CS311 Lecture: The IO System last revised December 1, 2017

Objectives:

1. To discuss issues and options in the design of bus systems.
2. To discuss IO control options: programmed, interrupt-driven, DMA

Materials:

Projectable of illustration of parallel and serial IO using toll booths

I. Introduction

A. A complete computer system consists of three major subsystems:

1. The CPU

2. The memory subsystem

3. The IO subsystem

Having considered the first two of these, we now turn to the third

B. The IO subsystem needs to cope with great diversity, since almost anything can be connected to a computer system as an IO device, and various kinds of devices have widely varying requirements.

1. Some IO devices transfer relatively small quantities of information at a time, while others transfer large quantities of information at a time.

Example: mouse button versus a disk block

2. Data rates vary.

Example: a keyboard typically transfers only a few bytes per second, even at maximum typing speed; a 1 Gbit ethernet interface may transfer over 100MB per second.

3. Some IO devices require the system to respond to a request for service within a short period of time, or there may be undesirable consequences such as data loss or worse

   Example: slow response to mouse clicks or key presses is annoying to a user, but a delay of a small fraction of a second will not be noticeable. (Such a delay could correspond to millions of instructions!)

   Example: a modem typically requires that one received byte be processed before the next byte completely arrives, or the first byte is lost

   Example: a "fly by wire" airplane may require that the system respond to a certain event within a tightly-contrained time period or the plane may crash.

C. The following key issues arise in connection with the IO subsystem.

1. The characteristics of the IO devices themselves

2. The way in which IO devices are connected to the rest of the system

3. The strategies used for interaction between the CPU and the IO devices.

4. Ancillary issues such as data compression and encryption.

D. We will not spend time on the first and last of these issues. Instead, we will focus on interconnection structures and control regimens.

1. A key design issue in building a computer system is how its various components are CONNECTED. The SPEED at which information flows between the various components of the system can be the determining factor in overall system performance. If system performance is limited by the speed of the interconnection system, then technical improvements to the individual components will not result in performance gains.

There are two different measurements which are important in assessing the speed of a given connection system.

a. LATENCY refers to the time it takes to actually perform an operation, which includes both setup time and the time to actually transmit the information.

   i. Certain latency is inherent in the different devices - e.g. the time needed for seek and search on a disk.

   ii. Some latency is associated with the communication system - e.g. the overhead necessary to initiate a connection between two devices.

   For a given operation, the total latency is the sum of these two components.

b. THROUGHPUT refers to the rate at which data is actually transmitted once the flow of data begins.

   i. Throughput may be dictated by the device - how fast can it generate or consume data.

   ii. Throughput may be dictated by the capacity of the communication system.

   For a given operation, the throughput is the MINIMUM of these two components.

   Example: A disk may have a latency of (say) 10 ms, but may have a throughput of (say) 10 million bytes / second. In this case, bus latency may be so small as to be insignificant, and bus throughput may be high enough that the transfer rate is determined solely by the disk.

2. Because the CPU and IO devices often operate at vastly different speeds, we will also discuss ways to SYNCHRONIZE the CPU with external devices.

a. For many devices, the CPU is much faster than the device, so we will discuss various approaches to avoiding having the CPU "cool its heels" while a much slower mechanical device is performing an operation. [This is especially important if the device has high latency].

b. In some cases, an IO Device is so fast that the CPU cannot keep up with it, so we will have to consider how to handle this as well.

E. Interconnection structures can be characterized in several ways.

1. Parallel versus serial.

a. In a PARALLEL CONNECTION, there is one data line for each bit of data being sent over the connection during a given transaction. All the bits of data are transmitted at the same time.

b. In a SERIAL CONNECTION, there is just one data line. The data bits are sent over this wire one after another over time. (Also, there usually aren't separate lines for sending addresses - instead, if an address needs to be sent it is sent serially.)

c. Thus, a parallel connection uses n wires to transmit n bits of data in one unit of time. A serial connection uses 1 wire to transmit n bits of data in n units of time.

PROJECT: Toll booth example

d. Historically, parallel connections were used for higher-speed devices (e.g. disks) and serial connections for slower devices (e.g. keyboards). Today, however, there are several very fast serial bus technologies, and serial connections are often used even in very high speed situations.

Examples: USB, Firewire, Ethernet

e. Generally the CPU connects itself to a parallel bus, with interfaces for various kinds of serial busses connected to it.

2. Simplex versus half duplex versus full duplex.

a. A SIMPLEX connection is one designed for transmitting information only in a single direction.

e.g. the cable connecting to a keyboard or mouse could be simplex - although often a bi-directional connection like USB is used on contemporary systems. (Older dedicated cables were simplex)

b. A HALF-DUPLEX connection is one designed for transmitting information in both directions, but only in one direction at a time. A turn around protocol is used for changing directions. (e.g. the use of the term "over" in radio communication.)

Example: USB

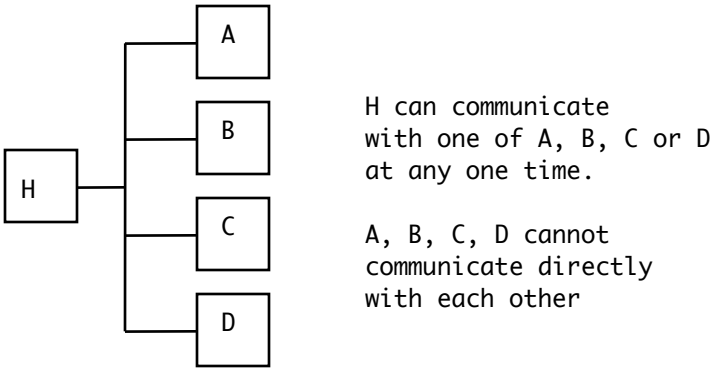
c. A FULL-DUPLEX connection allows transmission of information in both directions simultaneously. (In essence, it is a pair of simplex connections.)

Example: Ethernet connection between a computer and a switch.

d. Parallel connections are usually either simplex or half-duplex (generally the latter) - largely due to the high cost that would result from two wires per bit. Serial connections may be full duplex-requiring one wire (plus often a ground) for each direction, or they may also be half-duplex.

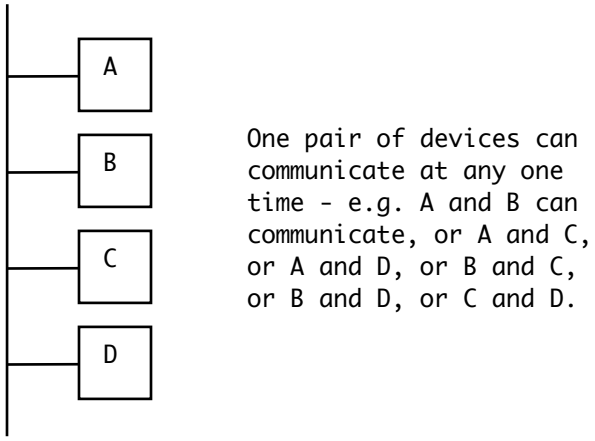
3. In principle, it is possible to use direct connections between pairs of devices; but by far the most commonly-used type of interconnect structure used is a bus structure in which multiple modules are interconnected by a common bus. There are two different configurations possible.

a. There is a single host at the root to the structure, such that any one device can communicate with the root at any time - but devices cannot communicate directly with each other.



Example: USB, Firewire busses

b. There is no special host - any one pair of devices can communicate at any time.



Example: Original Ethernet; Wi-Fi also does something similar (though, strictly speaking, it's not a bus per se)

c. The key characteristic of a bus (of either sort) is that only one pair of devices can communicate using the bus at any one time.

4. Bus structures can be OPEN or PROPRIETARY.

a. Today there are several industry-standard bus architectures which allow components from different manufacturers to be assembled together into a single system.

Example: The PCI Bus and the newer PCI Expess (PCIe) Bus is widely used in both Windows PCs and MacIntoshes.

