8. Interfacing Processors and Peripherals

Revisions

Date	Changes
March 23, 2003	More details on synchronous buses have been added to section 8.4
March 31, 2003	Details and more diagrams in the section on programmed I/O and DMA.

8.1. INTRODUCTION

Design Objectives:

- to maximize expandability
- to maximize performance, which as two different measures:
  - latency
  - throughput
- to maximize resilience

I/O Relegated to Second Class Status

I/O design is typically given the least attention yet it is what affects performance the most:

Impact of I/O on System Performance

Example:

A benchmark program executes in 100 seconds of elapsed time in which 90% of the time is spent in the CPU and 10% is in waiting for I/O.

If CPU execution speed improves by 50% each year for the next 5 years, but I/O performance does not improve, how much faster will the program run at the end of 5 years?

Solution

First, it is important to understand that an increase in speed of p percent implies that execution time is diminished by a factor of 100/(100 +p). To see this, suppose the current speed is M instructions per second. Then the new speed is M(1 + p/100) instructions per second. The amount of time it takes to execute M instructions is now

$$\frac{M \text{ instr}}{M(1+p/100) \text{ instr}/\text{sec}} = \frac{100}{100+p} \text{ sec}$$

In our example, a 50% increase in speed implies that the execution time is 2/3 of what it was. Each year, therefore, CPU time is reduced by 2/3. After n years have passed, it is $90 \cdot (2/3)^n$ seconds. In contrast I/O time stays the same, so after n years, total time is $10 + 90 \cdot (2/3)^n$ seconds.

The following table illustrates the changes in elapsed time and CPU time over the 5-year period of the problem

Years elapsed	CPU Time	I/O Time	Elapsed Time	% I/O Time
0	90	10	100	10
1	60	10	70	14
2	40	10	50	20
3	27	10	37	27
4	18	10	28	36
5	12	10	22	45

After 5 years, the elapsed time is 22 seconds; the speed increase is 100/22 = 4.5, even though the processor has increased by 90/12 = 7.5 times.

8.2. I/O Performance Measures

A confusion: I/O transfer rates are measured in different units: 1 MB = 10^6 bytes, not 2^20 bytes.

Supercomputer I/O Benchmarks

Dominated by file accesses, typically output is much greater than input.
I/O throughput is more important than response time or latency.

Transaction Processing I/O Benchmarks

Small I/O accesses, very frequent.
I/O rate = number of disk accesses/ second
Data rate = number of bytes/second
I/O rate is more important than data rate.
Sometimes, both are critical

File System I/O Benchmarks

UNIX example:

- most accesses are to small files (less than 10KB)
- most are sequential accesses
- more reads than writes

8.3. Types and Characteristics of I/O Devices

Large variety of I/O devices. Considerations:

- input or output or both

- partner: human or machine
- data rate: ranges from 0.01 KB/sec to 10,000KB/sec

Selected I/O Devices

Mouse

Either generates a sequence of pulses as it is moved, or increments and decrements counters. Also generates mouse button signals – button up, button down.

Because mouse signals are sparse relative to CPU cycles, **polling** is used to obtain data from the mouse.

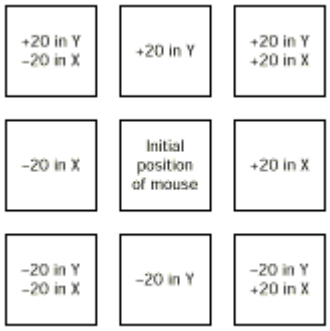


Figure 1 Mouse coordinates

Magnetic Disks

Features in common

- Rotating platter
- Magnetic coated surface
- Read/write head
- Nonvolatile

Differences

- hard disk is rigid and can be larger
- hard disk is denser because it can be controlled more precisely.
- hard disk has a higher data rate because it spins faster
- hard disks can have multiple platters

Hard Disks

Rotation speed from 3600 to 16,200 RPM. It is always a multiple of 1800. Common speeds are 7200 and 10,800 RPM.

- Platter — one disk of a set of coaxial disks
- Surface — one of the two surfaces of a platter
- Track — one of the concentric rings on a surface on which the data is recorded
- Sector — the consecutive sequence of bits on a track within a pre-specified arc of the circle.
- Cylinder — The set of tracks at the same position relative to the center, on all surfaces

Performance Costs

- Seek time — the time to position the disk heads on the specified cylinder
- Rotational Latency — the time to rotate the disks so that the sector to be read is under the head
- Transfer Rate — the rate at which data can be transferred between the disk and the disk controller's buffers.
- Controller overhead — the amount of time spent in the logic circuitry of the controller processing the requests.

Common Measures

Minimum seek time, maximum seek time, average seek time

Average rotational latency is between

½ rotation / rotations per sec = 0.5 /60 sec = 0.0083 ms

and

½ rotation / rotations per sec = 0.5 /120 sec = 0.0042 ms

Transfer time depends on sector size, recording density, and rotation speed.

Usually, disk controllers have a built-in cache that stores data from sectors passed over. Transfer rates from cache can be 40MB/sec as opposed to 2 to 15MB from the disk itself.

Example

What is the average time to read or write a 512 byte sector for a disk rotating at 5400 RPM, given advertised

- average seek time = 12 ms
- transfer rate = 5 MB/sec
- controller overhead = 2 ms

Time to read

= seek time + rotational latency + transfer time + controller overhead

= 12 ms + 0.5/(5400/60) sec + 512 bytes/(5 MB/sec) + 2 ms

= 12 ms + 5.6 ms + 0.1 ms + 2 ms

= 19.7 ms

Rotational delay can be the dominating cost if seek time is much less.

See Figure 8.6 for comparative performance characteristics.

Networks

Characteristics

- Distance: 0.01 to 10,000 km
- Speed: 0.001 to 100 MB/sec
- Topology: bus, ring, star, tree
- Shared Lines: none (point to point) or shared (multidrop)

RS232 standard

- 0.3 to 19.2 Kbit/sec terminal network
- point-to-point
- 10 to 100 meters in distance

Ethernet

- 10 Mbit to 100Mbit/sec bus
- packets range from 64 bytes to 1518 bytes
- shared line with multiple masters (distributed control)

Switched Networks

Switches are introduced to reduce the number of hosts per Ethernet segment.

Long-haul Networks

8.4. Buses

A ***bus*** is a shared communication path whose purpose is to allow the transfer of data among the devices connected to it. A bus includes address, control, and data lines as well as lines needed to support interrupts and bus arbitration. Control lines include a read/write line.

A ***bus protocol*** is a set of rules that govern the behavior of the bus. Devices connected to the bus have to use these rules to transfer data. The rules specify when to place data on the bus, when to assert and de-assert control signals, and so on.

Among methods of interconnecting components of a computer system, buses are

- versatile
- low-cost
- a major bottleneck

Bus speed is limited by:

- bus length
- number of attached devices

In any transfer operation, the device initiating the request is called the ***master***, and the one receiving the request is called the ***slave***. The processor is often the master, but memories can be masters as well.

Convention: Transfers are always from the perspective of the master, so an input operation transfers data from a slave device to the master and an output request transfers data from the master to a slave.

Buses can be ***synchronous*** or ***asynchronous***; synchronous buses use a clock to control transfers, whereas asynchronous buses do not.

Bus Types

Different kinds of buses are used to connect different parts of a computer system to each other.

Processor-Memory Bus

These are sometimes called ***processor buses***. They are short, high speed buses, usually designed for specific a memory and processor to maximize bandwidth. They can be used when all devices that can be attached are known in advance and have known characteristics. Typically these are **proprietary** buses.

I/O Bus

I/O buses have varying length and usually allow many different kinds of devices with varying speeds and data block sizes to be attached. I/O buses may be connected to the processor bus via a bridge or a separate controller interface, but usually they are connected to what is often called a system, or backplane, bus, described below. I/O buses are almost always **standard, off-the-shelf** components. Examples include SCSI buses and USB (Universal Serial Bus).

Backplane, or System, Bus

Most modern computers use a single, general-purpose bus to interconnect a variety of internal devices, including network interface cards, DMA controllers, and hard disk drives. These buses used to be called backplane buses but are now often called system buses. Usually, I/O buses are connected to the system bus, which in turn connects to the processor bus via a bridge. A picture to illustrate a typical combination is shown below.

