13
I/O and Storage Devices

We have now arrived at a very interesting point in our study of processors. We have learned how to design a full processor and its accompanying memory system using basic transistors. This processor can execute the entire *SimpleRisc* instruction set, and can run very complicated programs ranging from chess games to weather simulations. However, there is a vital aspect missing in our design. There is no way for us to communicate with our computer. To render our computer usable, we need to have a method to write input data, and display the outputs.

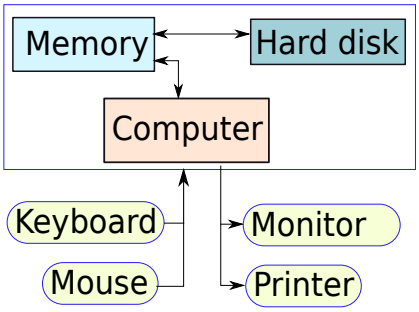


Figure 13.1: A typical computer system

We thus need an I/O (Input/Output) system in our computer. Let us look at the structure of a typical computer in Figure 13.1. The processor is the core of the computer. It is connected to a host of I/O devices for processing user inputs, and for displaying results. These I/O devices are known as *peripherals*. The most common user input devices are the keyboard and the mouse. Likewise, the most common display devices, are the monitor, and the printer. The computer can also communicate with a host of other devices such as cameras, scanners, mp3 players, camcorders, microphones, and speakers through a set of generic I/O ports. An I/O port consists of: (1) a set of metallic pins that help the processor to connect with external

devices, and (2) a port controller that manages the connection with the peripheral device. The computer can also communicate with the outside world through a special peripheral device called a network card. The network card contains the circuits to communicate with other computers via wired or wireless connections.

Definition 140
An I/O port consists of a set of metallic pins that are used to attach to connectors provided by external devices. Every port is associated with a port controller that co-ordinates the exchange of data on the communication link.

We give a special preference to a particular class of devices in this chapter known as *storage* devices. Storage devices such as the hard disk, and flash drives help us permanently store data even when the system is powered off. We looked at them briefly in Chapter 11 while discussing swap space. In this chapter, we shall study them in more detail, and look at the methods of data storage, and retrieval. The reason we stress on storage devices in this chapter is because they are integral to computer architecture. The nature of peripherals across computers varies. For example, a given computer might have a microphone, whereas another computer might not have a monitor because it is accessed remotely over the network. However, invariably all computers from small handheld phones to large servers have some form of permanent storage. This storage is used to save files, system configuration data, and the swap space during the operation of a program. Hence, architects pay special attention to the design an optimization of storage systems, and no book in computer architecture is complete without discussing this vital aspect of computer architecture.

13.1 I/O System – Overview

13.1.1 Overview

Let us now distance ourselves from the exact details of an I/O device. While designing a computer system, it is not possible for designers to consider all possible types of I/O devices. Even if they do, it is possible that a new class of devices might come up after the computer has been sold. For example, tablet PCs such as the Apple iPad were not there in 2005. Nonetheless, it is still possible to transfer data between an iPad and older PCs. This is possible because most designers provide standard interfaces in their computer system. For example, a typical desktop or laptop has a set of USB ports. Any device that is compliant with the USB specification can be connected to the USB port, and can then communicate with the host computer. Similarly, it is possible to attach almost any monitor or projector with any laptop computer. This is because laptops have a generic DVI port that can be connected to any monitor. Laptop companies obey their part of the DVI specification by implementing a DVI port that can seamlessly transfer data between the processor and the port. On similar lines, monitor companies obey their part of the DVI specification by ensuring that their monitors can seamlessly display all the data that is being sent on the DVI port. Thus, we need to ensure that our computer provides support for

a finite set of interfaces with peripherals. It should then be possible to attach any peripheral at run time.

The reader should note that just because it is possible to attach a generic I/O device by implementing the specification of a port, it does not mean that the I/O device will work. For example, we can always connect a printer to the USB port. However, the printer might not be able to print a page. This is because, we need additional support at the software level to operate the printer. This support is built into the printer device drivers in the operating system that can efficiently transfer data from user programs to the printer.

Roles of Software and Hardware

We thus need to clearly differentiate between the roles of software and hardware. Let us first look at software. Most operating systems define a very simple user interface for accessing I/O devices. For example, the Linux operating system has two system calls, *read* and *write*, with the following specifications.

```
read(int file_descriptor, void *buffer, int num_bytes)
write(int file_descriptor, void *buffer, int num_bytes)
```

Linux treats all devices as files, and allots them a file descriptor. The file descriptor is the first argument, and it specifies the id of the device. For example, the speaker has a certain file descriptor, and a printer has a different file descriptor. The second argument points to an area in memory that contains the source or destination of the data, and the last argument represents the number of bytes that need to be transferred. From the point of view of a user, this is all that needs to be done. It is the job of the operating system's device drivers, and the hardware to co-ordinate the rest of the process. This approach has proved to be an extremely versatile method for accessing I/O devices.

Unfortunately, the operating system needs to do more work. For each I/O call it needs to locate the appropriate device driver and pass on the request. It is possible that multiple processes might be trying to access the same I/O device. In this case, the different requests need to be properly scheduled.

The job of the device driver is to interface with native hardware and perform the desired action. The device driver typically uses assembly instructions to communicate with the hardware device. It first assesses its status, and if it is free, then it asks the peripheral device to perform the desired action. The device driver initiates the process of transfer of data between the memory system and the peripheral device.

Figure 13.2 encapsulates the discussion up till now. The upper part of the diagram shows the software modules (application, operating system, device driver), and the lower part of the diagram shows the hardware modules. The device driver uses I/O instructions to communicate with the processor, and the processor then routes the commands to the appropriate I/O device. When the I/O device has some data to send to the processor, it sends an interrupt, and then the interrupt service routine reads the data, and passes it on to the application.

We summarized the entire I/O process in just one paragraph. However, the reader should note that this is an extremely complicated process, and entire books are devoted to the study and design of device drivers. In this book, we shall limit our discussion to the hardware part of the I/O system, and take a cursory look at the software support that is required. The

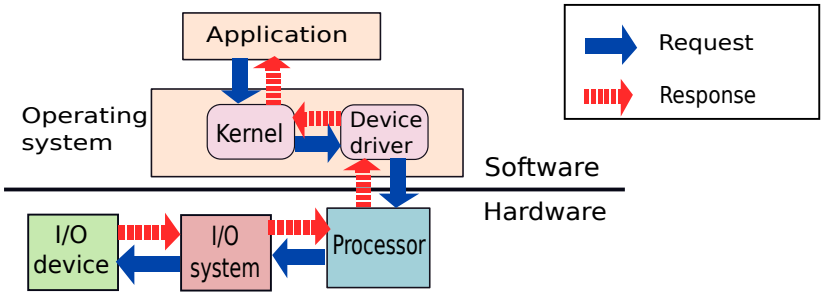


Figure 13.2: The I/O system (software and hardware)

important points of difference between the software and hardware components of the I/O system are enumerated in Point 21.

Important Point 21
The role of software and hardware in the I/O system:

1. *The software component of the I/O system consists of the application, and the operating system. The application is typically provided a very simple interface to access I/O devices. The role of the operating system is to collate the I/O requests from different applications, appropriately schedule them, and pass them on to the corresponding device drivers.*
2. *The device drivers communicate with the hardware device through special assembly instructions. They co-ordinate the transfer of data, control, and status information between the processor and the I/O devices.*
3. *The role of the hardware (processor, and associated circuitry) is to just act as a messenger between the operating system, and the I/O devices, which are connected to dedicated I/O ports. For example, if we connect a digital camera to a USB port, then the processor is unaware of the details of the connected device. Its only role is to ensure seamless communication between the device driver of the camera, and the USB port that is connected to the camera.*
4. *It is possible for the I/O device to initiate communication with the processor by sending an interrupt. This interrupts the currently executing program, and invokes the interrupt service routine. The interrupt service routine passes on control to the corresponding device driver. The device driver then processes the interrupt, and takes appropriate action.*

Let us now discuss the architecture of the hardware component of the I/O system in detail.

13.1.2 Requirements of the I/O System

Device	Bus Technology	Bandwidth	Typical Values
Video display via graphics card	PCI Express (version 4)	High	1-10 GB/s
Hard disks	ATA/SCSI/SAS	Medium	150-600 MB/s
Network card (wired/wireless)	PCI Express	Medium	10-100 MB/s
USB devices	USB	Medium	60-625 MB/s
DVD audio/video	PCI	Medium	1-4 MB/s
Speaker/Microphone	AC'97/Intel High. Def. Audio	Low	100 KB/s to 3 MB/s
Keyboard/Mouse	USB/PCI	Very Low	10-100 B/s

Table 13.1: List of I/O devices along with bandwidth requirements (as of 2012)

Let us now try to design the architecture of the I/O system. Let us start out by listing out all the devices that we want to support, and their bandwidth requirements in Table 13.1. The component that requires the maximum amount of bandwidth is the display device (monitor, projector, TV). It is attached to a graphics card. The *graphics card* contains the graphics processor that processes image and video data.

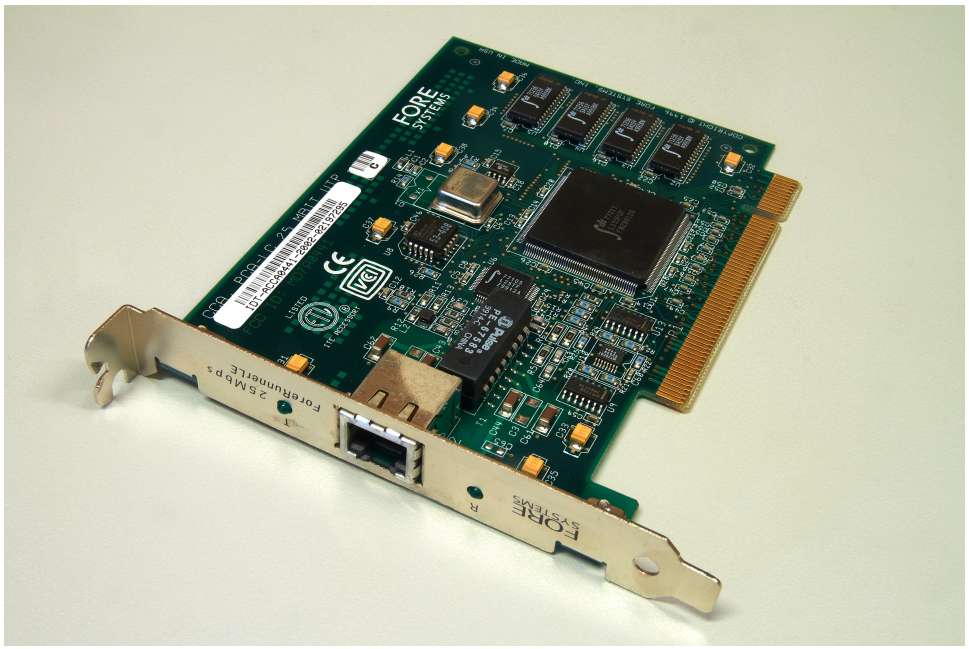


Figure 13.3: Photograph of a network card This article uses material from the Wikipedia article "Network Interface Controller" [nic,], which is released under the Creative Commons Attribution-Share-Alike License 3.0 [ccl,]

Note that we shall use the term *card* often in the discussion of I/O devices. A card is a printed circuit board(PCB), which can be attached to the I/O system of a computer to

implement a specific functionality. For example, a graphics card helps us process images and videos, a sound card helps us process high definition audio, and a network card helps us connect to the network. The picture of a network card is shown in Figure 13.3. We can see a set of chips interconnected on a printed circuit board. There are a set of ports that are used to connect external devices to the card.

Definition 141
A card *is a printed circuit board(PCB), which can be attached to the I/O system of a computer to implement a specific functionality.*

Along with the graphics card, the other high bandwidth device that needs to be connected to the CPU is the main memory. The main memory bandwidth is of the order of 10-20 GB/s. Hence, we need to design an I/O system that gives special treatment to the main memory and the graphics card.

The rest of the devices have a relatively lower bandwidth. The bandwidth requirement of the hard disk, USB devices, and the network card is limited to 500-600 MB/s. The keyboard, mouse, CD-DVD drives, and audio peripherals have an extremely minimal bandwidth requirement ($< 3 - 4$ MB/s).

13.1.3 Design of the I/O System

I/O Buses

Let us take a look at Table 13.1 again. We notice that there are different kinds of bus technologies such as USB, PCI Express, and SATA. Here, a bus is defined as a link between two or more than two elements in the I/O system. We use different kinds of buses for connecting different kinds of I/O devices. For example, we use the USB bus to connect USB devices such as pen drives, and cameras. We use SATA or SCSI buses to connect to the hard disk. We need to use so many different types of buses for several reasons. The first is that the bandwidth requirements of different I/O devices are very different. We need to use extremely high speed buses for graphics cards. Whereas, for the keyboard, and mouse, we can use a significantly simpler bus technology because the total bandwidth demand is minimal. The second reason is historical. Historically, hard disk vendors have used the SATA or IDE buses, whereas graphics card vendors have been using the AGP bus. After 2010, graphics card companies shifted to using the PCI Express bus. Hence, due to a combination of factors, I/O system designers need to support a large variety of buses.

Definition 142
A bus *is a set of wires that is used to connect multiple devices in parallel. The devices can use the bus to transmit data and control signals between each other.*

Now, let us look deeper into the structure of a bus. A bus is much more than a set of copper wires between two end points. It is actually a very complex structure and its specifications are typically hundreds of pages long. We need to be concerned about its electrical properties, error control, transmitter and receiver circuits, speed, power, and bandwidth. We shall have ample opportunity to discuss high speed buses in this chapter. Every node (source or destination) connected to a bus requires a *bus controller* to transmit and receive data. Although the design of a bus is fairly complicated, we can abstract it as a logical link that seamlessly and reliably transfers bytes from a single source to a set of destinations.

The Chipset and Motherboard

For designing the I/O system of a computer, we need to first provide external I/O ports that consist of a set of metallic pins or sockets. These I/O ports can be used to attach external devices. The reader can look at the side of her laptop or the back of her desktop to find the set of ports that are supported by her computer. Each port has a dedicated port controller that interfaces with the device, and then the port controller needs to send the data to the CPU using one of the buses listed in Table 13.1.

Here, the main design issue is that it is not possible to connect the CPU to each and every I/O port through an I/O bus. There are several reasons for this.

1. If we connect the CPU to each and every I/O port, then the CPU needs to have bus controllers for every single bus type. This will increase the complexity, area, and power utilization of the CPU.
2. The number of output pins of a CPU is limited. If the CPU is connected to a host of I/O devices, then it requires a lot of extra pins to support all the I/O buses. Most CPUs typically do not have enough pins to support this functionality.
3. From a commercial viewpoint, it is a good idea to separate the design of the CPU from the design of the I/O system. It is best to keep both of them separate. This way, it is possible to use the CPU in a large variety of computers.

Hence, most processors are connected to only a single bus, or at most 2 to 3 buses. We need to use ancillary chips that connect the processor to a host of different I/O buses. They need to aggregate the traffic from I/O devices, and properly route data generated by the CPU to the correct I/O devices and vice versa. These extra chips comprise the *chipset* of a given processor. The chips of the chipset are interconnected with each other on a printed circuit board known as the *motherboard*.

Definition 143

Chipset *These are a set of chips that are required by the main CPU to connect to main memory, the I/O devices, and to perform system management functions.*

Motherboard *All the chips in the chipset are connected to each other on a printed circuit board, known as the* motherboard.

Architecture of the Motherboard

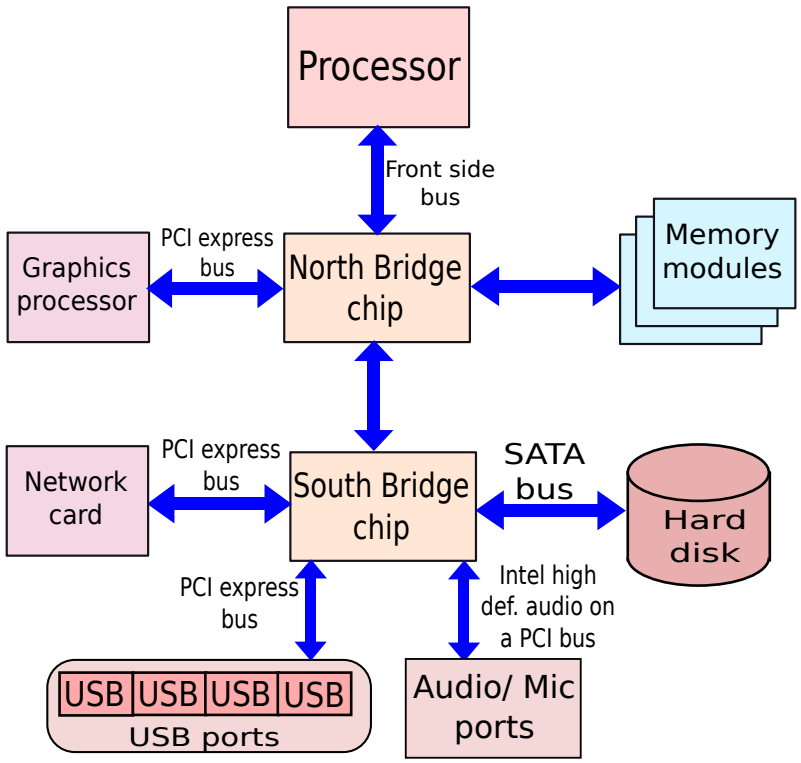


Figure 13.4: Architecture of the I/O system

Most processors typically have two important chips in their chipset – North Bridge and South Bridge – as shown in Figure 13.4. The CPU is connected to the North Bridge chip using the Front Side Bus (FSB). The North Bridge chip is connected to the DRAM memory modules, the graphics card, and the South Bridge chip. In comparison, the South Bridge chip is meant to handle much slower I/O devices. It is connected to all the USB devices including the keyboard and mouse, audio devices, network cards, and the hard disk.

For the sake of completeness, let us mention two other common types of buses in computer systems. The first type of bus is called a *back side bus*, which is used to connect the CPU to the L2 cache. In the early days, processors used off chip L2 caches. They communicated with them through the back side bus. Nowadays, the L2 cache has moved on chip, and consequently the back side bus has also moved on chip. It is typically clocked at the core frequency, and is a very fast bus. The second type of bus is known as a *backplane bus*. It is used in large computer or storage systems that typically have multiple motherboards, and peripheral devices such as hard disks. All these entities are connected in parallel to a single backplane bus. The backplane bus itself consists of multiple parallel copper wires with a set of connectors that can be used to attach devices.

Definition 144

Front side bus *A bus that connects the CPU to the memory controller, or the North Bridge chip in the case of Intel systems.*

Back side bus *A bus that connects the CPU to the L2 cache.*

Backplane bus *A system wide bus that is attached to multiple motherboards, storage, and peripheral devices.*

Both the North Bridge, and South Bridge chips need to have bus controllers for all the buses that they are attached with. Each bus controller co-ordinates the access to its associated bus. After successfully receiving a *packet* of data, it sends the packet to the destination (towards the CPU, or the I/O device). Since these chips interconnect various types of buses, and temporarily buffer data values if the destination bus is busy, they are known as *bridges* (bridge between buses).

The memory controller is a part of the North Bridge chip and implements read/write requests to main memory. In the last few years processor vendors have started to move the memory controller into the main CPU chip, and also make it more complicated. Most of the augmentations to the memory controller are focused on reducing main memory power, reducing the number of refresh cycles, and optimizing performance. Starting from the Intel Sandybridge processor the graphics processor has also moved on chip. The reason for moving things into the CPU chip is because (1) we have extra transistor's available, and (2) on-chip communication is much faster than off-chip communication. A lot of embedded processors also integrate large parts of the South Bridge chip, and port controllers, along with the CPU in a single chip. This helps in reducing the size of the motherboard, and allows more efficient communication between the I/O controllers and the CPU. Such kind of systems are known as SOCs (System on Chip).

Definition 145
A system on a chip (SOC) typically packages all the relevant parts of a computing system into one single chip. This includes the main processor, and most of the chips in the I/O system.

13.1.4 Layers in the I/O System

Most complex architectures are typically divided into layers such as the architecture of the internet. One layer is mostly independent of the other layer. Hence, we can choose to implement it in any way we want, as long as it adheres to a standard interface. The I/O architecture of a modern computer is also fairly complicated, and it is necessary to divide its functionality into different layers.

We can broadly divide the functionality of the I/O system into four different layers. Note that our classification of the functionality of an I/O system into layers is broadly inspired from the 7 layer OSI model for classifying the functions of wide area networks into layers. We try to conform as much as possible to the OSI model such that readers can relate our notion of layers to concepts that they would study in a course on networking.

Physical Layer The physical layer of a bus primarily defines the electrical specifications of the bus. It is divided into two sublayers namely the *transmission sublayer* and the *synchronization sublayer*. The transmission sublayer defines the specifications for transmitting a bit. For example, a bus can be active high (logical 1, if the voltage is high), and another bus can be active low (logical 1, if the voltage is zero). Today's high speed buses use high speed differential signaling. Here, we use two copper wires to transmit a single bit. A logical 0 or 1 is inferred by monitoring the sign of the difference of voltages between the two wires (similar to the concept of bit lines in SRAM cells). Modern buses extend this idea and encode a logical bit using a combination of electrical signals. The synchronization sublayer specifies the timing of signals, and methods to recover the data sent on the bus by the receiver.

Data Link Layer The data link layer is primarily designed to process logical bits that are read by the physical layer. This layer groups sets of bits into frames, performs error checking, controls the access to the bus, and helps implement I/O transactions. In specific, it ensures that at any point of time, only one entity can transmit signals on the bus, and it implements special features to leverage common message patterns.

Network Layer This layer is primarily concerned with the successful transmission of a set of frames from the processor to an I/O device or vice versa through various chips in the chip set. We uniquely define the address of an I/O device, and consider approaches to embed the addresses of I/O devices in I/O instructions. Broadly we discuss two approaches – I/O port based addressing, and memory-mapped addressing. In the latter case, we treat accesses to I/O devices, as regular accesses to designated memory locations.

Protocol Layer The top most layer referred to as the *protocol layer* is concerned with executing I/O requests end to end. This includes methods for high level communication between the processor, and the I/O devices in terms of the message semantics. For example, I/O devices can interrupt the processor, or the processor can explicitly request the status of each I/O device. Secondly, for transferring data between the processor and devices, we can either transfer data directly or delegate the responsibility of transferring data to dedicated chips in the chipset known as DMA controllers.

Figure 13.5 summaries the 4 layered I/O architecture of a typical processor.

13.2 Physical Layer – Transmission Sublayer

The physical layer is the lower most layer of the I/O system. This layer is concerned with the physical transmission of signals between the source and receiver. Let us divide the physical layer into two sublayers. Let us call the first sublayer as the *transmission sublayer* because it

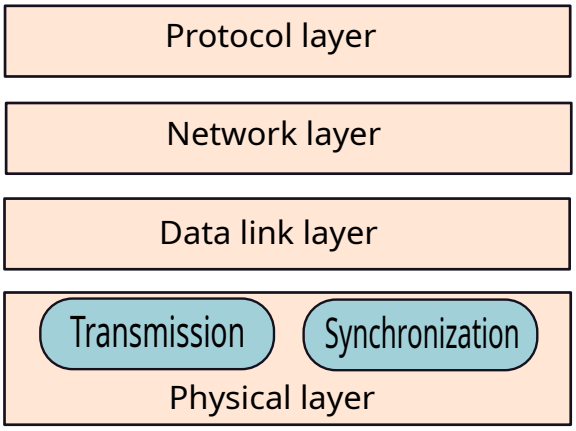


Figure 13.5: The 4 layers of the I/O system

deals with the transmission of bits from the source to the destination. This layer is concerned with the electrical properties of the links (voltage, resistance, capacitance), and the methods of representing a logical bit (0 or 1) using electrical signals.

Let us refer to the second sublayer as the *synchronization sublayer*. This sublayer is concerned with reading an entire *frame* of bits from the physical link. Here, a frame is defined as a group of bits demarcated by special markers. Since I/O channels are plagued with jitter (unpredictable signal propagation time), it is necessary to properly synchronize the arrival of data at the receiver, and read each frame correctly.

In this section, we shall discuss the transmission sublayer. We shall discuss the synchronization sublayer in the next section.

Note that the reason that we create multiple sublayers, instead of creating multiple layers is because sublayers need not be independent of each other. However, in general layers should be independent of each other. It should be theoretically possible to use any physical layer, with any other data link layer protocol. They should ideally be completely oblivious of each other. In this case, the transmission and synchronization sublayers have strong linkages, and thus it is not possible to separate them into separate layers.

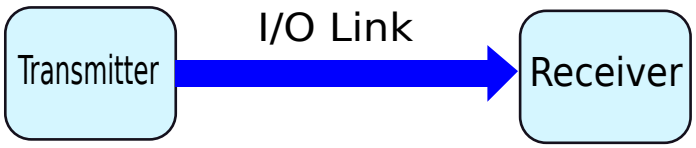


Figure 13.6: A generic view of an I/O link

Figure 13.6 shows the generic view of an I/O link. The source (transmitter) sends a sequence of bits to the destination (receiver). At the time of transmission, the data is always synchronized with respect to the clock of the source. This means that if the source runs at 1 GHz, then it

sends bits at the rate of 1 GHz. Note that the frequency of the source is not necessarily equal to the frequency of the processor, or I/O element that is sending the data. The transmission circuitry, is typically a separate submodule, which has a clock that is derived from the clock of the module that it is a part of. For example, the transmission circuitry of a processor might be transmitting data at 500 MHz, whereas the processor might be running at 4 GHz. In any case, we assume that the transmitter transmits data at its internal clock rate. This clock rate is also known as the *frequency of the bus, or bus frequency* , and this frequency is in general lower than the clock frequency of the processors, or other chips in the chipset. The receiver can run at the same frequency, or can use a faster frequency. Unless explicitly stated, we do not assume that the source and destination have the same frequency. Lastly, note that we shall use the terms *sender*, *source*, and *transmitter* interchangeably. Likewise, we shall use the terms *destination*, and *receiver* interchangeably.