Bus standards are typically established by an industry group such as IEEE. This allows many different kinds of devices to be built to plug into it. Each device that plugs into a given bus must know what signals to expect on what pins and what protocols will be used to exchange information over the bus.

b. In addition, some manufacturers have their own proprietary bus architectures.

5. Actually, a given computer may have several bus systems - with one or more busses connecting to external devices being connected to the main system bus.

II. Overview of Bus Systems

A. Though the topic of this lecture is input-output, a parallel bus inside the system "box" is often used for both memory and IO, so we will include memory in our discussion.

B. The transfer of a piece of information between two devices on the bus involves a BUS TRANSACTION.

1. For each bus transaction, one device on the bus is designated the BUS MASTER, and the other is designated the SLAVE. The master is the device that initiates the transaction.

a. The master may the same device all the time, or provision may be made to allow different devices to become bus masters at different times.

i. CPU's are almost always masters (but may be slaves in certain cases, as we shall see)

ii. Memories are almost always slaves.

iii. IO devices are generally slaves when receiving commands or data from the CPU, but can be masters when interrupting the CPU or doing DMA transfers to/from memory.

b. If multiple devices can be bus masters, then each transaction must be preceded by an arbitration period when one device is chosen to be the master for the current cycle. (This is generally done on a priority basis.)

2. A bus transaction involves two distinct phases: an address phase and a data phase. The address phase also includes control information giving details about the transaction.

a. For a parallel bus, the transaction the two phases follow each other in time sequence. an address phase and a data phase (with the control signals included in the address phase.)

b. For a serial bus, control and address information is packaged into a frame that the master sends to the slave, followed by one or more frames of data going in the appropriate direction.

3. Either way, the master specifies an address with the expectation that the slave device will recognize it and respond.

a. When the slave is a memory, the address may consist of two parts:

i. The high order bits designate a particular memory device (if there is more than one on the system).

ii. The low order bits designate a particular location in that memory.

b. When the slave is an IO device, the address may serve only to designate the particular device. Furthermore, if the same bus is used for both memory and IO, then for IO sometimes only a portion of the address bits are used for this purpose, since the number of IO devices on the system is usually small compared to the total number of individually-addressable memory locations.

4. The bus master uses control signals to indicate what type of data transfer is to be performed. These may specify:

a. Type of slave being addressed (memory, IO, or a coprocessor). (Note: some bus architectures don't need this - no distinction is drawn between types of devices.)

b. Direction: master to slave (write) or slave to master (read) or (sometimes) slave to master to slave (read-modify-write). (This is always couched in terms of the vantage point of the master - e.g. data transfer from a master to a slave is always considered a write.)

c. Quantity of data to be transferred (one byte, one word, or (sometimes) a block of contiguous locations).

5. The data phases may consist of one or more transfers that follow each other in time sequence.

a. In the simplest case, one unit of data (i.e. as many bits as there are in the data part of the bus) is transferred.

b. It is also possible to do BLOCK MODE transfers, in which several units of data are transferred from successive addresses, one after another. (The address specified in the address phase is the address of the first unit of data).

C. Bus transactions are used for a variety of different purposes. (Note: on-chip cache memory allows most of the accesses to memory to be done without the need for an actual bus transaction on the main bus.)

1. Each instruction executed by the CPU may involve a bus transaction for INSTRUCTION FETCH. Here the master is the CPU and the slave is memory.

2. Instructions executed by the CPU may involve additional transactions with the CPU as master and memory as the slave - for:

   a. OPERAND ADDRESS CALCULATION may or may not require a bus cycle. (A bus cycle is required when memory indirect addressing is used.)

   b. OPERAND FETCH

   c. OPERAND STORE

3. IO instructions will involve a bus transaction with the CPU as master and an IO controller (or the controller portion of a peripheral device) as the slave. This may involve:

   a. TRANSFER OF A COMMAND (to the controller).

   b. TRANSFER OF STATUS INFORMATION (from the controller.)

   c. TRANSFER OF DATA TO/FROM AN IO DEVICE.

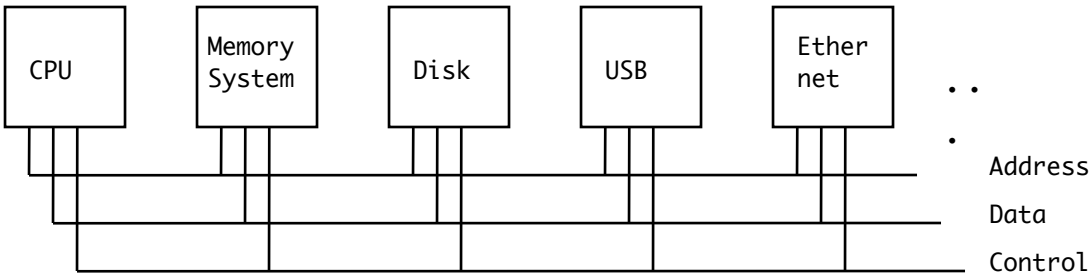
4. On many systems, IO controllers can also initiate bus transactions. These are of two types:

   a. INTERRUPTS (CPU is the slave).

   b. DMA (memory is the slave).

5. On multi-processor systems, bus transactions may also be used for INTER-PROCESSOR COMMUNICATION. (Coprocessors are typically part of the CPU chip, so they can be accessed without using the bus.)

D. A parallel bus system can be thought of of as a set of wires that the individual components connect to. These wires include some for carrying addresses, some for carrying information, and some for control.



Internal structure of a typical personal computer

Example: the Z80 system bus which you will work with in lab consists of 40 lines: 16 for address, 8 for data, and 16 for control (including power, ground, clock.)

Example: the PCI bus consists of 55 lines: 32 that are used for both address and data (time multiplexed) and 23 for control.

a. Address lines

i. If the bus system is used for accessing memory, the address lines will transmit a memory address, and the number needed is dictated by the size of the address space (e.g. 32 bits).

ii. If a bus is used only for IO, the address lines can be used to identify which of several devices that may be connected to the bus is actually being accessed. Clearly, many fewer bits ar needed in this case.

iii. In either case, the range of possible addresses is called the BUS ADDRESS SPACE.

b. Data lines

i. In a serial bus, there is one data line or - more commonly - one data line for each direction.

ii. In a parallel bus, there is generally one data line for each bit of data being transmitted. However, sometimes a bus will have a smaller number of data lines that are used multiple times during a cycle. This is called MULTIPLEXING.

iii. In both kinds of busses, it is possible to design the bus to use the same physical lines BOTH for addresses and for data in different parts of the cycle. This is another form of MULTIPLEXING.

- This is almost always the case with serial busses.

- It also occurs with parallel busses.

Example: PCI.

c. Control lines

i. These are used to specify things like the nature and direction of the transfer.

ii. One special control line that may be present is a BUS CLOCK.

- If it is present, all devices connected to the bus use this clock to synchronize their communication with each other.

- An alternative is the use self-clocking data - where edges in the state change of data are also used to communicate clock information. (This is more common on serial busses)

- In either case, note that the bus clock is separate and distinct from the internal clocks in various devices (e.g. the CPU).

E. In the case of a serial bus, there is only one line per direction - or just one line - which performs all of the above functions at different times - e.g. sometimes it is transmitting address bits (one at a time), at other time data bits (again one at a time), and at other times commands (also one bit at a time) Rather than having a separate clock, the clock is also implicitly present in the form of transitions on the line.

F. Further notes.

1. In either case, devices may connect to the bus system through a separate interface or controller - e.g. a system may include a disk controller that interfaces between the system bus and the disks. Or, each device may include its own interface components. (E.g. IDE disks - "integrated drive electronics".) The latter is almost always the case when busses are used to connect to external devices - e.g USB, Firewire.

2. It is quite common to find a single system having multiple buses. In this case, there is a central system bus, with adapters used to connect other busses to it.

