Input-Output Architecture

I. Introduction -- The input and output (I/O) subsystem of a computer should provide an efficient mode of communication between the CPU and the external world. A computer system serves no useful purpose without the ability to receive/transmit information from/to external devices. The I/O subsystem is analogous to the eyes, ears, hands, arms, etc. of the physical body—the CPU being the brain. The brain has a meaningless existence by itself. It must work in complete coordination with all the other members. Only when all the parts are functioning in complete coordination with the others do you have an efficient system.

Someone once stated, "Strictly speaking, computers only execute instructions," and to this, we fully agree. Yet, what are the instructions doing? They are arranged in such a way such that they accomplish a particular task, called a program. Programs are processing data and generating particular results. Where did the data that we are to process originally come from? What do we do with the results? Programs and data must be ***entered*** into the memory subsystem for processing, and results obtained from computations must be ***recorded*** or ***displayed***.

Peripheral Devices -- Input and output devices connected to the CPU and memory are referred to as *peripherals*. Typical input peripherals found in desktop computers are keyboards, mice, scanners, microphones, CD-ROM's and DVD's. Typical output peripherals are the video subsystem, printers and soundcard with speakers. Peripheral devices that serve as both input and output include hard drives, floppy drives, CD-RW drives. In addition, computer systems are increasingly dependent upon communication peripherals such as modems and networking cards. In fact, for years major computer manufacturers have been saying, "the network **is** the system."

The I/O architecture of a computer is a function of its intended application. The majority of computers being used in the "new millenium" do not have the peripheral devices required by a desktop computer. The most significant number of microprocessors are used in "*embedded applications*," such as automobiles, digital cell phones, digital cameras, personal digital assistants, etc. etc. etc. The onslaught of the "digital revolution" is centered around these embedded systems. The phrase, "*ubiquitous computing*" comes from the saturation of our society with these products. These embedded systems have varying kinds of I/O specifications that include analog-to-digital converters (A/D), digital to analog (D/A) converters, and other data acquisition and control components. Nevertheless, regardless of the type of application, there are certain fundamental concepts and principals used for transferring data to/from the peripherals and basic interface techniques (protocols) that are used in I/O architectures. These are our main focus in this chapter.

I/O Bandwidth (Data Rates) -- In our discussion of memory architecture, we saw that it does not make sense to have a high speed CPU (e.g. 1 Gigahertz clock frequency) if it is not "matched" with an equally efficient memory subsystem. Similarly, why "pay" for a high performance CPU that is frequently "tapping its foot" while it waits for a sub-performance I/O subsystem?

Recall that, due to varying access times of memory components, we layered memory in a "hierarchy." With respect to I/O architecture, the varying *bandwidth* (frequency of data transfer) of peripheral devices is much broader than memory devices. Compare the bandwidth of input from a user at a keyboard with the bandwidth of the hard drive. The great degree of differences seen in the bandwidth between peripheral devices must (somehow) be "invisible" to the CPU. The I/O architecture plays this crucial role. It must interface all of the different "*personalities*" found in the peripheral community with the CPU. Similar to the memory subsystem, the I/O subsystem must also "feed the beast." The complex challenges presented to the I/O architect are at least as great as those encountered by the memory architect.

I/O Bandwidth (cont.)

To give some perspective on the diversity of peripheral bandwidth, the table below lists typical peripherals found in desktop systems. It is also significant to note the "partner" (i.e. who is on the other end) with which a peripheral is communicating.

Device	***Type***	***Partner***	***Bandwidth (KB/sec)***
Keyboard	Input	Human	0.01
Mouse	Input	Human	0.02
Voice input	Input	Human	0.02
Scanner	Input	Human	400.00
Voice output	Output	Human	0.60
Line Printer	Output	Human	1.00
Laser Printer	Output	Human	200.00
Graphics Display	Output	Human	60,000.00
Modem	Input or Output	Machine	5.00 - 500.00
Network/LAN	Input or Output	Machine	500.00 - 6,000.00
Floppy Disk	Storage	Machine	100.00
Optical Disk	Storage	Machine	1,000.00
Magnetic Tape	Storage	Machine	2,000.00
Magnetic Disk	Storage	Machine	2,000.00 - 10,000.00

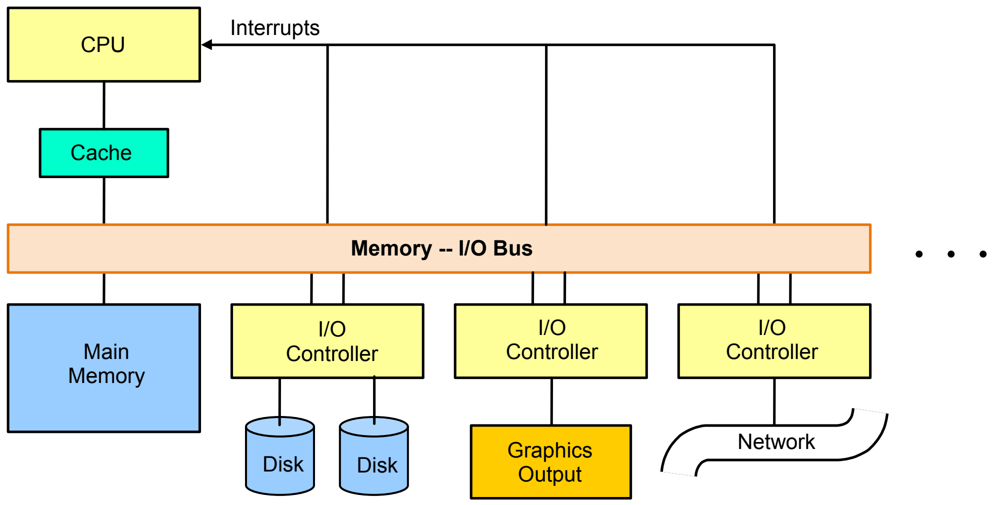
II. I/O Interface—Peripherals connected to a computer need special communication links to interface them with the CPU. The purpose of these links is to resolve differences in the properties of the CPU and memory and the properties of each peripheral. The major differences are listed below:

- Peripherals are usually electromechanical devices whose manner of operation is different from that of the CPU and memory, which are electronic devices.
- The bandwidth of peripherals is usually different from the bandwidth of the CPU and memory. Consequently, a synchronization mechanism may be needed.
- Data codes and formats of the peripherals tend to differ from the word formats in the CPU and memory. Thus, data from peripherals may have to be "repackaged" for efficiency of transfer to the CPU and memory.
- The operating modes of peripherals differ from each other, and each must be controlled in a way that does not disturb the operation of other peripherals connected to the CPU and memory.

To resolve these differences, I/O architects use special hardware components between the CPU and peripherals to supervise and synchronize all input and output transfers. These components are called *interface units* because they interface between the system bus of the CPU and the peripheral devices. In addition, each peripheral has its own controller, built around an embedded processor, to supervise the operations of the particular mechanism of that peripheral. For example, a hard disk has a complex processor with its own local memory that communicates with the electronics of the physical drive mechanism, directing it to specific surfaces, tracks and sectors from reading/writing data. This embedded controller on the hard disk must also communicate with the IDE controller on the motherboard of the desktop system.

II. I/O Interface (cont.)

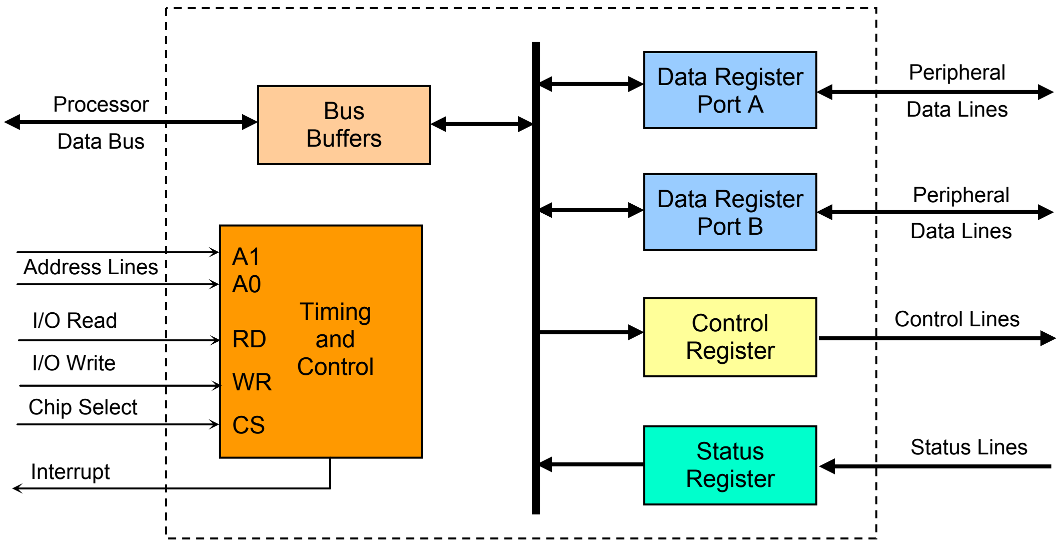
The typical structure between the CPU and several peripherals is illustrated in the diagram below:



Each peripheral has an interface unit associated with it. The common bus between from the CPU and memory is attached to all the peripheral interface units (I/O controllers). To communicate with a particular device, a program issues an input or output instruction. The CPU will place a device address on the address bus. Each I/O controller attached to the bus has an address decoder that monitors the address lines. Each I/O controller is also "programmed" to respond any addresses within a certain range. When the interface detects its own address, it activates the path between the system bus and the device that it controls, similar to a "valve" that has been opened. All other peripherals not corresponding to the address on the bus are disabled by their respective decode logic since *one-and-only-one* peripheral can be "attached" to the bus at a moment in time. At the same time that the address is made available on the address lines, the CPU provides a function code (supplied by the original I/O instruction) on the control/data lines. The selected I/O controller responds to the function code and proceeds to "*execute*" it. If data is to be transferred, the I/O controller communicates with both the device and the CPU data bus to synchronize the transfer.

