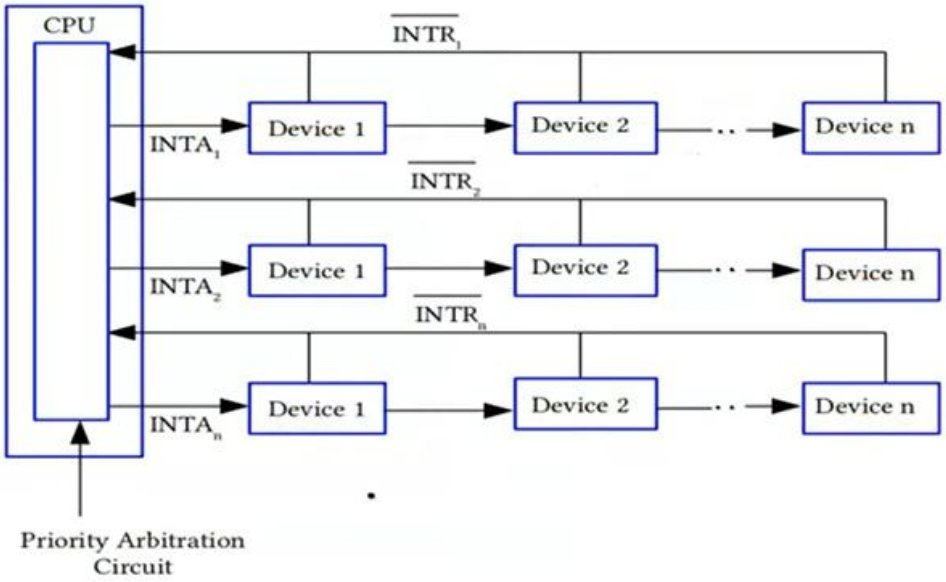
INTA is connected in daisy chain fashion where INTA signal propagates serially through the devices.

When IR is active on INTR line then INTA is sent to Device 1 and passes to device 2 if it does not require any service.

In daisy chain, the device that is electrically closest to the processor will have the highest priority.

Combining the Multiple-priority and daisy chain.

Where device organized in group and each have different priority level.



Direct Memory Access (DMA)

To transfer large blocks of data at high speed, an alternative approach is used.

A special control unit provided to allow transfer of a block of data directly between an external device and the main memory, without continuous intervention by the processor. This approach is called **DMA**.

DMA transfers are performed by a control circuit that is part of the I/O device interface called **DMA controller**.

For each word transferred, processor provides the memory address and all the bus signals that control data transfer.

Since DMA controller has to transfer blocks of data, the DMA controller must increment the memory address for successive words and keep track of the number of transfers.

Registers used in DMA Operation

DMA controller has number of registers that are accessed by the processor to initiate transfer operations.

Status and Control

Starting Address

Word Count

One register is used for storing the Starting address to/from which the communication will take place.

The word count register is used for storing the COUNT of units to be transferred.

The third register contains status and control flags.

The R/W bit determines the direction of the transfer.

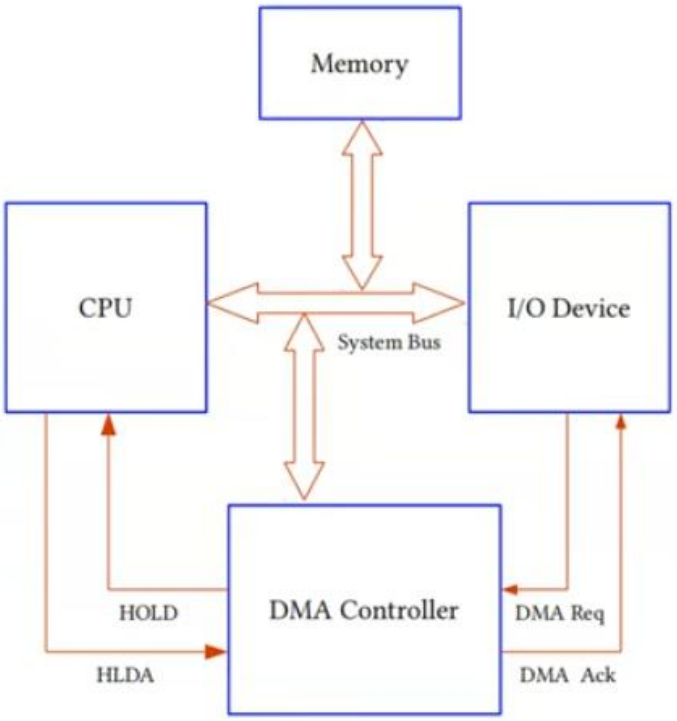
When this bit is set to **1** by a program instruction, the controller performs a read operation, that is, it transfers data from the memory to the I/O device, else write operation.

When the controller has completed transferring a block of data and is ready to receive another command, it sets the Done flag to 1.

Bit 30 is the Interrupt-enable flag, IE. When this flag is set to 1, it causes the controller to raise an interrupt after it has completed transferring a block of data.

The controller sets the IRQ bit to 1 when it has requested an interrupt.

DMA Operation



DMA Transfer

When transfer of data between peripheral and memory is needed, the peripheral places it's request to the DMA controller attached to it.

The DMA controller sends Bus Request(HOLD) signal to the CPU for releasing the control of the system bus.

The CPU responds to it by **terminating** the current instruction execution.

The CPU initializes the DMA controller by sending the following information through the data bus.

1. The starting address of the memory block where the data are available (for read) or where the data need to be stored (for write operation).
2. The word count which is the number of bytes in the memory block.
3. Control to specify mode of transfer such as read/ write.

Then the CPU sends the Bus Grant (HLDA) signal to the DMAC, and releases the control on the system bus.

After receiving the HLDA, the DMA Controller informs the peripheral and starts the operation.

It continues to transfer data between memory and peripheral unit until the entire block is transferred.

For each transfer, memory address is incremented and the word count is decremented.

When the count becomes zero, no more transfer take place.

After the transfer is over, the DMA Controller sends an interrupt request to the processor.

In response to this interrupts, processor takes back the control on the system bus.

Modes of DMA Transfer: Cycle Stealing and Burst Mode

In **Cycle Stealing,** the DMA Controller takes the control of buses from CPU for transferring one word of data in one cycle and returns the control to the CPU.

Alternatively, the DMA controller may be given exclusive access to the main memory to transfer a block of data without interruption. This is known as **block or burst mode.**