Example: A given system may contain an internal bus, with one or more USB adapters connecting to one or more USB busses, as well as Firewire and Ethernet adapters.

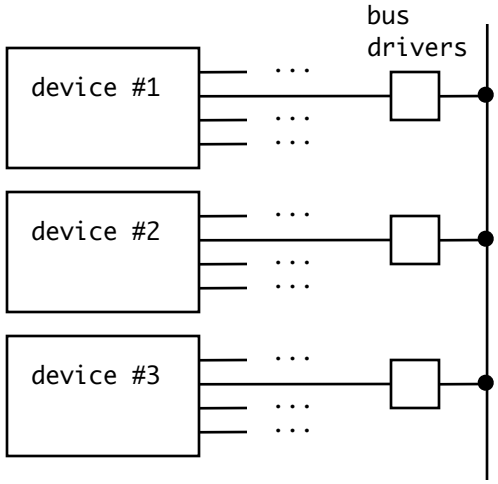
G. Implementation of a bus

1. This is an interesting problem because, in general, it must be possible for different devices to drive a given line at different times.

a. Example: For a write transaction, the master drives data on to the data bus; but for a read transaction, the slave does so. Further, different read transactions may involve different slaves.

b. Example: If a bus can have multiple masters, then each master must be capable of driving the address and control lines in the bus when it is in charge.

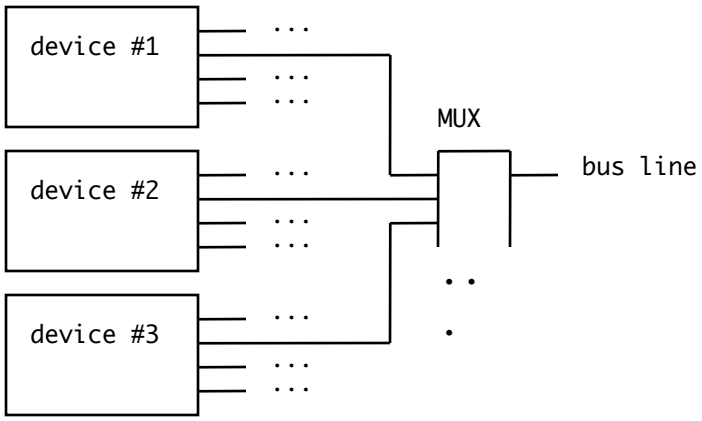
2. This suggests that each device that can drive a given line of a bus should contain a gate (called the driver) whose output is tied to that line - e.g.



However, this won't work if ordinary gates are used for the drivers.

ASK CLASS WHY

3. One solution might be to implement a bus using MUXes:



a. This technique can be used for INTERNAL BUSSES inside the CPU

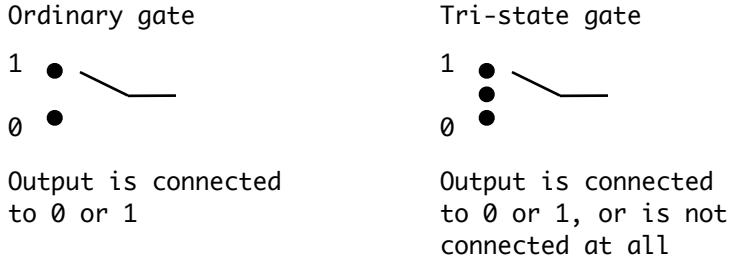
b. But it is not a good approach for system busses. ASK WHY

- The total number of devices to be connected to the bus must be known when it is built (inflexible)
- A lot more wires are needed - each bus slot must have its own set of connections to the MUXes

4. The most common approach is to use TRI-STATE gates.

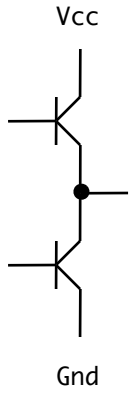
a. As the name implies, a tri-state gate is one whose output can be in any of three states: 0, 1, or Hi-impedance ("Z"). Recall the discussion of this possibility earlier in the course.

b. The hi-impedance state is the new one. When the output is in this state, it behaves like it is not connected at all - e.g. (viewing the gate output as being like a switch)



c. Tri-state gates can be built in many configurations (e.g. AND, OR, NAND, or can be used as the outputs of flip-flops) A tri-state gate has an additional input called ENABLE. When this is active, the output of the gate is determined by the other inputs or the flip-flop state, as usual; when it is inactive, the output of the gate is effectively disconnected from the circuit.

d. Tri-state gates are realized by modifying the output circuit of a standard gate. The following is the "totem-pole" circuit used by TTL gates. (CMOS gates use a similar structure).



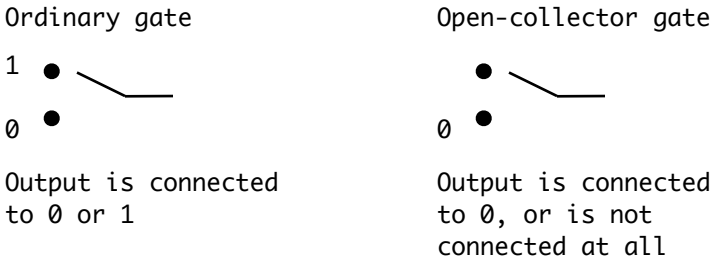
i. Each of the two transistors acts like a switch which is either off or on If the transistor is on, then the output of the gate is effectively connected to Vcc or ground (as the case may be.) (Clearly, we cannot allow both transistors to be on at the same time. This would effectively short the power supply to ground, resulting in the rapid destruction of one or both ' of the transistors.)

ii. In an ordinary gate one or the other of the two transistors connecting to the output is on at any given time, and the other is off, thus connecting the output to either Vcc or ground.

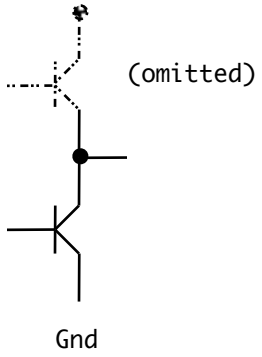
iii. In a disabled tri-state gate, BOTH transistors are off, thus leaving the output effectively unconnected (as if the gate weren't in the circuit at all)

5. A third approach is to use OPEN-COLLECTOR gates. This is useful for cases where a given device must drive a given line either to 0 or not at all (i.e. it never has to drive the line to 1).

   a. Open-collector gates are built in many standard configurations (e.g. AND, OR, NAND) However, the two states of the output are disconnected or 0 rather than 1 or 0. (The disconnected state occurs when the logic function the gate implements would otherwise call for a 1 to be output.)



   b. Open collector gates are realized by a different modification of the standard gate output circuit. For example, this is the way a basic "totem pole" would be turned into an open-collector gate by omitting one of the output transistors:



   c. Open collector gates are most often used when any number of devices must be able to assert the same line at the same time - e.g. an arbitration line representing a bus request.

      i. Because the only state to which a device can assert such a line is 0, such lines are most often configured as ACTIVE-LOW - i.e. the active state is 0 and the inactive state is 1.

ii. To make sure the line goes to 1 when no device is asserting it, such lines normally are terminated by a PULLUP RESISTOR to Vcc, which keeps the line at the 1 state unless some device(s) is/are driving it to 0.

H. Regardless of how the interfaces connect to the bus, the electrical characteristics of a bus system have an important influence on system performance.

1. Bus designers must take at least the following characteristics into consideration:

a. PROPAGATION DELAY: When a bus master or slave near one end of the bus places some information on the bus, it will take a measurable time for that information to propagate to the other end, due to effects of capacitance and inductance. This time increases with increasing physical length of the bus.

b. LOADING: When we talked about the realization of gates, we mentioned that a given gate can only drive so many of a given type. Since some signals generated by a bus master must be received by all other devices on the bus (e.g. to recognize their own address), there is a limit as to how many devices may be plugged into the bus. Note, too, that propagation delay tends to increase with increasing bus load.

c. SKEW: (Only an issue for parallel busses) The propagation delay for different lines of the bus is not necessarily the same; thus if several bits are changed at the same time near one end of the bus, the changes may be seen at different times at the other end. Also, if a common clock is used to synchronize events, the clock may actually arrive at different devices at different times.