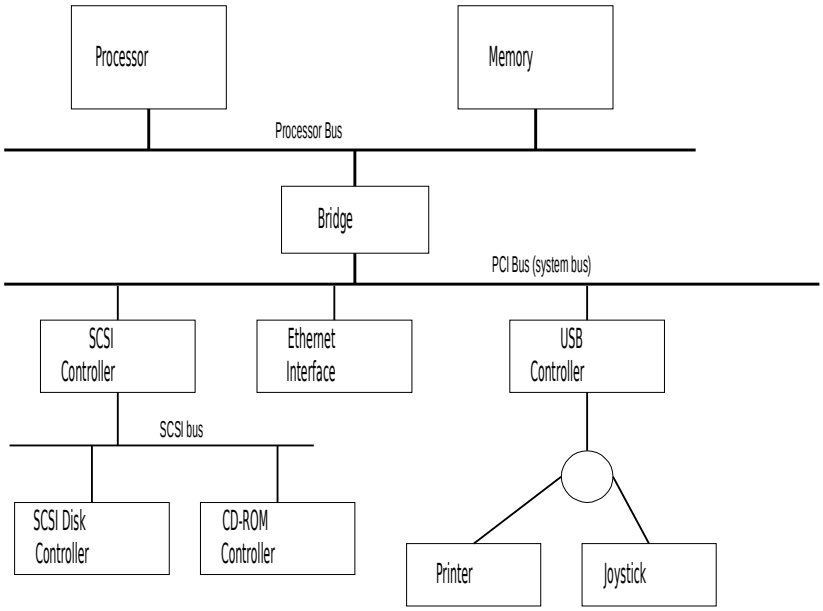


Figure 2 Typical bus organization

The book also shows more general combinations.

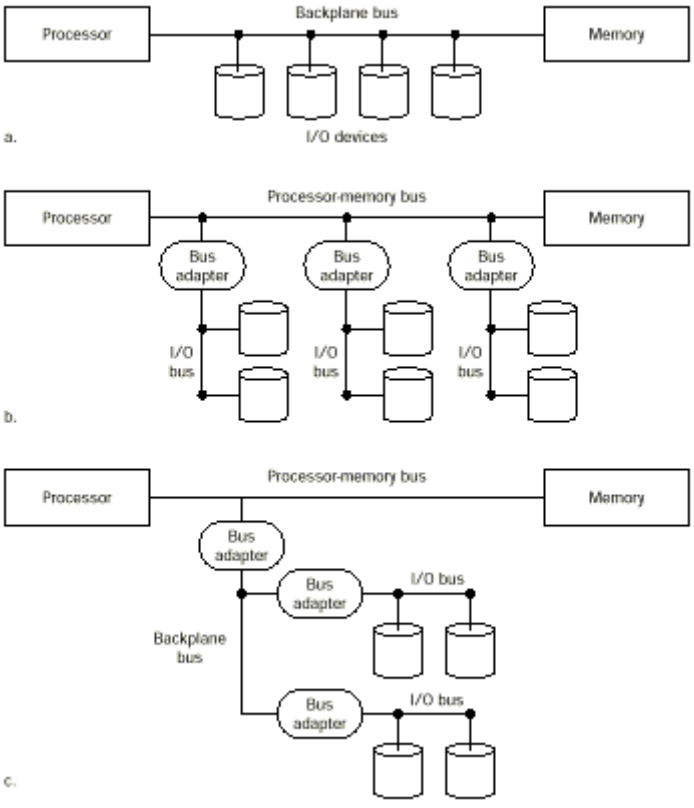


Figure 3 Three different bus organizations

a. Single bus for all communication (like in old PCs)

b. A separate processor-memory bus with speed-matching handled by bus adaptors to I/O buses. The processor memory bus is sometimes a PCI bus.

c. A single processor-memory bus separated from the I/O traffic by a backplane bus tapped in using a SCSI controller.

Synchronous Buses

A **synchronous** bus has a clock line accessible to all attached devices, and a clock that generates uniform **pulses** on the line. A pulse is an interval during which the clock line is in a high voltage state. The pulse is followed by equal interval of time during which the line is at low voltage state. The **pulse width** is the length of the pulse. A ***bus cycle*** is the interval between one or more consecutive pulses. In Figure 4, the pulse width is $t_1 - t_0$ and the cycle length is $t_2 - t_0$.

It is important to understand the steps involved in a bus transaction. The master places a command and an address on the bus. It takes a small amount of time for the command and address to appear on the bus. These signals must then propagate along the bus lines to all of the devices. The device whose address matches the address on the address lines must read the command from the command line and decode it. This step also takes a small amount of time. The device must then respond to the command. If it is a read command, for example, the device has to place the requested data on the data lines of the bus. This step also takes time. The data must then propagate to the master, which must then strobe the data into its own buffer. The data must be available on the bus for a period of time at least as long as the setup time of the buffer before the clock edge, and must remain on the bus after the clock edge for the hold time of the bus. (Recall that the setup time and hold time of a flip-flop are the amounts of time before and after the clock edge during which the data must be valid.)

The pulse width must be long enough to allow any signals (data, addresses, or control signals) to reach all devices and be decoded by them. Obviously, it must be longer than the maximum propagation delay between any pair of devices connected to the bus. This delay is determined by the bus's physical and electrical characteristics. The bus length is an example of a physical characteristic that affects the propagation delay. Bus impedance and voltage are examples of electrical characteristics affecting bus speed.

The clock pulse must also be longer than the time that the slowest device needs to decode address and control signals. Figure 4 shows conceptually how the bus works. The master puts a read command and an address on the address and command lines of the bus at time t_0, the start of a clock cycle. The pulse must be long enough so that the slave can detect the pulse, store the command and address signals in its buffers and decode them. The slave will not be allowed to place the data on the Data lines before the end of this pulse, because the line is unreliable at time t_1 because signals are changing state. The slave must wait until t_1. At t_1, the slave puts the data on the data line and leaves it there until t_2. The pulse width is long enough to guarantee that the master has strobed the data into its own buffers by t_2.

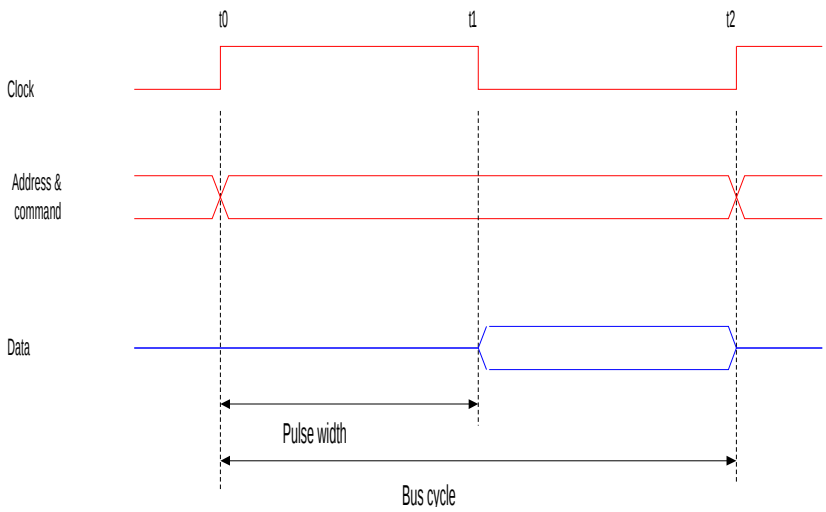


Figure 4 Conceptual diagram of synchronous bus

A more realistic timing diagram is shown in Figure 5. This figure shows the events as they are seen at the master and the slave, which are separated by some length of wire on the bus. The master sends the address and command at time t_0, but the signals do not actually appear on the bus until a short time later, at time t_{AM}, due to delays in the control logic inside the bus driver. The slave does not see the signals until time t_{AS} because of propagation delays. The slave decodes the address and command and sends the data at time t_1, when it sees the falling clock edge. The data do not actually appear on the bus until t_{DS}, and are not seen by the master until t_{DM}. The pulse is long enough so that $t_2 - t_{DM}$ is greater than the setup time of the master's buffer. The slave must keep the data on the bus long enough so that it is there after the hold time has transpired.