Example I/O Interface—A typical I/O Interface is shown in the block diagram below, consisting of data registers (at least one), a control register, and a status register. The interface is analogous to an "interpreter" that can speak both the language of the CPU and the language of the particular peripheral it interfaces with.

CS	A1	A0	Register
0	x	x	None
1	0	0	Port A
1	0	1	Port B
1	1	0	Control
1	1	1	Status



Example I/O Interface (cont.)

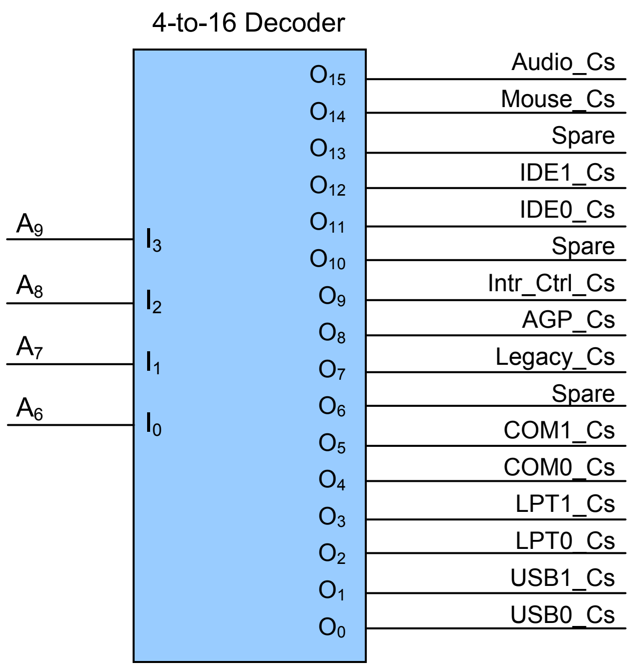
Functioning as an interpreter, the I/O Interface has all the appropriate inputs and outputs on the processor side to communicate in the hardware language of the specific processor to which it is attached. The Data lines, Address lines, I/O Read, I/O Write, Chip Select, and Interrupt lines serve this purpose. These particular signals are inappropriate for communicating directly with the peripheral. Communication with the peripheral is accomplished through the I/O data, control and status lines.

Data to be transferred to a peripheral will be written to the I/O Interface (typically with an output instruction), where the address specifies an *output port register*. Input to be received from a peripheral can be read from the I/O Interface (typically with an input instruction), where the address specifies an *input port register*.

The *control register* is used to initialize and configure the I/O Interface for certain operational modes. For example, a serial port may be configured to transfer 8 data bits, check for odd parity and one stop bit, generate interrupts upon the receipt of a byte, but not generate interrupts upon the transmission of a byte. The serial port "device driver" does this by setting/clearing specific bits within the control register.

The bits within the *status register* are used by the serial port "device driver" to indicate status and error conditions related to the interface with the peripheral. For example, a serial port will typically have a status bit to indicate data has been received, and another to indicate that data has been transmitted. In addition, there are status bits to indicate a "parity" error, a "framing" error, or a "receive overrun" error.

The chip select signal is generated by system decode logic based upon the address on the address bus, and the assertion of the I/O Read or I/O Write signal. As indicated earlier, each I/O controller is "programmed" to respond any addresses within a certain range. The decoding of the address bus is done directly by the decode logic, which then generates the chip select for the appropriate I/O controller. An example of the I/O decode logic, with respective address mapping, is shown below:

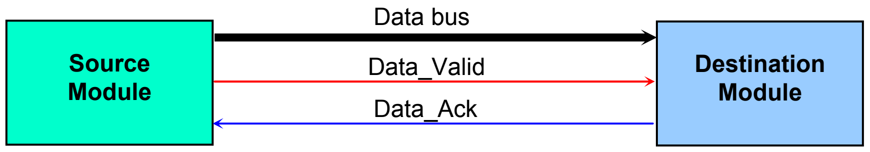


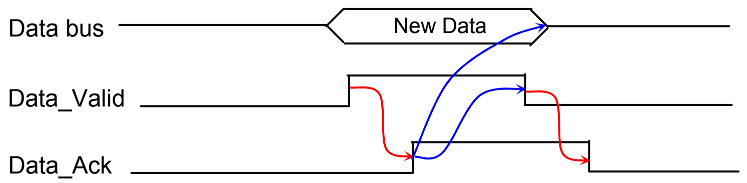
A_9	A_8	A_7	A_6	Base I/O Address	I/O Device
1	1	1	1	03C0h	Audio_Cs
1	1	1	0	0380h	Mouse_Cs
1	1	0	1	0340h	Spare
1	1	0	0	0300h	IDE1_Cs
1	0	1	1	02C0h	IDE0_Cs
1	0	1	0	0280h	Spare
1	0	0	1	0240h	Intr_Ctrl_Cs
1	0	0	0	0200h	AGP_Cs
0	1	1	1	01C0h	Legacy_Cs
0	1	1	0	0180h	Spare
0	1	0	1	0140h	COM1_Cs
0	1	0	0	0100h	COM0_Cs
0	0	1	1	00C0h	LPT1_Cs
0	0	1	0	0080h	LPT0_Cs
0	0	0	1	0040h	USB1_Cs
0	0	0	0	0000h	USB0 Cs

Note that addresses in the table are the "Base I/O Addresses" for each I/O device. Since address lines A_5 to A_0 are not used to select the various I/O devices, they are available as address inputs to the different I/O Controllers, allowing access to as many as 64 registers per I/O Interface.

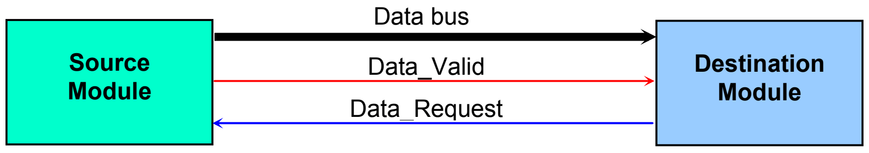
III. Basic Data Communications—The CPU, I/O Interface and I/O devices are likely to have different clocks that are not synchronized with each other. Thus, these digital modules are said to be *asynchronous* with respect to each other. Asynchronous data transfer between two independent modules requires either that (1) control signals be transmitted between the two units to indicate the time at which data is being transmitted, or (2) an asynchronous protocol, establishing some rules of communication. Two methods of performing data transfer using control signals is called "*handshaking*" and "*strobing*."

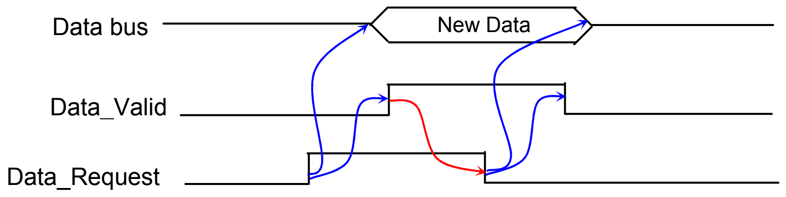
Handshaking—The handshaking method uses two control signals between the source and destination modules. The basic principle of a two-signal handshaking procedure is as follows. One control line from the initiating module is used to request a response from the other module. The second control line, from the other module, is used to reply to the initiating module that a response is occurring. In this way, each module "informs" the other of its status, and the result is that no data will be lost (missed) between the two modules. The diagram below shows the data transfer "protocol" based upon a "source-initiated" transfer. In this scheme, the source module places the data to transfer on the data bus and then asserts Data_Valid. It then waits for the destination module to assert its Data_Ack. This "acknowledgment" is a signal to the source that the destination module has read the data, causing the source to remove the data from the data bus and deassert the Data_Valid signal. The deassertion of Data_Valid causes the deassertion of Data_Ack. This scheme might be called "*supply side*" data transfer, since it is initiated when the source has data.





The diagram below shows the data transfer "protocol" based upon a "destination-initiated" transfer. In this scheme, the destination module asserts its Data_Request signal. When the source module has data to send, it will place the data on the bus and assert its Data_Valid signal. When the destination recognizes the Data_Valid, it will read the data from the bus and deassert its Data_Request. When the source recognizes the deassertion of Data_Request, it will take the data off the bus and deassert Data_Valid. This scheme could be called "*demand side*" data transfer since it is initiated when the destination wants data.

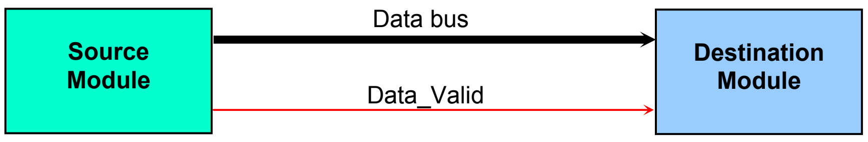


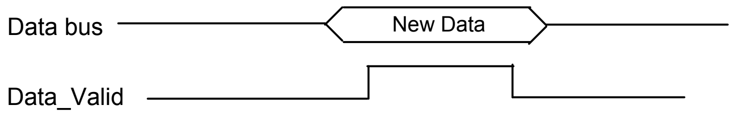


Handshaking (cont.)