(This problem is a significant reason for preferring serial buses over parallel ones.)

2. Bus designers take these factors into consideration when establishing bus timing.

   a. An appropriate interval must be allowed between the time a device asserts a signal and the time it can expect the signal to be received. This time is called the SETTLING TIME.

   b. In the case of addresses, because skew could cause the wrong device to respond to an address, a separate "address valid" control signal is often used, asserted some time AFTER the address itself is put on the bus (to ensure that all bits have settled.)

   c. For well more than a decade now, bus speeds have lagged behind CPU speeds, so that the basic bus cycle time is some multiple of the CPU cycle time. On-CPU-chip cache memory is used to enable many instructions to execute without having to do a bus transaction - otherwise the bus cycle time, not the CPU cycle time, would determine the rate at which instructions can be executed.

III. General Issues in the Design of Bus Systems

A. One fundamental choice, when designing an overall system, is whether to have one bus to serve both memory and IO devices, or separate memory and IO busses.

   1. The choice is sometimes made to have two separate busses. Considerations of speed are a reason for going this route - a memory bus (which gets the most intense use) can be made faster if it handles memory only, since the total length of the bus is smaller, and the devices connected to it are more uniform.

      (Actually, sometimes the "memory bus" is used only for Level 2 cache, with main memory (DRAM) connected to the same bus as the IO devices. In this case, the "memory bus" which is used for cache is sometimes called a "backside bus".)

2. However, pin count considerations generally dictate that the CPU only have one set of bus connections. If multiple internal busses are used, the CPU is directly connected to the memory system, and to a bus adapter that, in turn, connects to the main system bus.

B. If a single bus is used for both memory and IO, there is also a choice to be made between using MEMORY MAPPED IO and ISOLATED IO.

1. With memory-mapped IO, both memory and IO devices use the same address space - i.e. a "memory read" or "memory write" operation to certain addresses actually transfers data to or from a given device. Any CPU instruction that references memory can do IO if the address it specifies lies in the IO portion of the address space.

Example: As noted in the book, the MIPs architecture uses memory-mapped IO. historically very important bus architecture was the PDP-11 UNIBUS, which was designed for memory-mapped IO.

Addresses 00000000 to FFFEFFFF are memory addresses
FFFF0000 to FFFFFFFF are IO device registers

2. With isolated io, separate address spaces are used for memory and IO. The CPU has separate instructions for doing IO operations. This also requires that the CPU bus connections include some separate control lines for each kind of operation - one set is used when the CPU is accessing memory, the other when it is executing an IO operation. (There is one physical bus, but two logical busses.)

Example: The Z80 we will work with in lab uses addresses 0000 to FFFF (hex) for memory and 00 to FF (hex) for IO ports. There is a single physical bus, with includes a control

$\overline{\text{MREQ}}$ line called MREQ used only for memory operations, and one

called $\overline{\text{IORQ}}$ used only for IO operations. When $\overline{\text{MREQ}}$ is asserted, memories look at the address lines and IO devices ignore them, while $\overline{\text{IORQ}}$ causes the opposite to be done.

The instruction set includes IN and OUT instructions for IO.

Example: The x86 architecture uses isolated IO

C. Some fundamental choices for parallel busses.

1. The WIDTHS of the bus - the number of bits used for addresses, and the number of bits used for data.

   a. The address width ultimately determines how much memory and/or how many IO devices can be connected to the bus.

   b. The data width helps determine bus throughput (# of bytes transferred over the bus per second).

   Example: Many computers that used a 32 bit word used a 64 bit bus - one reason being that bus cycles are longer than CPU cycles.

2. Whether to DEDICATE all the lines in the bus to certain functions, or to MULTIPLEX certain lines. As we have already noted to reduce the total width of the bus (and thus the cost of each interface), and also to reduce the pin-count chip packages, the same lines can be used for both functions, but at different times during the bus transaction.

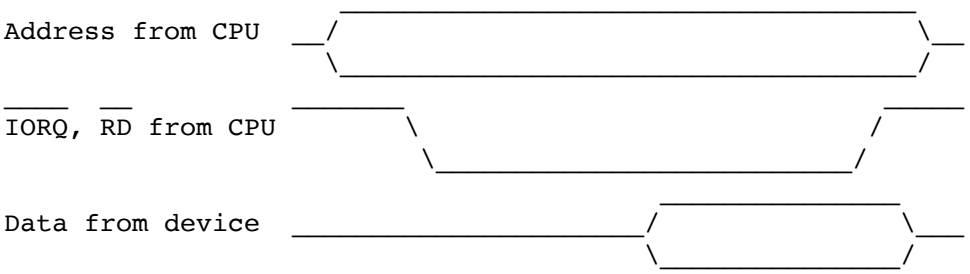
   Example: PCI multiplexes address and data onto the same lines at different times.

3. The previous choices have dealt with the physical configuration of the bus. Another important choice has to do with the bus PROTOCOL - the rules whereby the bus master and slave exchange signals with one

another. Here, the fundamental choice is between SYNCHRONOUS and ASYNCHRONOUS protocols.

a. In a synchronous protocol, all devices on the bus share a common clock. The bus master puts signals on the bus and expects the slave to respond within a certain time frame, without looking for explicit acknowledgement from the slave that it has done so.

Example: The Z80 memory IN protocol we will be using in lab:



i. Note the delay between putting the address on the bus and asserting $\overline{\text{IORQ}}$. This ensures that the address has settled so that only the right slave will respond.

ii. The protocol specifies a maximum interval between the falling edge of $\overline{\text{IORQ}}$ and the time the slave gets its data on the bus, by performing any access necessary and then enabling its data drivers.

iii. After the master removes $\overline{\text{IORQ}}$, it leaves the address on the bus for a time to prevent a slave from still seeing the control signal true (due to skew) after the address begins to change (could result in a wrong device thinking it's being addressed).

iv. The slave must disable its data drivers as soon as it sees that $\overline{\text{IORQ}}$ is deasserted.

v. Note that, in a synchronous protocol, all control signals are generated by the master.

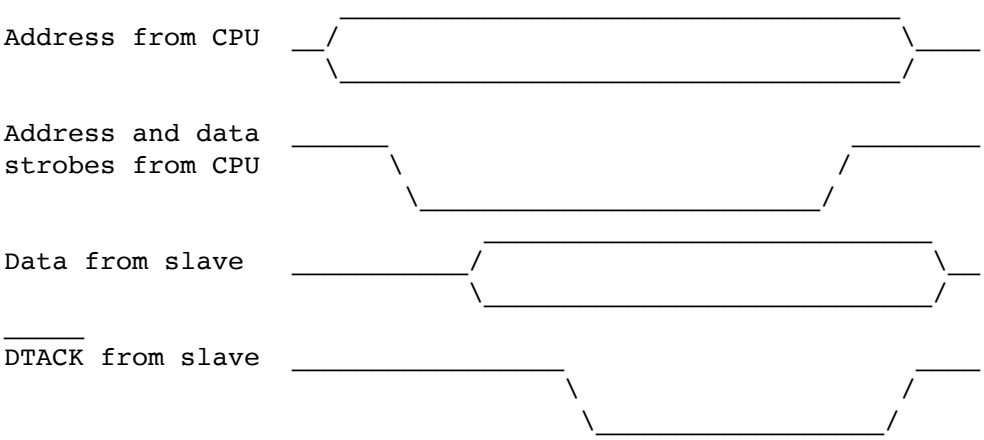
(a). On a read, the slave has to respond by providing the data, but does not send any control signals to the master. On a write, the slave is not responsible for any signals. This simplifies the construction of the slave interface.