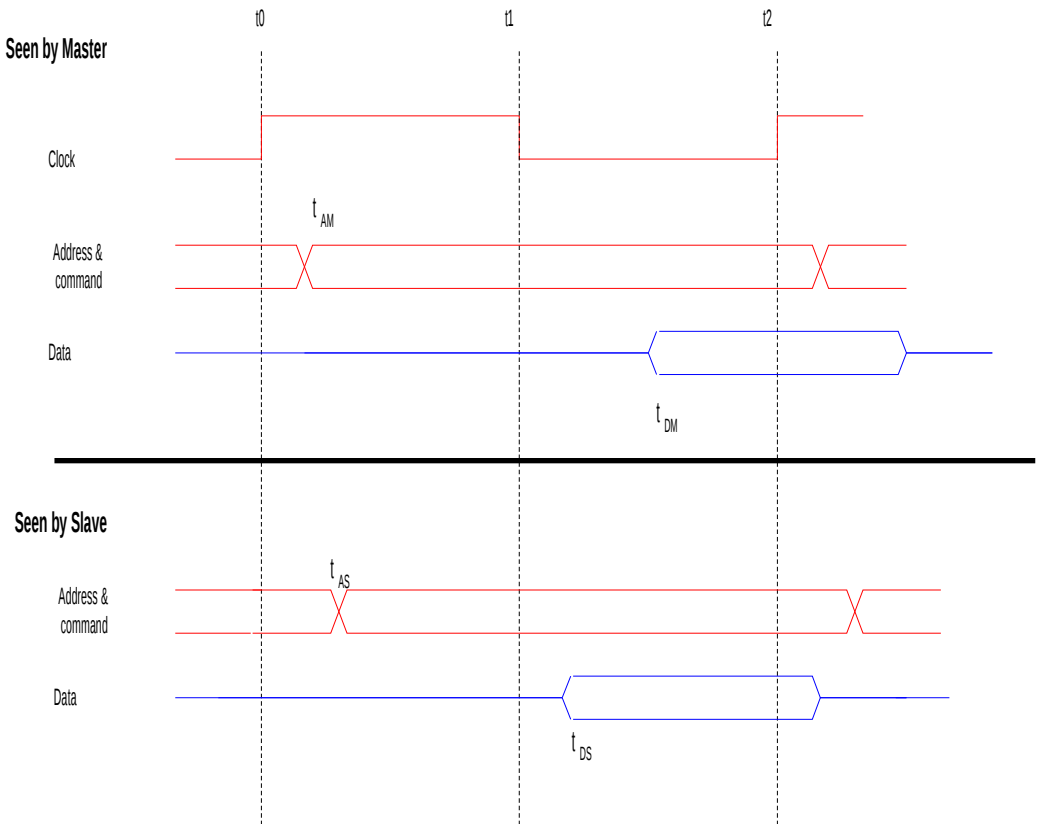


Figure 5 Realistic timing diagram of the bus transfer of Figure 4

The simplest synchronous bus makes the clock cycle long enough to accommodate all data transfers, for even the slowest devices. All data transfers take one clock cycle. This design is inefficient because all transfers take as long as the slowest device. More general synchronous buses use a higher frequency clock rate and allow different devices to use different numbers of clock cycles for a single transfer. One device might respond after two clock cycles and another might require five. In all synchronous buses, the bus protocol is based upon the use of clock cycles exclusively to coordinate transfers of data.

Example

An overly simplistic (and unrealistic) protocol to input a word from memory to the processor :

> transmit a read request on the R/W control line and address on data line
> wait 5 clock cycles;
> store data from memory into processor location

In reality, an additional control-line is used so that the master can determine whether the slave has actually responded. This signal is called the ***slave-ready*** signal. In a typical input transfer, the master puts the slave's address and a read command on the address and command lines respectively. (In the diagram below, this happens in clock cycle 1.) The slave receives this information, decodes it and responds by retrieving the data. Suppose the data is ready for the start of clock cycle 3. The slave puts the data on the data line and asserts the slave-ready line at the start of clock cycle 3. When the slave-ready signal reaches the master, which might be before or after the data arrives at the master, the master waits a small

amount of time to ensure that the data has arrived, and it strobes the data into its input buffer, in clock cycle 4. It then de-asserts the address and command lines.

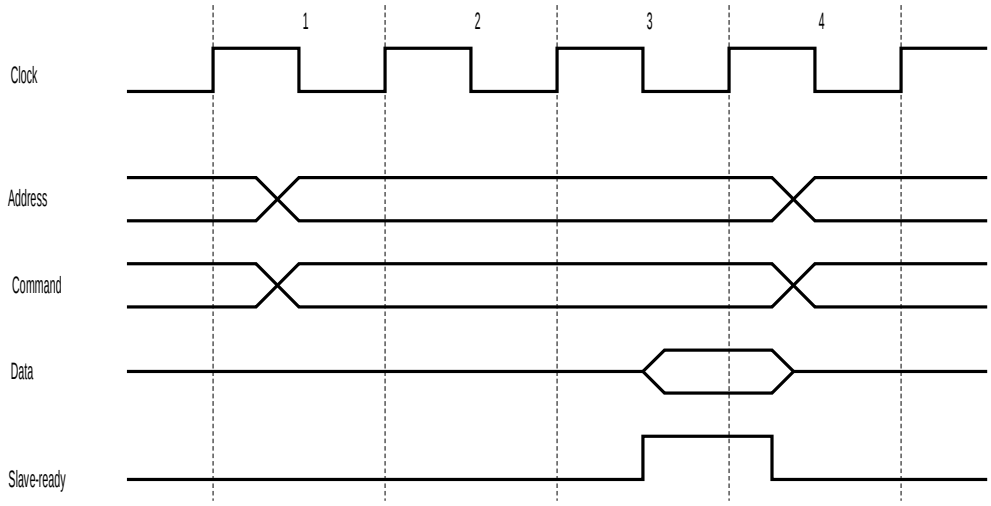


Figure 6 Read operation on a synchronous bus with a Slave-Ready line

Summary

Advantages of Synchronous Buses

- relatively simple control logic
- usually fast, because the protocol uses little bus overhead

Disadvantages of Synchronous Buses

- does not work if bus length is too great because clock skew (clock signals arriving at different times from different devices) can corrupt logic.
- must be designed to accommodate the slowest device.
- adding new devices complicates the design.

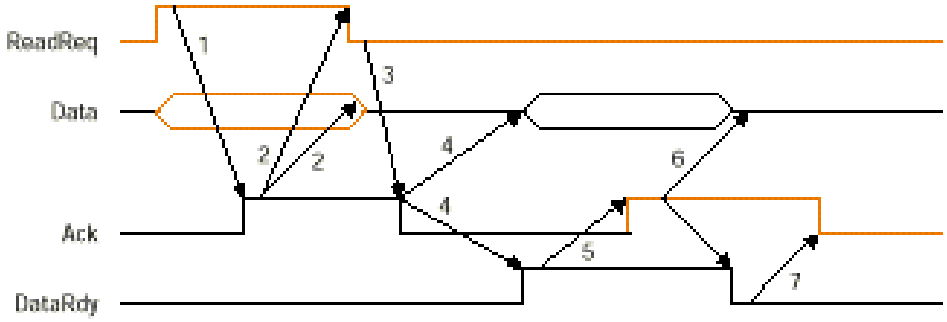
Asynchronous Buses

The PCI and ISA buses are examples of synchronous buses. An **asynchronous** bus is not clocked. Instead, devices communicate by exchanges messages with each other. These messages are called ***handshakes***. Because it is not clocked, it is a more versatile bus, but the complex, handshaking protocol can consume more overhead.

Handshaking is like the please-and-thank-you protocol that people use in everyday life. Each party in a communication expects a certain response to its message and until it receives that response, it does not advance to its next logical state. Think about a telephone call. When you call, you expect the person responding to answer with some verbal message such as "hello" or "this is so-and-so." That person expects you to reply with a greeting in return, and if no reply is forthcoming, he or she will hang up.

Example

Assume that an I/O device issues a read request to memory. The device is the master and memory is the slave. Assume 3 control lines: **ReadReq, DataRdy, Ack** and a data line.



The sequence is as follows. The times correspond to the end of the arrow, so t_4 is the time where the t_4 arrow ends.

Device raises **ReadReq** line and puts the address on the Data lines at time t_0.