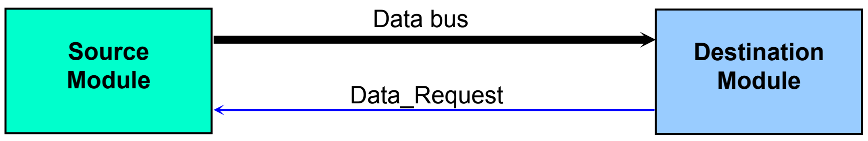
The handshaking scheme provides a high degree of flexibility and reliability because the successful completion of data transfer relies on the active participation—“handshaking”—by both modules. If one of the modules is faulty, the data transfer will not be completed. Such an error can be detected by means of a “time-out” mechanism, which produces an interrupt if the data transfer is not completed within a predetermined time interval. The time-out is implement by a “watch-dog” timer that starts counting time when a module asserts one of its handshaking control signals. If the return handshake does not occur with the given time period, the module assumes an error has occurred. An interrupt can be sent to the CPU so that an interrupt service routine that takes appropriate error recovery action can be executed. Also note that in the handshaking scheme, within the time-out limits, the response of each module to a change in the control signal of the other module can take an arbitrary amount of time, and the transfer will still be successful.

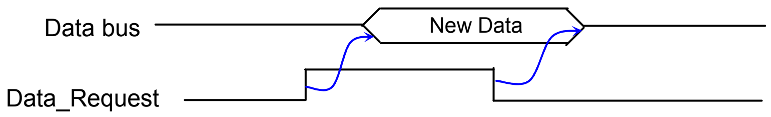
Strobing -- Data transfers using *strobing* only have one control signal between the communicating modules. The diagrams below show the differences between source initiated and destination initiated transfers.





In the source initiated transfer, shown above, the source module places its data on the bus an asserts its Data_Valid. After a "*specified*" amount of time, the source module will deassert Data_Valid and remove the data from the bus. In this scheme, the destination module is "aware" of the specifications of the Data_Valid signal and bears the responsibility to read the data within the specified time. The source module "assumes" the destination module has read the data. Thus, no handshaking occurs. Typically, the destination unit uses one of the "edges" of the Data_Valid signal to clock the data into one of its internal data registers.





In the destination initiated transfer, shown above, the destination module asserts Data_Request. Upon recognition of the request, the source module places the data on the data bus. The destination module expects the data to be on the bus, at worst, a specified amount of time after Data_Request goes from low to high. The destination module will read the data, capturing the data into an internal data register, and will then deasserted the Data_Request. In response to the deassertion of Data_Request, the source module will remove the data from the data bus.

IV. Serial Communications—The transfer of data between two modules may be performed in parallel or serial. In parallel transfer, each bit of the data has its "own path" on a bus, and the data is transmitted "***n***-bits" at a time (where ***n*** is the width of the data bus). In serial data communication, each bit of the data is sent one bit at a time. This method requires only one or two signal lines (e.g. transmit data line and receive data line). Parallel transmission has an obviously faster bandwidth, but requires many wires. Serial transmission is slower, but less expensive, since is requires only one signal per direction. In addition, parallel transmission of data is typically limited to short distances between the two communication modules. In contrast, serial communications typically allows the two communicating modules to be great distances apart (e.g. satellite communication, Internet communications etc.). For the above reasons, serial communications is the most widely used form of data communications used on the earth. Hence, it is well worth our time and energy to familiarize ourselves with some of the basics of serial communications.

We are all beneficiaries of serial communications when accessing the Internet. Typically, "home" access to the Internet is through modems, either analog (e.g. 56K modem) or digital (i.e. DSL, cable or fiber-optic modems). The modems use the telephone lines to connect to remote computers. Since these lines were designed for voice communications, and computers communicate in digital languages, there was the need of a "translator" that could convert from the "digital language" of the computer to the "analog language" used on the phone lines. This is the function of the modem. Modem is actually an acronym for **MO**dulation-**DEM**odulation. The modem receives bytes of digital data and "modulates" them into audio tones for appropriate transmission over telephone lines. Similarly, a modem receives the audio tones from the telephone lines and "demodulates" them into bytes of digital data. There are various modulation schemes, as well as different grades of transmission speeds.

Serial communications can be either synchronous or asynchronous. In *synchronous transmission*, the two modules share a common clock frequency, and bits are transmitted continuously at that frequency. In long distance serial transmission, the transmitter and receiver modules are each driven by different clocks of the same frequency. Synchronization signals are transmitted periodically between the two modules to keep their clock frequencies in step with each other. In *asynchronous transmission*, the binary information is sent only when it is available (i.e. *supply side transfers*), and the line remains "idle" when there is no data to transmit. This is in contrast with synchronous transmission, in which bits must be transmitted continuously to keep the clock frequencies of both modules synchronized.

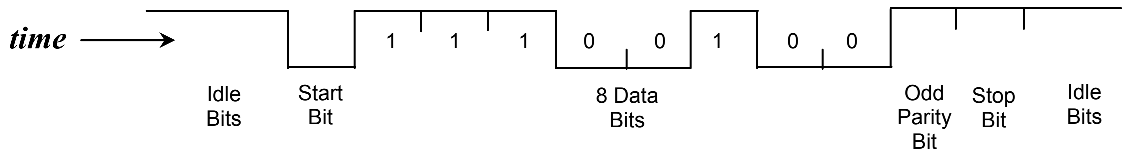
Asynchronous Communications -- One of the most common applications of serial transmission is the communication of one computer with another via modems that are connected through telephone lines. The protocol (i.e. rules of communication) is as follows. Each data to be transmitted consists of a 7 or 8-bit "byte," with additional bits concatenated at the beginning and end of the byte. In essence, these added bits are a "clock" that is encoded into the data being sent. These are known as the *start* and *stop* bits. The actual byte being transmitted is "sandwiched" between the start and stop bits. The protocol used in asynchronous communications is for the transmitter to rest at the logic 1 state when there is no data to be transmitted. This is referred to as an "*idle bit*" or "*marking bit*." When the transmitter has a byte to transmit, the first bit it will send will be the start bit, which is logic 0. Then the transmitter will "shift" out the byte, bit by bit, from the LSB to the MSB on the transmit data line (TxD). After the 7 or 8-bits have been sent, the transmitter may append a parity bit (either even or odd). Whether or not a parity bit is sent is based upon a previous agreement between the transmitter and receiver. Lastly, the transmitter will append the stop bit(s), which is always logic 1. Following the stop bit(s), if there is nothing more to transmit, the transmitter will keep TxD at logic level 1, the "idle state." If there was another byte to transmit, the transmitter could immediately output a start bit, and proceed sending the new byte.

Asynchronous Communications (cont.)

A transmitted byte can be detected by the receiver by applying the transmission protocol. The receiver is always checking the TxD line from the transmitter (its RxD line). When data is not being sent, the line is kept in the logic 1 state. The initiation of reception is detected by a start bit, which is always a logic 0. Thus, the receiver is waiting for the data line to transition from logic 1 to logic 0, i.e. a falling edge. Once a falling edge is detected, it makes sure that the start bit remains at the logic 0 state for "*one bit time*," where a bit time is the reciprocal of the baud rate (number of bits transmitted per second). For example, if the agreed upon transmission rate was 1000 bits per second, one bit time would be one one-thousandth of a second, or one millisecond. If the start bit is shorter than one bit time, the receiver interprets it as "noise" and will continue to wait for another valid start bit.

Once a valid start bit is detected, the receiver will begin to receive the actual data bits—either 7 or 8, depending upon a previous agreement between the two modules. It serially shifts these bits into a shift register. After the data bits have been captured into the shift register, the receiver will use the next bit, the parity bit (if applicable) to check for errors. If a parity error is detected, the receiver will set the appropriate bit in the status register, indicating a parity error. The receiver will then check for the stop bit(s). If a stop bit, the final bit at logic 1, was not received when expected, a "framing error" has occurred. If a framing error occurs, the receiver will set another bit in the status register, indicating a loss of synchronization with the transmitter.

By means of this protocol, the transmitter and receiver are synchronized at each byte being transmitted. The falling edge of the start bit is a "clock" that begins the reception of data. The receiver then samples the data line according to the previously agreed upon baud rate, shifting each data bit into the shift register. The receiver also knows the number of bits, to receive (7 or 8) and whether parity is being sent or not, and how many stop bits are being sent (1 or 2). The stop bit(s) allow both the transmitter and receiver a space of one (or two) bits times to resynchronize. Even if there is "clock skew" between the two modules, it is "absorbed" by only having the skew for 10 bits of data. This is typically not long enough to cause synchronization errors between the two modules. An example of asynchronous serial transfer is shown below:



Thus, the data being sent was equivalent to binary 00100111_2, or hexadecimal 27. Recall that the parity bit is used to make the "***total number of 1's***," inclusive of the parity bit, being sent either even or odd. In this case, there were five 1's sent, including the parity bit, so the receiver would not detect a parity error.

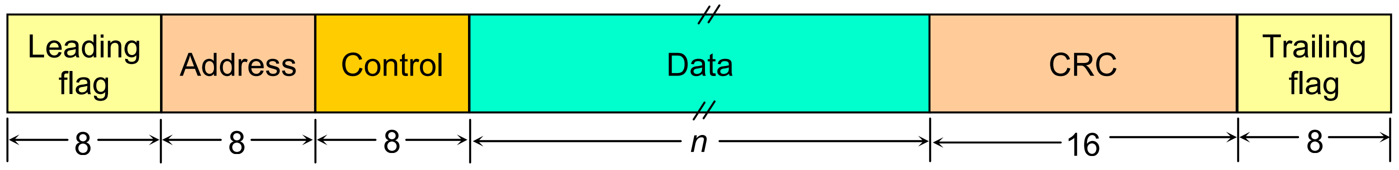
With respect to "bandwidth", consider serial transmission with a baud rate of 56,000 bits per second. Suppose that the transmitter and receiver agreed upon 8 data bits, odd parity, and one stop bit, for a total of 11 bits transmitted per byte. Since the bits are being transmitted at 56,000 bits per second, then one bit time will be 17.86 microseconds. Since there are 11 bits to transfer, it will take 196.43 microseconds per byte. This would be a maximum bandwidth of 5090 bytes per second, assuming no "network traffic" that may slow down the communications. Note that since three of the 11 bits are not actually data being sent, we have a 27% "overhead" for this example of asynchronous data transfer.