13.2.1 Single Ended Signaling

Let us consider a naive approach, where we send a sequence of 1s and 0s, by sending a sequence of pulses from the source to the destination. This method of signaling is known as *single ended signaling*, and it is the simplest approach.

In specific, we can associate a high voltage pulse with a 1, and a low voltage pulse with a 0. This convention is known as *active high*. Alternatively, we can associate a low voltage pulse with a logical 1, and a high voltage pulse with a logical 0. Conversely, this convention is known as *active low*. Both of these conventions are shown in Figure 13.7.

Sadly, both of these methods are extremely slow and outdated. Recall from our discussion of SRAM cells in Section 7.4.1 that a fast I/O bus needs to reduce the voltage difference between a logical 0, and 1 to as low a value as possible. This is because the voltage difference is detected after it has charged the detector that has an internal capacitance. The higher the voltage required, the longer it takes to charge the capacitors. If the voltage difference is 1 Volt, then it will take a long time to detect a transition from 0 to 1. This will limit the speed of the bus. However, if the voltage difference is 30 mV, then we can detect the transition in voltage much sooner, and we can thus increase the speed of the bus.

Hence, modern bus technologies try to minimize the voltage difference between a logical 0 and 1 to as low a value as possible. Note that we cannot arbitrarily reduce the voltage difference between a logical 0 and 1, in the quest for increasing bus speed. For example, we cannot make the required voltage difference 0.001 mV. This is because there is a certain amount of electrical noise in the system that is introduced due to several factors. Readers might have noticed that if a cell phone starts ringing when the speakers of a car or computer are on, then there is some amount of noise in the speakers also. If we take a cell phone close to a microwave oven while it is running, then there is a decrease in the sound quality of the cell phone. This happens because of electromagnetic interference. Likewise, there can be electromagnetic interference in processors also, and voltage spikes can be introduced. Let us assume that the maximum amplitude of such voltage spikes is 20 mV. Then the voltage difference between a 0 and 1, needs to be more than 20 mV. Otherwise, a voltage spike due to interference can flip the value of a signal leading to an error. Let us take a brief look at one of the most common technologies for on-chip signaling namely LVDS.

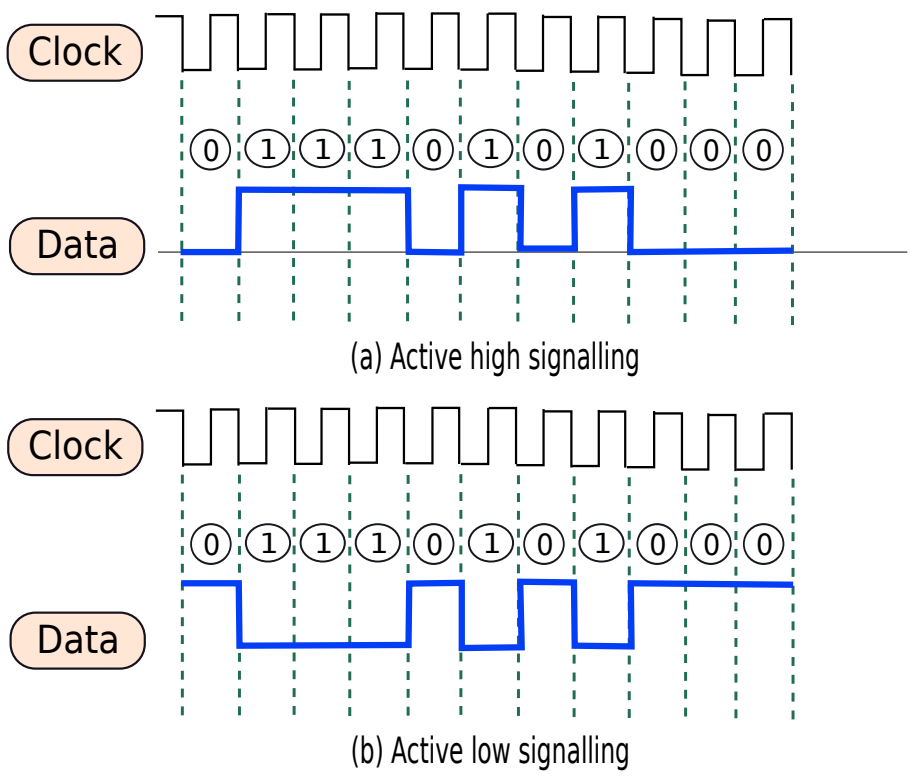


Figure 13.7: Active High and Active Low Signaling Methods

13.2.2 Low Voltage Differential Signaling (LVDS)

LVDS uses two wires to transmit a single signal. The difference between the voltages of these wires is monitored. The value transferred is inferred from the sign of the voltage difference.

The basic LVDS circuit is shown in Figure 13.8. There is a fixed current source of 3.5 mA. Depending on the value of the input A, the current flows to the destination through either line 1 or line 2. For example, if A is 1, then the current flows through line 1 since transistor $T1$ starts conducting, whereas T_2 is off. In this case, the current reaches the destination, passes through the resistor R_d, and then flows back through line 2. Typically, the voltage of both the lines when no current is flowing is maintained at 1.2 V. When current flows through them, there is a voltage swing. The voltage swing is equal to 3.5 mA times R_d. R_d is typically 100 Ohms. Hence, the total differential voltage swing is 350 mV. The role of the detector is to sense the sign of the voltage difference. If it is positive, it can declare a logical 1. Otherwise, it can declare a logical 0. Because of the low swing voltage (350 mV), LVDS is a very fast physical layer protocol.

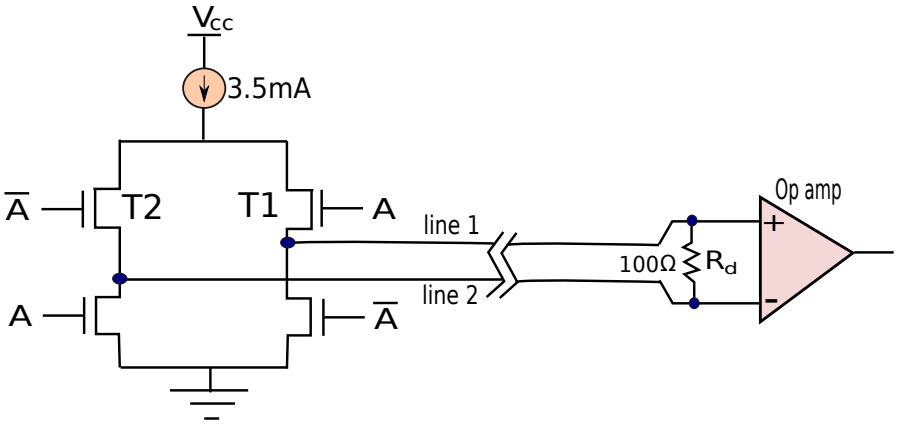


Figure 13.8: LVDS Circuit

13.2.3 Transmission of Multiple Bits

Let us now consider the problem of transmitting multiple bits in sequence. Most I/O channels are not busy all the time. They are busy only when data is being transmitted, and thus their duty cycle (percentage of time that a device is in operation) tends to be highly variable, and most of the time it is not very high. However, detectors are on almost all the time, and they keep sensing the voltage of the bus. This can have implications on both power consumption and correctness. *Power* is an issue because the detectors keep sensing either a logical 1 or 0 every cycle, and it thus becomes necessary for the higher level layers to process the data. To avoid this, most systems typically have an additional line that indicates if the data bits are valid or invalid. This line is traditionally known as the *strobe*. The sender can indicate the period of the validity of data to the receiver by setting the value of the strobe. Again, it becomes necessary to synchronize the data lines and the strobe. This is getting increasingly difficult for high speed I/O buses, because it is possible that signals on the data lines, and the strobe can suffer from different amounts of delay. Hence, there is a possibility that both the lines might move out of synchronization. It is thus a better idea to define three types of signals – zero, one, and *idle*. *Zero* and *one* refer to the transmission of a logical 0 and 1 on the bus. However, the *idle* state refers to the fact that no signal is being transmitted. This mode of signaling is also known as *ternary signaling* because we are using three states.

Definition 146
Ternary *signaling refers to a convention that uses three states for the transmission of signals – one (logical one), zero (logical zero), and* idle *(no signal).*

We can easily implement ternary signaling with LVDS. Let us refer to the wires in LVDS, as

A and B respectively. Let V_A be the voltage of line A. Likewise, let us define the term, V_B. If $\mid V_A - V_B \mid < \tau$, where τ is the detection threshold, then we infer that the lines are idle, and we are not transmitting anything. However, if $V_A - V_B > \tau$, we conclude that we are transmitting a logical 1. Similarly, if $V_B - V_A > \tau$, we conclude that we are transmitting a logical 0. We thus do not need to make any changes to our basic LVDS protocol.

Let us now describe a set of techniques that are optimized for transmitting multiple bits in the physical layer. We present examples that use ternary signaling. Some protocols can also be used with simple binary signaling (zero and one state) also.

13.2.4 Return to Zero (RZ) Protocols

In this protocol we transmit a pulse (positive or negative), and then pause for a while in a bit period. Here, we define the *bit period* as the time it takes to transmit a bit. Most I/O protocols assume that the bit period is independent of the value of the bit (0 or 1) that is being transmitted. Typically, a 1-bit period is equal to the length of one I/O clock cycle. The I/O clock is a dedicated clock that is used by the elements of the I/O system. We shall interchangeably use the terms *clock cycle* and *bit period*, where we do not wish to emphasize a difference between the terms.

Definition 147

bit period *The time it takes to transfer a single bit over a link.*

I/O clock *We assume that there is a dedicated I/O clock in our system that is typically synchronized with the processor clock. The I/O clock is slower than the processor clock, and is used by elements of the I/O subsystem.*

In the RZ protocol, if we wish to transmit a logical 1, then we send a positive voltage pulse on the link for a fraction of a bit period. Subsequently, we stop transmitting the pulse, and ensure that the voltage on the link returns to the *idle* state. Similarly, while transmitting a logical 0, we send a negative voltage pulse along the lines for a fraction of a cycle. Subsequently, we wait till the line returns to the *idle* state. This can be done by allowing the capacitors to discharge, or by applying a reverse voltage to bring the lines to the *idle* state. In any case, the key point here is that while we are transmitting, we transmit the actual value for some part of the bit period, and then we allow the lines to fall back to the default state, which in our discussion we have assumed to be the *idle* state. We shall see that returning to the *idle* state helps the receiver circuitry synchronize with the clock of the sender, and thus read the data correctly. The implicit assumption here is that the sender sends out one bit every cycle (sender cycle). Note that the clock period of the sender and the receiver may be different. We shall take a look at such timing issues in Section 13.4.

Figure 13.9 shows an example of the RZ protocol with ternary signaling. If we were to use binary signaling, then we can have an alternative scheme as follows. We could transmit a short pulse in a cycle for a logical 1, and not transmit anything for a logical 0. Here, the main issue

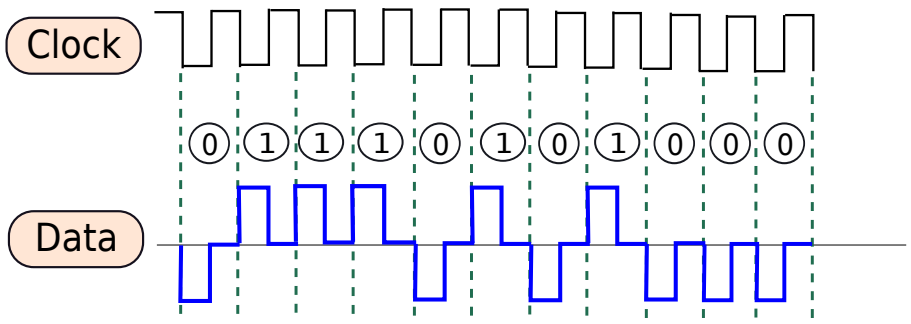


Figure 13.9: Return to zero (RZ) protocol (example)

is to figure out if a logical 0 is being sent or not by taking a look at the length of the pause after transmitting a logical 1. This requires complicated circuitry at the end of the receiver.

Nevertheless, a major criticism of the RZ (return to zero) approaches is that it wastes bandwidth. We need to introduce a short pause (period of idleness) after transmitting a logical 0 or 1. It turns out that we can design protocols that do not have this limitation.

13.2.5 Manchester Encoding

Before proceeding to discuss Manchester encoding, let us differentiate between a *physical bit*, and a *logical bit*. Up till now we have assumed that they mean the same thing. However, this will cease to be true from now onwards. A physical bit such as a physical one or zero, is representative of the voltage across a link. For example, in an active high signaling method, a high voltage indicates that we are transmitting the bit, 1, and a low voltage (physical bit 0) indicates that we are transmitting the 0 bit. However, this ceases to be the case now because we assume that a logical bit (logical 0 or 1) is a function of the values of physical bits. For example, we can infer a logical 0, if the current and the previous physical bit are equal to 10. Likewise, we can have a different rule for inferring a logical 1. It is the job of the receiver to translate physical signals (or rather physical bits), into logical bits, and pass them to the higher layers of the I/O system. The next layer (data link layer discussed in Section 13.4) accepts logical bits from the physical layer. It is oblivious to the nature of the signaling, and the connotations of physical bits transmitted on the link.

Let us now discuss one such mechanism known as Manchester encoding. Here, we encode logical bits as a transition of physical bits. Figure 13.10 shows an example. A $0 \rightarrow 1$ transition of physical bits encodes a logical 1, and conversely a $1 \rightarrow 0$ transition of physical bits encodes a logical 0.

A Manchester code always has a transition to encode data. Most of the time at the middle of a bit period, we have a transition. If there is no transition, we can conclude that no signal is being transmitted and the link is idle. One advantage of Manchester encoding is that it is easy to decode the information that is sent on the link. We just need to detect the nature of the transition. Secondly, we do not need external strobe signals to synchronize the data. The

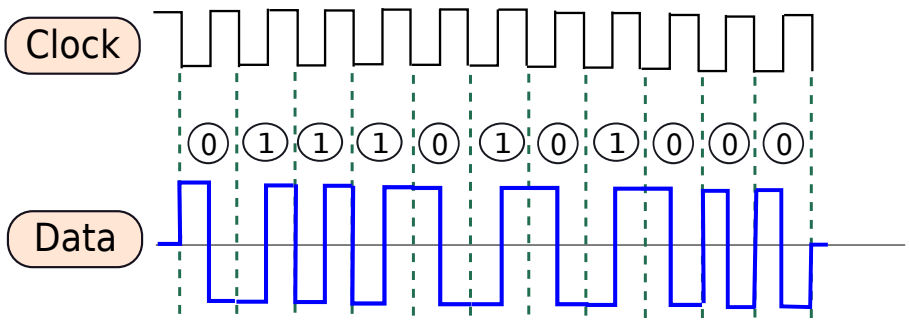


Figure 13.10: Manchester code (example)

data is said to be *self clocked*. This means that we can extract the clock of the sender from the data, and ensure that the receiver reads in data at the same speed at which it is sent by the sender.

Definition 148
A self clocked signal *allows the receiver to extract the clock of the sender by examining the transition of the physical bits in the signal. If there are periodic transitions in the signal, then the period of these transitions is equal to the clock period of the sender, and thus the receiver can read in data at the speed at which it is sent.*

Manchester encoding is used in the IEEE 802.3 communication protocol that forms the basis of today's Ethernet protocol for local area networks. Critics argue that since every logical bit is associated with a transition, we unnecessarily end up dissipating a lot of power. Every single transition requires us to charge/discharge a set of capacitors associated with the link, the drivers, and associated circuitry. The associated resistive loss is dissipated as heat. Let us thus try to reduce the number of transitions.

13.2.6 Non Return to Zero (NRZ) Protocol

Here, we take advantage of a run of 1s and 0s. For a transmitting a logical 1, we set the voltage of the link equal to high. Similarly, for transmitting a logical 0, we set the voltage of the link to low. Let us now consider a run of two 1 bits. For the second bit, we do not induce any transitions in the link, and we maintain the voltage of the link as high. Similarly, if we have a run of n 0s. Then for the last $(n-1)$ 0s we maintain the low voltage of the link, and thus we do not have transitions. Figure 13.11 shows an example. We observe that we have minimized the number of transitions by completely avoiding voltage transitions when the value of the logical bit that needs to be transmitted remains the same. This protocol is fast because we are

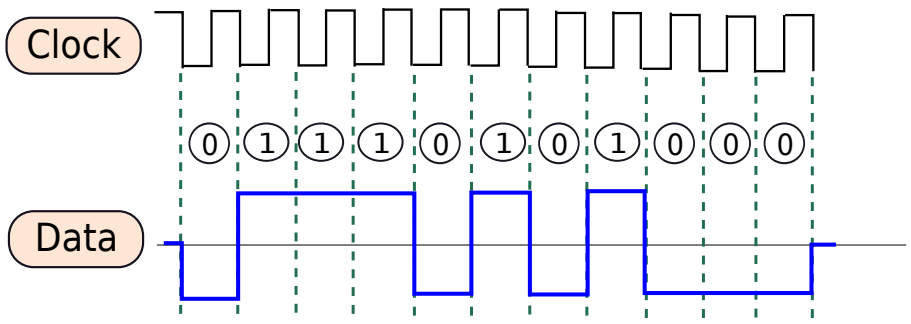


Figure 13.11: Non return to zero protocol (example)

not wasting any time (such as the RZ protocols), and is power efficient because we eliminate transitions for a run of the same bit (unlike RZ and Manchester codes).

However, the added speed and power efficiency comes at the cost of complexity. Let us assume that we want to transmit a string of hundred 1s. In this case, we will have a transition only for the first and last bit. Since the receiver does not have the clock of the sender, it has no way of knowing the length of a bit period. Even if the sender and receiver share the same clock, due to delays induced in the link, the receiver might conclude that we have a run of 99 or 101 bits with a non-zero probability. Hence, we have to send additional synchronization information such that the receiver can properly read all the data that is being sent on the link.

13.2.7 Non Return to Zero (NRZI) Inverted Protocol

This is a variant of the NRZ protocol. Here, we have a transition from 0 to 1, or 1 to 0, when we wish to encode a logical 1. For logical 0s, there are no transitions. Figure 13.12 shows an example.

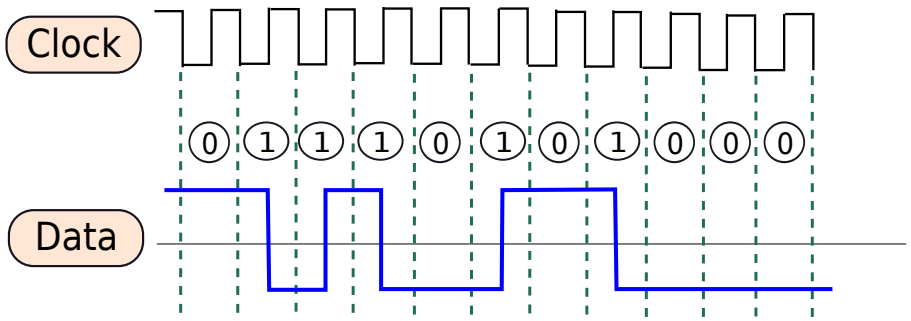


Figure 13.12: Non return to zero inverted protocol (example)

13.3 Physical Layer – Synchronization Sublayer

The transmission sublayer ensures that a sequence of pulses is successfully sent from the transmitter to either one receiver, or to a set of receivers. However, this is not enough. The receiver needs to read the signal at the right time, and needs to assume the correct bit period. If it reads the signal too early or too late, then it risks getting the wrong value of the signal. Secondly, if it assumes the wrong values of the bit period, then the NRZ protocol might not work. Hence, there is a need to maintain a notion of time between the source and destination. The destination needs to know exactly when to transfer the value into a latch. Let us consider solutions for a single source and destination. Extending the methods to a set of destinations is left as an exercise to the reader.

To summarize, the synchronization sublayer receives a sequence of logical bits from the transmission sublayer without any timing guarantees. It needs to figure out the values of the bit periods, and read in an entire *frame* (a fixed size chunk) of data sent by the sender, and send it to the data link layer. Note that the actual job of finding out the frame boundaries, and putting sets of bits in a frame is done by data link layer.

13.3.1 Synchronous Buses

Simple Synchronous Bus

Let us first consider the case of a synchronous system where the sender and the receiver share the same clock, and it takes a fraction of a cycle to transfer the data from the sender to the receiver. Moreover, let us assume that the sender is transmitting all the time. Let us call this system a *simple synchronous bus*.

In this case, the task of synchronizing between the sender and receiver is fairly easy. We know that data is sent at the negative edge of a clock, and in less than a cycle it reaches the receiver. The most important issue that we need to avoid is *metastability* (see Section 7.3.8). A flip flop enters a metastable state when the data makes a transition within a small window of time around the negative edge of the clock. In specific, we want the data to be stable for an interval known as the *setup time* before the clock edge, and the data needs to be stable for another interval known as the *hold time* after the clock edge. The interval consisting of the setup and hold intervals, is known as the keep-out region of the clock as defined in Section 7.3.8 and [Dally and Poulton, 1998].

In this case, we assume that the data reaches the receiver in less than $t_{clk} - t_{setup}$ units of time. Thus, there are no metastability issues, and we can read the data into a flip-flop at the receiver. Since digital circuits typically process data in larger chunks (bytes or words), we use a serial in – parallel out register at the receiver. We serially read in n bits, and read out an n-bit chunk in one go. Since the sender and the receiver clocks are the same, there is no rate mismatch. The circuit for the receiver is shown in Figure 13.13.

Mesochronous Bus

In a *mesochronous system*, the phase difference between the signal and the clock is a constant. The phase difference can be induced in the signal because of the propagation delay in the link, and because there might be a phase difference in the clocks of the sender and the receiver. In

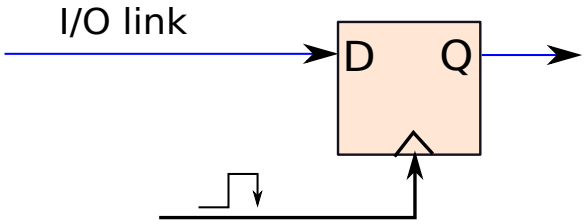


Figure 13.13: The receiver of a simple synchronous bus

this case, it is possible that we might have a metastability issue because the data might arrive in the crucial keep-out region of the receiver clock.

Hence, we need to add a delay element that can delay the signal by a fixed amount of time such that there are no transitions in the keep-out region of the receiver clock. The rest of the circuit remains the same as that used for the simple synchronous bus. The design of the circuit is shown in Figure 13.14.

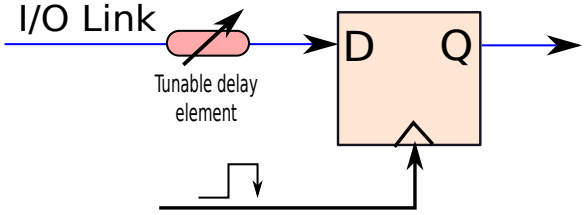


Figure 13.14: The receiver of a mesochronous bus

A delay element can be constructed by using a delay locked loop (DLL). DLLs can have different designs and some of them can be fairly complex. A simple DLL consists of a chain of inverters. Note that we need to have an even number of inverters to ensure that the output is equal to the input. To create a tunable delay element, we can tap the signals after every pair of inverters. These signals are logically equivalent to the input, but have a progressive phase delay due to the propagation delay of the inverters. We can then choose the signal with a specific amount of phase delay by using a multiplexer.

Plesiochronous Bus*

Let us now consider a more realistic scenario. In this case the clocks of the sender and receiver might not exactly be the same. We might have a small amount of clock drift. We can assume that over a period of tens or hundreds of cycles it is minimal. However, we can have a couple cycles of drift over millions of cycles. Secondly, let us assume that the sender does not transmit data all the time. There are idle periods in the bus. Such kind of buses are found in server computers where we have multiple motherboards that theoretically run at the same frequency, but do not share a common clock. There is some amount of clock drift (around 200 ppm [Dally

and Poulton, 1998]) between the processors when we consider timescales of the order of millions of cycles.

Let us now make some simplistic assumptions. Typically, a given frame of data contains 100s or possibly 1000s of bits. We need not worry about clock drift when we are transmitting a few bits (< 100). However, for more bits (> 100), we need to periodically resynchronize the clocks such that we do not miss data. Secondly, ensuring that there are no transitions in the keep-out region of the receiver's clock is a non-trivial problem.

To solve this problem, we use an additional signal known as the *strobe* that is synchronized with the sender's clock. We toggle a strobe pulse at the beginning of the transmission of a frame (or possibly a few cycles before sending the first data bit). We then periodically toggle the strobe pulse once every n cycles. In this case, the receiver uses a tunable delay element. It tunes its delay based on the interval between the time at which it receives the strobe pulse, and the clock transition. After sending the strobe pulse for a few cycles, we start transmitting data. Since the clocks can drift, we need to readjust or retune the delay element. Hence, it is necessary to periodically send strobe pulses to the receiver. We show a timing diagram for the data and the strobe in Figure 13.15.