1. At time t_1, memory sees **ReadReq** high; reads the address from the Data lines and raises **Ack** to acknowledge it has been seen. Memory can begin to retrieve the data as early as t_1.

2. At t_2, the **Ack** signal reaches the I/O device, which, on seeing **Ack** high, drops the **ReadReq** and Data lines.

3. At t_3, when memory sees the **ReadReq** drop, it knows that the device has seen the **Ack** it sent, so it drops the **Ack** line to signal to the device that it has seen the change in **ReadReq**. This is a way to tell the other party that this handshake is finished.

4. t_4 is the point at which memory has the data ready. Memory places the data on the Data lines and raises **DataRdy**. The amount of time it takes to do this is independent of the protocol, depending only on how long it takes memory to complete the access of its data.

5. At t_5, the I/O device sees that **DataRdy** is asserted, so it reads the data from the Data lines and raises **Ack** to signal that it has strobed the data.

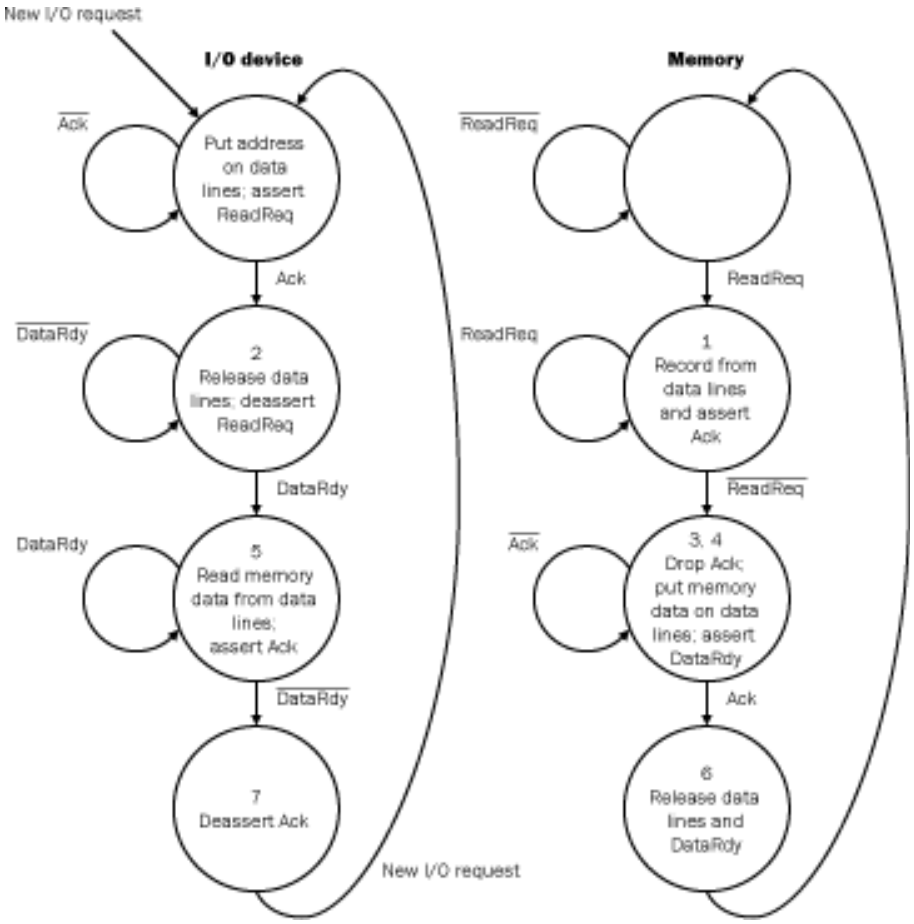
6. At t_6, memory sees **Ack** high, so it drops the **DataRdy** and releases the Data lines. This tells the I/O device that it has seen the **Ack** and is done.

7. The I/O device sees **DataRdy** go low at t_7 and drops the **Ack** line, indicating the end of the transmission.

This example uses full handshakes: a change of state in one control signal is followed by a change of state in the other. For example, raising **ReadReq** is followed by **Ack** being raised and dropping **ReadReq** is followed by **Ack** being dropped. Handshaking protocols can be modeled with finite state automata (FSA) also. Doing so is more complex because of the way devices receive signals. They sample the control lines. If clock rates are different, sampling can fail. **Synchronizers** can overcome this problem.

Advantages

- Handshake eliminates need for synchronization of clocks



- Can be unlimited length
- Can add devices or change delays without redesign of bus

Disadvantages

- Generally slower than synchronous buses

Performance Comparison of Synchronous Versus Asynchronous Buses

In an asynchronous bus, each full handshake requires two round-trip delays: the master signal must be received by the slave before it can reply, and the slave's reply to the master must be received by the master before it can proceed. Each transfer requires two full handshakes, so each transfer requires four round-trip delays. For transfers of small amounts of data this overhead can be a large percentage of the total transfer time. This overhead is absent in a synchronous bus.

On the other hand, in a synchronous bus, the number of clock cycles that the master must wait for a device's response is always the maximum possible delay to allow for the worst case delay. Synchronous buses therefore have a built-in overhead.

Which is faster depends on how much data is being transferred, though almost always a synchronous bus will be faster. *Most system and processor buses are synchronous buses*.

Example

Assume that a synchronous bus has a 50 ns bus cycle, and that a single bus transmissions takes 1 bus cycle.

Assume that an asynchronous bus has a 40 ns handshake delay, i.e., the one-way trip delay is 40 ns.

Assume that each has a 32-bit data line and are connected to a memory with a 200 ns response time to return one word of data, and compare the time to read one word from memory.

Synchronous Bus

1. Send address to memory:	50ns
2. Read memory:	200ns
3. Send data to the device:	50ns
Total time	**300ns**

Bandwidth = 1 word/300ns = 4 bytes/300ns = 13.3 MB/sec

Asynchronous Bus

There is overlap in times. The 40 ns trip time means that the difference $t_{n+1} - t_n$ in the step-by-step description is 40 ns for all n.

1. Read address from data line (t_1)	40 ns
2. Memory retrieves data during steps 2, 3, and 4 of protocol. The time at which the data can be sent is the larger of 3 handshakes (for 2,3,4) and the 200 ns access time, so it is max(120, 200)	200ns
3. Send the data to the I/O device: step t_7 is 3 handshakes later	120ns
Total time	**360ns**

Bandwidth = 4 bytes/360ns = 11.1MB/sec

This shows that the synchronous bus is faster in this case.

Increasing Bus Bandwidth

Bus bandwidth can be increased by several methods:

1. *Increasing the width of the data bus*: by increasing the number of data wires, multiple words can be transferred in fewer cycles.

2. *Separate data and address lines*: by providing separate bus lines for addresses and data, write operations can be performed faster since the address and data can be sent simultaneously.

3. *Block transfers*: by allowing the bus to transfer multiple words at a time without sending an address or releasing the bus with each one, overhead of transferring large blocks is reduced. This has no effect on the time to transfer single-word blocks.

Example: Performance of a Bus Using Block Transfers

Given the following characteristics:

1. Memory and the bus support bus accesses of from 4 to 16 32-bit words.

2. The bus is a 64-bit synchronous bus clocked at 200 MHz. Each 64-bit transfer takes one clock cycle.

3. It takes one cycle to send an address.

4. Two clock cycles are needed between each bus operation to reset the bus.

5. Memory needs 200 ns to access the first 4 words of a block transfer; each subsequent 4 words requires 20 ns.

6. Bus transfers and memory accesses can be overlapped.

Find the sustained bandwidth, latency, and transaction rate for a read of 256 words using

a. 4-word block transfers

b. 16-word block transfers.

The transaction rate is the number of bus transactions per second.

7. From the fact that the bus is clocked at 200MHz, and memory requires 200 ns for an access, an access takes 200/5 = 40 cycles for the first 4 words and 20/5 = 4 cycles for subsequent 4-word blocks.

Case a. 4-word transfers

There are 256/4 = 64 4-word transfers. Each 4-word transfer can be analyzed as follows:

1. To send the address,	1 clock cycle	(from 3 above)
2. Read 4 words from memory	40 clock cycles	(from 7 above)
3. Send the data to the master	2 clock cycles	2 64-bit transfers
4. idle cycles	2 clock cycles	(from 4 above)
Total	**45 cycles**	

Since there are 64 transactions, latency is 64*45 = 2880 cycles. Since each cycle is 5 ns, it is 2880*5 = 14,400 ns.