With respect to error handling when bytes are received in error, it is typical that the software application using the serial port will handle the errors. A common technique is for the transmitter to send a "block" of data to the receiver. Between transmission of blocks, the receiver sends a status byte back to the transmitter. If any byte in the block was in error, the status byte would indicate such, and the entire block retransmitted.

Synchronous Communications -- Synchronous transmission does not use start or stop bits to frame the bytes being sent. The modems employed in synchronous transmission have internal clocks that are set to the frequency at which bits are being transmitted. For proper operation, it is required that the clocks of the transmitter and receiver modems remain synchronous at all times. The communication line, however, only carries data bits. Frequency synchronization is achieved by the receiving modem from the signal transitions that occur in the data that is received. Any frequency shift that may occur between the transmitter and receiver clocks is continuously adjusted by maintaining the receiver clock at the frequency of the incoming bit stream. In this way, the same data rate is maintained in both modems.

In contrast to asynchronous communications, in which each byte is sent separately, framed by its start and stop bits, synchronous transmission must send a continuous message in order to maintain synchronism. Synchronous formats eliminate the intermittent start/stop bits around each byte and provide bytes that precede and sometimes follow that user data stream. The preliminary bytes are usually called *synchronization (sync) bytes, flags, or preambles*. Their principal function is to alert the receiver to an incoming frame (block) and provide a means to determine when all the bytes in the frame have been transmitted. This process is referred to as *framing*.

A common synchronous serial transmission protocol is High-level Data Link Control (HDLC). The frame format for HDLC is shown in the diagram below. The leading and trailing flags are used to synchronize and delimit the frames. The address and control fields specify the address of the destination and the function of the frame. The *cyclic redundancy check* (CRC) is used to catch errors in the transmission of a frame.



Frame Format for High-Level Data Link Control Protocol

Asynchronous transmission is cost effective for low-speed and low-volume transmissions such as those from keyboards or mice in a desktop computer system. For larger volumes, synchronous transmission is more efficient. To illustrate, let us assume a 1Kbyte (1024) block is to be transmitted. The asynchronous protocol will have 8 data bits, no parity, and one stop bit. If the HDLC synchronous protocol is used, the following comparison can be made:

ASYNCHRONOUS OVERHEAD	SYNCHRONOUS (HDLC) OVERHEAD
2 (start & stop) / 8 = .25	(1) 6 (bytes of control overhead) * 8 bits = 48 bits
	(2) 1024 (bytes of data) * 8 bits = 8192 bits
	(3) 48 / 8192 = 0.0059

As these figures show, asynchronous transmission has a 25% overhead, while synchronous transmission of the same data had only a 0.59% overhead. The decrease overhead of synchronous formats must be weighed against the more expensive circuitry necessary to support it.

The topic of synchronous data link protocols is an enormous one, with many textbooks written to address all the advantages and disadvantages of each. It is beyond the scope of these pages to go any further than our present discussion.

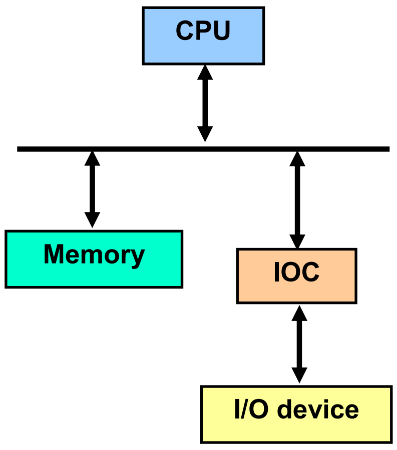
V. I/O Data Transfer Techniques

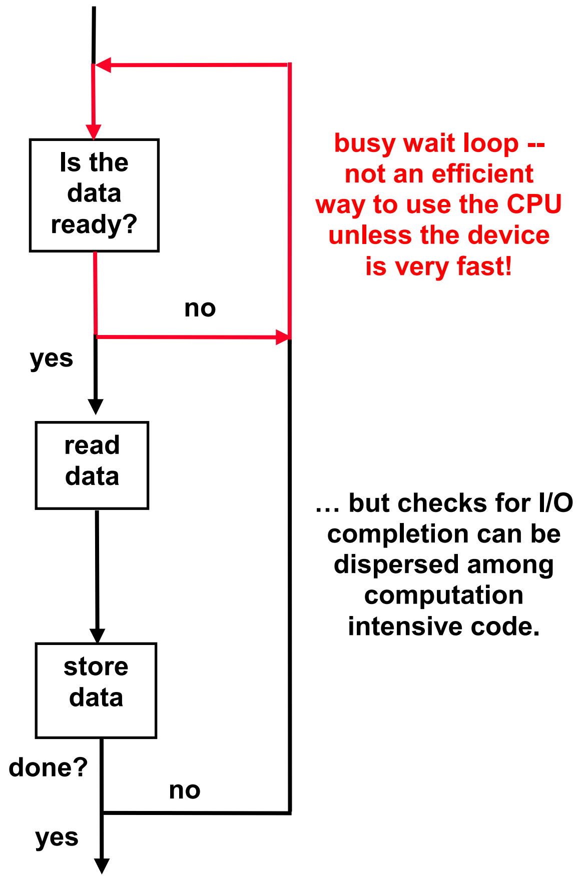
Data that is *transferred from* an I/O device is usually stored in main memory to be processed by an application. Data *transferred to* an I/O device usually comes from main memory. Reading or writing to an I/O device is done by executing an I/O instruction. Although the I/O instructions typically use CPU registers as the source or destination of the transfers, the intended source or destination is memory. When using I/O instructions, the CPU becomes an "*intermediary*" between I/O and memory. Thus, other techniques for I/O data transfers have been developed. Data transfers to and from peripherals are generally implemented by using one of these four modes:

- Program Controlled I/O (a.k.a. Polling)
- Interrupt Driven I/O
- Direct Memory Access (DMA)
- I/O Processors

We will discuss each of these approaches to I/O transfers. It should be noted that a computer system typically has more than one of these modes available to the operating system and applications.

A. Program Controlled I/O—This approach to data transfer is the simplest and least expensive. I/O transfers take place under program control using I/O instructions and loops called "polling loops," where the program "waits" for a peripheral status flag to become asserted (or deasserted) before it continues to execute further I/O. The diagram below illustrates the process. As is noted in the diagram, this is inefficient with respect to the CPU doing other tasks, since once it enters a "*busy wait loop*," it will not be doing any other processing. However, there may be many systems, or particular instances in a system, where this is allowable.

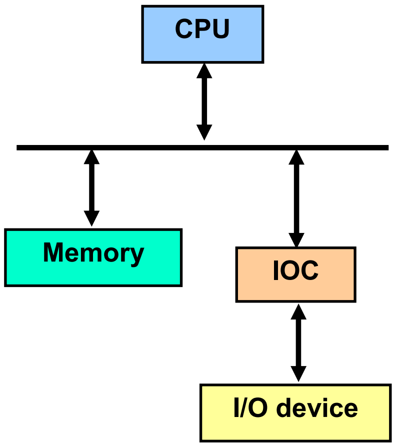


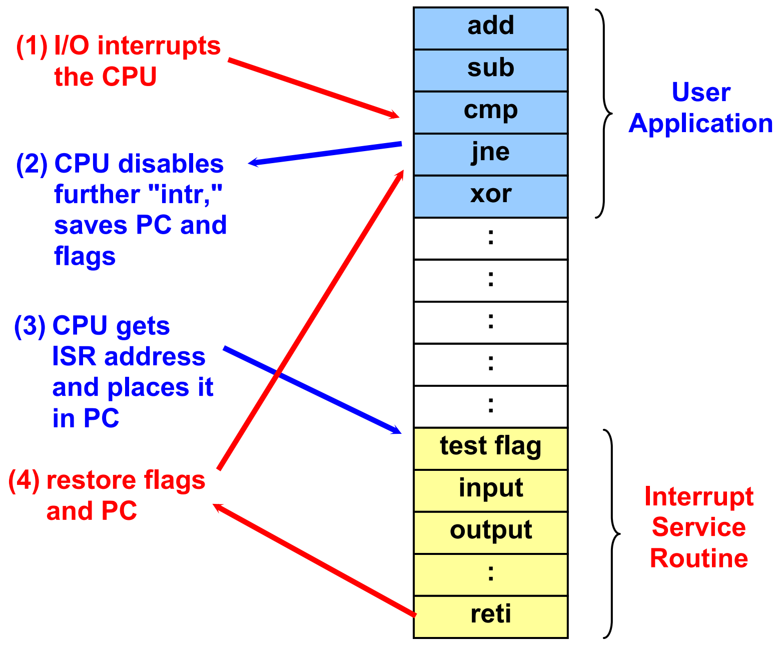

Is the data ready?
no
yes
read data
store data
done?
no
yes
busy wait loop -- not an efficient way to use the CPU unless the device is very fast!
… but checks for I/O completion can be dispersed among computation intensive code.