(b) However, if the slave failed to respond the CPU would never know. On a read, a totally floating bus looks like a byte of all 1's, so if the CPU addressed a non-functioning (or nonexistent) slave it would think that the slave was sending it the value F...F and that would be treated as the slave's data. On a write, the master would have no clue that no slave received its data.

(c) To address the possibility that a slave may not be able to respond as quickly as it needs to, a synchronous system may add a WAIT control line that the slave may assert to pause the process. Unlike the other control signals, this one goes from the slave to the master.

b. In an asynchronous protocol, the CPU and port EXCHANGE a series of signals.

For example, the following is the protocol for a memory or IO read on the MC68000 microprocessor. (This chip uses memory-mapped IO, so the distinction as to what type of operation is being done arises from the value of the address)



i. The CPU puts the address on the bus, waits for settling time, then asserts the strobes (three separate lines). It then waits for the memory/port to respond.

ii. The memory or port addressed places its data on the bus, waits for settling time, and then asserts $\overline{\text{DTACK}}$.

iii. The CPU captures the data, then releases its strobes. After a settling time it also releases its address.

iv. The port, seeing the strobes no longer asserted, releases $\overline{\text{DTACK}}$, then (after a settling time) its data.

v. This exchange of control signals is often referred to as "HANDSHAKING".

Note that the issue in the handshake is the TRANSMISSION of the data, not its PROCESSING by the device. For example, a printer may take several milliseconds to print a character. But its interface will handshake with the CPU when the character to be printed has been received, not when it has actually been printed. (The CPU must still poll a status bit separately to be sure that the printer has finished printing the preceding character.)

c. The choice of synchronous versus asynchronous protocols is basically made by the bus system designer. Both synchronous and asynchronous protocols have their pros and cons:

i. In favor of the synchronous approach:

(a) Interfacing is simpler: the slave does not need to send any signals back to the CPU. (However, this advantage goes away if the slave is slow and must request wait states.)

(b) The synchronous approach is faster overall, since fewer signals must be put on the bus. (Recall that each signal must be followed by a settling time before other activity can occur.)

ii. In favor of the asynchronous approach:

(a) This approach can accommodate a wide variety of interface speeds mixed on the same bus.

- This allows older and newer technology interfaces to be used on the same system, increasing the range possible devices that can be interfaced.

- If a slow interface is replaced with a fast interface, system speed immediately improves without changing anything else.

(b). This approach gives a positive assurance that the requested data transfer has actually occurred - i.e. the interface addressed exists, is working, and was able to respond. (Of course, if an attempt is made to access a nonexistent or nonfunctional device, the CPU could wait forever for a handshake signal that never comes. This is usually handled by having a bus timeout mechanism that causes a trap to a software routine that deals with the problem.)

Note: this is the rationale behind the error message on Unix systems "Bus Error" that occurs when one tries to access nonexistent memory. In actual systems, such an error might be caught short of causing a timeout on the bus; but the Unix message reflects the way a situation like this was originally caught - and might still be caught.

d. On systems having separate parallel busses for memory and IO, it is common to find that the memory bus is synchronous (for speed, and since the memories can be assumed to be of uniform technology), while the IO bus is asynchronous (for interface flexibility.)

D. In the case of a serial bus, the design of the formats of the frames used during the address and data frames - including the number of address and data bits, respectively - is the fundamental choice.

E. Finally, if more than one device can serve as a bus master, there is the matter of BUS ARBITRATION - how is a master chosen if more than one device wants to use the bus at the same time?

1. There are three basic approaches that can be taken.

a. A centralized approach: one device (often the CPU) is designated as the bus ARBITRATOR. All requests to use the bus are routed to it and it gives permission on a priority basis.z

In the case of a serial bus, the central arbitrator will need to poll the other devices to see if they need to use the bus.

b. With a parallel bus, a decentralized approach: all potential bus masters look at arbitration lines that are part of thebus, and the highest priority device recognizes that it has priority and proceeds while all others wait.

c. With a serial bus, a decentralized approach.

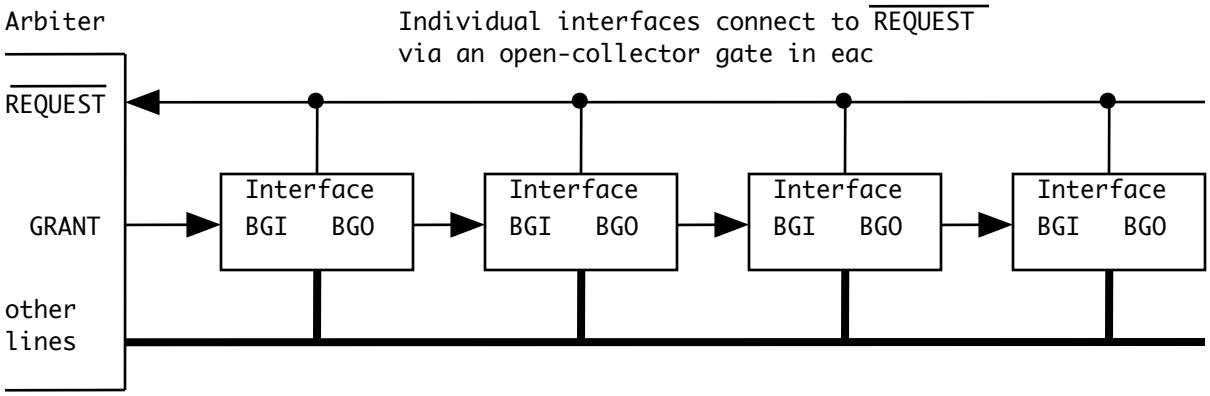
2. An example of a centralized parallel bus approach: DAISY-CHAINING:

a. The arbitrator has a single bus request input. The request line is connected to open-collector gates in each of the other devices, so that any device an request the use of the bus by pulling this line low. If multiple devices do this at the same time, the process described below determines which device gets to use the bus.

b. The arbitrator has a single bus grant output, which it asserts as soon as possible after seeing an incoming request.

c. Each device has an BUS-GRANT INPUT (BGI) and a BUS-GRANT OUTPUT (BGO).

d. The devices are connected in a chain, such that the BGO of one device connects to the BGI of its neighbor. The first device on the chain receives the external grant signal from the centralized arbiter) and the last device on the chain has no connection from its BGO.



e. When the arbiter sees an incoming bus use request and is able to grant it, it asserts BGI to the first device.

f. Each device behaves as follows:

i. If its BGI is not asserted, then it does not assert its BGO.

ii. If its BGI is asserted then

- If it wants the bus, it uses it and leaves BGO unasserted.
- Otherwise, it asserts BGO.

g. The result is that, if multiple devices request the bus, only the one nearest the arbiter gets to use it.

3. A decentralized approach for a serial bus.

a. The original Ethernet standard used CSMA (Carrier-Sense-Multiple-Access). A device that wanted to use the bus would listen until no other device was using the bus (sensing the presence of a carrier) and then would begin transmitting. (Sort of like what happens in a conversation when you wait until no one else is talking before you start to talk.)

b. Of course, under this approach, two devices could end up trying to transmit at the same time (sort of like two people starting to talk simultaneously in a group conversation.) In this case, each would detect that someone else was trying to use the bus. A device that sensed this would "back off" for a random period again and then try again as soon as the bus was clear. The use of a random backoff period made it unlikely that the same two devices would collide when they retried.

c. A similar approach is used in Wi-Fi - which is not a bus-system per se.

IV. Control Options

A. One important consideration in the design of an IO system is mechanisms for managing the RATE at which data is transmitted to/from the various devices.

B. IO devices vary widely in the rates at which they can handle data.

1. At one end of the spectrum, the rate at which a keyboard produces data is limited by the typing speed of a human typist, which is rarely more than 5 characters per second, and can be as slow as one character every few seconds (or less.) Pointing devices (e.g. mice) have similar characteristics.