The bandwidth is (256 words) / 14,400 ns = 256*4 bytes / 14,400 ns = 71.11 MB / sec.

The transaction rate is 64 transactions / 14,400 ns = 4.44×10^6 transactions/sec.

Case b. 16-word transfers

There are 256/16 = 16 transactions. Each transaction is analyzed as follows:

1. To send the address,	1 clock cycle	(from 3 above)
2. Read 4 words from memory	40 clock cycles	(from 7 above)
3. Send 1^{st} 4 words to the master	2 clock cycles	2 64-bit transfers
4. Idle cycles waiting for memory access	2 clock cycles	

During steps 3 and 4, the next 4 words are accessed from memory, since the

second access needs only 4 cycles, so this access does not add more time.

5. Send the 2nd 4 words 2 clocks

6. Reset the bus 2 clocks

During these 2 steps the 3rd 4 words are accessed, without adding more time. Steps 5 and 6 are repeated for the 3rd group of 4 words, while the 4th group is accessed.

4 clocks

7. Send the last 4 words, and reset the bus 4 clocks

Total **57 cycles**

The latency is 57 * 16 = 912 clock cycles, which is 4560 ns. The bandwidth is 256*4 bytes/ 4560 ns or 224.56 MB/sec. This is roughly 3 times faster than the first case. The transaction rate is 16 transactions / 4560 ns = 3.51×10^6 transactions / sec.

Bus Arbitration

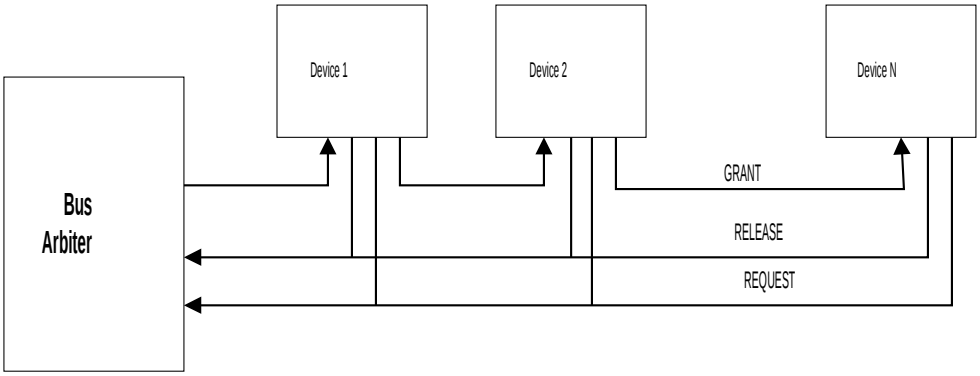
If a system has a single bus master, namely the processor, there is no question about which device is allowed to control the bus; it is always the processor. Memory is always a slave. Most modern computers allow multiple devices to be potential bus masters. DMA controllers, cache controllers, other bus interface controllers, and I/O devices can all vie for mastership of the bus.

The question is, when more than one device is trying to become a bus master, how is it decided which one becomes master first. The process of making this decision is called ***bus arbitration***.

Bus arbitration methods can be classified as *centralized* or *distributed*. In a centralized arbitration method, there is a single device called the bus arbiter whose purpose is to pick the next bus master. A centralized arbitration method always suffers from the fact that the single arbiter is a bottleneck. Centralized arbitration is almost always simpler to implement and therefore less costly. There are various methods of centralized arbitration, including daisy-chain arbitration, daisy-chain arbitration with polling, and parallel arbitration.

Daisy-Chain Arbitration

In daisy-chain arbitration, all devices are connected in a sequence starting at the arbiter and ending with the last device. Each device has a unique priority, which is simply its position relative to the arbiter: the closer the device is in the chain, the higher its priority. The diagram below illustrates.



The GRANT line runs through all devices from highest priority to lowest. Each device has a switch to intercept the signal in the GRANT line so that it does not reach the other devices. Each device is connected to the REQUEST and RELEASE lines. The GRANT line is maintained in a low voltage state when the bus is not in use. When a device wants to use the bus, it asserts the REQUEST line. The arbiter asserts the GRANT line in response. Devices gain mastership of the bus on rising edges of the GRANT line. The device closest to the arbiter that has a REQUEST intercepts the GRANT line, preventing it from reaching the other devices. It then drops the REQUEST line and uses the bus. When it is finished, it asserts the RELEASE line. The arbiter drops the GRANT line in response. If the REQUEST line is high, it asserts it again.

The daisy-chain is simple and slow. It favors devices of higher priority and can cause starvation of low priority devices.

An alternative to this simple daisy-chain design is a daisy-chain with polling. In the daisy-chain with polling, each device is given an integer code. The single GRANT line is replaced by n lines. A device that wants to use the bus asserts the REQUEST line. The device repeatedly puts one of 2^n possible integer codes on the n lines. If U_i is the current number on the polling lines, then the device whose code is U_i can become bus master. If this device is currently asserting the REQUEST line, it can use the bus; it asserts the BUSBUSY line. The arbiter stops asserting the poll lines while the bus is in use.

This latter design is a form of centralized, parallel arbitration, because although there is a single, central arbiter, there are multiple, parallel lines used for arbitration. In the first daisy-chain design, the arbitration is sequential – the grant signal passes from one device to the next, as opposed to be received by all of them simultaneously.

Distributed Arbitration by Self-Selection

In this design, every device is an arbiter; there is no central arbiter. Each device has a code that identifies it, and this code acts as its priority number – the higher the code, the higher the priority. The device with the highest code that has an outstanding request is the device that becomes the next bus master. The SCSI bus uses this strategy.

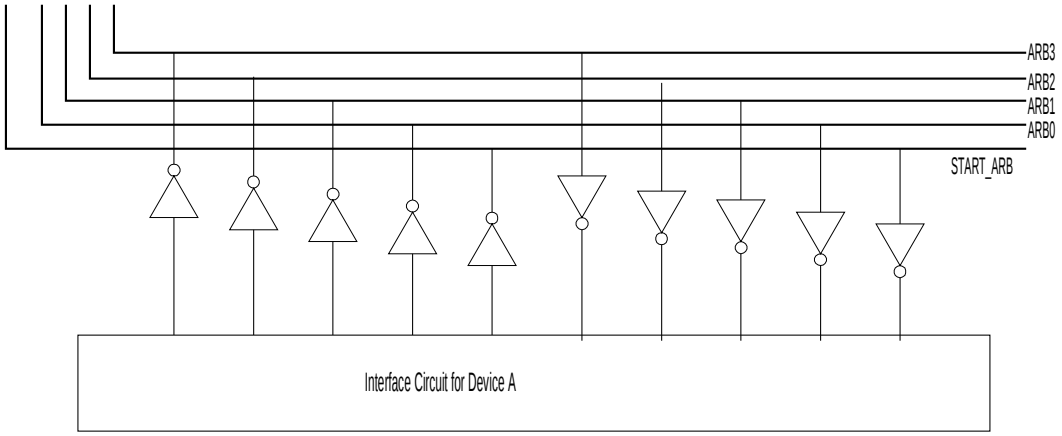
The figure below illustrates one method of implementing this method. In this example, there are four control lines, labeled ARB0, ARB1, ARB2, and ARB3. These are maintained in the high voltage state (they are called *open collector lines*). To drive one of these lines, a device opens a switch to ground, dropping the voltage. Each line is therefore the OR of the driving

devices, since if any device is driving the line, it is in a low voltage state, and only if none are is it in a high voltage state.

In this example, each device has a 4-bit code, allowing up to 16 devices to be supported by this bus. When a device wants to use the bus, it puts its 4-bit code on the collector lines and asserts the START_ARB line. Each device uses the following algorithm to decide if it is bus master:

Starting with the most significant bit position, compare the code bit of the device with the current value on the corresponding ARB line. If the values are the same, check the next lower bit. If there are no lower bits, the device is the bus master. If the values are different, disable the driver for this bit and all bits of lower significance.

Disabling the driver means that if it is sending a 1 bit, it sends a 0. If it was sending a 0, it has no effect.