Polling: Program Controlled I/O

- **Advantage**
  Simple: the processor is totally in control and does all the work
- **Disadvantage**
  Polling overhead can consume a tremendous of CPU time

B. Interrupt Driven I/O—Polling loops can be avoided by using external interrupts. In this approach, the I/O interface is configured to generate an interrupt signal to the CPU when it needs service, typically for an application to read its data register when full or write to its data register when empty. This allows the CPU to continue executing the instructions for other applications as the I/O is done in the "background." The I/O interface monitors the peripheral device. When the I/O interface determines that a data transfer should take place, it generates the interrupt request to the CPU. Upon detecting the interrupt request, the CPU momentarily stops the task it is performing, branches to an appropriate interrupt service routine to handle the cause of the interrupt, and then returns to the original task. The procedure is diagrammed below:




(1) I/O interrupts the CPU
(2) CPU disables further "intr," saves PC and flags
(3) CPU gets ISR address and places it in PC
(4) restore flags and PC
add
sub
cmp
jne
xor
:
:
:
:
:
test flag
input
output
:
reti
User Application
Interrupt Service Routine

Interrupt Driven I/O

- **Advantage**
  CPU is only halted during actual I/O transfer

- **Disadvantage -- Special hardware to:**
  1) **cause an interrupt (I/O device)**
  2) **detect an interrupt (CPU)**
  3) **save the "current state" of the CPU, to resume after interrupt**

Exceptions are internal interrupts that occur during the execution of instructions. Examples are division by zero, an illegal opcode fetch, or an illegal memory reference. Thus, exceptions are "synchronous" in nature. The following summarizes some of the differences between exceptions and external interrupts:

I/O interrupts are just like the a CPU exceptions except:

- An I/O interrupt is asynchronous
- Further information needs to be conveyed (i.e. ISR address)

I/O interrupts are asynchronous with respect to instruction execution:

- I/O interrupt is not associated with any instruction
- I/O interrupt does not prevent any instruction from completion
  -- You can pick your own convenient point (i.e. TCU state) to check for interrupts

I/O interrupts are more complicated than exceptions:

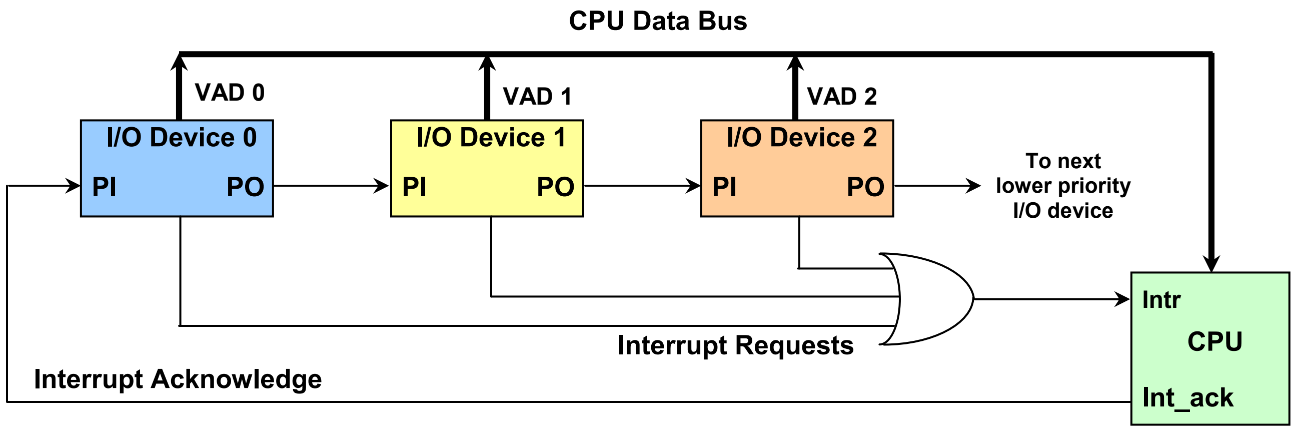
- There is a need to convey the "identity" of the device generating the interrupt
- Interrupt requests can have different urgencies:
  -- Interrupt requests need to be prioritized.

B. Interrupt Driven I/O (cont.)

Since there are occurrences of simultaneous interrupts in a computer system, the system architect must address whether prioritization is necessary or not. A priority system establishes an order among the various interrupting devices. The interrupt subsystem may also determine which interrupt requests are permitted to interrupt the CPU while another interrupt is in the process of being serviced. Prioritization can be done by either hardware or software. We will focus our time on some common hardware schemes.

A hardware priority module functions as an overall manager of the environment of interrupting devices. It receives the interrupts from all the I/O devices, resolves priority, provides a single interrupt to the CPU, and communicates with the CPU to supply necessary addresses (or vectors) for the appropriate interrupt service routine. Although there are numerous ways of designing the "manager" of I/O interrupts, we will discuss two of the more common hardware prioritization schemes: (1) daisy chain priority, and (2) parallel priority.

1. **Daisy Chain Priority**—The *daisy chain* method of establishing priority consists of a serial connection of all the I/O devices that request an interrupt. The I/O device with the highest priority is placed in the first position, followed by I/O devices of priority in descending order, down to the lowest priority device, which is placed last in the chain. This method of connection is illustrated with three I/O devices and the CPU below:

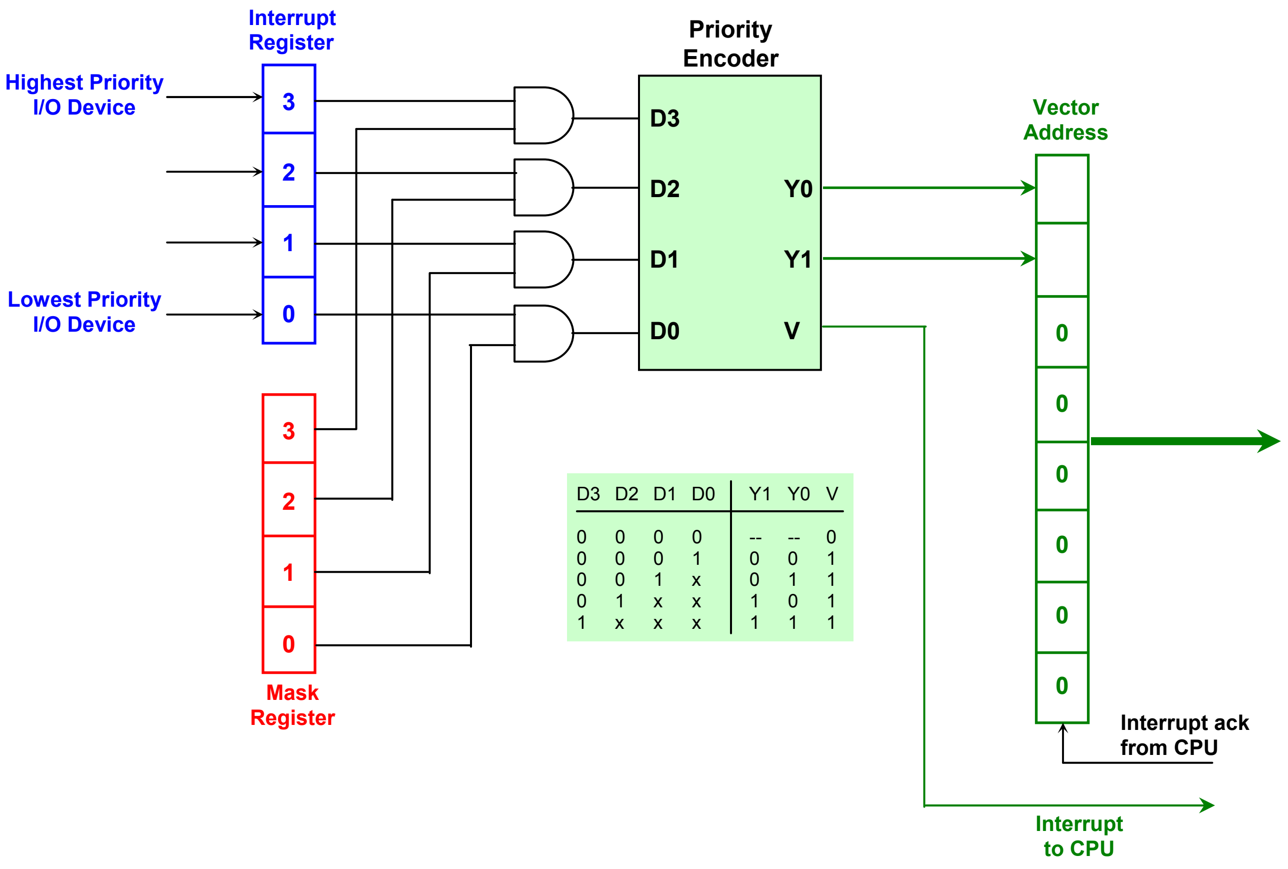


If any I/O device asserts its interrupt request line, the "Intr" to the CPU will be asserted. If the CPU has interrupts enabled, it will respond with an interrupt acknowledge. The "Int_ack" is received by I/O Device 0 at its **PI** (priority in) input. If it was not the source of the interrupt, it will "pass" the signal down the daisy chain to the next I/O device. If I/O Device 0 had a pending interrupt, it will "block" the acknowledgement signal from reaching the next I/O device, and place its own interrupt vector address (**VAD 0**) onto the data bus during the interrupt cycle. An I/O device with 0 on its PI input will generate a 0 on its **PO** (priority out) output to inform the device with the next lower priority that the acknowledge has been blocked. Thus, the device with **PI** = 1 and **PO** = 0 is the one with the highest priority that is requesting an interrupt, and this device places its **VAD** on the data bus.