2. At the other end of the spectrum, some devices can transfer data at rates at 100's of millions of bytes/second.

3. Compare these numbers with CPU speeds, where the clock rate may be on the order of 2 GHz. In some cases, the CPU could execute millions of instructions in the time the IO device takes to process one byte of data; in other cases, the CPU cannot process data at nearly the rate that the IO device produces or consumes it - given that several instructions are probably needed as a minimum to read and store the data (or to fetch it to write).

C. For any device then, we need some way of coordinating the device speed and the CPU speed. In most cases, the CPU will have to wait for the slower IO device to perform its work; but in other cases the reverse may be true.

D. For illustration, we will use an old-technology printer that has no internal buffer, and can print at a rate of 100 CPS (characters per second) connected to a CPU with a 1 GHz clock - so the printer's data rate is roughly 1 ten-millionth of the clock rate of the CPU. (This example, though dated, is chosen because facilitates illustration of all possibilities.)

1. If we assume that the CPU executes a loop in which it sends characters to the printer as fast as it can, and the loop contains 10 instructions, then the CPU can send 100 million characters per second and the printer can handle only 100. If we did this, well over 99% of the characters would be lost!

2. To prevent such things from occurring, devices such as printers are typically built with some sort of status flag which indicates whether the printer is able to accept a character. This flag will be part of a status byte that the CPU can read (possibly along with other flags such as "out of paper" etc.) Thus, the interface to the printer will include at least two separate ports: an output port to which an ASCII character may be sent, and an input port which may be read in order to determine the device's status, and the connection to the CPU must be either half-duplex or full-duplex.

   a. We will use, for our example, a "ready" flag that would be 1 to indicate that the device is able to accept a new character, and 0 to indicate that it cannot accept a character because it is still printing a previous one.

   b. The printer manages this flag as follows:

      - When it is first turned on, the printer sets its ready flag to 1.
      - When it receives a character to be printed, it sets its ready flag to 0.
      - When it has finished printing the character, it sets its ready flag back to 1.

   c. To prevent loss of data, we impose the following requirement: a character to be printed can only be send to the printer's output port when the printer's ready flag is 1.

E. Four basic approaches may be taken to synchronizing the CPU and its external devices.

1. A strictly software-based approach, in which the CPU tests the device status before sending data.

   a. The simplest form of this is an approach known as busy waiting.

```
do
{
    read the status flag
} while status is not ready;

print the character;
```

Note that the do while loop would typically be executed literally thousands of times for each character printed. The CPU would do nothing else during this time.

b. Another variant of this is an approach that might be used in a system that is servicing multiple devices like our printer.

```
for (int i = 1; i < NumberOfDevices; i ++)
{
    test status of device i;
    if it is ready, then service it;
}
```

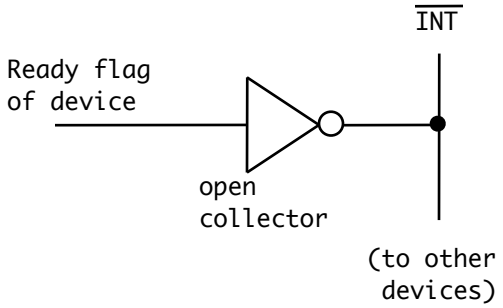
c. With somewhat more difficulty, polling of device(s) might be intermixed with other kinds of computation. This is made difficult by the need to have the other computational routines "remember" to call the polling routines from time to time.

d. Except for embedded systems or CPU's totally dedicated to IO (e.g. as part of a "smart" interface to some device), total software control of IO is rarely satisfactory.

2. A second approach to IO control uses a mixture of hardware and software techniques, by utilizing the interrupt capabilities of the CPU. This approach is called interrupt-driven IO.

a. Most CPU's have one or more interrupt control lines, which may be asserted by an external device. (For now, we assume just one.) We

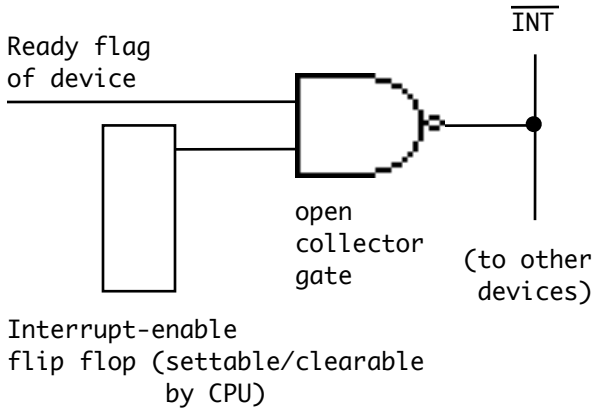
begin by connecting the device's ready flag to the CPU's interrupt input, so that an interrupt is requested whenever the device needs CPU attention:



b. We further arrange for the CPU to respond to the interrupt request by executing a software routine that performs an appropriate data transfer to/from the device, thus clearing the ready flag and removing the interrupt condition until the operation is complete, at which time a new interrupt will be generated.

c. Interrupt-driven IO has several complexities that must be dealt with in a complete system:

i. With the simple hardware configuration shown above, we assumed that we would always want the device to interrupt when it becomes ready. In the case of a device like a printer, however, there may be times when we have no work for it to do. In such a case, we want to be able to tell the device to quit interrupting until some more work comes along. This is conventionally handled by including an interrupt-enable flag in each interface, which the CPU can set and clear to determine whether that particular device may interrupt.



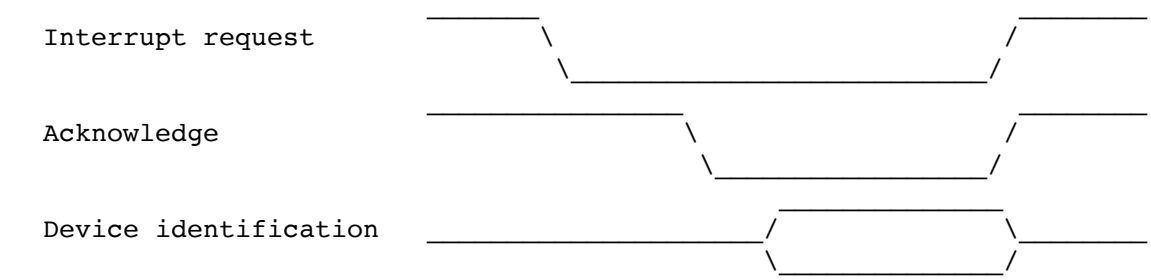
ii. If the system has more than one IO device (as it generally does), some provision must be made to cause the software for the proper device to be invoked when the interrupt is received.

iii. Further, if two or more devices become ready at the same time we want to guarantee that each is serviced in turn without interference from the other. We may wish to prioritize the interrupts so that the highest priority device gets served first, and we may even wish to allow a higher priority device to interrupt a lesser-priority one.

d. The problem of identifying the device responsible for the interrupt can be handled in one of two ways:

i. The simplest approach (from a hardware standpoint) is simply to have the CPU poll all the devices to see which one is in need of service. However, this makes the interrupt service routine slow, and so is not generally desirable.

ii. Instead, most systems make some provision for the interrupting device to place some data on the system bus to identify itself. This is done in response to an interrupt acknowledge signal from the CPU - e.g.



(Note: we assume that the device also uses the acknowledge as an indicator to remove its request.)

iii. Anything that will uniquely identify the device can be chosen for the device's response; but the most typical choice is to require the device to put on the bus a memory address which is either:

- The starting address of a service routine for the device.

- The address of a memory location which contains the starting address of a service routine for the device.

(The second option is more flexible since it allows system software to be restructured without rewiring the devices, so long as a table of service routine addresses is kept in a fixed, known location.)

This approach is known as VECTORED interrupts.