The effect of this algorithm is that the device whose code is the largest will always win the competition for the bus. If A has code 5 (0101) and B has code 6 (0110) and they both put their codes on the collector lines at the same time, then the collector lines will have the code OR(0101, 0110) = 0111. A will compare 0101 to 0111 and set its drivers to 0100. This will cause the collector lines to become OR(0100, 0110) = 0110. B will have the matching code. Order has no effect – if B first compares to 0111, it will set its least bit to 0, but it is already 0.

Distributed Arbitration by Collision Detection

This is an arbitration method used by the Ethernet, which is not a computer bus, but a network data link. It is included in the book because an Ethernet is a bus, technically speaking, and it does have an arbitration strategy. In this strategy, a collision is said to occur if a device attempts to use the bus but it is already in use. In this case, the device waits an arbitrary interval and tries again. The length of the interval is randomized.

Summary of Cost versus Performance

In general, higher performance costs more. Performance can be improved through five different independent means. See the chart below.

Characteristic	High Performance Feature	Lost Cost Feature

bus width	separate address and data lines	multiplexed address and data lines
data width	more lines	fewer lines
transfer size	multi-word blocks	single word blocks
bus mastering	multiple masters	single master
clocking	synchronous	asynchronous

Software and the I/O Interface

This section addresses the following questions.

- How is an I/O request transformed into device-specific commands?
- How is the I/O request actually communicated to the I/O device?
- How is data transferred to or from the memory?
- What are the respective roles of the application level software, the operating system, and the hardware?

The answers to these questions will vary depending upon how the computer system is designed to be used. Most modern computers and operating systems support multi-processing, interactive use, and multiple users. To provide these features, the operating system must perform under the following constraints:

- The I/O system is shared among multiple processes.
- If processes are allowed to control devices directly, then throughput and response time will not be under the control of the operating system, and the computer system will perform poorly.
- If user processes are required to issue I/O requests directly to devices, then the programming task becomes much more tedious for the software engineer.
- I/O requests should therefore be handled by the operating system, and the operating system must determine when the requested I/O has completed.
- Because I/O is handled by the operating system, the operating system must provide equitable access to I/O resources and device abstraction.

Therefore the most common paradigm used in operating systems is a multi-layered approach to resources. Users make requests for I/O to application software. Application software translates these requests into calls to the operating system. The device drivers are the operating system's lowest level software; they are really two parts. One part issues I/O requests to the devices. The other part takes care of what happens when the I/O completes. Together they maintain the queues needed to coordinate the requests. The operating system notifies applications when the I/O is completed.

Methods of Controlling I/O

There are three methods of controlling I/O operations, one or more of which may be used in any single computer system:

Program-controlled I/O. I/O programs running on the processor repeatedly check the status of the device to synchronize with it during the exchange of data. This type of input/output control

was the only method available until the advent of interrupts. It is still an acceptable method of control for small, low-speed computer systems in which hardware costs must be kept small, and for transferring small amounts of data. In program-controlled I/O, a program running on the CPU executes instructions that direct the I/O device to transfer a word of data to or from the CPU's registers. The I/O device does not access memory; instead the CPU transfers the data between memory and its registers. Needless to say, this is not very efficient, because the CPU is completely consumed waiting for I/O transfers that are orders of magnitude slower than the CPU, and because every single word is moved from memory to the CPU to the device or vice-versa.

Interrupt-driven I/O. An I/O program running on the processor issues the I/O request and then goes into a waiting state. While that program is waiting it is removed from the processor and another program is run instead. Meanwhile the I/O device carries out the I/O operation. When the I/O completes, the I/O device sends a signal to the processor. This is better than program-controlled I/O because it frees the processor to do useful tasks instead of waiting. It requires more expensive hardware, which is standard on all modern computers.

Direct Memory Access (DMA). The processor, under program control, effectively authorizes the I/O device to take charge of the I/O transfers to memory, allowing it to be the bus master until the I/O is completed. An I/O device with this capability has what is called a DMA controller.

Program-controlled I/O.

In programmed I/O, every byte of data is transferred under the control of the CPU. An I/O program issues instructions to the I/O device to transfer the data. The data is transferred to or from memory by the program. On a read, for example, the program must request the input operation and then repeatedly test the status of a bit or register to see if the input is available. It does this in a "busy-waiting" loop in which it "polls" the device to see if it is ready. In effect, it is the nagging child on the long trip, "are we there yet, are we there yet, are we there yet,…" until at long last we have arrived.

This method is appropriate if the performance requirements are not great and the hardware does not support the other methods. In general, it is wasteful of computing cycles. An example of an IA-32 I/O program to read from the keyboard and echo the characters on the screen is shown below. IA-32 is a 32-bit Intel instruction format.

Example (Intel IA-32 instructions)

In the Intel-32 instruction set, there are two device status registers, INSTATUS and OUTSTATUS. The following program assumes that the keyboard synchronization flag is stored in bit 3 of INSTATUS and the display synchronization flag is stored in bit 3 of OUTSTATUS.

```
      LEA   EBP,LOC       # Register EBP points to LOC, the memory area
READ: BT    INSTATUS,3    # INSTATUS bit 3 is set if there is data in
      JNC   READ          # DATAIN; this loops waits for data
      MOV   AL,DATAIN     # Transfer char into register AL
      MOV   [EBP],AL      # Transfer AL contents to address in EBP
      INC   EBP           # and increment EBP pointer
ECHO: BT    OUTSTATUS,3   # Wait for display to be ready
      JNC   ECHO
      MOV   DATAOUT,AL    # Send char to display
      CMP   AL,CR         # If not carriage return,
      JNE   READ          #  read more chars
```

Notes.

- LEA *reg, addr* is an instruction to load the address *addr* into the pointer register *reg*.
- *BT is a bit-test instruction. It loads the value in bit 3 of the specified register into the carry bit; JNC will branch if the carry bit is 0.*

This program demonstrates another principle: **memory-mapped I/O**. In this program, the identifiers `DATAIN` and `DATAOUT` are mnemonic names for memory addresses that are mapped to I/O device registers. The ordinary `MOV` machine instruction

```
MOV AL, DATAIN
```

moves a character from the device register into register `AL`. In memory-mapped I/O, device locations are mapped to the memory address space and ordinary instructions are used to perform I/O. When the processor executes an instruction that references a memory-mapped I/O address, the hardware translates this to the corresponding I/O device register. Figure 7 illustrates the bus in memory-mapped I/O.

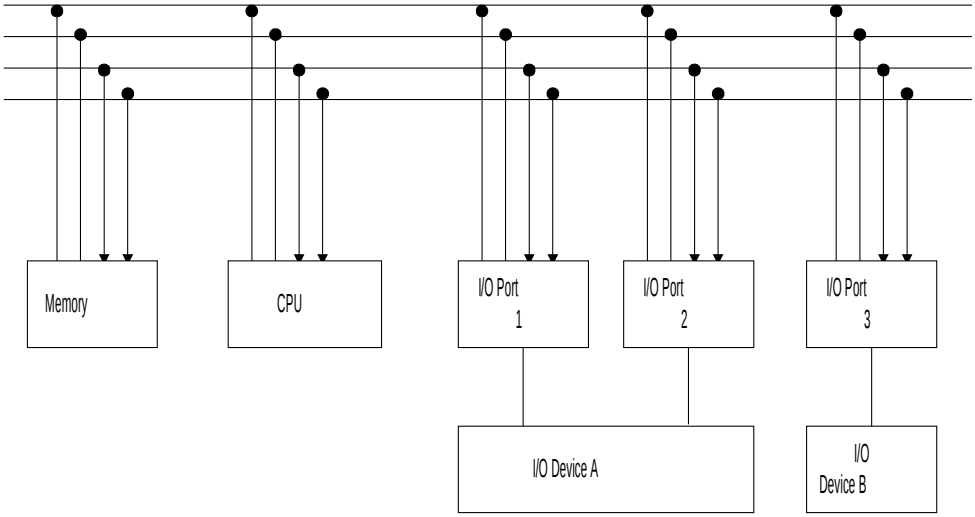


Figure 7 Memory-mapped I/O

In **isolated I/O**, or **I/O-mapped I/O**, the processor has special instructions that perform I/O and a separate address space for the I/O devices. The I/O address space is typically 256 addresses, with some systems, such as the Intel, supporting up to 64K indirect addresses. In isolated I/O, the same address lines are used to address memory and the I/O devices; the processor asserts a control line to indicate that the I/O devices should read the address lines. All I/O devices read the address but only one responds. For example, in the IA-32 instruction format,

```
IN REGISTER,DEVICE_ADDR
```

and

```
OUT DEVICE_ADDR,REGISTER
```

are the input and output instructions respectively. `DEVICE_ADDR` is an 8-bit address, and `REGISTER` is either `AL` or `EAX`. Notice that the IA-32 supports both memory-mapped and isolated I/O.