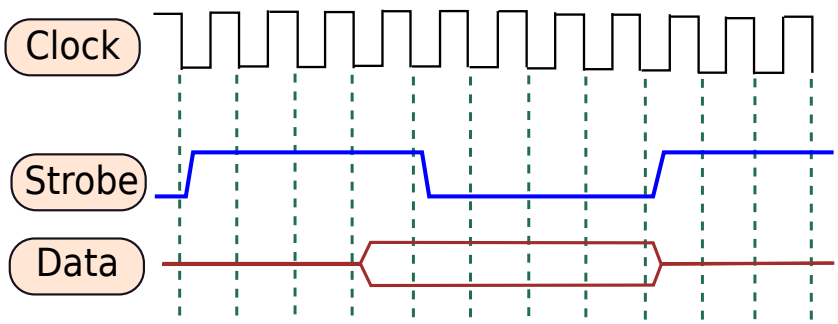


Figure 13.15: The timing diagram of a plesiochronous bus

Similar to the case of the mesochronous bus, every n cycles the receiver can read out all the n bits in parallel using a serial in – parallel out register. The circuit for the receiver is shown in Figure 13.16. We have a delay calculator circuit that takes the strobe and the receiver clock ($rclk$) as input. Based on the phase delay, it tunes the delay element such that data from the source arrives at the middle of the receiver's clock cycle. This needs to be done because of the following reason. Since the sender and receiver clock periods are not exactly the same, there can be an issue of rate mismatch. It is possible that we might get two valid data bits in one receiver clock cycle, or get no bits at all. This will happen, when a bit arrives towards the beginning or end of a clock cycle. Hence, we want to ensure that bits arrive at the middle of a clock cycle. Additionally, there are also metastability avoidance issues.

Sadly, the phase can gradually change and bits might start arriving at the receiver at the beginning of a clock cycle. It can then become possible to receive two bits in the same cycle. In this case, dedicated circuitry needs to predict this event, and a priori send a message to the sender to pause sending bits. Meanwhile, the delay element should be retuned to ensure that

bits arrive at the middle of a cycle.

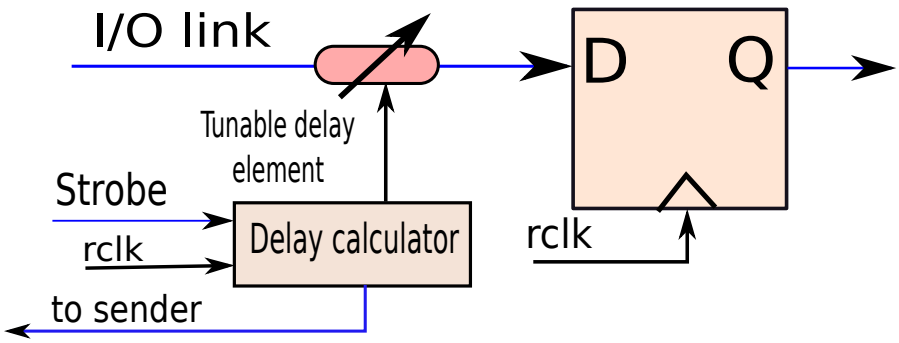


Figure 13.16: The receiver of a plesiochronous bus

13.3.2 Source Synchronous Bus*

Sadly, even plesiochronous buses are hard to manufacture. We often have large and unpredictable delays while transmitting signals, and even ensuring tight clock synchronization is difficult. For example, the AMD Hypertransport [Consortium et al., 2006] protocol that is used to provide a fast I/O path between different processors on the same motherboard does not assume synchronized or plesiosynchronized clocks. Secondly, the protocol assumes an additional jitter (unpredictability in the signal propagation time) of up to 1 cycle.

In such cases, we need to use a more complicated strobe signal. In a source synchronous bus, we typically send the sender clock as the strobe signal. The main insight is that if delays are introduced in the signal propagation time, then the signal and the strobe will be equally affected. This is a very realistic assumption, and thus most high performance I/O buses use source synchronous buses as of 2013. The circuit for a source synchronous bus is again not very complicated. We clock in data to the serial in – parallel out register using the clock of the sender (sent as the strobe). It is referred to as $xclk$. We read the data out using the clock of the receiver as shown in Figure 13.17. As a rule whenever a signal travels across clock boundaries we need a tunable delay element to keep transitions out of the keep-out region. We thus have a delay calculator circuit that computes the parameters of the delay element depending upon the phase difference between the sender clock received as a strobe ($xclk$), and the receiver clock ($rclk$).

Note that it is possible to have multiple parallel data links such that a set of bits can be sent simultaneously. All the data lines can share the strobe that carries the synchronizing clock signal.

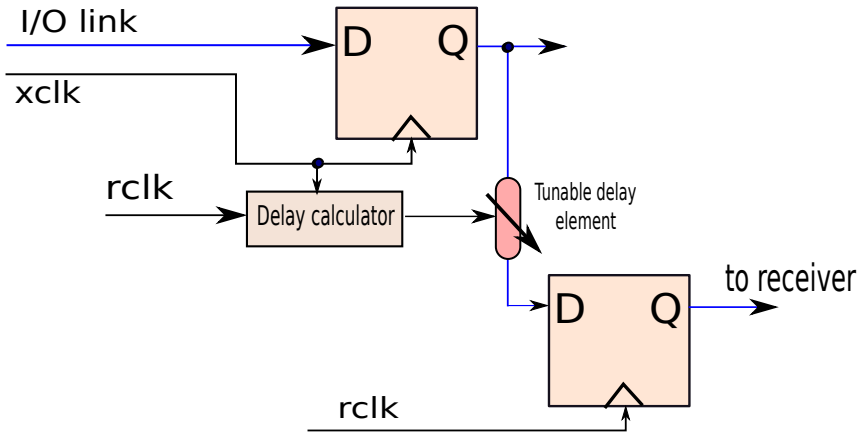


Figure 13.17: The receiver of a source synchronous bus

13.3.3 Asynchronous Buses

Clock Detection and Recovery*

Now, let us consider the most general class of buses known as *asynchronous buses*. Here, we do not make any guarantees regarding the synchronization of the clocks of the sender and the receiver. Nor, do we send the clock of the sender along with the signal. It is the job of the receiver, to extract the clock of the sender from the signal, and read the data in correctly. Let us take a look at the circuit for reading in the data as shown in Figure 13.18.

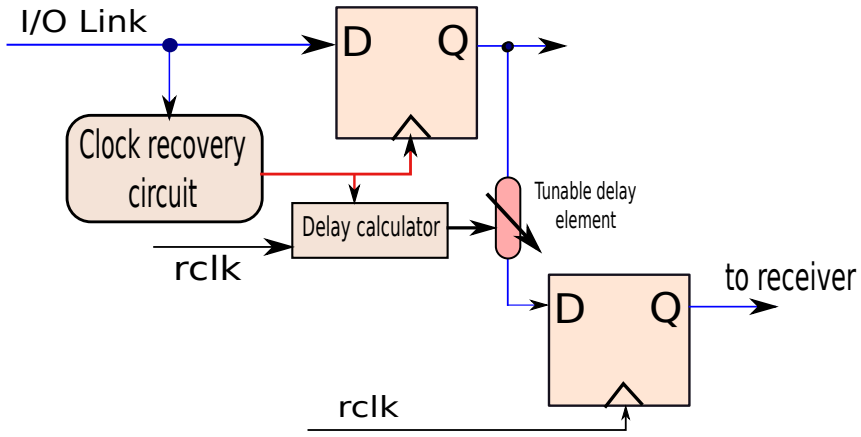


Figure 13.18: The receiver circuit in an asynchronous bus

For the sake of explanation, let us assume that we use the NRZ method of encoding bits. Extending the design to other kinds of encodings is fairly easy, and we leave it as an exercise for the reader. The logical bit stream passed on by the transmission sublayer is sent to the first D

flip-flop, and simultaneously to the clock detector and recovery circuit. These circuits examine the transitions in the I/O signal and try to guess the clock of the sender. Specifically, the clock recovery circuit contains a PLL (phase locked loop). A *PLL* is an oscillator that generates a clock signal, and tries to adjust its phase and frequency such that it is as close as possible to the sequence of transitions in the input signal. Note that this is a rather involved operation.

In the case of the RZ or Manchester encodings, we have periodic transitions. Hence, it is easier to synchronize the PLL circuits at the receiver. However, for the NRZ encoding, we do not have periodic transitions. Hence, it is possible that the PLL circuits at the receiver might fall out of synchrony. A lot of protocols that use the NRZ encoding (notably the USB protocol) insert periodic transitions or dummy bits in the signal to resynchronize the PLLs at the receiver. Secondly, the PLL in the clock recovery circuit also needs to deal with the issue of long periods of inactivity in the bus. During this time, it can fall out of synchronization. There are advanced schemes to ensure that we can correctly recover the clock from an asynchronous signal. These topics are taught in advanced courses in communication, and digital systems. We shall only take a cursory look in this chapter, and assume that the clock recovery circuit does its job correctly.

We connect the output of the clock detection and recovery circuit to the clock input of the first D flip-flop. We thus clock in data according to the sender's clock. To avoid metastability issues we introduce delay elements between the two D flip-flops. The second D flip-flop is in the receiver's clock domain. This part of the circuit is similar to that of source synchronous buses.

Note that in the case of ternary signaling, it is easy to find out when a bus is active (when we see a physical 0 or 1 on the bus). However, in the case of binary signaling, we do not know when the bus is active, because in principle we have a 0 or 1 bit being transmitted all the time. Hence, it is necessary to use an additional strobe signal to indicate the availability of data. Let us now look at protocols that use a strobe signal to indicate the availability of data on the bus. The strobe signals can also be optionally used by ternary buses to indicate the beginning and end of an I/O request. In any case, the reader needs to note that both the methods that we present using strobe signals are rather basic, and have been superseded by more advanced methods.

Asynchronous Communication with Strobe Signals

Let us assume that the source wishes to send data to the destination. It first places data on the bus, and after a small delay sets (sets to 1) the strobe as shown in the timing diagram in Figure 13.19. This is done to ensure that the data is stable on the bus before the receiver perceives the strobe to be set. The receiver immediately starts to read data values. Till the strobe is on, the receiver continues to read data, places it in a register, and transfers chunks of data to higher layers. When the source decides to stop sending data, it resets (sets to 0) the strobe. Note that timing is important here. We typically reset the strobe just before we cease sending data. This needs to be done because we want the receiver to treat the contents of the bus after the strobe is reset as the last bit. In general, we want the data signal to hold its value for some time after we have read it (for metastability constraints).

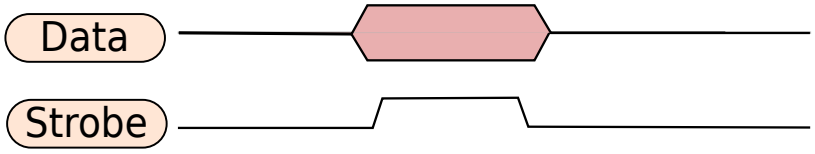


Figure 13.19: Timing diagram of a strobe based asynchronous communication system

Asynchronous Communication with Handshaking (4 Phase)

Note that in simple asynchronous communication with strobe signals the source has no way of knowing if the receiver has read the data. We thus introduce a handshaking protocol where the source is explicitly made aware of the fact that the receiver has read all its data. The associated timing diagram is shown in Figure 13.20.

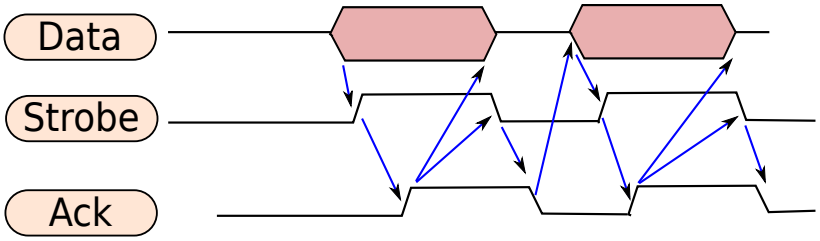


Figure 13.20: Timing diagram of a strobe based asynchronous communication system with handshaking

At the outset, the sender places data on the bus, and then sets the strobe. The receiver begins to read data off the bus, as soon as it observes the strobe to be set. After it has read the data, it sets the *ack* line to 1. After the transmitter observes the *ack* line set to 1, it can be sure of the fact that the receiver has read the data. Hence, the transmitter resets the strobe, and stops sending data. When the receiver observes that the strobe has been reset, it resets the *ack* line. Subsequently, the transmitter is ready to transmit again using the same sequence of steps.

This sequence of steps ensures that the transmitter is aware of the fact that the receiver has read the data. Note that this diagram makes sense when the receiver can ascertain that it has read all the data that the transmitter wished to transmit. Consequently, designers mostly use this protocol for transmitting single bits. In this case, after the receiver has read the bit, it can assert the *ack* line. Secondly, this approach is also more relevant for the RZ and Manchester coding approaches because the transmitter needs to return to the default state before transmitting a new bit. After it receives the acknowledgement, the transmitter can begin the process of returning to the default state, as shown in Figure 13.19.

To transmit multiple bits in parallel, we need to have a strobe for each data line. We can

however, have a common acknowledgement line. We need to set the *ack* signal when all the receivers have read their bits, and we need to reset the *ack* line, when all the strobe lines have been reset. Lastly, let us note that there are four separate events in this protocol (as shown in the diagram). Hence, this protocol is known as a 4-phase handshake protocol.

Asynchronous Communication with Handshaking (2 Phase)

If we are using the NRZ protocols, then we do not need to return to the default state. We can immediately start transmitting the next bit after receiving the acknowledgement. However, in this case, we need to slightly change the semantics of the strobe and acknowledgement signals. Figure 13.21 shows the timing diagram.

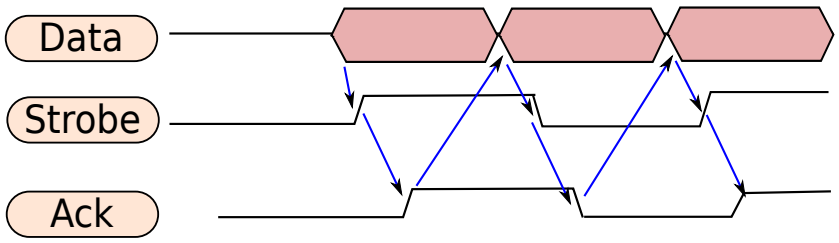


Figure 13.21: Timing diagram of a strobe based asynchronous communication system with 2-phase handshaking

In this case, after placing the data on the bus, the transmitter toggles the value of the strobe. Subsequently, after reading the data, the receiver toggles the value of the *ack* line. After the transmitter detects that the *ack* line has been toggled, it starts transmitting the next bit. After a short duration, it toggles the value of the strobe to indicate the presence of data. Again, after reading the bit, the receiver toggles the *ack* line, and the protocol thus continues. Note that in this case, instead of setting and resetting the *ack* and strobe lines, we toggle them instead. This reduces the number of events that we need to track on the bus. However, this requires us to keep some additional state at the side of the sender and the receiver. This is a negligible overhead. Our 4-phase protocol thus gets significantly simplified. The NRZ protocols are more amenable to this approach because they have continuous data transmission, without any intervening *pause* periods.

Definition 149

Simple Synchronous Bus *A simple synchronous bus that assumes that the transmitter and the receiver share the same clock, and there is no skew (deviation) between the clocks.*

Mesochronous Bus *Here, the transmitter and receiver have the same clock frequency, but there can be a phase delay between the clocks.*

Plesiochronous Bus *In a plesiochronous bus, there is a small amount of mismatch between the frequencies of the clocks of the transmitter and receiver.*

Source Synchronous Bus *In a source synchronous bus, there is no relationship between the clocks of the transmitter and receiver. Consequently, we send the clock of the transmitter to the receiver along with the message, such that it can use it to sample the bits in the message.*

Asynchronous Bus *An asynchronous bus does not assume any relationship between the clocks of the transmitter and receiver. It typically has sophisticated circuitry to recover the clock of the transmitter by analyzing the voltage transitions in the message.*

13.4 Data Link Layer

Now, we are ready to discuss the data link layer. The data link layer gets sequences of logical bits from the physical layer. If the width of the serial in – parallel out register is n bits, then we are guaranteed to get n bits at one go. The job of the data link layer is to break the data into frames, and buffer frames for transmission on other outgoing links. Secondly, it performs rudimentary error checking and correction. It is possible that due to electromagnetic interference, errors might be induced in the signal. For example, a logical 1 might flip to a logical 0, and vice versa. It is possible to correct such single bit errors in the data link layer. If there are a lot of errors, and it is not possible to correct the errors, then at this stage, the receiver can send a message to the transmitter requesting for a retransmission. After error checking the frame is ready to be forwarded on another link if required.

It is possible that multiple senders might be trying to access a bus at the same time. In this case, we need to arbitrate between the requests, and ensure that only one sender can send data at any single point of time. This process is known as *arbitration*, and is also typically performed in the data link layer. Lastly, the arbitration logic needs to have special support for handling requests that are part of a transaction. For example, the bus to the memory units might contain a load request as a part of a memory transaction. In response, the memory unit sends a response message containing the contents of the memory locations. We need a little bit of additional support at the level of the bus controller to support such message patterns.

To summarize the data link layer breaks data received from the physical layer into frames, performs error checking, manages the bus by allowing a single transmitter at a single time, and optimizes communication for common message patterns.

13.4.1 Framing and Buffering

The processing in the data link layer begins by reading sets of bits from the physical layer. We can either have one serial link, or multiple serial links that transmit bits simultaneously. A set of multiple serial links is known as a *parallel link*. In both cases, we read in data, save them in serial in – parallel out shift registers, and send chunks of bits to the data link layer. The role of the data link layer is to create frames of bits from the values that it gets from the

physical layer. A *frame* might be one byte for links that transfer data from the keyboard and mouse, and might be as high as 128 bytes for links that transfer data between the processor and the main memory, or the main memory and the graphics card. In any case, the data link layer for each bus controller is aware of the frame size. The main problem is to demarcate the boundaries of a frame.

Demarcation by Inserting Long Pauses Between two consecutive frames, the bus controller can insert long pauses. By examining, the duration of these pauses, the receiver can infer frame boundaries. However, because of jitter in the I/O channel, the duration of these pauses can change, and new pauses can be introduced. This is not a very reliable method, and it also wastes valuable bandwidth.

Bit Count We can fix the number of bits in a frame a priori. We can simply count the number of bits that are sent, and declare a frame to be over once the required number of bits have reached the receiver. However, the main issue is that sometimes pulses can get deleted because of signal distortion, and it is very easy to go out of synchronization.

Bit/Byte Stuffing This is the most flexible approach and is used in most commercial implementations of I/O buses. Here, we use a pre-specified sequence of bits to designate the start and end of a frame. For example, we can use the pattern 0xDEADBEEF to indicate the start of a frame, and 0x12345678 to indicate the end of a frame. The probability that any 32-bit sequence in the frame will match the special sequences at the start and end is very small. The probability is equal to 2^{-32}, or $2.5e-10$. Sadly, the probability is still non zero. Hence, we can adopt a simple solution to solve this problem. If the sequence, 0xDEADBEEF appears in the content of the frame, then we add 32 more dummy bits and repeat this pattern. For example, the bit pattern 0xDEADBEEF gets replaced with 0xDEADBEEFDEADBEEF. The link layer of the receiver can find out that the pattern repeats an even number of times. Half of the bits in the pattern are a part of the frame, and the rest are dummy bits. The receiver can then proceed to remove the dummy bits. This method is flexible because it can be made very resilient to jitter and reliability problems. These sequences are also known as *commas*.

Once, the data link layer creates a frame, it sends it to the error checking module, and also buffers it.

13.4.2 Error Detection and Correction

Errors can get introduced in signal transmission for a variety of reasons. We can have external electromagnetic interference due to other electronic gadgets operating nearby. Readers would have noticed a loss in the voice quality of a mobile phone after they switch on an electronic gadget such as a microwave oven. This happens because electromagnetic waves get coupled to the copper wires of the I/O channel and introduce current pulses. We can also have additional interference from nearby wires (known as *crosstalk*), and changes in the transmission delay of a wire due to temperature. Cumulatively, interference can induce jitter (introduce variabilities in the propagation time of the signal), and introduce distortion (change the shape of the pulses). We can thus wrongly interpret a 0 as a 1, and vice versa. It is thus necessary to add redundant information, such that the correct value can be recovered.

The reader needs to note that the probability of an error is very low in practice. It is typically less than 1 in every million transfers for interconnects on motherboards. However, this is not a very small number either. If we have a million I/O operations per second, which is plausible, then we will typically have 1 error per second. This is actually a very high error rate. Hence, we need to add extra information to bits such that we can detect and recover from errors. This approach is known as forward error correction. In comparison, in backward error correction, we detect an error, discard the message, and request the sender to retransmit. Let us now discuss the prevalent error detection and recovery schemes.

Definition 150

Forward Error Correction *In this method, we add additional bits to a frame. These additional bits contain enough information to detect and recover from single or double bit errors if required.*

Backward Error Correction *In this method also, we add additional bits to a frame, and these bits help us detect single or double bit errors. However, they do not allow us to correct errors. We can discard the message, and ask the transmitter for a retransmission.*

Single Error Detection

Since single bit errors are fairly improbable, it is extremely unlikely that we shall have two errors in the same frame. Let us thus focus on detecting a single error, and also assume that only one bit flips its state due to an error.

Let us simplify our problem. Let us assume that a frame contains 8 bits, and we wish to detect if there is a single bit error. Let us number the bits in the frame as $D_1, D_2, \ldots, D_8$ respectively. Let us now add an additional bit known as the *parity bit*. The parity bit, P is equal to:

$$P = D_1 \oplus D_2 \oplus \ldots \oplus D_8 \tag{13.1}$$

Here, the $\oplus$ operation is the XOR operator. In simple terms, the parity bit represents the XOR of all the data bits ($D_1 \ldots D_8$). For every 8 bits, we send an additional bit, which is the parity bit. Thus, we convert an 8-bit message to an equivalent 9 bit message. In this case, we are effectively adding a 12.5% overhead in terms of available bandwidth, at the price of higher reliability. Figure 13.22 shows the structure of a frame or message using our 8-bit parity scheme. Note that we can support larger frame sizes also by associating a separate parity bit with each sequence of 8 data bits.

When the receiver receives the message, it computes the parity by computing the XOR of the 8 data bits. If this value matches the parity bit, then we can conclude that there is no error. However, if the parity bit in the message does not match the value of the computed parity bit, then we can conclude that there is a single bit error. The error can be in any of the data bits

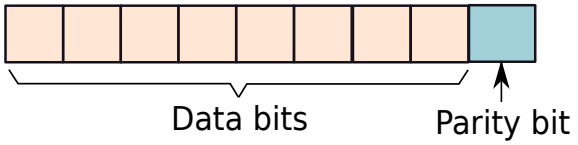


Figure 13.22: An 8-bit message with a parity bit

in the message, or can even be in the parity bit. In this case, we have no way of knowing. All that we can detect is that there is a single bit error. Let us now try to correct the error also.

Single Error Correction

To correct a single bit error, we need to know the index of the bit that has been flipped if there is an error. Let us now count the set of possible outcomes. For an n-bit block, we need to know the index of the bit that has an error. We can have n possible indices in this case. We also need to account for the case, in which we do not have an error. Thus, for a single error correction (SEC) circuit there are a total of $n + 1$ possible outcomes (n outcomes with errors, and one outcome with no error). Thus, from a theoretical point of view, we need $\lceil log(n + 1) \rceil$ additional bits. For example, for an 8-bit frame, we need $\lceil log(8 + 1) \rceil = 4$ bits. Let us design a (8,4) code that has four additional bits for every 8-bit data word.

Let us start out by extending the parity scheme. Let us assume that each of the four additional bits are parity bits. However, they are not the parity functions of the entire set of data bits. Instead, each bit is the parity of a subset of data bits. Let us name the four parity bits P_1, P_2, P_3, and P_4. Moreover, let us arrange the 8 data bits, and the 4 parity bits as shown in Figure 13.23.

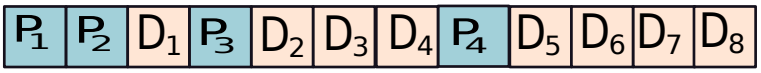


Figure 13.23: Arrangement of data and parity bits

We keep the parity bits, P_1, P_2, P_3, and P_4 in positions 1, 2, 4 and 8 respectively. We arrange the data bits, $D_1 \dots D_8$, in positions 3, 5, 6, 7, 9, 10, 11, and 12 respectively. The next step is to assign a set of data bits to each parity bit. Let us represent the position of each data bit in binary. In this case, we need 4 binary bits because the largest number that we need to represent is 12. Now, let us associate the first parity bit, P_1, with all the data bits whose positions (represented in binary) have 1 as their LSB. In this case, the data bits with 1 as their LSB are D_1 (3), D_2 (5), D_4 (7), D_5 (9), and D_7 (11). We thus compute the parity bit P_1 as:

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7 \tag{13.2}$$

Similarly, we associate the second parity bit, P_2, with all the data bits that have a 1 in their 2^{nd} position (assumption is that the LSB is in the first position). We use similar definitions for

the 3^{rd}, and 4^{th} parity bits.