2. **Parallel Priority**—The *parallel* priority interrupt scheme uses an "*interrupt register*" with bits set separately by the interrupt signal from each I/O device. Priority is established according to the position of the bits in the register. In addition to the interrupt register, there is typically a "*mask register*" to enable or disable each individual I/O interrupt. The mask register allows the OS to "program" whether lower priority interrupts are allowed during higher priority interrupt service, or vice-versa. This allows the "pre-empting" of one interrupt service routine by another interrupt.

2. Parallel Priority (cont).

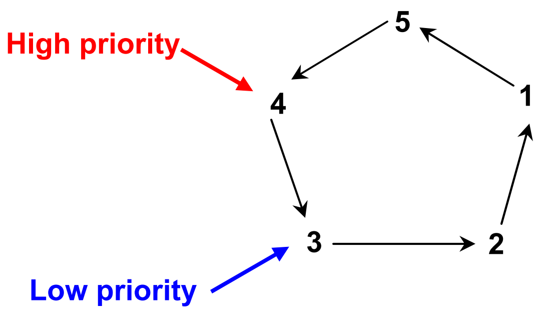
The priority logic for a system with four interrupt sources is shown below. In this diagram, interrupt input 3 has the highest priority and interrupt input 0 has the lowest. Note that the mask register has the same number of bits as the interrupt register. The “heart” of the logic is the priority encoder, which generates a two-bit code based upon the highest priority interrupt that is pending at any point in time. The “**V**” output of the encoder, which becomes the interrupt request to the CPU, will only be asserted if one of the inputs are asserted, otherwise **V** will be deasserted.



D3	D2	D1	D0	Y1	Y0	V
0	0	0	0	--	--	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

In the scheme illustrated above, the output of the encoder is used to form an 8-bit vector address that will go back to the CPU. The vector address register will be “loaded” upon receipt of the interrupt acknowledge from the CPU. The other six bits could be “hard-coded” with any values. Since the six extra bits are filled with 0, the vector address in this case will be 00_h to 03_h, depending on the highest priority interrupt request.

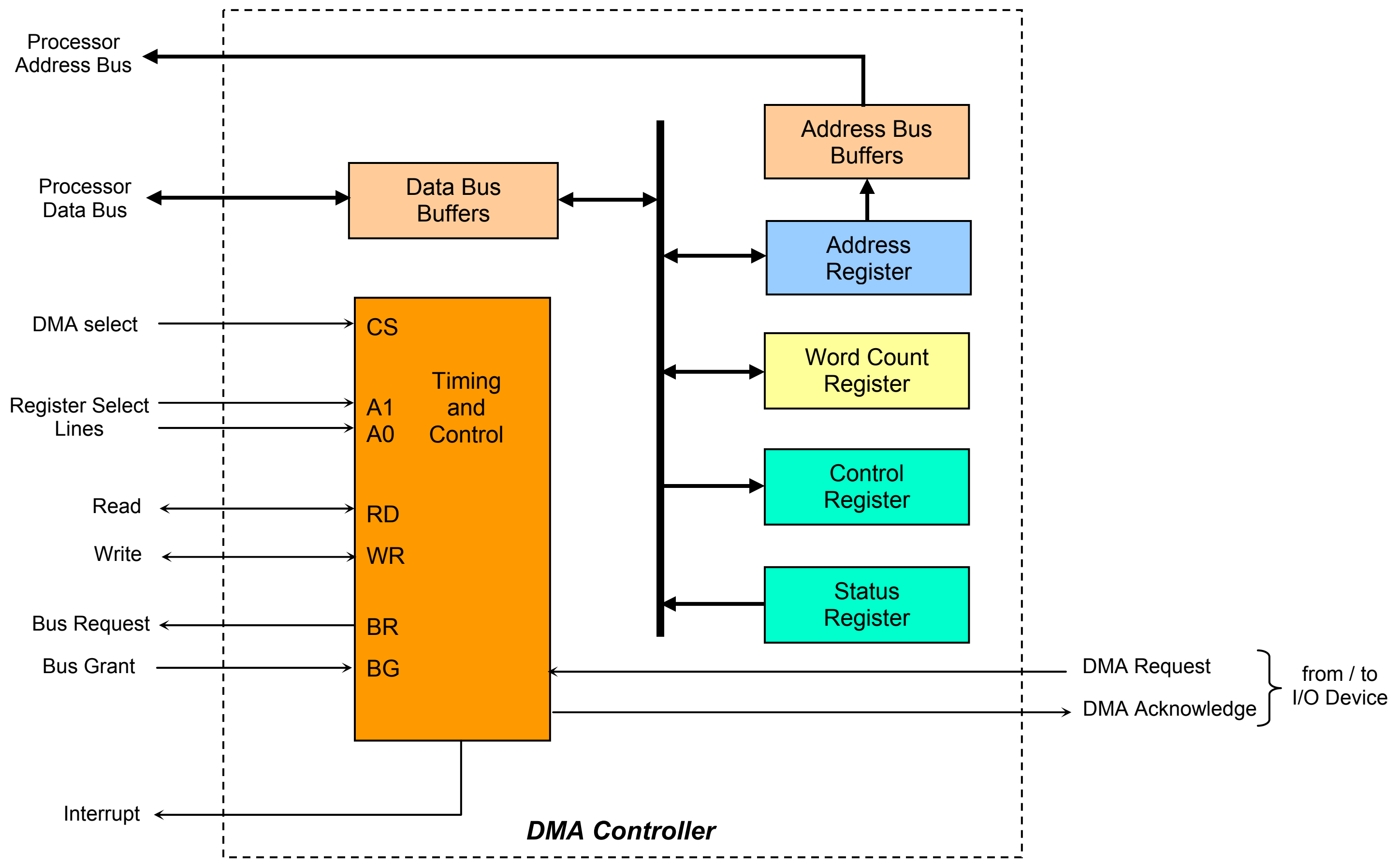
The two hardware priority schemes just discussed are designed with “*static*” priority. In a “*rotating*” priority scheme, the interrupt requests are arranged in a “circle.” There is a pointer which points to the lowest priority interrupt. The priority of each interrupt increases as one travels around the circle, with the highest priority interrupt being adjacent to the lowest. The lowest priority interrupt pointer is changed to point to the interrupt that was just serviced. For example, if interrupt 3 was just serviced, interrupt 4 will have the highest priority. This structure is advantageous when all the interrupts have similar priority and service bandwidth requirements.



Rotating Priority Scheme

C. Direct Memory Access (DMA) -- If the transfer of data between a high-speed I/O device (e.g. the hard disk) and the CPU were to use input and output instructions, a majority of time would be spent doing I/O. Access to the secondary memory is horrendously slow when compared to main memory access, yet large volumes of data are transferred to and from the disk drives. Since I/O instructions involve the CPU being a "intermediary" between the I/O device and memory, they are inefficient for data transfers involving high-speed I/O devices. Removing the CPU "*from the loop*," and letting the high-speed I/O peripherals manage the memory busses directly will relieve the CPU from many I/O operations and allow it to do what it does best -- execute instructions! Using this technique, called direct memory access, a DMA controller takes control of the address, data and control busses to manage the transfer of data between the I/O device and memory -- *without CPU intervention*. Consequently, the CPU is temporarily deprived of access to memory.

The DMA controller may gain control of the system busses in a number of ways. Most processors have been designed with DMA in mind. They have a "*bus request*" (**BR**) input and a "*bus grant*" (**BG**) output for interfacing with "*bus masters*" (DMA controllers, I/O processors, and other CPU's). A bus master is any device that can take responsibility for controlling the system bus to and from memory. The **BR** input is asserted by a bus master (e.g. a DMA controller) when it wants to gain access to the system bus. When the CPU has finished executing its current instruction, it will place the address, data, and control busses in the "high impedance" state, and asserts **BG**, indicating to the bus master that the CPU has disconnected from the system bus. Once it receives a bus grant, the bus master takes control of the system bus and will directly communicate with memory. The transfer of data can be made for an entire block of memory words, suspending operation of the CPU until the entire block is transferred. This is referred to as "*burst mode*." Alternatively, the transfer may be made for only one memory word at a time, between executions of CPU instructions. This is referred to as "*single-cycle transfer*" or "*cycle stealing*." The CPU is delayed for only one memory cycle to allow the DMA controller to steal one memory cycle. A block diagram of a DMA controller is shown below.