Example: Intel 80x86 processors reserve a block of 2048 bytes of memory as an interrupt descriptor table containing 256 entries of 8 bytes each. A CPU register (the interrupt descriptor table register) points to the beginning of this block. An interrupting device puts an 8 bit interrupt type on the bus, which the CPU uses as an index into this table. The appropriate entry contains the address of a service

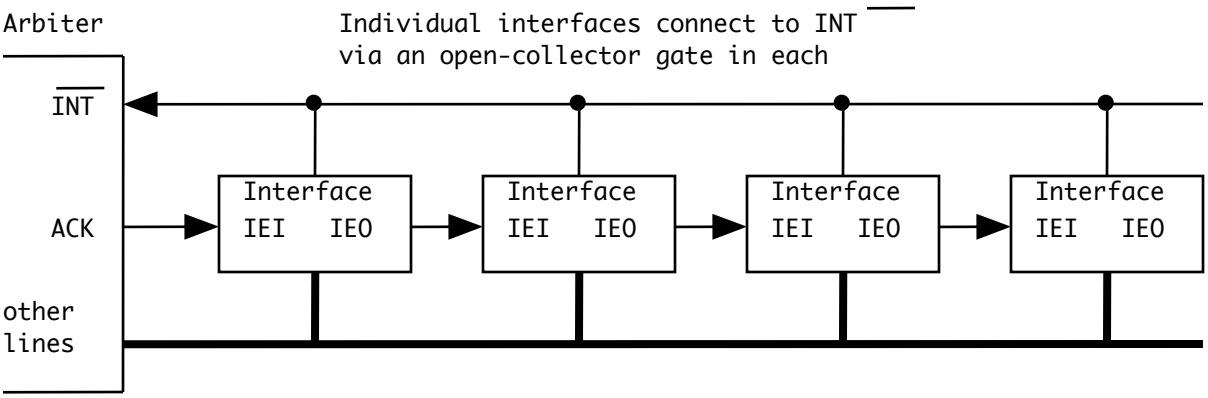
routine for this kind of interrupt. Each device is generally assigned a unique interrupt type from among the 256 possible values.

e. The problem of multiple devices interrupting at the same time can be handled in several different ways.

i. First, all CPU's have some mechanism whereby interrupt recognition can be temporarily disabled. (The request is present, but the CPU ignores it until interrupts are re-enabled.) This allows an interrupt service routine to protect itself from interrupts by other devices.

ii. But we still have to ensure that when an interrupt is accepted only one device will respond. One way to do this is by daisy-chaining. We add one new input and one new output to each interface, known as IEI (interrupt-enable-in) and IEO (interrupt-enable out.)

The various interfaces are connected as follows:



Note that the first device receives an input of 1 when the CPU acknowledges an interrupt, and either keeps it or passes it on to the next device.

- Any device may assert $\overline{INT}$, and multiple devices may do so at the same time.

- However, only the requesting device nearest the CPU will see the acknowledge signal, and so it alone will put its vector on the bus.

- To prevent race conditions, however, we must ensure that no device near the CPU decides to request an interrupt (and thus "steal" IEI) when a device further down the chain is in the process of being acknowledged. This can be done by wiring the internal request so that it cannot be set when IEI coming into the interface is high.

iii. Another way to handle the problem of multiple devices interrupting at the same time is by the use of a special purpose support chip called a priority encoder.

- As an example, a one out of eight priority encoder has 8 inputs and 4 outputs. The inputs are numbered 0, 1, 2 ... 7, with 7 being the highest priority input and 0 the lowest.

- One of the outputs is asserted if at least one of the inputs are asserted. This output is called GS.

- The remaining three outputs encode the number of the highest priority input that is currently asserted. (If no input is asserted, these outputs encode are normally ignored.) These outputs are designated A2, A1, A0.

Examples:

- No input is asserted. GS is not asserted, A2..A0 ignored.
- Input 4 is asserted. GS is asserted, A2..A0 encode 4.

- Inputs 4,5 asserted. GS is asserted, A2..A0 encode 5.
- All inputs asserted. GS is asserted, A2..A0 encode 7.

iv. Some CPU's effectively internalize what the priority encoder does, by having multiple interrupt lines coming in at different levels.

- For example, the PDP-11 had 4 such lines, designated BR4 .. BR7, with BR7 being the highest priority. Each level also has its own acknowledge line.

- The CPU contained a "processor status word" that included a three bit field that encoded the processor priority - the priority level of the task the CPU is currently working on. (This could range from 0..7). Under normal conditions, the CPU priority would be 3 or less.

- An interrupt would only be acknowledged when the CPU priority is less than that of the incoming request - e.g. a BR4 request will only be acknowledged if the CPU priority is 3 or less.

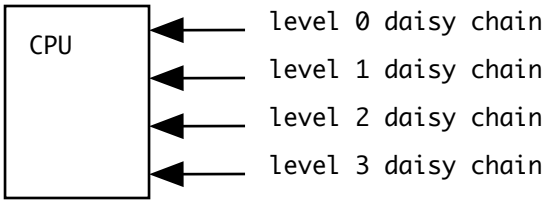
- If multiple requests are coming in, the highest priority request is acknowledged.

- Generally, the service routine for a given device will see that the PSW is set to a priority level equal to the priority of the interrupt that called for the service. This means that a service routine for a level 4 device would run at processor priority 4, and could not be interrupted by by any other level 4 device, but could be interrupted by level 5 or higher devices. When the service routine exits, it resets the CPU priority to what it was on entrance.

This approach is known as MULTIPLE-LEVEL interrupts.

v. Of course, it is still possible to have more devices than levels. (And generally this will be true.) In this case, a daisy chain can be used to prioritize devices on the same level.

Example:



f. One concern in the design of systems using interrupt-driven IO is INTERRUPT LATENCY - the maximum time that can occur between the time that a device requests an interrupt and the time the servicing routine for the interrupt begins executing (due to issues like finishing the execution of the instruction that was in process when the interrupt occurred.)

3. A third approach to IO control is direct memory access (DMA). This is an approach that is totally based in hardware.

a. We have noted that the speed of IO devices ranges from several thousand times slower than the CPU to as fast as the CPU itself. When device speeds approach those of the CPU, the other forms of IO control we have discussed cease to be useable, since any form of software IO requires several machine instructions (at least) to transfer a single item of data. Thus, when device speeds approach 10% or so of CPU speed, software control of IO becomes impractical.

b. The alternative for fast devices is to allow the device interface to gain control of the system bus each time it has a data item to transfer. This means, in essence, that the interface must contain some of the capabilities of a CPU.

- It must be able to generate the various system bus signals for a MEMORY operation and to gate its own address and data information onto the bus.

- Typically, the interface needs at least two registers of its own:

  - A memory address register to keep track of where the next transfer is to go to/come from. This register must be incremented after each transfer.

  - A counter to keep track of the number of data items transferred. Typically, the DMA interface will interrupt the CPU when this count reaches 0.

- Often, a third register is needed. If the device itself is addressable (as would be true in the case of a disk, say), then the interface also needs a DEVICE ADDRESS register to keep track of the location on the device to/from which the transfers occur.

c. When a DMA device is in use, the only task of the software is to load up the registers in the interface and start it doing the transfer.

d. Because DMA interfaces are complex, DMA is typically used only in cases where device speeds make it necessary.

e. One other issue arises in conjunction with DMA interfaces: cycle-stealing versus burst mode transfer:

i. Most interfaces are designed so that they have to go through the process of requesting the bus and waiting for acknowledgement for EACH data transfer done. This is fine, so long as the data rate of the device is low enough. This mode is called CYCLE-STEALING, because each transfer "steals" one memory cycle from the CPU.

ii. For very fast devices (e.g. some disks), there might not be enough time to allow the interface to request and wait for the bus for each transfer. Such interfaces may work in a BURST MODE

in which, once the interface has control of the bus, it keeps it until a whole block of transfers is done - i.e. it goes through repeated memory cycles, but holds the bus request active the whole time without ever releasing it.