Figure 8 illustrates schematically how isolated I/O would look. There are separate physical lines for communicating with the I/O devices and memory. They have been labeled READIO and WRITEIO to symbolize that special commands would be issued to use these lines.

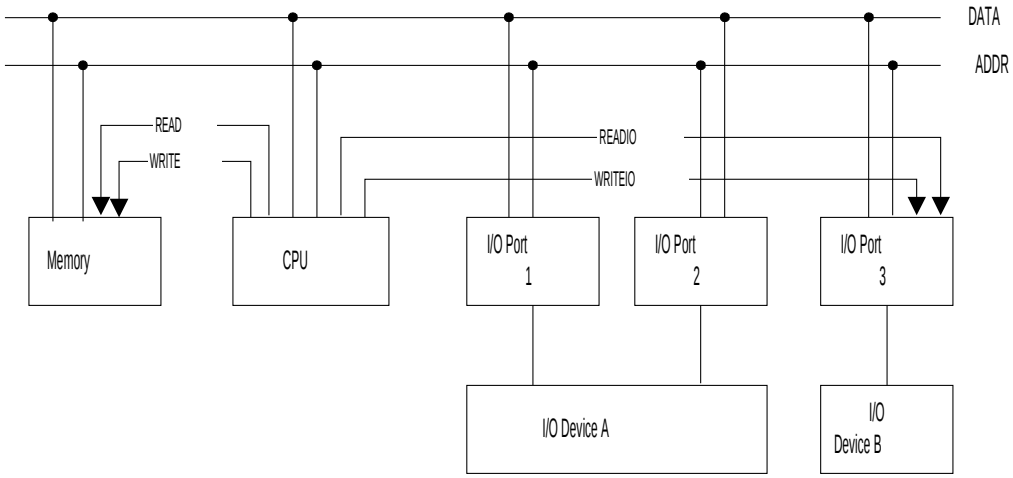


Figure 8 Isolated I/O

Interrupt-driven I/O

Programmed I/O's use of polling wastes much CPU time if the frequency of polling is great. The following example demonstrates this.

Example.

Assume that a computer has a 500 MHz clock, and that the instructions in the operating system that are executed to perform the polling operation use 400 clock cycles. What is the overhead to poll each of the following three devices, with the stated characteristics?

1. Mouse: can produce an event 30 times per second.
2. Floppy disk: can transfer 2-byte blocks at a rate of 50KB/second.
3. Hard disk: can transfer 4-word blocks at a rate of 4MB/second.

Solution.
The operating system uses 400 2ns clock cycles per polling operation, or 800 ns.

Mouse: 30*800 ns /second = 0.000024 or 0.0024%.

Floppy Disk: 50KBytes/sec at 2 bytes per transfer implies that it performs 25,000 transfers per second. The overhead is 25,000*800 ns/second = 0.02 or 2%.

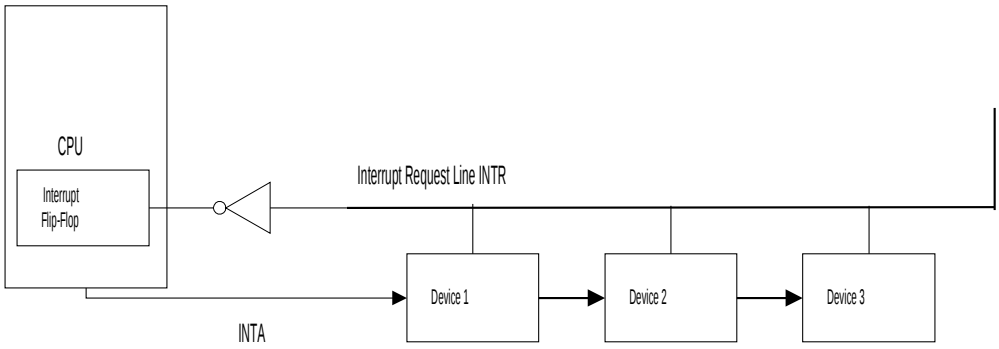
Hard Disk: 4MB/second at 4 words per transfer implies that it transfers 250,000 times per second. The overhead is 250,000*800 ns/ second = 0.2 or 20%.

This shows that the faster the device, the more overhead is involved in polling as a means of controlling I/O.

An alternative to programmed I/O is to add more signals to the bus and more logic to the I/O devices to allow them to notify the CPU when the device is ready for a new I/O operation, or

when an I/O operation is complete. There are various ways to arrange this, some more complex and flexible than others.

The simplest scheme is a single control line on the bus, usually denoted INTR. All devices share this line. The line is the logical OR of the interrupt requests of the attached devices. If any device has issued an interrupt, the processor receives the signal. The processor then has to determine which device issued the request, which it does by polling the status registers of the devices. Once it has determined which device issued the request, it



If interrupts arrive at the same time, the processor has to decide which to accept. Generally, it does this by assigning priorities to the devices. With a single interrupt line, priorities can be assigned by using a daisy chain scheme like the one used for bus arbitration. That is the purpose of the INTA line in the figure above, which runs through the devices. The processor sends an Interrupt Acknowledge signal through the line. The closest device that has an outstanding interrupt request intercepts the signal.

Another solution is to use a multiple-line interrupt system. In this case each device has a dedicated INTR line and a dedicated INTA line. The lines run into a priority arbitration circuit in the CPU. The CPU chooses the highest priority interrupt.

Handling Interrupts

When an interrupt occurs, the CPU needs to execute the subroutine that handles, or *services*, the interrupt. To do this, the contents of some of its registers must be saved. Modern processors typically allow two types of interrupts, one that forces a save of the entire register set, and another that saves only a minimal set of registers. The latter is the more efficient solution since only a few registers are saved, and the subroutine can choose to save whatever other registers it needs to in order to execute. This is faster than saving all registers, and avoids unnecessary work if there is no need to save them. The registers may be saved in special dedicated storage, or put on the system stack. The interrupt service routine (ISR) runs, saves whatever CPU state it must, and on completion, either the previous instruction sequence is resumed, or the scheduler is run to pick a different process to run.

How does the operating system know which ISR to execute? The answer is that once it knows which device caused the interrupt it knows which subroutine to run. In vectored interrupts, there is a portion of memory that contains an array, i.e., a vector, each of whose cells is the starting address of an interrupt service routine (ISR). Each device has an entry in this vector. When an interrupt occurs, the address of the device is used as an index into this array, and the starting

address of the interrupt service routine is automatically loaded, once the registers have been saved.

There are other issues related to interrupts. They include:

- Should interrupts be disabled while an interrupt service routine is running? If so, is the interrupt lost, or is it just that the response is delayed?
- If not disabled, what happens if an interrupt occurs while an interrupt service routine is running?

There are various answers to these questions, depending on the complexity of the system. Generally speaking, most modern machines have the ability to set the interrupt priority level of the processor. If an interrupt occurs that is lower priority than the current priority level, it is ignored. If one occurs that is a higher or equal priority, the currently running process is preempted in favor of the interrupt service routine that handles the new interrupt, regardless of what it was doing. Each device has an associated priority level, and the ISR for that device runs at that priority level. For example, the power supply can send an interrupt if it senses an impending loss of power. This is the highest priority level on many machines. The system timer is also very high priority; it must keep accurate time and uses very little CPU time when it runs, so it is reasonable to allow it to run whenever it needs to, which is on the order of 60 times per second.

Many architectures have an interrupt mask, which is a register with a bit for each interrupt line. If the bit is set, the interrupt is enabled; if not it is disabled. This is a way to select which interrupts the processor is willing to receive.

Overhead

The overhead of interrupts is much lower than that of program-controlled I/O. To illustrate, consider the hard disk from the example above. It has an overhead of 20%. Now suppose that the operating system uses 500 clock cycles of processing time to handle an interrupt and that the disk is only busy 5% of the time. Then for each of the 250,000 transfer per second that the disk generates, the interrupt service routine, uses 500*2 = 1000 ns, implying that the overhead is 5% of 250,000* 1000 ns per second, which is 1.25%. Recall that polling used 20% of the CPU time, so this is 93.75% reduction in overhead.