Parity Bits	Data Bits							
	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
	0011	0101	0110	0111	1001	1010	1011	1100
P_1	X	X		X	X		X	
P_2	X		X	X		X	X	
P_3		X	X	X				X
P_4					X	X	X	X

Table 13.2: Relationship between data and parity bits

Table 13.2 shows the association between data and parity bits. An "X" indicates that a given parity bit is a function of the data bit. Based, on this table, we arrive at the following equations for computing the parity bits.

$$P_1 = D_1 \oplus D_2 \oplus D_4 \oplus D_5 \oplus D_7 \tag{13.3}$$

$$P_2 = D_1 \oplus D_3 \oplus D_4 \oplus D_6 \oplus D_7 \tag{13.4}$$

$$P_3 = D_2 \oplus D_3 \oplus D_4 \oplus D_8 \tag{13.5}$$

$$P_4 = D_5 \oplus D_6 \oplus D_7 \oplus D_8 \tag{13.6}$$

The algorithm for message transmission is as follows. We compute the parity bits according to Equations 13.3 – 13.6. Then, we insert the parity bits in the positions 1, 2, 4, and 8 respectively, and form a message according to Figure 13.23 by adding the data bits. Once the data link layer of the receiver gets the message it first extracts the parity bits, and forms a number of the form $P = P_4P_3P_2P_1$, that is composed of the four parity bits. For example, if $P_1 = 0$, $P_2 = 0$, $P_3 = 1$, and $P_4 = 1$, then $P = 1100$. Subsequently, the error detection circuit at the receiver computes a new set of parity bits (P_1', P_2', P_3', P_4') from the received data bits, and forms another number of the form $P' = P_4'P_3'P_2'P_1'$. Ideally P should be equal to P'. However, if there is an error in the data or parity bits, then this will not be the case. Let us compute $P \oplus P'$. This value is also known as the *syndrome*.

Let us now try to correlate the value of the syndrome with the position of the erroneous bit. Let us first assume that there is an error in a parity bit. In this case, the first four entries in Table 13.3 show the position of the erroneous bit in the message, and the value of the syndrome. The value of the syndrome is equal to the position of the erroneous bit in the message. This should come as no surprise to the reader, because we designed our message to explicitly ensure this. The parity bits are at positions 1, 2, 4, and 8 respectively. Consequently, if any parity bit has an error, its corresponding bit in the syndrome gets set to 1, and the rest of the bits remain 0. Consequently, the syndrome matches the position of the erroneous bit.

Let us now consider the case of single bit errors in data bits. Again from Table 13.3, we can conclude that the syndrome matches the position of the data bit. This is because once a data bit has an error, all its associated parity bits get flipped. For example, if D_5 has an error then the parity bits, P_1 and P_4, get flipped. Recall that the reason we associate P_1 and P_4 with

Bit	Position	Syndrome	Bit	Position	Syndrome
P_1	1	0001	D_3	6	0110
P_2	2	0010	D_4	7	0111
P_3	4	0100	D_5	9	1001
P_4	8	1000	D_6	10	1010
D_1	3	0011	D_7	11	1011
D_2	5	0101	D_8	12	1100

Table 13.3: Relationship between the position of an error and the syndrome

D_5 is because D_5 is bit number 9 (1001), and the two 1s in the binary representation of 9 are in positions 1 and 4 respectively. Subsequently, when there is an error in D_5, the syndrome is equal to 1001, which is also the index of the bit in the message. Similarly, there is a unique syndrome for every data and parity bit (refer to Table 13.2).

Thus, we can conclude that if there is an error, then the syndrome points to the index of the erroneous bit (data or parity). Now, if there is no error, then the syndrome is equal to 0. We thus have a method to detect and correct a single error. This method of encoding messages with additional parity bits is known as the SEC (single error correction) code.

Single Error Correction, Double Error Detection (SECDED)

Let us now try to use the SEC code to additionally detect double errors (errors in two bits). Let us show a counterexample, and prove that our method based on syndromes will not work. Let us assume that there are errors in bits D_2, and D_3. The syndrome will be equal to 0111. However, if there is an error in D_4, the syndrome will also be equal to 0111. There is thus no way of knowing whether we have a single bit error (D_4), or a double bit error (D_2 and D_3).

Let us slightly augment our algorithm to detect double errors also. Let us add an additional parity bit, P_5, that computes the parity of all the data bits ($D_1 \ldots D_8$), and the four parity bits ($P_1 \ldots P_4$) used in the SEC code, and then let us add P_5 to the message. Let us save it in the 13^{th} position in our message, and exclude it from the process of calculation of the syndrome. The new algorithm is as follows. We first calculate the syndrome using the same process as used for the SEC (single error correction) code. If the syndrome is 0, then there can be no error (single or double). The proof for the case of a single error can be readily verified by taking a look at Table 13.2. For a double error, let us assume that two parity bits have gotten flipped. In this case, the syndrome will have two 1s. Similarly, if two data bits have been flipped, then the syndrome will have at least one 1 bit, because no two data bits have identical columns in Table 13.2. Now, if a data and a parity bit have been flipped, then also the syndrome will be non-zero, because a data bit is associated with multiple parity bits. The correct parity bits will indicate that there is an error.

Hence, if the syndrome is non-zero, we suspect an error; otherwise, we assume that there are no errors. If there is an error, we take a look at the bit P_5 in the message, and also recompute it at the receiver. Let us designate the recomputed parity bit as P_5'. Now, if $P_5 = P_5'$, then we can conclude that there is a double bit error. Two single bit errors are essentially canceling each other while computing the final parity. Conversely, if $P_5 \neq P_5'$, then it means that we have

a single bit error. We can thus use this check to detect if we have errors in two bits or one bit. If we have a single bit error, then we can also correct it. However, for a double bit error, we can just detect it, and possible ask the source for retransmission. This code is popularly known as the SECDED code .

Hamming Codes

All the codes described up till now are known as *Hamming codes*. This is because they implicitly rely on the *Hamming distance*. The Hamming distance is the number of corresponding bits that are different between two sequences of binary bits. For example, the Hamming distance between 0011 and 1010 is 2 (MSB and LSB are different).

Let us now consider a 4-bit parity code. If a message is 0001, then the parity bit is equal to 1, and the transmitted message with the parity bit in the MSB position is 10001. Let us refer to the transmitted message as the code word. Note that 00001 is not a valid code word, and the receiver will rely on this fact to adjudge if there is an error or not. In fact, there is no other valid code word within a Hamming distance of 1 of a valid code word. The reader needs to prove this fact. Likewise for a SEC code, the minimum Hamming distance between code words is 2, and for a SECDED code it is 3. Let us now consider a different class of codes that are also very popular.

Cyclic Redundancy Check (CRC) Codes

CRC codes are mostly used for detecting errors, even though they can be used to correct single bit errors in most cases. To motivate the use of CRC codes let us take a look at the patterns of errors in practical I/O systems. Typically, in I/O channels, we have interference for a duration of time that is longer than a bit period. For example, if there is some external electro-magnetic interference, then it might last for several cycles, and it is possible that several bits might get flipped. This pattern of errors is known as a *burst error*. For example, a 32-bit CRC code can detect burst errors as long as 32 bits. It typically can detect most 2-bit errors, and all single bit errors.

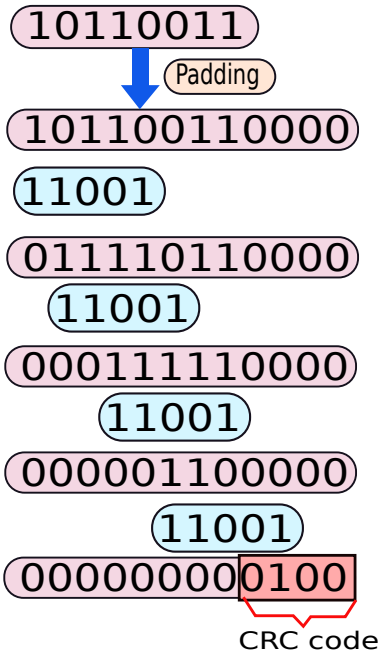
The mathematics behind CRC codes is complicated, and interested readers are referred to texts on coding theory [Neubauer et al., 2007]. Let us show a small example in this section.

Let us assume, that we wish to compute a 4-bit CRC code, for an 8-bit message. Let the message be equal to 10110011_2 in binary. The first step is to pad the message by 4 bits, which is the length of the CRC code. Thus, the new message is equal to 10110011 0000 (a space has been added for improving readability). The CRC code requires another 5 bit number, which is known as the *generator polynomial* or the *divisor*. In principle, we need to divide the number represented by the message with the number represented by the divisor. The remainder is the CRC code. However, this division is different from regular division. It is known as modulo-2 division. In this case, let us assume that the divisor is 11001_2. Note that for an n-bit CRC code, the length of the divisor is $n + 1$ bits.

Let us now show the algorithm. We start out by aligning the MSB of the divisor with the MSB of the message. If the MSB of the message is equal to 1, then we compute a XOR of the first $n + 1$ (5 in this case) bits, and the divisor, and replace the corresponding bits in the message with the result. Otherwise, if the MSB is 0, we do not do anything. In the next step, we shift the divisor one step to the right, treat the bit in the message aligned with the

MSB of the divisor as the MSB of the message, and repeat the same process. We continue this sequence of steps till the LSB of the divisor is aligned with the LSB of the message. We show the sequence of steps in Example 156. At the end, the least significant n (4 bits) contain the CRC code. For sending a message, we append the CRC code with the message. The receiver recomputes the CRC code, and matches it with the code that is appended with the message.

Example 156
Show the steps for computing a 4-bit CRC code, where the message is equal to 10110011_2*, and the divisor is equal to* 11001_2*.*
Answer:



In this figure, we ignore the steps in which the MSB of the relevant part of the message is 0, because in these cases nothing needs to be done.

13.4.3 Arbitration

Let us now consider the problem of *bus arbitration*. The word "arbitration" literally means "resolution of disputes." Let us consider a *multidrop* bus, where we can potentially have multiple transmitters. Now, if multiple transmitters are interested in sending a value over the bus, we need to ensure that only one transmitter can send a value on the bus at any point of time. Thus, we need an arbitration policy to choose a device that can send data over the bus. If we have point-to-point buses, where we have one sender and one receiver, then arbitration is not required. If we have messages of different types waiting to be transmitted, then we need to

schedule the transmission of messages on the link with respect to some optimality criteria.

Definition 151
We need to ensure that only one transmitter sends values on the bus at any point of time. Secondly, we need to ensure that there is fairness, and a transmitter does not need to wait for an indefinite amount of time for getting access to the bus. Furthermore, different devices connected to a bus, typically have different priorities. It is necessary to respect these priorities also. For example, the graphics card, should have more priority than the hard disk. If we delay the messages to the graphics card, the user will perceive jitter on her screen, and this will lead to a bad user experience. We thus need a bus allocation policy that is fair to all the transmitters, and is responsive to the needs of the computer system. This bus allocation policy is popularly known as the arbitration policy*.*

We envision a dedicated structure known as an *arbiter*, which performs the job of bus arbitration. All the devices are connected to the bus, and to the arbiter. They indicate their willingness to transmit data by sending a message to the arbiter. The arbiter chooses one of the devices. There are two topologies for connecting devices to an arbiter. We can either use a star like topology, or we can use a *daisy chain* topology. Let us discuss both the schemes in the subsequent sections.

Star Topology

In this centralized protocol, we have a single central entity called the *arbiter*. It is a dedicated piece of circuitry that accepts *bus-request* requests from all the devices that are desirous of transmitting on the bus. It enforces priorities and fairness policies, and grants the right to individual devices to send data on the bus. Specifically, after a request finishes, the arbiter takes a look at all the current requests, and then asserts the *bus grant* signal for the device that is selected to send data. The selected device subsequently becomes the *bus master* and gets exclusive control of the bus. It can then configure the bus appropriately, and transmit data. An overview the system is shown in Figure 13.24.

We can follow two kinds of approaches to find out when a current request has finished. The first approach is that every device connected to the bus transmits for a given number of cycles, n. In this case, after n cycles have elapsed, the *arbiter* can automatically presume that the bus is free, and it can schedule another request. However, this might not always be the case. We might have different speeds of transmission, and different message sizes. In this case, it is the responsibility of each transmitting device to let the arbiter know that it is done. We envision an additional signal *bus release*. Every device has a dedicated line to the arbiter that is used to send the *bus release* signal. Once it is done with the process of transmitting, it asserts this line (sets it equal to 1). Subsequently, the arbiter allocates the bus to another device. It typically follows standard policies such as round-robin or FIFO.

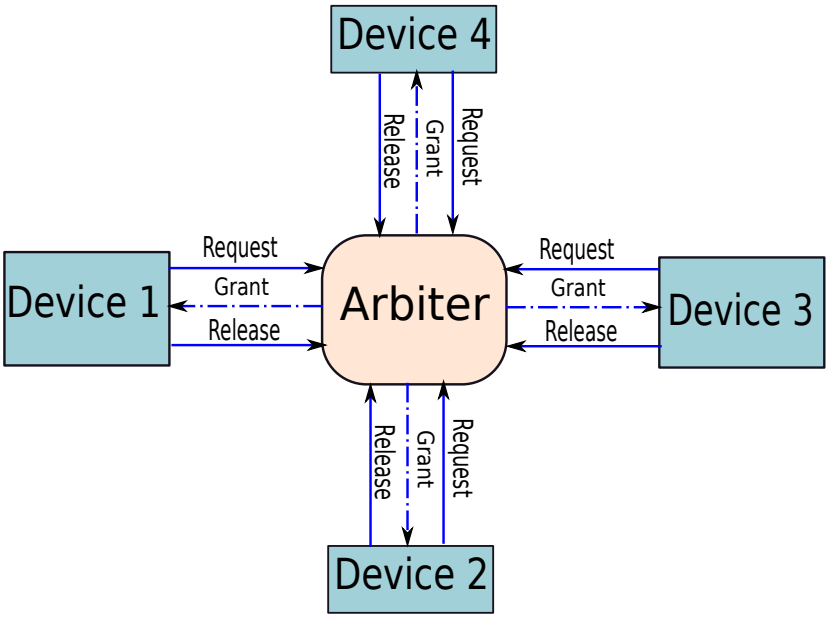


Figure 13.24: Centralized arbiter-based architecture

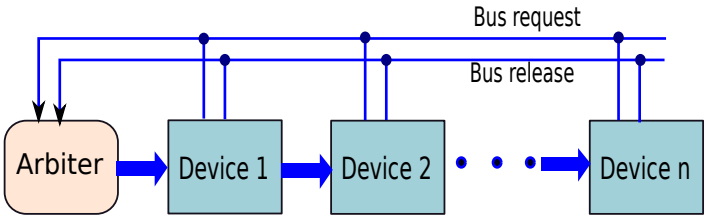


Figure 13.25: Daisy chain architecture

Daisy Chain Based Arbitration

If we have multiple devices connected to a single bus, the arbiter needs to be aware of all of them, and their relative priorities. Moreover, as we increase the number of devices connected to the bus, we start having high contention at the arbiter, and it becomes slow. Hence, we wish to have a scheme, where we can easily enforce priorities, guarantee some degree of fairness, and not incur slowdowns in making bus allocation decisions as we increase the number of connected devices. The *daisy chain* bus was proposed with all of these requirements in mind.

Figure 13.25 shows the topology of a daisy chain based bus. The topology resembles a linear chain, with the arbiter at one end. Each device other than the last one has two connections. The protocol starts as follows. A device starts out by asserting its *bus request* lines. The *bus*

request lines of all the devices are connected in a wired OR fashion. The *request* line that goes to the arbiter essentially computes a logical OR of all the bus request lines. Subsequently, the arbiter passes a token to the device connected to it if it has the token. Otherwise, we need to wait till the arbiter gets the release signal. Once a device gets the token, it becomes the *bus master*. It can transmit data on the bus if required. After transmitting messages, each device passes the token to the next device on the chain. This device also follows the same protocol. It transmits data if it needs to, otherwise, it just passes the token. Finally, the token reaches the end of the chain. The last device on the chain asserts the *bus release* signal, and destroys the token. The *release* signal is a logical OR of all the *bus release* signals. Once, the arbiter observes the *release* signal to be asserted, it creates a token. It re-inserts this token into the daisy chain after it sees the *request* line set to 1.

There are several subtle advantages to this scheme. The first is that we have an implicit notion of priority. The device that is connected to the arbiter has the highest priority. Gradually, as we move away form the arbiter the priority decreases. Secondly, the protocol has a degree of fairness because after a high priority device has relinquished the token, it cannot get it back again, until all the low priority devices have gotten the token. Thus, it is not possible for a device to wait indefinitely. Secondly, it is easy to plug in and remove devices to the bus. We never maintain any individual state of a device. All the communication to the arbiter is aggregated, and we only compute OR functions for the *bus request*, and *bus release* lines. The only state that a device has to maintain is the information regarding its relative position in the daisy chain, and the address of its immediate neighbor.

We can also have purely distributed schemes that avoid a centralized arbiter completely. In such schemes, all the nodes take decisions independently. However, such schemes are rarely used, and thus we shall refrain from discussing them.

13.4.4 Transaction-Oriented Buses

Up till now, we have been only focusing on unidirectional communication, where only one node can transmit to the other nodes at any single point of time. Let us now consider more realistic buses. In reality, most high performance I/O buses are not multidrop buses. Multidrop buses potentially allow multiple transmitters, albeit not at the same point of time. Modern I/O buses are instead point-to-point buses, which typically have two end points. Secondly, an I/O bus typically consists of two physical buses such that we can have bidirectional communication. For example, if we have an I/O bus connecting nodes A and B. Then it is possible for them to send messages to each other simultaneously.

Some early systems had a bus that connected the processor directly to the memory. In this case, the processor was designated as the *master*, because it could only initiate the transfer of a bus message. The memory was referred to as the *slave*, which could only respond to requests. Nowadays, the notion of a master and a slave has become diluted. However, the notion of concurrent bidirectional communication is still common. A bidirectional bus is known as a *duplex* bus or full duplex bus. In comparison, we can have a *half duplex bus*, which only allows one side to transmit at any point of time.

Definition 152

Full Duplex Bus *It is a bus that allows both of the nodes connected at its endpoints to transmit data at the same time.*

Half Duplex Bus *It only allows one of its endpoints to transmit at any point of time.*

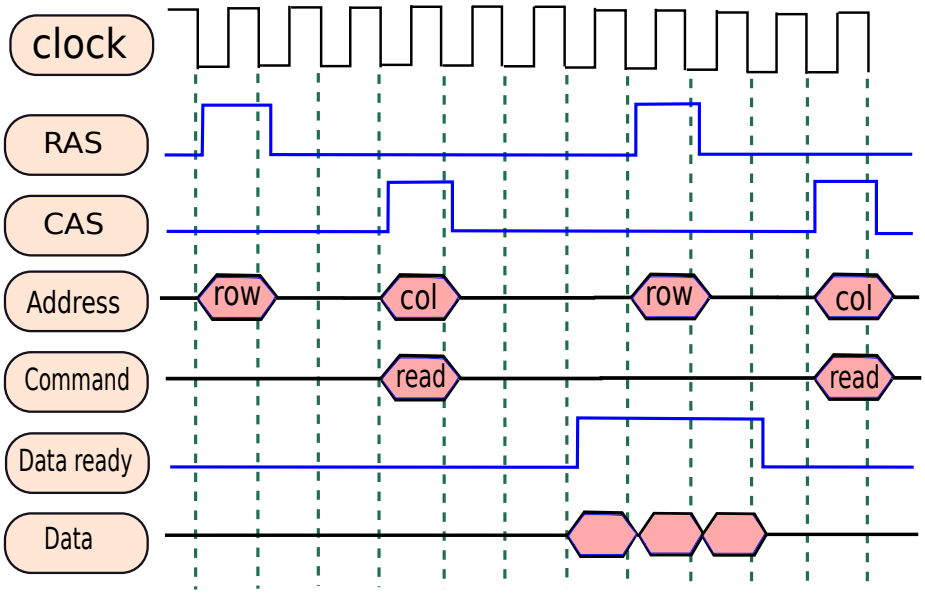


Figure 13.26: DRAM read timing

Let us look at a typical scenario of duplex communication between the memory controller chip, and the DRAM module in Figure 13.26. Figure 13.26 shows the sequence and timing of messages for a memory read operation. In practice, we have two buses. The first bus connects the memory controller to the DRAM module. It consists of address lines (lines to carry the memory address), and lines to carry dedicated control signals. The control signals indicate the timing of operations, and the nature of operation that needs to be performed on the DRAM arrays. The second bus connects the DRAM module to the memory controller. This contains data lines (lines to carry the data read from the DRAM), and timing lines (lines to convey timing information).

The protocol is as follows. The memory controller starts out by asserting the RAS (row address strobe) signal. The RAS signal activates the decoder that sets the values of the word lines. Simultaneously, the memory controller places the address of the row on the address lines. It has an estimate of the time (t_{row}) it takes for the DRAM module to buffer the row address. After t_{row} units of time, it asserts the CAS signal (column address strobe), and

places the address of the columns in the DRAM array on the bus. It also enables the *read* signal indicating to the DRAM module that it needs to perform a read access. Subsequently, the DRAM module reads the contents of the memory locations and transfers it to its output buffers. It then asserts the *ready* signal, and places the data on the bus. However, at this point of time, the memory controller is not idle. It begins to place the row address of the next request on the bus. Note that the timing of a DRAM access is very intricate. Often the processing of consecutive messages is overlapped. For example, we can proceed to decode the row address of the $(n+1)^{th}$ request, when the n^{th} request is transferring its data. This reduces the DRAM latency. However, to support this functionality, we need a duplex bus, and a complex sequence of messages.

Let us note a salient feature of the basic DRAM access protocol that we showed in Figure 13.26. Here, the request and response are very strongly coupled with each other. The source (memory controller) is aware of the intricacies of the destination (DRAM module), and there is a strong interrelationship between the nature and timing of the messages sent by both the source and destination. Secondly, the I/O link between the memory controller and the DRAM module is locked for the duration of the request. We cannot service any intervening request between the original request and response. Such a sequence of messages is referred to as a *bus transaction*.

Definition 153
A bus transaction *is defined as a sequence of messages on a duplex or multidrop bus by more than one node, where there is a strong relationship between the messages in terms of timing and semantics. It is in general not possible to send another unrelated sequence of messages in the middle, and then resume sending the original sequence of messages. The bus is locked for the entire duration.*

There are pros and cons of transaction oriented buses. The first is complexity. They make a lot of assumptions regarding the timing of the receiver. Hence, the message transfer protocol becomes very specific to each type of receiver. This is detrimental to portability. It becomes very difficult to plug in a device that has different message semantics. Moreover, it is possible that the bus might get locked for a long duration, with idle periods. This wastes bandwidth. However, in some scenarios such as the example that we showed, transaction-oriented buses perform very well and are preferred over other types of buses.

13.4.5 Split Transaction Buses

Let us now look at *split transaction* buses that try to rectify the shortcomings of transaction oriented buses. Here, we do not assume a strict sequence of messages between different nodes. For example, for the DRAM and memory controller example, we break the message transfer into two smaller transactions. First, the memory controller sends the memory request to the DRAM. The DRAM module buffers the message, and proceeds with the memory access. Subsequently, it sends a separate message to the memory controller with the data from memory. The interval between both the message sequences can be arbitrarily large. Such a bus is known as a *split*

transaction bus, which breaks a larger transaction into smaller and shorter individual message sequences.

The advantage here is simplicity and portability. All our transfers are essentially unidirectional. We send a message, and then we do not wait for its reply by locking the bus. The sender proceeds with other messages. Whenever, the receiver is ready with the response, it sends a separate message. Along with simplicity, this method also allows us to connect a variety of receivers to the bus. We just need to define a simple message semantics, and any receiver circuit that conforms with the semantics can be connected to the bus. We cannot use this bus to do complicated operations such as overlapping multiple requests, and responses, and fine grained timing control. For such requirements, we can always use a bus that supports transactions.

13.5 Network Layer

In Sections 13.2,13.3, and 13.4, we studied how to design a full duplex bus. In specific, we looked at signaling, signal encoding, timing, framing, error checking, and transaction related issues. Now, we have arrived at a point, where we can assume I/O buses that correctly transfer messages between end points, and ensure timely and correct delivery. Let us now look at the entire chipset, which is essentially a large network of I/O buses.

The problems that we intend to solve in this section, are related to I/O addressing. For example, if the processor wishes to send a message to a USB port, then it needs to have a way of uniquely addressing the USB port. Subsequently, the chipset needs to ensure that it properly routes the message to the appropriate I/O device. Similarly, if a device such as the keyboard, needs to send the ASCII code (see Section 2.5) of the key pressed to the processor, it needs to have a method of addressing the processor. We shall look at routing messages in the chipset in this section.

13.5.1 I/O Port Addressing

Software Interface of an I/O Port

In Definition 140, we defined a hardware I/O port as a connection endpoint for an externally attached device. Let us now consider a *software port*, which we define to be an abstract entity that is visible to software as a single register, or a set of registers. For example, the USB port physically contains a set of metallic pins, and a port controller to run the USB protocol. However, the "software version of the USB port", is an addressable set of registers. If we wish to write to the USB device, then we write to the set of registers exposed by the USB port to software. The USB port controller implements the software abstraction, by physically writing the data sent by the processor to the connected I/O device. Likewise, for reading the value sent by an I/O device through the USB port, the processor issues a read instruction to the software interface of the USB port. The corresponding port controller forwards the output of the I/O device to the processor.