C. Direct Memory Access (cont.)

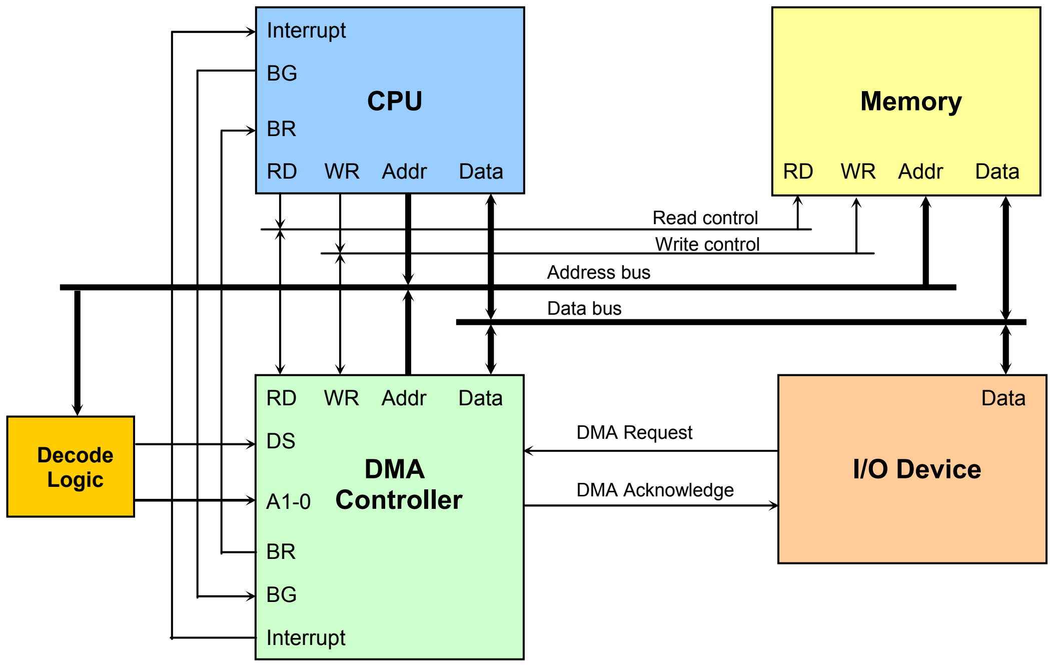
The DMA controller needs the same circuits that the CPU needs to interface with system memory. Thus, each DMA channel must have the following capabilities:

- An address register to specify the memory address to access (with increment/decrement capability).
- A word count register for counting the number of words left to transfer.
- A control register for configuring the mode for a particular DMA transfer.
- A status register for users to be able to read the current status of a DMA transfer.
- Ability to drive the address bus, or place its address outputs in the Hi-Z state when not connected.
- Ability to control the bi-directional data bus.
- Bi-directional READ and WRITE lines.
  -- Inputs when being initialized or accessed for status.
  -- Outputs when performing DMA transfers.

All of the registers in a DMA channel appear to the CPU as I/O interface registers, having their own unique I/O addresses. Thus, the operating system (OS) or user application can write to or read from the DMA registers under program control. After initialization by a program, using I/O instructions, the DMA starts and continues to control the transfer of data between memory and the I/O device until the entire block is transferred. The DMA controller is initialized by sending it the following information:

- The starting address of the memory block for reading or writing.
- The word count, which is the number of words to transfer.
- A "control word" to specify the operational mode for the transfer.
  -- burst mode vs. single-cycle mode
  -- interrupts upon completion vs. no interrupts
  -- 8-bit vs. 16-bit vs. 32-bit, vs. 64-bit transfers
- A control bit to "kick off" the DMA transfer.

Typically, once the DMA controller has begun the DMA transfer, the CPU will stop communicating with it. Once the transfer is finished, the DMA controller will generate an interrupt to the CPU, indicating that the transfer is complete. The position of the DMA controller among the other "players" in the computer system is shown in the diagram below:



C. Direct Memory Access (cont.)

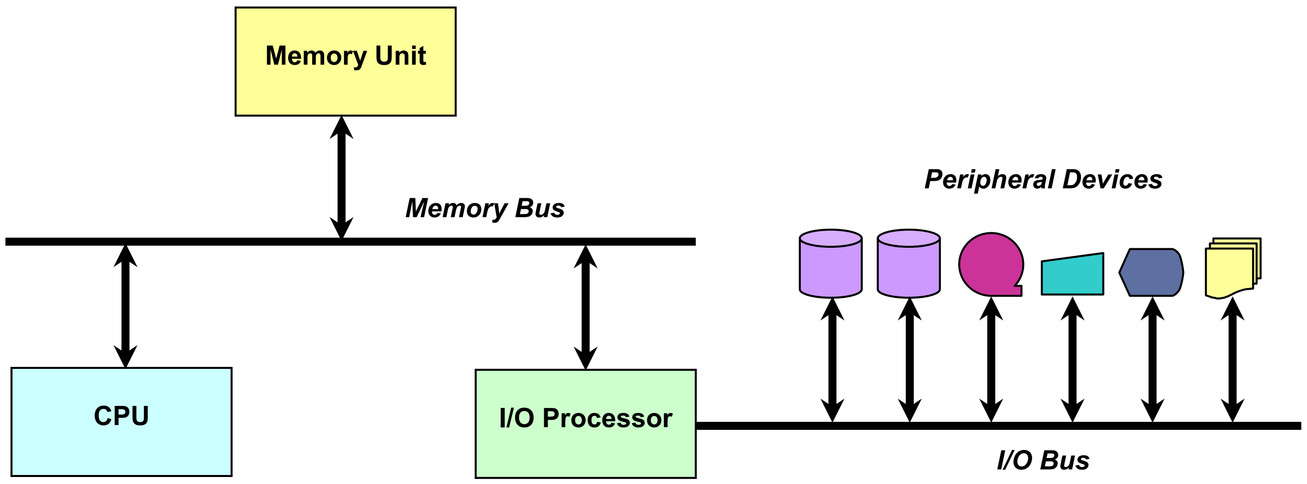
As previously indicated, the CPU communicates with the DMA controller through the address and data busses, as with any I/O device. Based upon the system decode logic, the DMA has it's own range of I/O addresses for which it will be selected. Once the DMA controller receives the "start" control bit, it can begin transferring data between the peripheral device and memory. The sequence of events is as follows:

(1) The peripheral device sends a DMA request to the DMA controller.
(2) The DMA controller asserts the **BR** (bus request) line to the CPU.
(3) The CPU responds by asserting **BG** (bus grant), indicating that the system bus is available.
(4) The DMA controller puts the current address (from the address register) onto the address bus, and asserts either the **RD** or **WR** control signals.
(5) The DMA controller asserts the DMA acknowledge for the peripheral.
(6) In response to the DMA acknowledge, the peripheral either reads the data on the data bus from memory, or puts a word on the data bus to write to memory.

Thus, the DMA module controls the read or write operation and supplies the address for the memory operation. The peripheral unit can communicate with memory through the data bus for a direct transfer of data between the two units while the CPU access to the bus is momentarily disabled.

For each word that is transferred, the DMA module increments (decrements) the address register, and decrements its word counter register. If the word count has not reached zero, the DMA controller will wait for another DMA request from the peripheral. With a high-speed peripheral, a new DMA request will be activated as soon as the previous transfer is complete. This process continues until the entire block is transferred. If the word count reaches zero, the DMA stops any further transfer and deasserts the **BR** (bus request) signal. It also may inform the CPU of the completion by assertion of the interrupt. When the interrupt service routine that handles DMA interrupts is executed, the code could read the status register to determine that the data transfer has been completed.

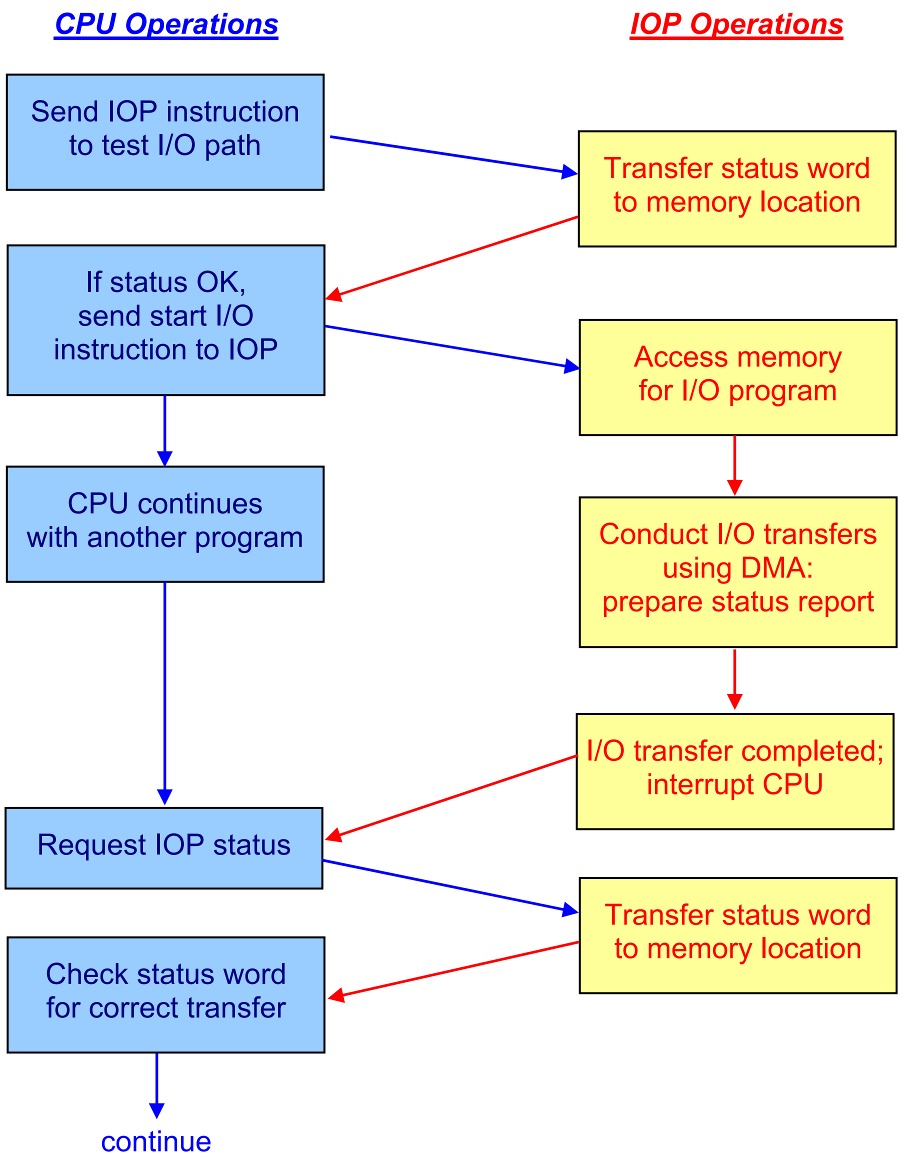
D. Input-Output Processors (IOP's) -- An IOP is similar to a CPU, except that it is designed to handle the details of I/O processing. Unlike the DMA controller, which must be set up entirely by program control, the IOP can fetch and execute its own instructions. IOP instructions are specifically designed to facilitate I/O transfers. In addition, the IOP can perform other processing tasks such as arithmetic, logic, branching, and translation of code. The diagram below illustrates a system with a CPU and IOP.