Direct Memory Access (DMA)

DMA, the acronym for "direct memory access" is a method I/O that reduces significantly the involvement of the processor in data transfers between memory and I/O devices. A ***DMA controller*** is an I/O processor that has the ability to communicate directly with memory, transferring large blocks of data between memory and the I/O devices to which it is attached. It achieves this because it is attached to the processor-memory bus on one side, and either an I/O bus or a dedicated device on the other, and it can be bus master on the memory bus. Typically, a single DMA controller will service multiple I/O devices. Certain devices, usually high-speed devices such as hard disks, CD-ROM drives, or network interfaces, may be equipped with DMA controllers. For example, a SCSI bus controller will have a DMA controller in its interface, making it possible for all devices on the SCSI bus to transfer data directly to or from memory with little CPU involvement. .

A program running on the CPU will give the DMA controller a memory address, a count of the number of bytes to transfer, and a flag indicating whether it is a read or a write. It will also give it the address of the I/O device involved in the I/O. The DMA controller becomes the bus master on the memory bus. If it is an input operation, the device will then start sending data to the DMA controller, which will buffer the data, and store it in successive memory locations as it becomes available. If it is an output location, it buffers the data from memory and send it to the I/O device as it becomes ready to receive it. In effect, it does the job the CPU would do, but the CPU is free to do other things in the meanwhile. Figure 9 depicts the circuitry in a typical DMA controller interface.

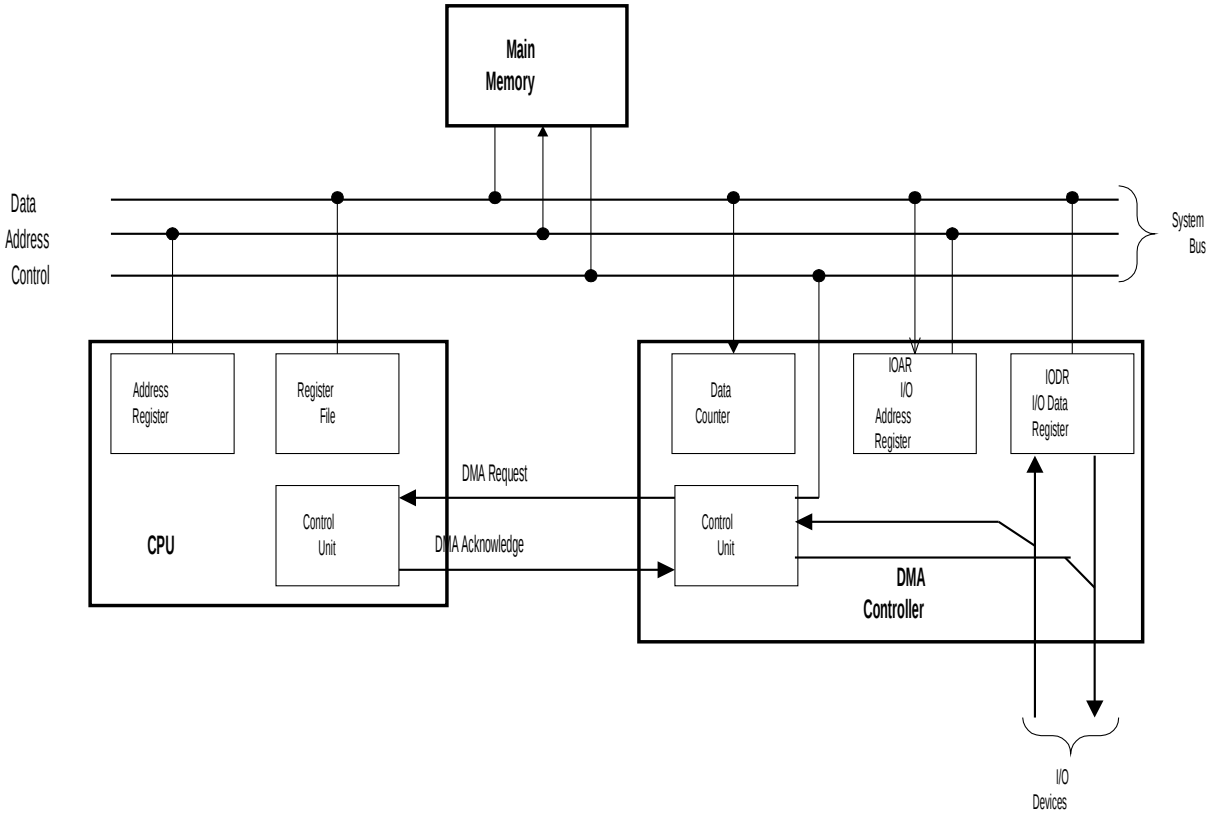


Figure 9 Circuitry for DMA

The typical sequence of operations in an input transfer would be:

1. The CPU executes two instructions to load the IOAR and the Data Counter. The IOAR gets the start address in memory of the first byte to be stored and the Data Counter gets a count of the number of bytes to be transferred.
2. When the DMA controller is ready, it activates the DMA request signal.
3. The CPU relinquishes control of the bus and activates DMA Acknowledge as part of the handshake.
4. The DMA controller begins the transfer of data to memory using a hardware loop to update the IOAR and Data Counter.
5. If the I/O device is not ready, but the transfer is not complete, the DMA controller relinquishes the bus so that the processor can use it.

6. If the I/O device was not ready and it becomes ready, the DMA controller re-acquires the bus in the same way it did in above.
7. When the Data Counter reaches 0, the DMA controller releases the bus and sends an interrupt to the CPU.

Because the DMA controller has the bus tied up during a transfer, the CPU will not be able to access memory. If the CPU or the cache controller needs to access memory, it will be delayed. For small data transfers this may be acceptable, but not for large transfers, because it somewhat defeats the purpose of using DMA in the first place. Therefore, DMA controllers usually operate in two modes, one for small transfers and one for larger transfers, in which "cycle-stealing" is allowed. In cycle-stealing, the CPU is allowed to interrupt the DMA transfer and acquire the bus.

DMA Overhead Example

Suppose that a system uses DMA for its hard disk. The system characteristics are:

- System Clock: 500 MHz (2 ns per cycle)
- Hard Disk can transfer at 4MB/second using an 8KB block size.
- 1000 clock cycles are used in the CPU to setup the I/O and 500 clock cycles are used afterwards in the CPU. What is the overhead of DMA transfers?

Each transfer takes 8KB / (4MB/ second) = 0.002 seconds (2 ms). Therefore there are

1.0 / 0.002 = 500 transfers per second.

If the disk is busy then it takes (1000 + 500) * 2 ns per transfer, or 3000 ns per transfer. Since there are 500 transfers per second, the total overhead is 500*3000 ns per second, or 1.5 ms per second, which is 0.15%.

DMA and Virtual Memory

In a system with virtual memory, DMA poses a problem – should the DMA controller use virtual or physical addresses? If it uses physical addresses, then it cannot perform reads or writes that cross page boundaries. To see this, imagine that pages are each 1 KB. If the DMA controller tries to write 3KB of data, then it will write into 3 or 4 physical pages. These pages may not belong to the same process, and may not be logically adjacent, and should be placed in the logically correct memory locations. On the other hand, if it uses virtual addresses, it will need to translate every address, slowing things down considerably and requiring a large RAM of its own. The solution is for the DMA controller to keep a cache of translations in its memory and update it using a replacement strategy such as LRU.

This is still inadequate because the page translations it has may go stale if the processor updates the page tables independently. For DMA to work properly, the processor must be prevented from changing the page tables during a DMA transfer.

DMA and Cache

DMA also creates problems with the cache. If the DMA controller is reading data directly from the disk into memory, then cache blocks may become stale, because the cache blocks will not be consistent with their corresponding memory blocks. Similarly, the DMA controller might read from memory and get stale data because the system has a write-update cache that has not yet been flushed to memory. There are a few solutions:

- Route all I/O through the cache.
- Flush the cache for I/O writes and invalidate it for I/O reads.

To make this efficient, special hardware is provided for flushing the cache.