4. A key motivation for DMA is to facilitate the OVERLAP OF IO AND COMPUTATION.

a. Non-interactive tasks typically involve reading input from disk, performing computation, and producing output on disk.

b. For any given task of this sort, the following will hold:

total time = input-output time + computation time

i. If the input-output time is much greater than the computation time, then we say that the task is IO BOUND.

Example: many business data processing tasks

ii. If the computation time is much greater than the input-output , time, then we say that the task is COMPUTE BOUND.

Example: many simulations of physical systems

c. Very early in the history of computation, it was realized that major performance gains could be realized by using "smart" disk controllers to facilitate performing disk IO and computation at the same time and even to perform multiple IO operations using distinct devices at the same time.

i. Often, this involves doing operations on behalf of two or more distinct tasks - e.g. the system might be reading or writing disk on behalf of one task while doing computation on behalf of the other.

ii. Keeping all the systems resources busy depends on having a good mix of IO Bound and Compute Bound tasks - e.g. if all of the tasks on the system are IO Bound, then the CPU will have some unavoidable idle time, and vice versa.

d. Sometimes it is possible to overlap disk IO and computation within a single task.

i. In the case of output, it is fairly easy to see how this could be done. But what about input? How can the system be doing computation on the behalf of a task while it is waiting to complete the reading of its input?

ii. Sometimes this is possible, if the input consists of units such that the processing of one unit is independent of the content of the next unit.

Example: Compiling a program using a one pass compiler. (The source program is read once, and is translated as it is being read. This is possible if the language requires all identifiers to be declared before they are used)

In this case, something like the following becomes possible:

```
READ    READ    READ    READ    READ    READ    READ
BLOCK 1 BLOCK 2 BLOCK 3 BLOCK 4 BLOCK 5 BLOCK 6 BLOCK 7

        COMPUTE COMPUTE COMPUTE COMPUTE COMPUTE COMPUTE
        BLOCK 1 BLOCK 2 BLOCK 3 BLOCK 4 BLOCK 5 BLOCK 6

                WRITE   WRITE   WRITE   WRITE   WRITE
                BLOCK 1 BLOCK 2 BLOCK 3 BLOCK 4 BLOCK 5
                RESULTS RESULTS RESULTS RESULTS RESULTS
```

iii. In some ways, this resembles pipelining, but is now applied to an entire task, not just to computation.

V. Software Interaction with IO Devices

A. In the very earliest days of computing, programs were required to directly control the relevant IO devices. That is no longer the case.

1. Directly controlling devices implies, of course, that if a device is replaced, all of the software that accessed this device would have to be modified.

2. Today, most computer systems attempt to achieve DEVICE-INDEPENDENCE by the various means we will describe here, so that - to the extent possible - a program is independent of the specific hardware devices it uses.

   (The major exception to this is embedded systems, where such dependency may be unavoidable.)

3. Given that relatively few people ever write low-level IO code (device drivers or embedded systems, the discussion here will focus on making good use of the facilities found in typical systems.

4. Device independence is achieved by making use of two fundamental abstractions - the abstract notion of a DEVICE and the abstract notion of a FILE.

B. The abstract notion of a device

1. This fundamental abstraction allows the software to treat any IO device as a "black box" that supports some combination of the following operations:

   a. Read - transfer information from the device to memory

   b. Write - transfer information from memory to the device

c. Seek - physically position the device, so that the next read or write operation will occur at a certain point

d. Control - perform some device-specific operation

2. Of course, the operations that are possible will vary from device to device.

a. Some devices allow read, but not write (e.g. read-only optical disk drives.

b. Some devices allow write, but not read (e.g. printers)

c. Devices also vary in terms of how much data can be transferred by a single read or write operation.

i. Character devices support the transfer of individual characters, or an arbitrary (but usually relatively small) number of characters.

Example: a keyboard

ii. Block devices require all transfers to in multiples of some device-specific block size.

Example: a disk

d. The concept of "seek" is only meaningful if the concept of physical position on the device means something - e.g. it is meaningful for disk type devices but not for keyboards or mice.

e. The permissible control operations are, of course, highly device-specific.

For example, "terminal" devices support a rich set of control options.

i. Historically, terminals were physical devices that were physically connected to the computer system. A time-sharing system might support dozens or hundreds of them.

ii. Today, a terminal is typically an abstract device (a pseudo terminal) that may be associated either with a terminal emulation program running in a window on the current machine, or a network connection (allowing remote login to a terminal emulation program on another machine.)

iii. Either way, terminals support options like

- buffered versus raw line input
- line editing (delete key, arrows, etc)
- echoing or not echoing of keystrokes
- special interpretation of characters like Control-C
- flow control (Control-S/Q)

3. This abstraction is provided by a software component known as a DEVICE-DRIVER (an early example of the bridge design pattern, though invented long before patterns).

a. A device driver is an operating system component. It may either be compiled into the system, or it may be installed later.

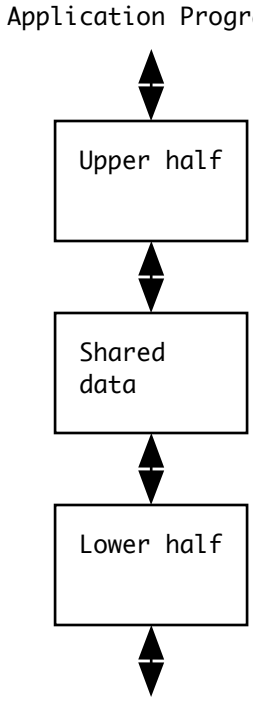
b. The way a program accesses the driver for a given device is system specific.

For example, on Unix systems devices are assigned a two part number - a major number that identifies a particular type of device, and a minor number that identifies a particular unit (which allows the same driver to service multiple instances of a given type of device on the same system - e.g. [historically] multiple terminal lines or multiple disks / disk partitions.

Each device is associated with a "special file" that resides in the /dev directory. The entries in this directory are actually just "hooks" to the various device drivers; each entry simply records the major and minor number of some device.

DEMO: ls -l /dev - note that ls displays major/minor instead of a size, since there really isn't a file at all, just a directory entry. Also note how the first character on the line is "c" for a character device and "b" for a block device (and "d" for subdirectories).

c. A driver actually consists of two major parts, connected by a shared data structure. (If multiple instances of a type of device exist on a system, then all share the same driver code, but each has its own data structure)



i. Application programs interact with the upper half of the driver, and receives requests to perform basic operations like read, write, seek, and control.

ii. The lower half of the driver interacts with the physical device, including issuing commands to the device based on operations requested by an application program, responding to interrupts, etc.

iii. A system may have configured into it many device drivers that do not correspond to any device actually present on a particular system. (Note the size of the listing produced by ls /dev earlier!)

When the system starts up, the operating system startup code "probes" the list of devices that might be present, and enables the drivers that correspond to devices that are actually present.

C. The abstract notion of a file.

1. Even the abstract notion of a device is too low-level for many purposes.

In fact, if the device is file-structured (like a disk), allowing ordinary users to access the device at this level would make file protection mechanisms useless. (Note how the special files that correspond to such devices allowed full access only to the file owner - root)

2. Therefore, most systems provide a higher level abstraction known as a FILE.

a. A file is named by a path whose format is system-specific.

b. A file may be associated either with

i. A physical device (e.g. on Unix systems a file name of the form /dev/___). (In this case, a further layer of abstraction is inserted between the program and the physical device.)

ii. A file recorded in the directory of a file-structured device. Such a file can only be accessed by ordinary programs at this level.

3. A specific capability added by this layer abstraction is the ability to do BUFFERED-IO - i.e. to transfer data between the device and a buffer in memory (using either DMA or interrupt-driven IO, as supported by the device, but the program need not know.)

4. This layer of abstraction is provided by the file system of the operating system - and is a topic for another course.