Let us graphically illustrate this concept in Figure 13.27. We have a physical hardware port that has a set of metallic pins, and associated electrical circuitry that implements the physical and data link layers. The port controller implements the network layer by fulfilling requests sent by the processor. It also exposes a set of 8 to 32-bit registers. These registers

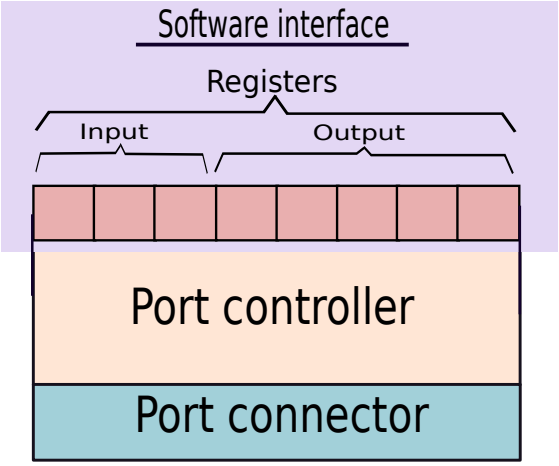


Figure 13.27: Software interface of an I/O port

can be either read-only, write-only, or read-write. For example, the port for the display device such as the monitor contains write-only registers, because we do not get any inputs from it. Similarly, the port controller of a mouse, contains read-only registers, and the port controller of a scanner contains read-write registers. This is because we typically send configuration data and commands to the scanner, and read the image of the document from the scanner.

For example, Intel processors define 64K (2^{16}) 8-bit I/O ports. It is possible to fuse 4 consecutive ports to have a 32-bit port. These ports are equivalent to registers that are accessible to assembly code. Secondly, a given physical port such as the Ethernet port or the USB port can have multiple such software ports assigned to them. For example, if we wish to write a large piece of data to the Ethernet in one go, then we might use hundreds of ports. Each port in the Intel processor is addressed using a 16-bit number that varies from 0 to 0xFFFF. Similarly, other architectures define a set of I/O ports that act as software interfaces for actual hardware ports.

Let us define the term *I/O address space* as the set of all the I/O port addresses that are accessible to the operating system and user programs. Each location in the I/O address space corresponds to an I/O port, which is the software interface to a physical I/O port controller.

Definition 154
The I/O address space *is defined as the set of all the I/O port addresses that are accessible to the operating system and user programs. Each location in the I/O address space corresponds to an I/O port, which is the software interface to a physical I/O port controller.*

ISA Support for I/O Ports

Most instruction set architectures have two instructions: *in* and *out*. The semantics of the instructions are as follows.

Instruction	Semantics
in r1, ⟨I/O port⟩	$r1 \leftarrow$ contents of ⟨I/O port⟩
out r1, ⟨I/O port⟩	contents of ⟨I/O port⟩ $\leftarrow r1$

Table 13.4: Semantics of the *in* and *out* instructions

The *in* instruction transfers data from an I/O port to a register. Conversely, the *out* instruction transfers data from a register to an I/O port. This is a very generic and versatile mechanism for programming I/O devices. For example, if we want to print a page, then we can transfer the contents of the entire page to the I/O ports of the printer. Finally, we write the print command to the I/O port that accepts commands for the printer. Subsequently, the printer can start printing.

Routing Messages to I/O Ports

Let us now implement the *in* and *out* instructions. The first task is to ensure that a message reaches the appropriate port controller, and the second task is to route the response back to the processor in the case of an *out* instruction.

Let us again take a look at the architecture of the motherboard in Figure 13.4. The CPU is connected to the North Bridge chip via the front side bus. The DRAM memory modules, and the graphics card are also connected to the North Bridge chip. Additionally, the North Bridge chip is connected to the South Bridge chip that handles slower devices. The South Bridge chip is connected to the USB ports, the PCI Express Bus (and all the devices connected to it), the hard disk, the mouse, keyboard, speakers and the network card. Each of these devices has a set of associated I/O ports, and I/O port numbers.

Typically, the motherboard designers have a scheme for allocating I/O ports. Let us try to construct one such scheme. Let us suppose that we have 64K 8-bit I/O ports like the Intel processors. The addresses of the I/O ports thus range from 0 to 0xFFFF. Let us first allocate I/O ports to high-bandwidth devices that are connected to the North Bridge chip. Let us give them port addresses in the range of 0 to 0x00FF. Let us partition the rest of the addresses for the devices connected to the South Bridge chip. Let us assume that the hard disk has a range of ports from 0x0100 to 0x0800. Let the USB ports have a range from 0x0801 to 0x0FFF. Let us assign the network card the following range: 0x1000 to 0x4000. Let us assign a few of the remaining ports to the rest of the devices, and keep a part of the range empty for any new devices that we might want to attach later.

Now, when the processor issues an I/O instruction (*in* or *out*), the processor recognizes that it is an I/O instruction, sends the I/O port address, and the instruction type to the North Bridge chip through the FSB (front side bus). The North Bridge chip maintains a table of ranges for each I/O port type, and their locations. Once it sees the message from the processor, it accesses this table and finds out the relative location of the destination. If the destination is a

device that is directly connected to it, then the North Bridge chip forwards the message to the destination. Otherwise, it forwards the request to the South Bridge chip. The South Bridge chip maintains a similar table of I/O port ranges, and device locations. After performing a lookup in this table, it forwards the received message to the appropriate device. These tables are called I/O routing tables. I/O routing tables are conceptually similar to network routing tables used by large networks and the internet.

For the reverse path, the response is typically sent to the processor. We assign a unique identifier to the processor, and the messages gets routed appropriately by the North Bridge and South Bridge chips. Sometimes it is necessary to route the message to the memory modules (see Section 13.6.3). We use a similar addressing scheme.

This scheme essentially maps the set of physical I/O ports to locations in the I/O address space, and the dedicated I/O instructions use the port addresses to communicate with them. This method of accessing and addressing I/O devices is commonly known as port-mapped I/O (PMIO).

Definition 155 *Port-mapped I/O is a scheme for addressing and accessing I/O devices by assigning each physical I/O port a unique address in the I/O space, and by using dedicated I/O instructions to transfer data to/from locations in the I/O address space.*

13.5.2 Memory-Mapped Addressing

Let us now take a look at the *in* and *out* I/O instructions again. The executing program needs to be aware of the naming schemes for I/O ports. It is possible that different chipsets and motherboards use different addresses for the I/O ports. For example, one motherboard might assign the USB ports the I/O port address range, 0xFF80 to 0xFFC0, and another motherboard might assign the range, 0xFEA0 to 0xFFB0. Consequently, a program that runs on the first motherboard might not work on the second motherboard.

To solve this issue, we need to add an additional layer between the I/O ports and software. Let us propose a solution similar to virtual memory. In fact, virtualization is a standard technique for solving various problems in computer architecture. Let us proceed to design a virtual layer between user programs and the I/O address space.

Let us assume that we have a dedicated device driver in the operating system that is specific to the chipset and motherboard. It needs to be aware of the semantics of the I/O ports, and their mappings to actual devices. Now, let us consider a program (user program or OS) that wishes to access the USB ports. At the outset, it is not aware about the I/O port addresses of the USB ports. Hence, it needs to first request the relevant module in the operating system to map a memory region in its virtual address space to the relevant portion of the I/O address space. For example, if the I/O ports for the USB devices are between 0xF000 to 0xFFFF, then this 4 KB region in the I/O address space can be mapped to a page in the program's virtual address space. We need to add a special bit in the TLB and page table entries to indicate that this page actually maps to I/O ports. Secondly, instead of storing the address of the physical frame, we need to store the I/O port addresses. It is the role of the motherboard driver that is a part of the operating system to create this mapping. After the operating system has mapped

the I/O address space to a process's virtual address space, the process can proceed with the I/O access. Note that before creating the mapping, we need to ensure that the program has sufficient privileges to access the I/O device.

After the mapping has been created, the program is free to access the I/O ports. Instead of using I/O instructions such as *in*, and *out*, it uses regular load and store instructions to write to locations in its virtual address space. After such instructions reach the memory access (MA) stage of the pipeline, the effective address is sent to the TLB for translation. If there is a TLB hit, then the pipeline also becomes aware of the fact that the virtual address maps to the I/O address space rather than the physical address space. Secondly, the TLB also translates the virtual address to an I/O port address. Note that at this stage it is not necessary to use the TLB, we can use another dedicated module to translate the address. In any case, the processor receives the equivalent I/O port address in the MA stage. Subsequently, it creates an I/O request and dispatches the request to the I/O port. This part of the processing is exactly similar to the case of port-mapped I/O.

Definition 156 *Memory-mapped I/O is a scheme for addressing and accessing I/O devices by assigning each address in the I/O address space to a unique address in the process's virtual address space. For accessing an I/O port, the process uses regular load and store instructions.*

This scheme is known as memory-mapped I/O. Its main advantage is that it uses regular load and store instructions to access I/O devices instead of dedicated I/O instructions. Secondly, the programmer need not be aware of the actual addresses of the I/O ports in the I/O address space. Since dedicated modules in the operating system, and the memory system, set up a mapping between the I/O address space and the process's virtual address space, the program can be completely oblivious of the semantics of addressing the I/O ports.

13.6 Protocol Layer

Let us now discuss the last layer in the I/O system. The first three layers ensure that a message is correctly delivered from one device to another in the I/O system. Let us now look at the level of a complete I/O request such as printing an entire page, scanning an entire document, or reading a large block of data from the hard disk. Let us consider the example of printing a document.

Assume that the printer is connected to a USB port. The printer device driver starts out by instructing the processor to send the contents of the document to the buffers associated with the USB port. Let us assume that each such buffer is assigned a unique port address, and the entire document fits within the set of buffers. Moreover, let us assume that the device driver is aware that the buffers are empty. To send the contents of the document, the device driver can use a sequence of *out* instructions, or can use memory-mapped I/O. After transferring the contents of the document, the last step is to write the `PRINT` command to a pre-specified I/O port. The USB controller manages all the I/O ports associated with it, and ensures that

messages sent to these ports are sent to the attached printer. The printer starts the job of printing after receiving the `PRINT` command from the USB controller.

Let us now assume that the user clicks the print button for another document. Before sending the new document to the printer, the driver needs to ensure that the printer has finished printing the previous document. The assumption here is that we have a simple printer that can only handle one document at a time. There should thus be a method for the driver to know if the printer is free.

Before looking at different mechanisms for the printer to communicate with its driver, let us consider an analogy. Let us consider a scenario in which Sofia is waiting for a letter to be delivered to her. If the letter is being sent through one of Sofia's friends, then Sofia can keep calling her friend to find out when she will be back in town. Once she is back, Sofia can go to her house, and collect the letter. Alternatively, the sender can send the letter through a courier service. In this case, Sofia simply needs to wait for the courier delivery boy to come and deliver the letter. The former mechanism of receiving messages is known as *polling*, and the latter is known as *interrupts*. Let us now elaborate.

13.6.1 Polling

Let us assume that there is a dedicated register called the *status register* in the printer that maintains the status of the printer. Whenever there is a change in the status of the printer, it updates the value of the status register. Let us assume that the status register can contain two values namely 0 ($free$) and 1 ($busy$). When the printer is printing a document, the value of the status register is 1 ($busy$). Subsequently, when the printer completes printing the document, it sets the value of the status register to 0 ($free$).

Now, let us assume that the printer driver wishes to read the value of the *status* register of the printer. It sends a message to the printer asking it for the value of the *status* register. The first step in sending a message is to send a sequence of bytes to the relevant I/O ports of the USB port controller. The port controller in turn sends the bytes to the printer. If it uses a split transaction bus, then it waits for the response to arrive. Meanwhile, the printer interprets the message, and sends the value of the *status* register as the response, which the USB port controller forwards to the processor through the I/O system.

If the printer is free, then the device driver can proceed to print the next document. Otherwise, it needs to wait for the printer to finish. It can keep on requesting the printer for its status till it is free. This method of repeatedly querying a device for its state till its state has a certain value is called *polling*.

Definition 157
Polling *is a method for waiting till an I/O device reaches a given state. It is implemented by repeatedly querying the device for its state in a loop.*

Let us show a snippet of *SimpleRisc* code that implements polling in a hypothetical system. We assume that the message for getting the status of the printer is 0xDEADBEEF. We need to

first send the message to the I/O port 0xFF00, and then subsequently read the response from the I/O port 0xFF04.

Assembly Code for Polling

```
/* load DEADBEEF in r0 */
movh r0, 0xDEAD
addu r0, r0, 0xBEEF

/* polling loop */
.loop:
        out r0, 0xFF00
        in  r1, 0xFF04
        cmp r1, 1
        beq .loop /* keep looping till status = 1 */
```

13.6.2 Interrupts

There are several shortcomings of the polling based approach. It keeps the processor busy, wastes power, and increases I/O traffic. We can use interrupts instead. Here, the idea is to send a message to the printer to notify the processor when it becomes free. After the printer becomes free, or if it is already free, the printer sends an interrupt to the processor. The I/O system typically treats the interrupt as a regular message. It then delivers the interrupt to the processor, or a dedicated interrupt controller. These entities realize that an interrupt has come from the I/O system. Subsequently, the processor stops executing the current program as described in Section 10.8, and jumps to the interrupt handler.

Note that every interrupt needs to identify itself, or the device that has generated it. Every device that is on the motherboard typically has a unique code. This code is a part of the interrupt. In some cases, when we connect devices to generic ports such as the USB port, the interrupt code contains two parts. One part is the address of the port on the motherboard that is connected to the external device. The other part is an id that is assigned to the device by the I/O port on the motherboard. Such interrupts that contain a unique code are known as *vectored interrupts*.

In some systems such as x86 machines, the first stage of interrupt processing is done by a *programmable interrupt controller* (PIC). These interrupt controllers are called APICs (advanced programmable interrupt controllers) in x86 processors. The role of these interrupt controllers is to buffer interrupt messages, and send them to the processor according to a set of rules.

Let us take a look at the set of rules that PICs follow. Most processors disable interrupt processing during some critical stages of computation. For example, when an interrupt handler is saving the state of the original program, we cannot allow the processor to get interrupted. After the state is successfully saved, interrupt handlers might re-enable interrupts. In some systems, interrupts are completely disabled whenever an interrupt handler is running. A closely related concept is interrupt masking that selectively enables some interrupts, and disables some other interrupts. For example, we might allow high priority interrupts from the temperature controller during the processing of an interrupt handler, and choose to temporarily ignore low priority interrupts from the hard disk. The PIC typically has a vector that has one entry per

interrupt type. It is known as the *interrupt mask vector*. For an interrupt, if the corresponding bit in the interrupt mask vector is 1, then the interrupt is enabled, otherwise it is disabled.

Lastly, PICs need to respect the priority of interrupts if we have multiple interrupts arriving in the same window of time. For example, interrupts from a device with real time constraints such as an attached high speed communication device have a high priority, whereas keyboard and mouse interrupts have lower priority. The PIC orders interrupts using heuristics that take into account their priority and time of arrival, and presents them to the processor in that order. Subsequently, the processor processes the interrupt according to the methods explained in Section 10.8.

Definition 158

Vectored Interrupt *An interrupt that contains the id of the device that generated it, or the I/O port address that is connected to the external device.*

Programmable Interrupt Controller(PIC) *A dedicated module called the programmable interrupt controller (PIC) buffers, filters, and manages the interrupts sent to a processor.*

Interrupt Masking *The user, or operating system can choose to selectively disable a set of interrupts at some critical phases of programs such as while running device drivers and interrupt handlers. This mechanism is known as interrupt masking. The interrupt mask vector in the PIC is typically a bit vector (one bit per each interrupt type). If a bit is set to 1, then the interrupt is enabled, otherwise it is disabled, and the interrupt will either be ignored, or buffered in the PIC and processed later.*

13.6.3 DMA

For accessing I/O devices, we can use both polling and interrupts. In any case, for each I/O instruction we transfer typically 4 bytes at a time. This means that if we need to transfer a 4 KB block to an I/O device, we need to issue 1024 *out* instructions. Similarly, if we wish to read in 4 KB of data, we need to issue 1024 *in* instructions. Each I/O instruction typically takes more than ten cycles, because it reaches an I/O port after several levels of indirection. Secondly, the frequency of I/O buses is typically a third to a quarter of the processor frequency. Thus, I/O for large blocks of data is a fairly slow process, and it can keep the processor busy for a long time. Our objective is to keep sensitive code such as device drivers and interrupt handlers as short as possible.

Hence, let us try to devise a solution that can offload some work of the processor. Let us consider an analogy. Let us assume that a professor is teaching a class of more than 100 students. After an exam, she needs to grade more than 100 scripts. This will keep her busy for at least a week, and the process of grading scripts is a very tiring and time consuming process. Hence, she can offload the work of grading exam scripts to teaching assistants. This will ensure

that the professor has free time, and she can focus on solving state-of-the-art research problems. We can take cues from this example, and design a similar scheme for processors.

Let us envision a dedicated unit called a DMA (direct memory access) engine that can do some work on behalf of the processor. In specific, if the processor wishes to transfer a large amount of data in memory to an I/O device, or vice versa, then instead of issuing a large number of I/O instructions, the DMA engine can take over the responsibility. The procedure for using a DMA engine is as follows. At the outset, the device driver program, determines that there is a necessity to transfer a large amount of data between memory and an I/O device. Subsequently, it sends the details of the memory region (range of bytes), and the details of the I/O device (I/O port addresses) to the DMA engine. It further specifies, whether the data transfer is from memory to I/O or in the reverse direction. Subsequently, the device driver program suspends itself, and the processor is free to run other programs. Meanwhile, the DMA engine or the DMA controller begins the process of transferring data between the main memory and I/O devices. Depending on the direction of the transfer, it reads the data, temporarily buffers it, and sends it to the destination. Once the transfer is over, it sends an interrupt to the processor indicating that the transfer is over. Subsequently, the device driver of the I/O device is ready to resume operation and complete any remaining steps.

The DMA based approach is typically used by modern processors to transfer a large amount of data between main memory, and the hard disk, or the network card. The transfer of data is done in the background, and the processor is mostly oblivious of this process. Secondly, most operating systems have libraries to program the DMA engine to perform data transfers.

There are two subtle points that need to be discussed in the context of DMA engines. The first is that the DMA controller needs to occasionally become the bus master. In most designs, the DMA engine is typically a part of the North Bridge chip. The DMA engine needs to become the bus master of the bus to memory, and the bus to the South Bridge chip, when required. It can either transfer all the data in one go (also known as a *burst*), or it can wait for idle periods in the bus, and use these cycles to schedule its own transfers. The former approach is known as the *burst* mode, and the latter approach is known as the *cycle stealing mode*.

The second subtle point is that there might be correctness issues if we are not careful. For example, it is possible that we have a given location in the cache, and simultaneously, the DMA engine is writing to the location in main memory. In this case, the value in the cache will become stale, and sadly, the processor will have no way of knowing this fact. Hence, it is important to ensure that locations accessed by DMA controllers are not present in the cache. This is typically achieved through a dedicated piece of logic called a DMA *snoop circuit* that dynamically evicts locations present in the cache, if they are written to by the DMA engine.

13.7 Case Studies – I/O Protocols

In this section, we shall describe the operation of several state-of-the-art I/O protocols. We shall provide a brief overview of each of these protocols in this book. For a detailed study, or wherever there is a doubt, the reader should take a look at their formal specifications posted on the web. The formal specifications are typically released by a consortium of companies that support the I/O protocol. Most of the material that we present is sourced from these specifications.

13.7.1 PCI Express®

Overview

Most motherboards require local buses that can be used to attach devices such as dedicated sound cards, network cards, and graphics cards to the North Bridge or South Bridge chips. In response to this requirement, a consortium of companies created the PCI (Peripheral Component Interconnect) bus specification in 1993. In 1996, Intel created the AGP (Accelerated Graphics Port) bus for connecting graphics cards. In the late nineties, many new bus types were being proposed for connecting a variety of hardware devices to the North Bridge and South Bridge chips. Designers quickly realized that having numerous bus protocols hampers standardization efforts, and compels device vendors to support multiple bus protocols. Hence, a consortium of companies started a standardization effort, and created the PCI Express bus standard in 2004. This technology superseded most of the earlier technologies, and till date it is the most popular bus on the motherboard.

The basic idea of the PCI Express bus is that it is a high speed point to point serial (single bit) interconnect. A point to point interconnect has only two end points. To connect multiple devices to the South Bridge chip, we create a tree of PCI Express devices. The internal nodes of the tree are PCI Express switches that can multiplex traffic from multiple devices. Secondly, as compared to older protocols, each PCI Express bus sends bits serially on a single bit line. Typically, high speed buses avoid transmitting multiple bits in parallel using several copper wires, because different links experience different degrees of jitter and signal distortion. It becomes very hard to keep all the signals in the different wires in synchrony with each other. Hence, modern buses are mostly serial.

A single PCI Express bus is actually composed of many individual serial buses known as *lanes*. Each lane has its separate physical layer. A PCI Express packet is *striped* across the lanes. *Striping* means dividing a block of data (packet) into smaller blocks of data and distributing them across the lanes. For example, in a bus with 8 lanes, and an 8-bit packet, we can send each bit of the packet on a separate lane. The reader needs to note that sending multiple bits in parallel across different lanes is not the same as a parallel bus that has multiple wires to send data. This is because a parallel bus has one physical layer circuit for all the copper wires, whereas in this case, each lane has its separate synchronization, and timing. The data link layer does the job of framing by aggregating the subparts of each packet collected from the different lanes.

Definition 159
The process of striping *refers to dividing a block of data into smaller blocks of data and distributing them across a set of entities.*

A *lane* consists of two LVDS based wires for full duplex signaling. One wire is used to send a message from the first end point to the second, and the second wire is to send a signal in the reverse direction. A set of lanes are grouped together to form an I/O link that is assumed to transfer a full packet (or frame) of data. The physical layer then transfers a packet to the data link layer that performs error correction, flow control, and implements transactions. The PCI

Express protocol is a layered protocol, where the functionality of each layer is roughly similar to the I/O layers that we have defined. Instead of considering transactions to be a part of the data link layer, it has a separate transaction layer. We shall however use the terminology that we have defined in this chapter for explaining all the I/O protocols unless mentioned otherwise.

Summary

PCI Express (Peripheral Component Interconnect Express)	
Usage	As a motherboard bus
Specification	[pci,]
Topology	
Connection	Point to point with multiple *lanes*
Lane	A single bit full duplex channel with data striping
Number of Lanes	1 – 32
Physical Layer	
Signaling	LVDS based differential signaling
Encoding	8 bit/ 10 bit
Timing	Source synchronous
Data Link Layer	
Frame Size	1 byte
Error Correction	32-bit CRC
Transactions	Split transaction bus
Bandwidth	250 MB/s per lane
Network Layer	
Routing Nodes	Switches

Table 13.5: The PCI Express I/O Protocol

A summary of the specifications of the PCI Express protocol is shown in Table 13.5. We can have 1-32 lanes. Here, each lane is an asynchronous bus, which uses a sophisticated version of data encoding called the 8bit/10bit encoding. The 8bit/10bit encoding can be conceptually thought of as an extension of the NRZ protocol. It maps a sequence of 8 logical bits to a sequence of 10 physical bits. It ensures that we do not have more than five 1s or 0s consecutively such that we can efficiently recover the clock. Recall that the receiver recovers the sender's clock by analyzing transitions in the data. Secondly, the encoding ensures that we have almost the same number of physical 1s and 0s in the transmitted signal. In the data link layer the PCI Express protocol implements a split transaction bus with a 1-128 byte frame, and 32-bit CRC based error correction.

The PCI Express bus is normally used to connect generic I/O devices. Sometimes some slots are left unused such that users can later connect cards for their specific applications. For example, if a user is interested in working with specialized medical devices, then she can attach an I/O card that can connect with medical devices externally, and to the PCI Express bus internally. Such free PCI Express slots are known as *expansion slots*.

13.7.2 SATA

Overview

Let us now take a look at a bus, which was primarily developed for connecting storage devices such as hard disks, and optical drives. Since the mid-eighties, designers, and storage vendors, began designing such buses. Several such buses developed over time such as the IDE (Integrated Drive Electronics) and PATA (Parallel Advanced Technology Attachment) buses. These buses were predominantly parallel buses, and their constituent communication links suffered from different amounts of jitter and distortion. Thus, these technologies got replaced by a serial standard known as SATA (Serial ATA), that is a point to point link like PCI Express.

The SATA protocol for accessing storage devices is now used in an overwhelming majority of laptop and desktop processors. It has become the de facto standard. The SATA protocol has three layers – physical, data link, and transport. We map the transport layer of the SATA protocol to our protocol layer. Each SATA link contains a pair of single-bit links that use LVDS signaling. Unlike PCI Express, it is not possible for an end point in the SATA protocol to read and write data at the same time. Only one of the actions can be performed at any point of time. It is thus a half duplex bus. It uses 8b/10b encoding, and it is an asynchronous bus. The data link layer does the job of framing. Let us now discuss the network layer. Since SATA is a point to point protocol, a set of SATA devices can be connected in a tree structure. Each internal node of the tree is known as a *multiplier*. It routes requests from the parent to one of its children, of from one of its children to its parent. Finally, the protocol layer acts on the frames and ensures that they are transmitted in the correct sequence, and implements SATA commands. In specific, it implements DMA requests, accesses the storage devices, buffers data, and sends it to the processor in a predefined order.

Summary

Table 13.6 shows the specification of the SATA protocol. We need to note that the SATA protocol has a very rich protocol layer. It defines a wide variety of commands for storage based devices. For example, it has dedicated commands to perform DMA accesses, perform direct hard disk accesses, encode and encrypt data, and control the internals of storage devices. The SATA bus is a split transaction bus, and the data link layer differentiates between commands and their responses. The protocol layer implements the semantics of all the commands.