D. Input-Output Processors (cont.)

In such a “dual processor” system, the IOP provides a path for the transfer of data between various peripheral devices and memory. The CPU is usually assigned the task of initiating the I/O program. From then on, the IOP operates independently of the CPU and continues to transfer data between the external devices and memory. The data formats of the peripherals often differ from those of the CPU. The IOP must structure data from many different sources. For example, it may be necessary to take four bytes from an input device and pack them into one 32-bit word before the transfer to memory.

The communication between the IOP and the peripherals it controls is similar to the program controlled method of transfer. Communication with memory is based upon DMA transfers. Communication between the CPU and IOP depends upon the sophistication of the system. In most systems, the memory acts as a “message center,” where each processor leaves information for the other. IOP instructions (a.k.a. *commands*) are prepared by a user program and stored in memory. The CPU informs the IOP where to find the commands by passing a “pointer” to the IOP (via a “start I/O” instruction). The IOP responds to the CPU prompts by storing “status” words in memory. These status words indicate the state of the IOP and I/O device, such as “device busy with another transfer,” or “device ready for transfer.” The sequence of operations of CPU-IOP communication is illustrated below:


CPU Operations
IOP Operations
Send IOP instruction to test I/O path
Transfer status word to memory location
If status OK, send start I/O instruction to IOP
Access memory for I/O program
CPU continues with another program
Conduct I/O transfers using DMA: prepare status report
I/O transfer completed; interrupt CPU
Request IOP status
Transfer status word to memory location
Check status word for correct transfer
continue

VI. Bus Arbitration—In systems with multiple bus masters (e.g. a CPU and an IOP), there must be some technique for resolving simultaneous requests for use of the bus. Bus arbitration, the process of obtaining access to the system bus, is one of the most important issues in system bus design. The question that must be answered is "How is the bus acquired by a device that requests to use it?"

One of the simplest arrangements to avoid "bus chaos" is to have a master-slave arrangement. In this scheme, only the bus master can control access to the bus—it initiates and controls all bus requests. A slave responds to the read and write requests of the master. Typically, in the simplest arrangement, the CPU is the only bus master and all bus requests are controlled by the CPU. The major drawback is that the CPU is involved in every bus transaction. Hence, architects have overcome this drawback by having systems with "*multiple potential bus masters*." However, multiple bus requests require a "*bus arbiter*" to resolve simultaneous requests for bus access.

Bus arbitration using a bus arbiter has a protocol that is similar to a DMA device requesting control of the system bus by a request to the CPU. However, in this scheme, the CPU is not the bus arbiter; it is simply a bus master among multiple bus masters. A bus master wanting to use the bus will assert its "bus request" signal. The bus master cannot use the bus until it receives a "bus grant" from the arbiter. Once a bus grant has been received, the bus master will use the bus, signaling to the arbiter whenever the bus is free.

Bus arbitration schemes usually try to balance two factors:

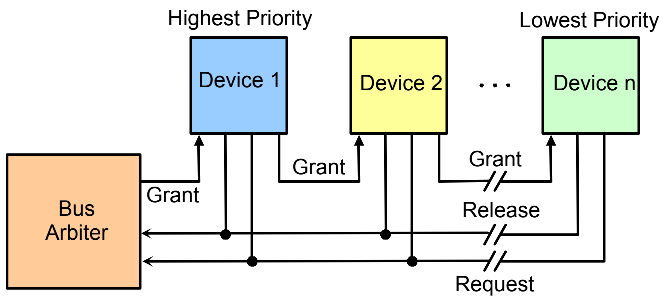
- Bus priority: the highest priority device should be serviced first
- Fairness: Even the lowest priority device should never be completely locked out from the bus

Bus arbitration schemes can be divided into four broad classes. We will cover the first two listed below:

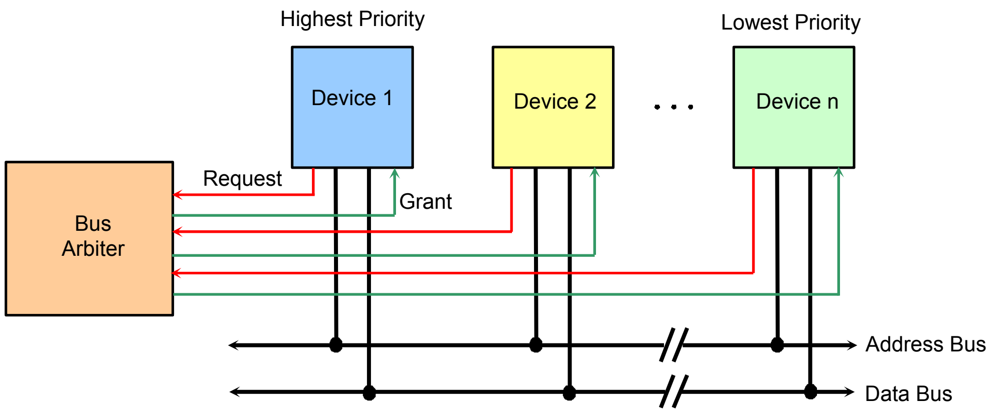
- Daisy chain arbitration: single device with all request lines (see illustration below).
- Centralized, parallel arbitration: see next-next slide (see illustration on next page).
- Distributed arbitration by self-selection: each device wanting the bus places a code indicating its identity on the bus.
- Distributed arbitration by collision detection: Ethernet uses this.

A. The Daisy Chain Bus Arbitration Scheme—This is similar to the daisy chain priority arrangement with respect to multiple interrupt requests. The advantage is its simplicity. Its disadvantages are (1) it cannot assure "fairness," a low priority device may be locked out indefinitely, and (2) the use of the daisy chain grant signal also limits the bus speed.

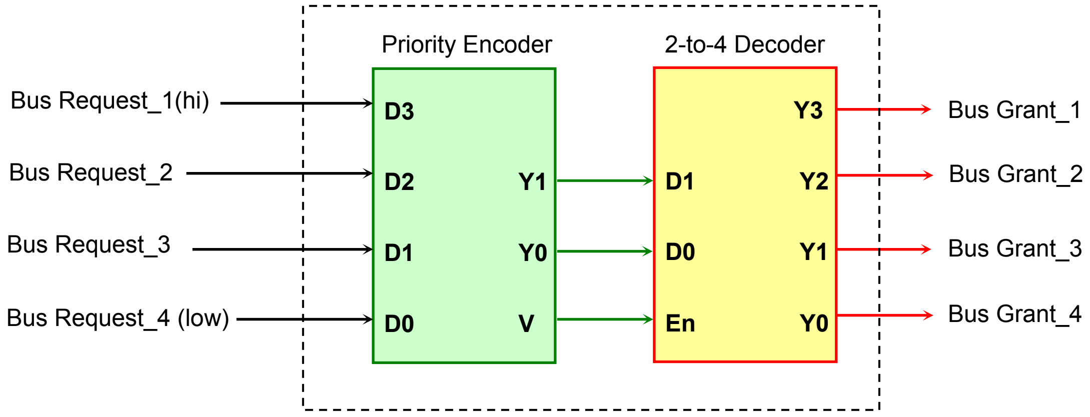
In this scheme, when a bus master needs to use the bus, it will assert its bus request signal. If the bus is free, the arbiter will assert the bus grant signal, which is passed through each bus master that does not have a pending request. Once a bus master with a pending request receives a bus grant, it will take control of the bus. When it is finished, it will assert its "bus release" signal. This is illustrated below:



B. Centralized Parallel Arbitration -- This scheme is similar to the parallel priority technique used with multiple interrupt requests. It is used in essentially all processor-memory busses and in high-speed I/O busses where there are multiple bus masters. This is illustrated in the diagram below:



The bus arbiter logic can be implemented with a priority encoder that can prioritize multiple requests. This is illustrated in the diagram below.



VII. Input-Output Responsibilities of the Operating System—The operating system (OS) acts as the interface between the I/O hardware and the program that requests I/O. There are three characteristics of the I/O system with respect to the OS: (1) the I/O system is shared by multiple programs using the processor; (2) I/O systems often use interrupts to communicate information about I/O operations; and (3) the low-level control of an I/O device is complex, requiring the management of a set of concurrent events.

Operating System Requirements—The OS bears the responsibility of providing protection to shared I/O resources. It must guarantee that a user's program can only access the portions of an I/O device to which the user has rights. The OS must also provide "abstraction" for accessing devices by supplying routines that handle low-level device operation. The OS must also handles the interrupts generated by I/O devices. Moreover, the OS must provide equitable access to the shared I/O resources—all user programs must have equal access to the I/O resources. In addition, the OS must schedule I/O accesses in order to enhance overall system throughput.

Operating System and I/O System Communication Requirements—The OS must be able to prevent the user program from communicating with the I/O devices directly. If user programs could perform I/O directly, protection to the shared I/O resources could not be provided. Thus, three types of communication are required: (1) The OS must be able to give commands to the I/O devices, (2) the I/O device must be able to notify the OS when the I/O device has completed an operation or has encountered an error, and (3) data must be transferred between memory and an I/O device.