13.7.3 SCSI and SAS

Overview of SCSI

Let us now discuss another I/O protocol also meant for peripheral devices known as the SCSI protocol (pronounced as "scuzzy"). SCSI was originally meant to be a competitor of PCI. However, over time the SCSI protocol metamorphosed to a protocol for connecting storage devices.

The original SCSI bus was a multidrop parallel bus that could have 8 to 16 connections. The SCSI protocol differentiates between a *host* and a peripheral device. For example, the South Bridge chip is a host, whereas the controller of a CD drive is a peripheral. Any pair of nodes (host or peripheral) can communicate between each other. The original SCSI bus was

SATA (Serial ATA)	
Usage	Used to connect storage devices such as hard disks
Source	[sat,]
Topology	
Connection	Point to point, half duplex
Topology	Tree based, internal nodes known as *multipliers*
Physical Layer	
Signaling	LVDS based differential signaling
Number of parallel links	4
Encoding	8 bit/ 10 bit
Timing	Asynchronous (clock recovery + comma symbols)
Data Link Layer	
Frame Size	variable
Error Correction	CRC
Transactions	Split transaction bus, command driven
Bandwidth	150-600 MB/s
Network Layer	
Routing Nodes	Multipliers
Protocol Layer	
Each SATA node has dedicated support for processing commands, and their responses. Examples of commands can be DMA reads, or I/O transfers	

Table 13.6: The SATA Protocol

synchronous and ran at a relatively low frequency as compared to today's high speed buses. SCSI has still survived till date and state-of-the-art SCSI buses use an 80-160 MHz clock to transmit 16 bits in parallel. They thus have a theoretical maximum bandwidth of 320-640 MB/s. Note that serial buses can go up till 1 GHz, are more versatile, and can support larger bandwidths.

Given the fact that there are issues with multidrop parallel buses, designers started retargeting the SCSI protocol for point to point serial buses. Recall that PCI Express and SATA buses were also created for the same reason. Consequently, designers proposed a host of buses that extended the original SCSI protocols, but were essentially point to point serial buses. Two such important technologies are the SAS (Serially Attached SCSI), and FC (fiber channel) buses. FC buses are mainly used for very high-end systems such as supercomputers. The SAS bus is more commonly used for enterprise and scientific applications.

Let us thus primarily focus on the SAS protocol, because it is the most popular variant of the SCSI protocol in use today. SAS is a serial point to point technology that is also compatible with previous versions of SATA based devices, and its specification is very close to the specification of SATA.

Overview of SAS

SAS was designed to be backward compatible with SATA. Hence, both the protocols are not very different in the physical and data link layers. However, there are still some differences. The biggest difference is that SAS allows full duplex transmission, whereas SATA allows only half duplex transmission. Secondly, SAS can in general support larger frame sizes, and it supports a larger cable length between the end points as compared to SATA (8 meters for SAS, as compared to 1 meter for SATA).

The network layer is different from SATA. Instead of using a multiplier (used in SATA), SAS uses a much more sophisticated structure known as an *expander* for connecting to multiple SAS targets. Traditionally, the bus master of a SAS bus is known as the *initiator*, and the other node is known as the *target*. There are two kinds of expanders – *edge expander*, and *fanout expander*. An *edge expander* can be used to connect up to 255 SAS devices, and a *fanout expander* can be used to connect up to 255 edge expanders. We can add a large number of devices in a tree-based topology using a root node, and a set of expanders. Each device at boot-up time is assigned a unique SCSI id. A device might further be subdivided into several logical partitions. For example, your author at this moment is working on a storage system that is split into two logical partitions. Each partition has a logical unit number (LUN). The routing algorithm is as follows. The initiator sends a command to either a device directly if there is a direct connection or to an expander. The expander has a detailed routing table that maintains the location of the device as a function of its SCSI id. It looks up this routing table and forwards the packet to either the device, or to an edge expander. This edge expander has another routing table, which it uses to forward the command to the appropriate SCSI device. The SCSI device then forwards the command to the corresponding LUN. For sending a message to another SCSI device, or to the processor, a request follows the reverse path.

Lastly, the protocol layer is very flexible for SAS buses. It supports three kinds of protocols. We can either use SATA commands, SCSI commands, or SMP (SAS Management Protocol) commands. SMP commands are specialized commands for configuring and maintaining the network of SAS devices. The SCSI command set is very extensive, and is designed to control a host of devices (mostly storage devices). Note that a device has to be compatible with the SCSI protocol layer before we can send SCSI commands to it. If a device does not understand a certain command, then there is a possibility that something catastrophic might happen. For example, if we wish to read a CD, and the CD driver does not understand the command, then it might eject the CD. Even worse, it is possible that it might never eject the CD because it does not understand the eject command. The same argument holds true for the case of SATA also. We need to have SATA compatible devices such as SATA compatible hard drives and SATA compatible optical drives, if we wish to use SATA commands. SAS buses are by design compatible with both SATA devices and SAS/SCSI devices because of the flexibility of the protocol layer. For the protocol layer, SAS initiators send SCSI commands to SAS/SCSI devices, and SATA commands to SATA devices.

Nearline SAS (NL-SAS) drives are essentially SATA drives, but have a SCSI interface that translates SCSI commands to SATA commands. NL-SAS drives can thus be seamlessly used on SAS buses. Since the SCSI command set is more expressive and more efficient, NL-SAS drives are 10-20% faster than pure SATA drives.

Let us now very briefly describe the SCSI command set in exactly 4 sentences. The initiator

begins by sending a command to the target. Each command has a 1-byte header, and it has a variable length payload. The target then sends a reply with the execution status of the command. The SCSI specifications defines at least 60 different commands for device control, and transferring data. For additional information, the readers can look up the SCSI specification at [scs,].

13.7.4 USB

Overview

Let us now consider the USB protocol, which was primarily designed for connecting external devices to a laptop or desktop computer such as keyboards, mice, speakers, web cameras, and printers. In the mid-nineties vendors realized that there are many kinds of I/O bus protocols and connectors. Consequently, motherboard designers, and device driver writers were finding it hard to support a large range of devices. There was thus a need for standardization. Hence, a consortium of companies (DEC, IBM, Intel, Nortel, NEC, and Microsoft) conceived the USB protocol (Universal Serial Bus).

The main aim of the USB protocol was to define a standard interface for all kinds of devices. The designers started out by classifying devices into three types namely low speed (keyboards, mice), full speed (high definition audio), and high speed (scanners, and video cameras). Three versions of the USB protocol have been proposed till 2012 namely versions 1.0, 2.0, and 3.0. The basic USB protocol is more or less the same. The protocols are backward compatible. This means that a modern computer that has a USB 3.0 port supports USB 1.0 devices. Unlike the SAS or SATA protocols that are designed for a specific set of hardware, and can thus make a lot of assumptions regarding the behavior of the target device, the USB protocol was designed to be very generic. Consequently, designers needed to provide extensive support for the operating system to discover the type of the device, its requirements, and configure it appropriately. Secondly, a lot of USB devices do not have their power source such as keyboards and mice. It is thus necessary to include a power line for running connected devices in the USB cable. The designers of the USB protocol kept all of these requirements in mind.

From the outset, the designers wanted USB to be a fast protocol that could support high speed devices such as high definition video in the future. They thus decided to use a point to point serial bus (similar to PCI Express, SATA, and SAS). Every laptop, desktop, and mid-sized server, has an array of USB ports on the front or back panels. Each USB port, is considered a *host* that can connect with a set of USB devices. Since we are using serial links, we can create a tree of USB devices similar to trees of PCI Express and SAS devices. Most of the time we connect only one device to a USB port. However, this is not the only configuration. We can alternatively connect a USB hub, which acts like an internal node of the tree. A *USB hub* is in principle similar to a SATA multiplier and SAS expander.

A USB hub is most of the time a passive device, and typically has four ports to connect to other devices and hubs *downstream*. The most common configuration for a hub consists of one upstream port (connection to the parent), and four downstream ports. We can in this manner create a tree of USB hubs, and connect multiple devices to a single USB host on the motherboard. The USB protocol supports 127 devices per host, and we can at the most connect 5 hubs serially. Hubs can either be powered by the host, or be self powered. If a hub is self powered it can connect more devices. This is because, the USB protocol has a limit on the

amount of current that it can deliver to any single device. At the moment, it is limited to 500 mA, and power is allocated in blocks of 100 mA. Hence, a hub that is powered by the host can have at the most 4 ports because it can give each device 100 mA, and keep 100 mA for itself. Occasionally, a hub needs to become an active device. Whenever, a USB device is disconnected from a hub, the hub detects this event, and sends a message to the processor.

Layers of the USB Protocol

Physical Layer

Let us now discuss the protocol in some more detail, and start with the physical layer. The standard USB connector has 4 pins. The first pin is a power line that provides a fixed 5V DC voltage. It is typically referred to as V_{cc} of V_{bus}. We shall use V_{cc}. There are two pins namely D^+ and D^- for differential signaling. Their default voltage is set to 3.3V. The fourth pin is the ground pin (GND). The mini and micro USB connectors have an additional pin called ID that helps differentiate between a connection to the host, and to a device.

The USB protocol uses differential signaling. It uses a variant of the NRZI protocol. For encoding logical bits, it assumes that a logical 0 is represented by a transition in physical bits, whereas a logical 1 is represented by no transitions (reverse of the traditional NRZI protocol). A USB bus is an asynchronous bus that recovers the clock. To aid in clock recovery, the synchronization sublayer introduces dummy transitions if there are no transitions in the data. For example, if we have a continuous run of 1s, then there will be no transitions in the transmitted signal. In this case, the USB protocol introduces a 0 bit after every run of six 1s. This strategy ensures that we have some guaranteed transitions in the signal, and the receiver can recover the clock of the transmitter without falling out of synchrony. The USB connectors only have one pair of wires for differential signaling. Hence, full duplex signaling is not possible. Instead, USB links use half duplex signaling.

Data Link Layer

For the data link layer, the USB protocol uses CRC based error checking, and variable frame lengths. It uses bit stuffing (dedicated frame begin and end symbols) to demarcate frame boundaries. Arbitration is a rather complex issue in USB hubs. This is because, we have many kinds of traffic and many kinds of devices. The USB protocol defines four kinds of traffic.

Control Control messages that are used to configure devices.

Interrupt A small amount of data that needs to be sent to a device urgently.

Bulk A large amount of data without any guarantees of latency and bandwidth. E.g., image data in scanners.

Isochronous A fixed rate data transfer with latency and bandwidth guarantees. E.g., audio/video in web cameras.

Along with the different kinds of traffic, we have different categories of USB devices namely low speed devices (192 KB/s), full speed devices (1.5 MB/s), and high speed devices (60 MB/s). The latest USB 3.0 protocol has also introduced super speed devices that require up to 384

MB/s. However, this category is still not very popular (as of 2012); hence, we shall refrain from discussing it.

Now, it is possible to have a high-speed and a low-speed device connected to the same hub. Let us assume that the high speed device is doing a bulk transfer, and the low speed device is sending an interrupt. In this case, we need to prioritize the access to the upstream link of the hub. Arbitration is difficult because we need to conform to the specifications of each class of traffic and each class of devices. We have a dilemma between performing the bulk transfer, and sending the interrupt. We would ideally like to strike a balance between conflicting requirements by having different heuristics for traffic prioritization. A detailed explanation of the arbitration mechanisms can be found in the USB specification [usb,].

Let us now consider the issue of transactions. Let us assume that a high speed hub is connected to the host. The high speed hub is also connected to full and low speed devices downstream. In this case, if the host starts a transaction to a low speed device through the high speed hub, then it will have to wait to get the reply from the device. This is because the link between the high speed hub and the device is slow. There is no reason to lock up the bus between the host, and the hub in this case. We can instead implement a split transaction. The first part of the split transaction sends the command to the low speed device. The second part of the split transaction consists of a message from the low speed device to the host. In the interval between the split transactions, the host can communicate with other devices. A USB bus implements similar split transactions for many other kinds of scenarios (refer to the USB Specification [usb,]).

Network Layer

Let us now consider the network layer. Each USB device including the hubs is assigned a unique ID by the host. Since we can support up to 127 devices per host, we need a 7-bit device id. Secondly, each device has multiple I/O ports. Each such I/O port is known as an end point. We can either have data end points (interrupt, bulk, or isochronous), or control end points. Additionally, we can classify end points as IN or OUT. The IN end point represents an I/O port that can only send data to the processor, and the OUT end point accepts data from the processor. Every USB device can have at the most 16 IN end points, and 16 OUT end points. Any USB request clearly specifies the type of end point that it needs to access (IN or OUT). Given that the type of the end point is fixed by the request, we need only 4 bits to specify the address of the end point.

All USB devices have a default set of IN and OUT end points whose id is equal to 0. These end points are used for activating the device, and establishing communication with it. Subsequently, each device defines its custom set of end points. Simple devices such as a mouse or keyboard that typically send data to the processor define just one IN end point. However, more complicated devices such as web cameras define multiple end points. One end point is for the video feed, one end point is for the audio feed, and there can be multiple end points for exchanging control and status data.

The responsibility of routing messages to the correct USB device lies with the hubs. The hubs maintain routing tables that associate USB devices with local port ids. Once a message reaches the device, it routes it to the correct end point.

Protocol Layer

The USB protocol layer is fairly elaborate. It starts out by defining two kinds of connections between end points known as *pipes*. It defines a *stream pipe* to be a stream of data without any specific message structure. In comparison, *message pipes* are more structured and define a message sequence that the sender and receiver must both follow. A typical message in the *message pipe* consists of three kinds of packets. The communication starts with a *token* packet that contains the device id, id of the end point, nature of communication, and additional information regarding the connection. The hubs on the path route the token packet to the destination, and a connection is thus set up. Then depending upon the direction of the transfer (host to device or device to host), the host or the device sends a sequence of *data* packets. Finally, at the end of the sequence of data packets, the receiver of the packets sends a *handshake* packet to indicate the successful completion of the I/O request.

Summary

Table 13.7 summarizes our discussion on USB up till now. The reader can refer to the specifications of the USB protocol [usb,] for additional information.

13.7.5 FireWire Protocol

Overview

FireWire started out with being a high speed serial bus in Apple computers. However, nowadays, it is being perceived as a competitor to USB. Even though it is not as popular as USB, it is still commonly used. Most laptops have FireWire ports. The FireWire ports are primarily used for connecting video cameras, and high speed optical drives. FireWire is now an IEEE standard (IEEE 1394), and its specifications are thus open and standardized. Let us take a brief look at the FireWire protocol.

Like all the buses that we have studied, FireWire is a high speed serial bus. For the same generation, FireWire is typically faster than USB. For example, FireWire (S800) by default has a bandwidth of 100 MB/s, as compared to 60 MB/s for high speed USB devices. Secondly, the FireWire bus was designed to be a hybrid of a peripheral bus and a computer network. A single FireWire bus can support up to 63 devices. It is possible to construct a tree of devices by interconnecting multiple FireWire buses using FireWire bridges.

The most interesting thing about the FireWire protocol is that it does not presume a connection to a computer. Peripherals can communicate among themselves. For example, a printer can talk to a scanner without going through the computer. It implements a real network in the true sense. Consequently, whenever a FireWire network boots up, all the nodes co-operate and elect a leader. The leader node, or the root node is the root of a tree. Subsequently, the root node sends out messages, and each node is aware of its position in the tree.

The physical layer of the FireWire protocol consists of two LVDS links (one for transmitting data, and one for transmitting the strobe). The channel is thus half duplex. Note that latest versions of the Firewall protocol that have bandwidths greater than 100 MB/s also support full duplex transmission. They however, have a different connector that requires more pins. For encoding logical bits, most FireWire buses use a method of encoding known as *data strobe (DS) encoding*. The DS encoding has two lines. One line contains the data (NRZ encoding), and the other line contains the strobe. The strobe is equal to the data XORed with the clock. Let the

USB (Universal Serial Bus)	
Usage	Connecting peripheral devices such as keyboards, mice, web cameras, and pen drives
Source	[usb,]
Topology	
Connection	Point to point, serial
Width	Single bit, half duplex
Physical Layer	
Signaling	LVDS based differential signaling.
Encoding	NRZI (transition represents a logical 0)
Timing	Asynchronous (a 0 added after six continuous 1s for clock recovery)
Data Link Layer	
Frame Size	46 – 1058 bits
Error Correction	CRC
Transactions	Split transaction bus
Bandwidth	192 KB/s (low speed), 1.5 MB/s (full speed), 60 MB/s (high speed)
Network Layer	
Address	7-bit device id, 4-bit end point id
Routing	Using a tree of hubs
Hub	Has one upstream port, and up to 4 downstream ports
USB network	Can support a maximum of 127 devices
Protocol Layer	
Connections	Message pipe (structured), and stream pipe (unstructured)
Types of traffic	Control, Interrupt, Bulk, Isochronous

Table 13.7: The USB Protocol

data signal be D (sequence of 0s and 1s), the strobe signal be S, and the clock of the sender be C. We have:

$$\begin{aligned} S &= D \oplus C \\ \Rightarrow D \oplus S &= C \end{aligned} \tag{13.7}$$

At the side of the receiver, it can recover the clock of the sender by computing $D \oplus S$ (XOR of the data and the strobe). Thus, we can think of the DS encoding as a variant of source synchronous transmission, where instead of sending the clock of the sender, we send the strobe.

The link layer of the FireWire protocol implements CRC based error checking, and split transactions. FireWire protocols have a unique way of performing arbitration. We divide time

into 125 μs cycles. The root node broadcasts a start packet to all the nodes. Nodes that wish to transmit isochronous data (data at a constant bandwidth) send their requests along with bandwidth requirements to the root node. The root node typically uses FIFO scheduling. It gives one device permission to use the bus and transmit data for a portion of the 125 μs cycle. Once the request is over, it gives permission to the next isochronous request and so on. Note that in a given cycle, we can only allot 80% of the time for isochronous transmission. Once, all the requests are over, or we complete 80% of the cycle, all isochronous transactions stop. The root subsequently considers asynchronous requests (single message transfers).

Devices that wish to send asynchronous data send their requests to the root through internal nodes in the tree (other FireWire devices). If an internal node represented by a FireWire device wishes to send an asynchronous packet in the current cycle, it denies the request to all the requesters that are in its subtree. Once a request reaches the root node, it sends a packet back to the requester to grant it permission to transmit a packet. The receiver is supposed to acknowledge the receipt of the packet. After a packet transmission has finished, the root node schedules the next request. The aspect of denying requests made by downstream nodes is similar to the concept of a daisy chain (see Section 13.4.3).

For the network layer, each FireWire device also defines its internal set of I/O ports. All the devices export a large I/O space to the processor. Each I/O port contains a device address and a port address within the device. The tree of devices first routes a request to the right device, and then the device routes the request to the correct internal port. Typically, the entire FireWire I/O address space is mapped to memory, and most of the time we use memory-mapped I/O for FireWire devices.

Summary

Table 13.8 summarizes our discussion on the FireWire protocol.

13.8 Storage

Out of all the peripheral devices that are typically attached to a processor, storage devices have a special place. This is primarily because they are integral to the functioning of the computer system.

The storage devices maintain *persistent state*. Persistent state refers to all the data that is stored in the computer system even when it is powered off. Notably, the storage systems store the operating system, all the programs, and their associated data. This includes all our documents, songs, images, and videos. From the point of view of a computer architect, the storage system plays an active role in the boot process, saving files and data, and virtual memory. Let us discuss each of these roles one by one.

When a processor starts (process is known as booting), it needs to load the code of the operating system. Typically, the code of the operating system is available at the beginning of the address space of the primary hard disk. The processor then loads the code of the operating system into main memory, and starts executing it. After the boot process, the operating system is available to users, who can use it to run programs, and access data. Programs are saved as regular files in the storage system, and data is also saved in files. Files are essentially blocks

FireWire (IEEE 1394)	
Usage	Connection to video cameras and optical drives
Source	[fir,]
Topology	
Connection	Point to Point, serial, daisy chain based tree
Width	Single bit, half duplex (till FireWire 400, full duplex beyond that)
Physical Layer	
Signaling	LVDS based differential signaling.
Encoding	Data Strobe Encoding (FireWire 800 and above also support 8bit/10 bit encoding)
Timing	Source Synchronous (sends a data strobe rather than a clock)
Data Link Layer	
Frame Size	12.5 KB (FireWire 800 protocol)
Error Correction	CRC
Transactions	Split Transaction Bus
Arbitration	(1) Elect a leader, or root node. (2) Each 125 μs cycle, the root sends a start packet, and each device willing to transmit sends its requirements to the root. (3) The root allots 100 μs for isochronous traffic, and the rest for asynchronous traffic
Bandwidth	100 MB/s (FireWire 800)
Network Layer	
Address Space	The tree of FireWire devices export a large I/O address space.
Routing	Using a tree of bridges

Table 13.8: The FireWire Protocol

of data in the hard disk, or similar storage devices. These blocks of data need to be read into main memory such that they are accessible by the processor.

Lastly storage devices play a very important role in implementing virtual memory. They store the swap space (see Section 11.4.4). Recall that the swap space contains all the frames that cannot be contained in main memory. It effectively helps to extend the physical address space to match the size of the virtual address space. A part of the frames are stored in main memory, and the remaining frames are stored in the swap space. They are brought into (swapped in), when there is a page fault.

Almost all types of computers have attached storage devices. There, however can be some exceptions. Some machines especially, in a lab setting, might access a hard disk over the network. They typically use a network boot protocol to boot from a remote hard disk, and

access all the files including the swap space over the network. Conceptually, they still have an attached storage device. It is just not physically attached to the motherboard. It is nonetheless, accessible over the network.

Now, let us take a look at the main storage technologies. Traditionally, magnetic storage has been the dominant technology. This storage technology records the values of bits in tiny areas of a large ferro-magnetic disk. Depending on the state of magnetization, we can either infer a logical 0 or 1. Instead of magnetic disk technology, we can use optical technology such as CD/DVD/Blu-ray drives. A CD/DVD/Blu-ray disk contains a sequence of pits (aberrations on the surface) that encodes a sequence of binary values. The optical disk drive uses a laser to read the values stored on the disk. Most of the operations of a computer typically access the hard disk, whereas optical disks are mainly used to archive videos and music. However, it is not uncommon to boot from the optical drives.

A fast emerging alternative to magnetic disks, and optical drives is solid state drives. Unlike magnetic, and optical drives that have moving parts, solid state drives are made of semiconductors. The most common technology used in solid state drives is *flash*. A flash memory device uses charge stored in a semiconductor to signify a logical 0 or 1. They are much faster than traditional hard drives. However, they can store far less data, and as of 2012, are 5-6 times more expensive. Hence, high-end servers opt for hybrid solutions. They have a fast SSD drive that acts as a cache for a much larger hard drive.

Let us now take a look at each of these technologies. Note that in this book, our aim is to give the reader an understanding of the basic storage technologies such that she can optimize the computer architecture. For a deeper understanding of storage technologies the reader can take a look at [Brewer and Gill, 2008, Micheloni et al., 2010].

13.8.1 Hard Disks

A hard disk is an integral part of most computer systems starting from laptops to servers. It is a storage device made of ferromagnetic material and mechanical components that can provide a large amount of storage capacity at low cost. Consequently, for the last three decades hard disks have been exclusively used to save persistent state in personal computers, servers, and enterprise class systems.

Surprisingly, the basic physics of data storage is very simple. We save 0s and 1s in a series of magnets. Let us quickly review the basic physics of data storage in hard disks.

Physics of Data Storage in Hard Disks

Let us consider a typical magnet. It has a *north pole*, and a *south pole*. Like poles repel each other, and opposite poles attract each other. Along with mechanical properties, magnets have electrical properties also. For example, when the magnetic field passing through a coil of wire changes due to the relative motion between the magnet and the coil, an EMF (voltage) is induced across the two ends of the wire according to Faraday's law. Hard disks use Faraday's law as the basis of their operation.

The basic element of a hard disk is a small magnet. Magnets used in hard disks are typically made of iron oxides and exhibit permanent magnetism. This means that their magnetic properties hold all the time. They are called permanent magnets or Ferromagnets (because of iron oxides). In comparison, we can have electromagnets that consist of coils of current carrying

wires wrapped around iron bars. Electromagnets lose their magnetism after the current is switched off.

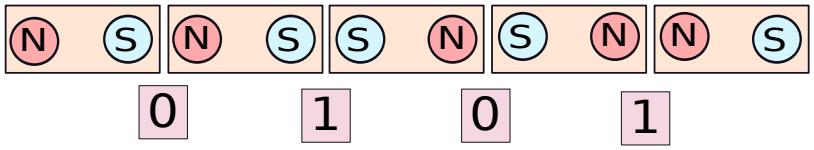


Figure 13.28: A sequence of tiny magnets on the surface of a hard disk

Now, let us consider a set of magnets in series as shown in Figure 13.28. There are two options for their relative orientation namely N-S (north–south), or S-N (south–north). Let us now move a small coil of wire over the arrangement of magnets. Whenever, it crosses the boundary of two magnets that have opposite orientations, there is a change in the magnetic field. Hence, as a direct consequence of Faraday's law, an EMF is induced across the two ends of the coil. However, when there is no change in the orientation of the magnetic field, the EMF induced across the ends of the coil is negligible. The transition in the orientation of the tiny magnets corresponds to a logical 1 bit, and no transition represents a logical 0 bit. Thus, the magnets in Figure 13.28 represent the bit pattern 0101. In principle, this is similar to the NRZI encoding for I/O channels.

Since we encode data in transitions, we need to save blocks of data. Consequently, hard disks save a block of data in a *sector*. A sector has traditionally between 512 bytes for hard disks. It is treated as an atomic block, and an entire sector is typically read or written in one go. The structure that contains the small coil, and passes over the magnets is known as the *read head*.

Let us now look at writing data to the hard disk. In this case, the task is to set the orientation of the magnets. We have another structure called the *write head* that contains a tiny electromagnet. An electromagnet can induce magnetization of a permanent magnet if it passes over it. Secondly, the direction of magnetization is dependent on the direction of the current. If we reverse the direction of the current, the direction of magnetization changes.

For the sake of brevity, we shall refer to the combined assembly of the read head, and write head, as the *head*.

Structure of the Platter

A hard disk typically consists of a set of *platters*. A platter is a circular disk with a hole in the middle. A spindle is attached to the platter through the circular hole in the middle. The platter is divided into a set of concentric rings called tracks. A track is further divided into fixed length sectors as shown in Figure 13.29.

Definition 160
A hard disk consists of multiple platters. *A platter is a circular disk that is attached to a*

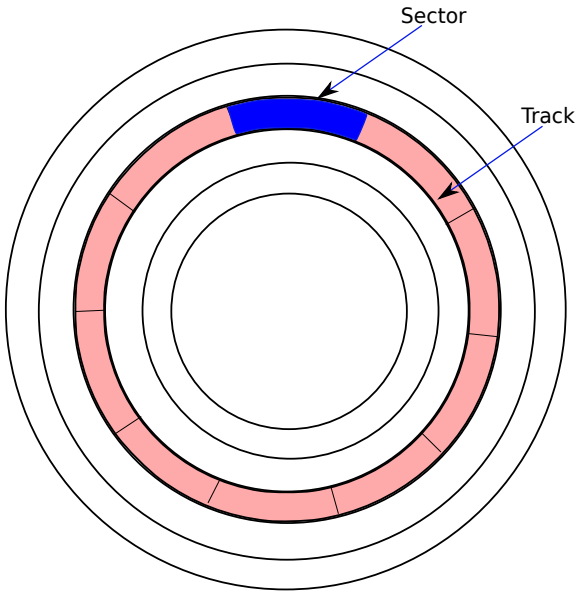


Figure 13.29: The structure of a platter

spindle. A platter further consists of a set of concentric rings called tracks*, and each track consists of a set of sectors. A sector typically contains a fixed number of bytes irrespective of the track.*

Now, let us outline the basic operation of a hard disk. The platters are attached to a spindle. During the operation of a hard disk, the spindle, and its attached platters are constantly in rotation. Let us for the sake of simplicity assume a single platter disk. Now, the first step is to position the head on the track that contains the desired data. Next, the head needs to wait at this position till the desired sector arrives under the head. Since the platter is rotating at a constant speed, we can calculate the amount of time that we need to wait based on the current position of the head. Once the desired sector, arrives under the head, we can proceed to read or write the data.

There is an important question that needs to be considered here. Do we have the same number of sectors per track, or do we have a different number of sectors per track? Note that there are technological limitations on the number of bits that can be saved per track. Hence, if we have the same number of sectors per track, then we are effectively wasting storage capacity in the tracks towards the periphery. This is because we are limited by the number of bits that we can store in the track that is closest to the center. Consequently, modern hard disks avoid this approach.

Let us try to store a variable number of sectors per track. Tracks towards the center contain fewer sectors, and tracks towards the periphery contain more sectors. This scheme also has its share of problems. Let us compare the innermost and outermost tracks, and let us assume that

the innermost track contains N sectors, and the outermost track contains $2N$ sectors. If we assume that the number of rotations per minute is constant, then we need to read data twice as fast on the outermost track as compared to the innermost track. In fact for every track, the rate of data retrieval is different. This will complicate the electronic circuitry in the disk. We can explore another option, which is to rotate the disk at different speeds for each track, such that the rate of data transfer is constant. In this case, the electronic circuitry is simpler, but the sophistication required to run the spindle motor at a variety of different speeds is prohibitive. Hence, both the solutions are impractical.

How about, combining two impractical solutions to make it practical !!! We have been following similar approaches throughout this book. Let us divide the set of tracks into a set of zones. Each *zone* consists of a consecutive set of m tracks. If we have n tracks in the platter, then we have n/m zones. In each zone, the number of sectors per track is the same. The platter rotates with a constant angular velocity for all the tracks in a zone. In a zone, data is more densely packed for tracks that are closer to the center as compared to tracks towards the periphery of the platter. In other words, sectors have physically different sizes for tracks in a zone. This is not a problem since the disk drive assumes that it takes the same amount of time to pass over each sector in a zone, and rotation at a constant angular velocity ensures this.

Figure 13.30 shows a conceptual breakup of the platter into zones. Note that the number of sectors per track varies across zones. This method is known as *Zoned-Bit Recording*(ZBR). The two impractical designs that we refrained from considering, are special cases of ZBR. The first design assumes that we have one zone, and the second design assumes that each track belongs to a different zone.

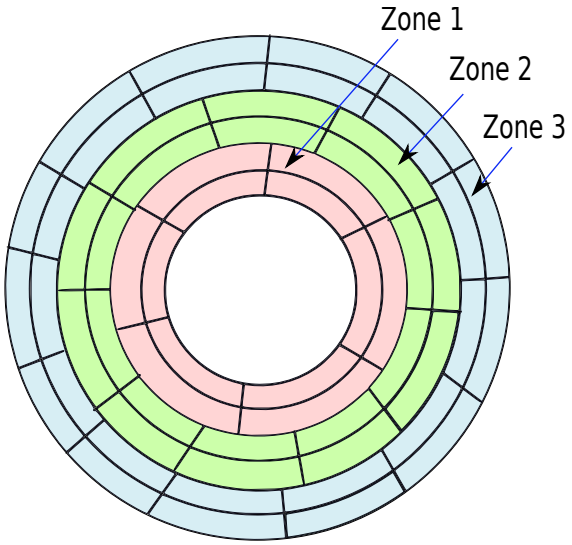


Figure 13.30: Zoned-Bit Recording

Let us now try to see why this scheme works. Since we have multiple zones, the storage space wasted is not as high as the design with just a single zone. Secondly, since the number of

zones is typically not very large, the motor of the spindle does not need to readjust its speed frequently. In fact because of spatial locality, the chances of staying within the same zone are fairly high.

Structure of the Hard Disk

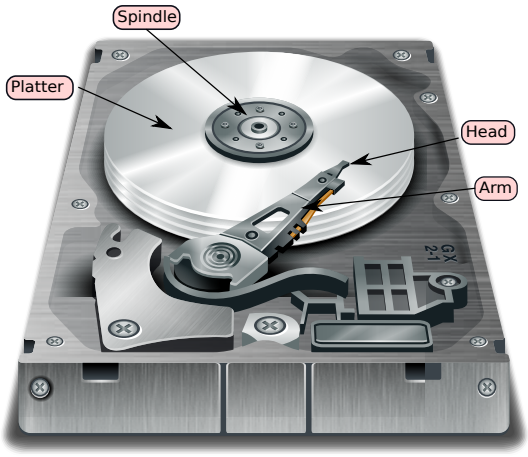


Figure 13.31: The structure of a hard disk (source [har,])

Let us now put all the parts together and take a look at the structure of the hard disk in Figure 13.31 and Figure 13.32. We have a set of platters connected to a single rotating spindle, and a set of disk arms (one for each side of the platter) that contain a head at the end. Typically, all the arms move together, and all the heads are vertically aligned on the same *cylinder*. Here, a cylinder is defined as a set of tracks from multiple platters, which have the same radius. In most hard disks only one head is activated at a point of time. It performs a read or write access on a given sector. In the case of a read access, the data is transmitted back to the drive electronics for post processing (framing, error correction), and then sent on the bus to the processor through the bus interface.

Let us now consider some subtle points in the design of a hard disk (refer to Figure 13.32). It shows two platters connected to a spindle, and each platter has two recording surfaces. The spindle is connected to a motor (known as the spindle motor), which adjusts its speed depending on the zone that we wish to access. The set of all the arms move together, and are connected using a spindle to the *actuator*. The actuator is a small motor used for moving the arms clock wise or anti-clockwise. The role of the actuator is to position the head of an arm on a given track by rotating it clockwise or ant-clockwise a given number of degrees.

A typical disk drive in a desktop processor has a track density of about 10,000 tracks per inch. This means that the distance between tracks is 2.5 μm, and thus the actuator has to be incredibly accurate. Typically, there are some markings on a sector indicating the number of the track. Consequently, the actuator typically needs to make slight adjustments to come to

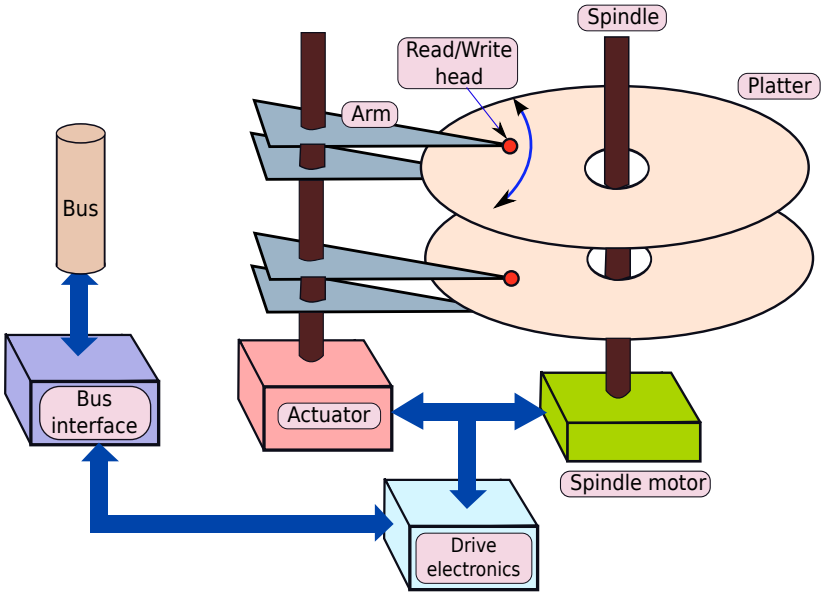


Figure 13.32: Internals of a hard disk

the exact point. This control mechanism is known as *servo control*. Both the actuator and the spindle motor are controlled by electronic circuits that are inside the chassis of the hard disk. Once the actuator has placed the head on the right track, it needs to wait for the desired sector to come under the head. A track has markings to indicate the number of the sector. The head keeps reading the markings after it is positioned on a track. Based on these markings it can accurately predict when the desired sector will be underneath the head.

Along with the mechanical components, a hard disk has electronic components including small processors. They receive and transmit data on the bus, schedule requests on the hard disk, and perform error correction. The reader needs to appreciate the fact that we have just scratched the surface in this book. A hard disk is an incredible feat of human engineering. The hard disk can most of the time seamlessly tolerate errors, dynamically invalidate bad sectors (sectors with faults), and remap data to good sectors. The reader is referred to [Jacob et al., 2007] for further study.

Mathematical Model of a Hard Disk Access

Let us now construct a quick mathematical model for the time a request takes to complete its access to the hard disk. We can divide the time taken into three parts. The first is the *seek time*, which is defined as the time required for the head to reach the right track. Subsequently, the head needs to wait for the desired sector to arrive under it. This time interval is known as the *rotational latency*. Lastly, the head needs to read the data, process it to remove errors and redundant information, and then transmit the data on the bus. This is known as the *transfer time*. Thus, we have the simple equation.

$$T_{disk\_access} = T_{seek} + T_{rot\_latency} + T_{transfer} \tag{13.8}$$

Definition 161

Seek Time *The time required for the actuator to move the head to the right track.*

Rotational Latency *The time required for the desired sector to arrive under the head, after the head is correctly positioned on the right track.*

Transfer Time *The time required to transfer the data from the hard disk to the processor.*

Example 157
Assume that a hard disk has an average seek time of 12 ms, rotates at 600 rpm, and has a bandwidth of 10^6 *B/s. Find the average time to transfer 10,000 bytes of data (assuming that the data is in consecutive sectors).*
Answer: T_{seek} *= 12 ms*
Since the disk rotates at 600 rpm, it takes 100 ms per rotation. On an average, the rotational latency is half this amount because the offset between the current position of the head, and the desired offset is assumed to be uniformly distributed between 0° and 360°. Thus, the rotational latency($T_{rot\_latency}$*) is 50 ms. Now, the time it takes to transfer* 10^4 *contiguous bytes is 0.01s or 10 ms, because the bandwidth is* 10^6 *B/s.*
Hence, the average time per disk access is 12 ms + 50 ms + 10 ms = 72 ms

13.8.2 RAID Arrays

Most enterprise systems have an array of hard disks because their storage, bandwidth, and reliability requirements are very high. Such arrays of hard disks are known as RAID arrays (Redundant Arrays of Inexpensive Disks). Let us review the design space of RAID based solutions in this section.

Definition 162
RAID (Redundant Array of Inexpensive Disks) is a class of technologies for deploying large arrays of disks. There are different RAID levels *in the design space of RAID solutions. Each level makes separate bandwidth, capacity, and reliability guarantees.*

RAID 0

Let us consider the simplest RAID solution known as RAID 0. Here, the aim is to increase bandwidth, and reliability is not a concern. It is typically used in personal computers optimized for high performance gaming.

The basic idea is known as *data striping*. Here, we distribute blocks of data across disks. A *block* is a contiguous sequence of data similar to cache blocks. Its size is typically 512 B; however, its size may vary depending on the RAID system. Let us consider a two disk system with RAID 0 as shown in Figure 13.33. We store all the odd numbered blocks ($B1$, $B3$, ...) in disk 1, and all the even numbered blocks ($B2$, $B4$, ...) in disk 2. If a processor has a page fault, then it can read blocks from both the disks in parallel, and thus in effect, the hard disk bandwidth is doubled. We can extend this idea and implement RAID 0, using N disks. The disk bandwidth can thus be theoretically increased N times.

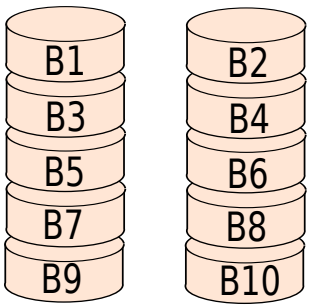


Figure 13.33: RAID 0

RAID 1

Let us now add reliability to RAID 0. Note that the process of reading and writing bits in a hard disk is a mechanical process. It is possible that some bits might not be read or written correctly because there might be a slight amount of deviation from ideal operation in the actuator, or the spindle motor. Secondly, external electro-magnetic radiation, and cosmic particle strikes can flip bits in the hard disk and its associated electronic components. The latter type of errors are also known as *soft errors*. Consequently, each sector of the disk typically has error detecting and correcting codes. Since an entire sector is read or written atomically, error checking and correction is a part of hard disk access. We typically care about more catastrophic failures such as the failure of an entire hard disk drive. This means that the there is break down in the actuator, spindle motor, or any other major component that prevents us from reading or writing to most of the hard disk. Let us consider disk failures from this angle. Secondly, let us also assume that disks follow the *fail stop* model of failure. This means that whenever there is a failure, the disks are not operational anymore, and the system is aware of it.

In RAID 1, we typically have a 2 disk system (see Figure 13.34), and we mirror data of one disk on the other. They are essentially duplicates of each other. We are definitely wasting half of our storage space here. In the case of reads, we can leverage this structure to theoretically

double the disk bandwidth. Let us assume that we wish to read the blocks 1 and 3. In the case of RAID 0, we needed to serialize the accesses, because both the blocks map to the same disk. However, in this case, since each disk has a copy of all the data, we can read block 1 from disk 1, and read block 3 from disk 3 in parallel. Thus, the read bandwidth is potentially double that of a single disk. However, the write bandwidth is still the same as that of a single disk, because we need to write to both the disks. Note that here it is not necessary to read both the disks and compare the contents of a block in the interest of reliability. We assume that if a disk is operational, it contains correct data.

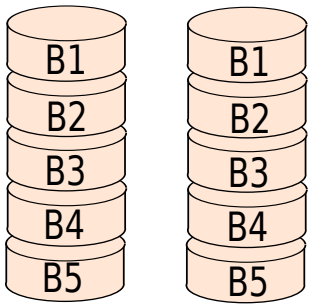


Figure 13.34: RAID 1

RAID 2

Let us now try to increase the efficiency of RAID 1. In this case, we consider a system of N disks. Instead of striping data at the block level, we stripe data at the bit level. We dedicate a disk for saving the parity bit. Let us consider a system with 5 disks as shown in Figure 13.35. We have 4 data disks, and 1 parity disk. We distribute contiguous sequences of 4 bits in a logical block across the 4 data disks. A *logical block* is defined as a block of contiguous data as seen by software. Software should be oblivious of the fact that we are using a RAID system instead of a single disk. All software programs perceive a storage system as an array of logical blocks. Now, the first bit of a logical block is saved in disk 1, the second bit is saved in disk 2, and finally the fourth bit is saved in disk 4. Disk 5, contains the parity of the first 4 bits. Each physical block in a RAID 2 disk thus contains a subset of bits of the logical blocks. For example, $B1$ contains bit numbers $1, 5, 9, \ldots$ of the first logical block saved in the RAID array. Similarly, $B2$ contains bit numbers $2, 6, 10, \ldots$. To read a logical block, the RAID controller assembles the physical blocks and creates a logical block. Similarly, to write a logical block, the RAID controller breaks it down into its constituent physical blocks, computes the parity bits, and writes to all the disks.

Reads are fast in RAID 2. This is because we can read all the 9 disks in parallel. Writes are also fast, because we can write parts of a block to different disks, in parallel. RAID 2 is currently not used because it does not allow parallel access to different logical blocks, introduces complexity because of bit level striping, and every I/O request requires access to all the disks. We would like to iterate again that the parity disk is not accessed on a read. The parity disk is accessed on a write because its contents need to be updated. Its main utility is to keep the

system operational if there is a single disk failure. If a disk fails, then the contents of a block can be recovered by reading other blocks in the same row from the other disks, and by reading the parity disk.

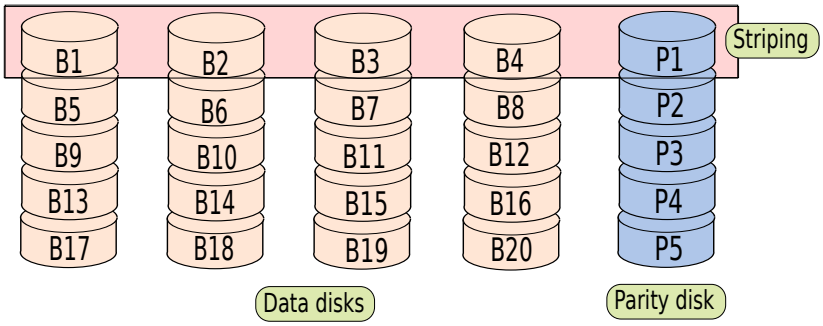


Figure 13.35: RAID 2

RAID 3

RAID 3 is almost the same as RAID 2. Instead of striping at the bit level, it stripes data at the byte level. It has the same pros and cons as RAID 2, and is thus seldom used.

RAID 4

RAID 4 is designed on the same lines as RAID 2 and 3. It stripes data at the block level. It has a dedicated parity disk that saves the parity of all the blocks on the same row. In this scheme, a read access for a single block is not as fast as RAID 2 and 3 because we cannot access different parts of a block in parallel. A write access is also slower for the same reason. However, we can read from multiple blocks at the same time if they do not map to the same disk. We cannot unfortunately do this for writes.

For a write access, we need to access two disks – the disk that contains the block, and the disk that contains the parity. An astute reader might try to argue that we need to access all the disks because we need to compute the parity of all the blocks in the same row. However, this is not true. Let us assume that there are m data disks, and the contents of the blocks in a row are $B_1 \dots B_m$ respectively. Then the parity block, P, is equal to $B_1 \oplus B_2 \oplus \dots \oplus B_m$. Now, let us assume that we change the first block from B_1 to B_1'. The new parity is given by $P' = B_1' \oplus B_2 \dots \oplus B_m$. We thus have:

$$\begin{aligned} P' &= B_1' \oplus B_2 \dots \oplus B_m \\ &= B_1 \oplus B_1 \oplus B_1' \oplus B_2 \dots \oplus B_m \\ &= B_1 \oplus B_1' \oplus P \end{aligned} \tag{13.9}$$

The results used in Equation 13.9 are: $B_1 \oplus B_1 = 0$ and $0 \oplus P' = P'$. Thus, to compute P', we need the values of B_1, and P. Hence, for performing a write to a block, we need two read

accesses (for reading B_1 and P), and two write accesses (for writing B_1' and P') to the array of hard disks. Since all the parity blocks are saved in one disk, this becomes a point of contention, and the write performance becomes very slow. Hence, RAID 4 is also seldom used.

RAID 5

RAID 5 mitigates the shortcomings of RAID 4. It distributes the parity blocks across all the disks for different rows as shown in Figure 13.36. For example, the 5^{th} disk stores the parity for the first row, and then the 1^{st} disk stores the parity for the second row, and the pattern thus continues in a round robin fashion. This ensures that no disk becomes a point of contention, and the parity blocks are evenly distributed across all the disks.

Note that RAID 5 provides high bandwidth because it allows parallel access for reads, has relatively faster write speed, and is immune to one disk failure. Hence, it is heavily used in commercial systems.

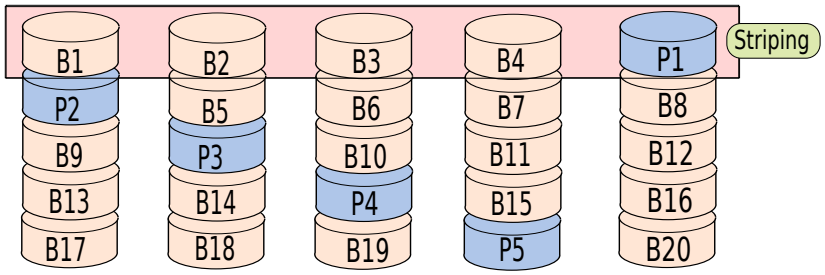


Figure 13.36: RAID 5

RAID 6

Sometimes, we might desire additional reliability. In this case, we can add a second parity block, and distribute both the parity blocks across all the disks. In this case, a write to the RAID array becomes slightly slower at the cost of higher reliability. RAID 6 is mostly used in enterprises that desire highly reliable storage. It is important to note that the two parity blocks in RAID 6 are not a simple XOR of bits. The contents of the two parity blocks for each row differ from each other, and are complex functions of the data. The reader requires background in field theory to understand the operation of the error detection blocks in RAID 6.

13.8.3 Optical Disks – CD, DVD, Blu-ray

We typically use optical disks such as CDs, DVDs, and Blu-ray disks to store videos, music, and software. As a matter of fact, optical disks have become the default distribution media for videos and music (other than the internet of course). Consequently, almost all desktops and laptops have a built-in CD or DVD drive. The reader needs to note that the physics of optical disks is very different from that of hard disks. We read the data stored in a hard disk by

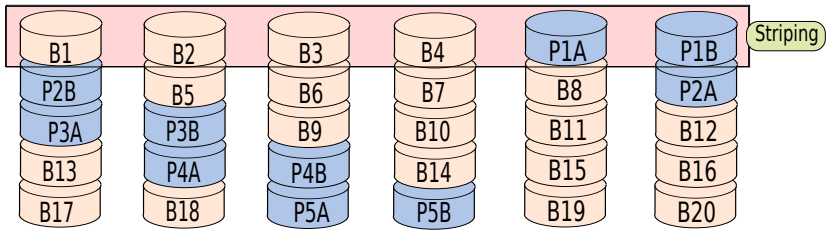


Figure 13.37: RAID 6

measuring the change in magnetic field due to the relative motion of tiny magnets embedded in the platters of the hard disk. In comparison, in an optical disk, we read data by using photo-detectors (light detectors) to measure the intensity of optical signals reflected off the surface of the disk.

The reader needs to note that CDs (compact disks), DVDs (Digital Video Disks, or Digital Versatile Disks), and Blu-ray disks, basically use the same technology. CDs represent first generation optical disks, DVDs represent second generation optical disks, and Blu-ray disks are representative of the third generation. Successive generations are typically faster and can provide more storage capacity. Let us now consider the physics of optical storage media.

Basic Physics of Optical Storage Media

An optical disk is shown in Figure 13.38. It is a circular disk, and is typically 12 cm in diameter. It has a hole in the center that is meant for attaching to a spindle (similar to hard disks). The hole is 1.5 cm in diameter, and the entire optical disk is 1.2 mm thick. An optical disk is made of multiple layers. We are primarily concerned with the reflective layer that reflects laser light to a set of detectors. We encode data bits by modifying the surface of the reflective layer. Let us elaborate.

The data is saved in a spiral pattern that starts from the innermost track, covers the entire surface of the disk, and ends at the outermost track. The width of the track, and the spacing between the tracks depends on the optical disk generation. Let us outline the basic mechanism that is used to encode data on the spiral path. The spiral path has two kinds of regions namely *lands* and *pits*. Lands reflect the optical signal, and thus represent the physical bit, 1. Lands are represented by a flat region in the reflective layer. In comparison, pits have lower reflectivity, and the reflected light is typically out of phase with the light reflected off the lands, and thus they represent the physical bit, 0. A *pit* is a depression on the surface of the reflective layer. The data on a CD is encoded using the NRZI encoding scheme (see Section 13.2.7). We infer a logical 1 when there is a pit to land, or land to pit transition. However, if there are no transitions, then we keep on inferring logical 0s.

Optical Disk Layers

An optical disk typically has four layers (refer to Figure 13.39).

Figure 13.38: An optical disk (source [dis,])

Polycarbonate Layer The polycarbonate layer is a layer of polycarbonate plastic. Lands and pits are created at its top using an injection molding process.

Reflective Layer The reflective layer consists a thin layer of aluminum or gold that reflects the laser light.

Lacquer Layer The lacquer based layer on top of the reflective layer protects the reflective layer from scratches, and other forms of accidental damage.

Surface Layer Most vendors typically add a plastic layer over the lacquer layer such that it is possible to add a label to the optical disk. For example, most optical disks typically have a poster of the movie on their top surface.

The optical disk reader sends a laser signal that passes through the polycarbonate layer and gets focused on the lands or pits. In CDs, the polycarbonate layer is typically very deep, and it occupies most of the volume. In comparison, the polycarbonate layer occupies roughly half the volume in DVDs. For third generation optical disks the reflective layer is very close to the bottom surface.

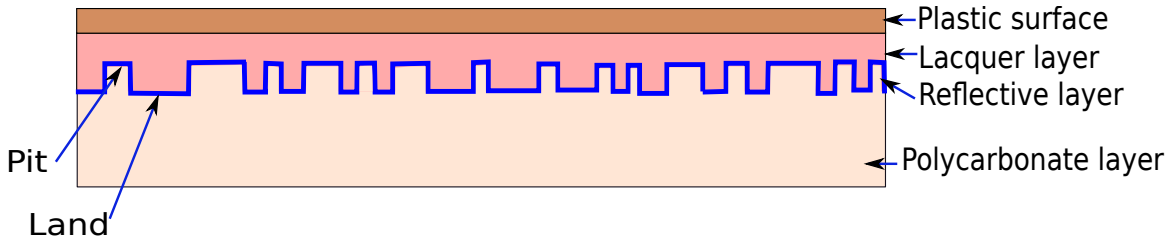


Figure 13.39: Optical disk layers

Optical Disk Reader

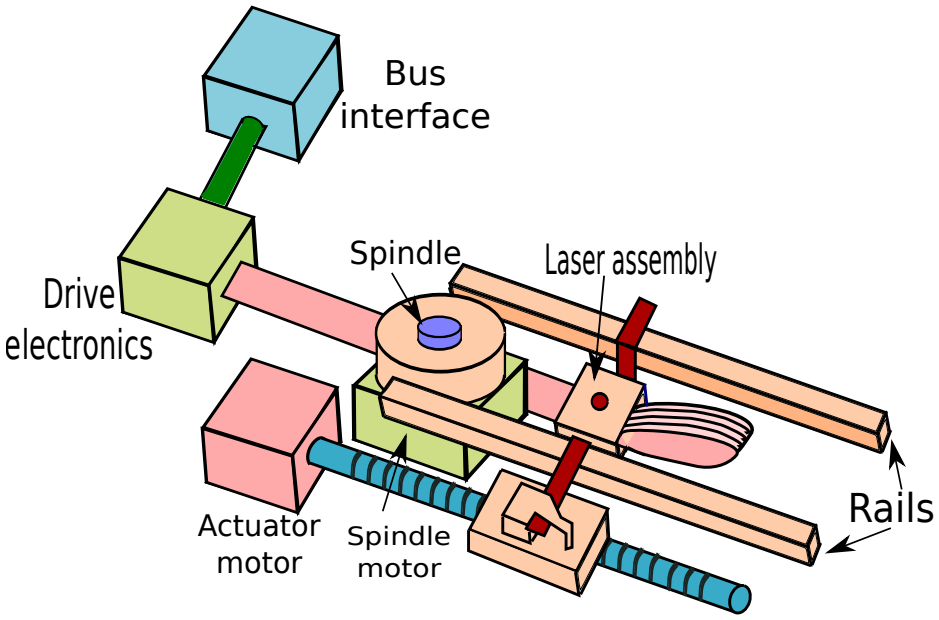


Figure 13.40: Optical disk reader

An optical disk reader is very similar to a hard disk drive (refer to Figure 13.40). The optical disk rotates on a spindle. The label of the optical disk is oriented towards the top. The actuator and head assembly are located at the bottom. Unlike a hard disk that uses a rotary actuator (rotating arm), an optical disk drive uses a linear actuator [Abramovitch, 2001] that slides radially in or out. Figure 13.40 shows a laser assembly that slides on a set of rails. The laser assembly is connected to an actuator via a system of gears and mechanical components. The actuator motor can very precisely rotate its spindle, and the system of gears translate rotational motion into linear motion of the laser assembly.

The head is a part of the laser assembly. The head typically contains a light source (laser) that is focused on the reflective layer through a system of lenses. The reflective layer then reflects the light, and a part of the reflected light gets captured by the optical disk head. The reflected light is converted to electrical signals within the head by photodetectors. The sequence of electrical signals is processed by dedicated circuitry in the drive, and converted to a sequence of logical bits. Similar to hard disks, optical drives perform error detection and correction.

One important point of difference from hard disks is that the optical disk rotates at constant linear velocity. This means that pits and lands traverse under the head at the same velocity irrespective of the track. In other words, the data transfer rate is the same irrespective of the position of the head. To support this feature, it is necessary to change the rotational speed of the spindle according to the position of the head. When the head is traveling towards the periphery of the disk, it is necessary to slow the disk down. The spindle motor has sophisticated support

for acceleration and deceleration in optical drives. To simplify the logic, we can implement zoning here similar to the zoning in hard disk drives (see Section 13.8.1). However, in the case of optical drives, zoning is mostly used in high performance drives.

Advanced Features

Most audio, video, and software CDs/DVDs are written once by the original vendors, and are sold as read-only media. Users are not expected to overwrite the optical storage media. Such kind of optical disks use 4 layers as described in Figure 13.39. Optical disks are also used to archive data. Such disks are typically meant to be written once, and read multiple times (CD-R and DVD-R formats). To create such recordable media, the polycarbonate layer is coated with an organic dye that is sensitive to light. The organic dye layer is coated with the reflective metallic layer, the lacquer layer, and the surface layer. While writing the CD, a high powered write-laser focuses light on the dye and changes its reflectivity. *Lands* are regions of high reflectivity, and *pits* are regions of low reflectivity. Such write-once optical media were superseded by optical media (CDs or DVDs) that can be read and written many times. Here, the reflective layer is made of a silver-indium-antimony-tellurium alloy. When it is heated to 500°C, spots in the reflective layer lose their reflectivity because the structure of the alloy becomes amorphous. To make the spots reflective they are heated to 200°C such that the state of the alloy changes to the polycrystalline state. We can thus encode lands and pits in the reflective layer, erase them, and rewrite them as required.

Modern disks can additionally have an extra layer of lands and pits. The first layer is coated with a chemical that is partially transparent to light. For example, pits can be coated with fluorescent material. When irradiated with red light they glow and emit light of a certain wavelength. However, most of the red light passes to the second layer, and then interacts with the fluorescent material in the pits, which is different from the material in the first layer. By analyzing the nature of reflected light, and by using sophisticated image filtering algorithms, it is possible to read the encoded data in both the layers. A simpler solution is to encode data on both sides of the optical disk. This is often as simple as taking two single side disks and pasting their surface layers together. To read such a disk, we need two laser assemblies.

Comparison of CDs, DVDs, and Blu-ray Disks

	CD	DVD	Blu-ray
Generation	1^{st}	2^{nd}	3^{rd}
Capacity	700 MB	4.7 GB	25 GB
Uses	Audio	Video	High definition Video
Laser wavelength	780 nm	650 nm	405 nm
Raw 1X transfer rate	153 KB/s	1.39 MB/s	4.5 MB/s

Table 13.9: Comparison between CD, DVD, and Blu-ray disks

Refer to Table 13.9 for a comparison of CD, DVD, and Blu-ray disks.

13.8.4 Flash Memory

Hard disks, and optical drives are fairly bulky, and need to be handled carefully because they contain sensitive mechanical parts. An additional shortcoming of optical storage media is that they are very sensitive to scratches and other forms of minor accidental damage. Consequently, these devices are not ideally suited for portable and mobile applications. We need a storage device that does not consist of sensitive mechanical parts, can be carried in a pocket, can be attached to any computer, and is extremely durable. Flash drives such as USB pen drives satisfy all these requirements. A typical pen drive can fit in a wallet, can be attached to all kinds of devices, and is extremely robust and durable. It does not lose its data when it is disconnected from the computer. We have flash based storage devices in most portable devices, medical devices, industrial electronics, disk caches in high-end servers, and small data storage devices. Flash memory is an example of an EEPROM (Electrically Erasable Programmable Read Only Memory) or EPROM (Erasable Programmable Read Only Memory). Note that traditionally EPROM based memories used ultraviolet light for erasing data. They have been superseded by flash based devices.

Let us look at flash based technology in this section. The basic element of storage is a floating gate transistor.

The Floating Gate Transistor

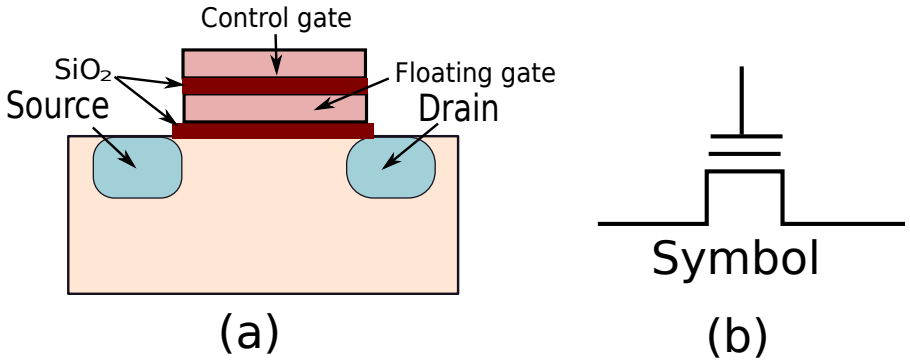


Figure 13.41: A floating gate transistor

Figure 13.41 shows a floating gate transistor. The figure shows a regular NMOS transistor with two gates instead of one. The gate on top is known as the control gate, and is equivalent to the gate in normal MOS transistors. The gate below the control gate is known as the floating gate. It is surrounded on all sides by an SiO_2 based electrical insulation layer. Hence, the floating gate is electrically isolated from the rest of the device. By some means if we are able to implant a certain amount of charge in the floating gate, then the floating gate will maintain its potential for a very long time. In practice, there is a negligible amount of current flow between the floating gate and the rest of the components in the floating gate transistor under normal conditions. Let us consider two scenarios. In the first scenario, the floating gate is not charged. In this case, the floating gate transistor acts as a regular NMOS transistor. However, if the

floating gate has accumulated electrons containing negative charge, then we have a negative potential gradient between the channel and the control gate. Recall that to create an n-type channel in the transistor, it is necessary to apply a positive voltage to the gate, where this voltage is greater than the threshold voltage. In this case the threshold voltage is effectively higher because of the accumulation of electrons in the floating gate. In other words, to induce a channel in the substrate, we need to apply a larger positive voltage at the control gate.

Let the threshold voltage when the floating gate is not charged with electrons be V_T, and let the threshold voltage when the floating gate contains negative charge be V_T^+ ($V_T^+ > V_T$). If we apply a voltage that is in between V_T and V_T^+ to the control gate, then the NMOS transistor conducts current if no charge is stored in the floating gate (threshold voltage is V_T). If the threshold voltage of the transistor is equal to V_T^+, then the transistor remains in the *off* state. It thus does not conduct any current. We typically assume that the default state (no charge on the floating gate) corresponds to the 1 state. When the floating gate is charged with electrons, we assume that the transistor is in the 0 state. When we set the voltage at the control gate to a value between V_T and V_T^+, we enable the floating gate transistor.

Now, to write a value of 0 or *program* the transistor, we need to deposit electrons in the floating gate. This can be done by applying a strong positive voltage to the control gate, and a smaller positive voltage to the drain terminal. Since there is a positive potential difference between the drain and source, a channel gets established between the drain and source. The control gate has an even higher voltage, and thus the resulting electric field pulls electrons from the n-type channel and deposits some of them in the floating gate.

Similarly, to *erase* the stored 0 bit, we apply a strong negative voltage between the control gate and the source terminal. The resulting electric field pulls the electrons away from the floating gate into the substrate, and source terminal. At the end of this process, the floating gate loses all its negative charge, and the flash device comes back to its original state. It now stores a logical 1.

To summarize, programming a flash cell means writing a logical 0, and erasing it means writing a logical 1. There are two fundamental ways in which we can arrange such floating gate transistors to make a basic flash memory cell. These methods are known as NOR flash and NAND flash respectively.

NOR FLash

Figure 13.42 shows the topology of a two transistor NOR flash cell that saves 2 bits. Each floating gate transistor is connected to a bit line on one side, and to the ground on the other side. Each of the control gates are connected to distinct word lines. After we enable a floating gate transistor it pulls the bit line low if it stores a logical 1, otherwise it does not have any effect because it is in the *off* state. Thus, the voltage transition in the bit line is logically the reverse of the value stored in the transistor. The bit line is connected to a sense amplifier that senses its voltage, flips the bit, and reports it as the output. Similarly, for writing and erasing we need to set the word lines, bit lines, and source lines to appropriate voltages. The advantage of NOR flash is that it is very similar to a traditional DRAM cell. We can build an array of NOR flash cells similar to a DRAM array. The array based layout allows us to access each individual location in the array uniquely.

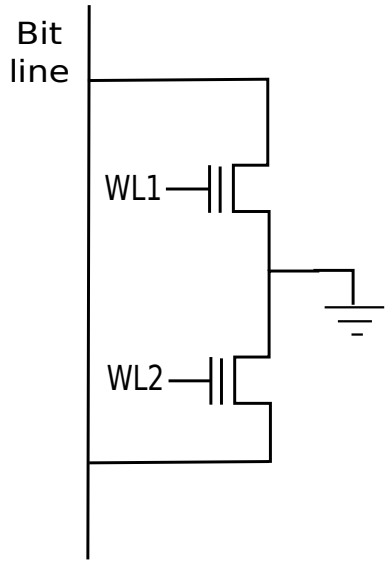


Figure 13.42: NOR Flash Cell

NAND Flash

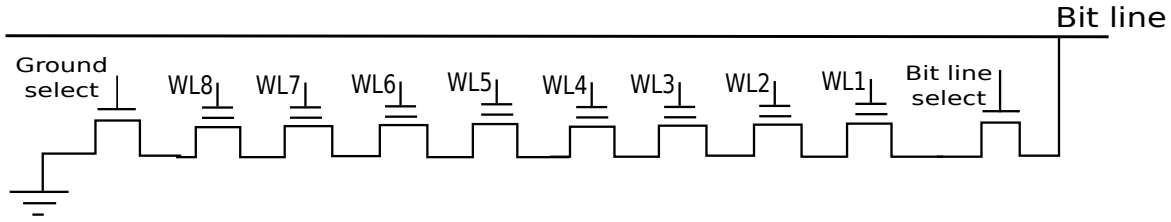


Figure 13.43: NAND Flash Cell

A NAND flash cell has a different topology. It consists of a set of NMOS floating gate transistors in series similar to series connections in CMOS NAND gates (refer to Figure 13.43). There are two dedicated transistors at both ends known as the *bit line select transistor*, and *ground select* transistor. A typical array of transistors connected in the NAND configuration contains 8 or 16 transistors. To read the value saved in a certain transistor in a NAND flash array, there are three steps. The first step is to enable the ground select, and bit line transistors. The second step is to turn on the rest of the floating gate transistors other than the one we wish to read by setting their word line voltages to V_T^+. Lastly, we read a specific transistor by setting its word line voltage to some value between V_T and V_T^+. If the cell is not programmed (contains a 1), it drives the bit line low, otherwise it does not change the voltage on the bit line. Sense amplifiers infer the value of the logical bit saved in the transistor. Such arrays of

floating gate transistors known as NAND flash cells are connected in a configuration similar to NOR flash cells.

This scheme might look complicated at the outset; however, it has a lot of advantages. Consequently, most of the flash devices in use today use NAND flash memories instead of NOR flash memories. The bit storage density is much higher. A typical NAND flash cell uses a lesser number of wires than a NOR flash cell because all the floating gate transistors are directly connected to each other, and there is just one connection to the bit line and ground terminal. Hence, NAND flash memories have at least 40-60% higher density as compared to NOR flash cells. Let us thus only consider NAND flash memories from now on, and refrain from discussing NOR flash memories.

Blocks and Pages

The most important point to note here is that a (NAND) flash memory device is not a memory device, it is a storage device. Memory devices provide byte level access. In comparison, storage devices typically provide block level access, where one block can be hundreds of kilobytes long. Due to temporal and spatial locality in accesses to storage media, the working set of most programs is restricted to a few blocks. Secondly, to reduce the number of accesses to storage devices, most operating systems have in-memory storage caches such as hard disk caches. Most of the time, the operating system reads and writes to the in-memory caches. This reduces the I/O access time.

However, after certain events it is necessary to synchronize the cache with the underlying storage device. For example, after executing a *sync*() system call in Linux, the hard disk cache writes its updates to the hard disk. Depending on the semantics of the operating system, and file system, writes are sent to the underlying storage media after a variety of events. For example, when we right-click the icon for a USB drive in the "My Computer" screen on Windows and select the eject option, the operating system ensures that all the outstanding write requests are sent to the USB device. Most of the time users simply unplug a USB device. This practice can occasionally lead to data corruption, and unfortunately your author has committed this mistake several times. This is because, when we pull out a USB drive, some uncommitted changes are still present in the in-memory cache. Consequently, the USB pen drive contains stale and possibly half-written data.

Data in NAND flash devices is organized in the granularity of pages and blocks. A *page* of data typically contains 512 – 4096 bytes (in powers of 2). Most NAND flash devices can typically read or write data at the granularity of pages. Each page additionally has extra bits for error correction based on CRC codes. A set of pages are organized into a block. Blocks can contain 32 – 128 pages, and their total size ranges from 16-512 KB. Most NAND flash devices can erase data at the level of blocks. Let us now look at some salient points of NAND flash devices.

Program/Erase Cycles

Writing to a flash device essentially means writing a logical 0 bit since by default each floating gate transistor contains a logical 1. In general, after we have written data to a block, we cannot write data again to the same block without performing additional steps. For example, if we have written 0110 to a set of locations in a block, we cannot write 1001 to the same

set of locations without erasing the original data. This is because, we cannot convert a 0 to a 1, without erasing data. Erasing is a slow operation, and consumes a lot of power. Hence, the designers of NAND flash memories decided to erase data at large granularities, i.e., at the granularity of a block. We can think of accesses to flash memory as consisting of a *program phase*, where data is written, and an *erase phase*, where the data stored in all the transistors of the block is erased. In other words, after the erase phase, each transistor in the block contains a logical 1. We can have an indefinite number of read accesses between the program phase, and the erase phase. A pair of program and erase operations is known as a program/erase cycle, or P/E cycle.

Unfortunately, flash devices can endure a finite number of P/E cycles. As of 2013, this number is between 100,000 to 1 million. This is because each P/E cycle damages the silicon dioxide layer surrounding the floating gate. There is a gradual breakdown of this layer, and ultimately after hundreds of thousands of P/E cycles it does not remain an electrical insulator anymore. It starts to conduct current, and thus a flash cell loses its ability to hold charge. This gradual damage to the insulator layer is known as *wear and tear*. To mitigate this problem, designers use a technique called *wear leveling*.

Wear Leveling

The main objective of *wear leveling* is to ensure that accesses are symmetrically distributed across blocks. If accesses are non-uniformly distributed, then the blocks that receive a large number of requests will wear out faster, and develop faults. Since data accesses follow both temporal and spatial locality we expect a small set of blocks to be accessed most often. This is precisely the behavior that we wish to prevent. Let us further elaborate with an example. Consider a pen drive that contains songs. Most people typically do not listen to all the songs in a round robin fashion. Instead, they most of the time listen to their favorite songs. This means that a few blocks that contain their favorite songs are accessed most often, and these blocks will ultimately develop faults. Hence, to maximize the lifetime of the flash device, we need to ensure that all the blocks are accessed with roughly the same frequency. This is the best case scenario, and is known as *wear leveling*.

The basic idea of wear leveling is that we define a logical address and a physical address for a flash device. A *physical address* corresponds to the address of a block within the flash device. The logical address is used by the processor and operating system to refer to data in the flash drive. We can think of the logical address as virtual memory, and the physical address as physical memory. Every flash device contains a circuit that maps logical addresses to physical addresses. Now, we need to ensure that accesses to blocks are uniformly distributed. Most flash devices have an access counter associated with each block. This counter is incremented once every P/E cycle. Once the access count for a block exceeds the access counts of other blocks by a predefined threshold, it is time to swap the contents of the frequently accessed block with another less frequently accessed block. Flash devices use a separate temporary block for implementing the swap. First the contents of block 1 are copied to it. Subsequently, block 1 is erased, and the contents of block 2 are copied to block 1. The last step is to erase block 2, and copy the contents of the temporary block to it. Optionally, at the end, we can erase the contents of the temporary block. By doing such periodic swaps, flash devices ensure that no single block wears out faster than others. The logical to physical block mapping needs to be

updated to reflect the change.

Definition 163
A technique to ensure that no single block wears out faster than other blocks is known as wear leveling*. Most flash devices implement wear leveling by swapping the contents of a block that is frequently accessed with a block that is less frequently accessed.*

Read Disturbance

Another reliability issue in flash memories is known as *read disturbance*. If we read the contents of one page continuously, then the neighboring transistors in each NAND cell start getting programmed. Recall that the control gate voltage of the neighboring transistors needs to be greater than V_T^+ such that they can pass current. In this case, the voltage of the gate is not as high as the voltage that is required to program a transistor, and it also lasts for a shorter duration. Nonetheless, a few electrons do accumulate in the floating gate. After thousands of read accesses to just one transistor, the neighboring transistors start accumulating negative charge in their floating gates, and ultimately get programmed to store a 0 bit.

To mitigate this problem, most designs have a read counter with each page or block. If the read counter exceeds a certain threshold, then the flash controller needs to move the contents of the block to another location. Before copying the data, the new block needs to be erased. Subsequently, we transfer the contents of the old block to the new block. In the new block, all the transistors that are not programmed start out with a negligible amount of negative charge in their floating gates. As the number of read accesses to the new block increases, transistors start getting programmed. Before we reach a threshold, we need to migrate the block again.

13.9 Summary and Further Reading

13.9.1 Summary

Summary 13

1. *The I/O system connects the processor to the I/O devices. The processor and all the chips for processing I/O data (chipset), are attached to a printed circuit board known as the* motherboard*. The motherboard also contains ports (hardware connectors) for attaching I/O devices.*
   (a) *The most important chips in the chipset are known as the North Bridge and Soutbridge chips in Intel-based systems.*
   (b) *The North Bridge chip connects the processor to the graphics card, and main memory.*

(c) *The South Bridge chip is connected to the North Bridge chip and a host of I/O and storage devices such as the keyboard, mouse, hard disk, and network card.*

2. *Most operating system define two basic I/O operations namely* read *and* write. *An I/O request typically passes from the application to the I/O device through the kernel, device driver, the processor, and elements of the I/O system.*

3. *We divide the functionality of the I/O system into 4 layers.*

   (a) *The physical layer defines the electrical specifications, signaling and timing protocols of a bus. It is further divided into two sublayers namely the transmission sublayer, and the synchronization sublayer.*

   (b) *The data link layer gets a sequence of logical bits from the physical layer, and then performs the tasks of framing, buffering, and error correction. If multiple devices want to access the bus, then the process of scheduling the requests is known as arbitration. The data link layer implements arbitration, and also has support for I/O transactions (sequence of messages between sender and receiver), and split transactions (transactions divided into multiple mini transactions).*

   (c) *The network layer helps to route data from the processor to I/O devices and back.*

   (d) *The protocol layer is concerned with implementing an entire I/O request end-to-end.*

4. *Physical Layer:*

   (a) *Transmission Sublayer Protocols: active high, active low, return to zero (RZ), non return to zero (NRZ), non return to zero inverted (NRZI), and Manchester encoding.*

   (b) *Synchronization Sublayer Protocols: synchronous (same clock for both ends), mesochronous (fixed phase delay), plesiochronous (slow drift in the clock), source synchronous (clock passed along with the data), and asynchronous (2-phase handshake, and 4-phase handshake).*

5. *Data Link Layer*

   (a) *Framing protocols: bit stuffing, pauses, bit count*

   (b) *Error Detection/ Correction: parity, SEC, SECDED, CRC*

   (c) *Arbitration: centralized, daisy chain (supports priority, and notion of tokens)*

   (d) *Transaction: single transaction (example, DRAM bus), split transaction (break a transaction into smaller transactions)*

6. *Network Layer*

   (a) *Port-Mapped I/O: Each I/O port is mapped to a set of registers that have unique addresses. The in and out instructions are used to read and write data to the ports respectively.*

(b) Memory-Mapped I/O: Here, we map the I/O ports to the virtual address space.

7. *Protocol Layer:*

   (a) Polling: Keep querying the device for a change in its state.

   (b) Interrupts: The device sends a message to the processor, when its status changes.

   (c) DMA (Direct Memory Access): Instead of transferring data from an I/O device to main memory by issuing I/O instructions, the device driver instructs the DMA engine to transfer a chunk of data between I/O devices and main memory. The DMA engine interrupts the processor after it is done.

8. *Case Studies:*

Protocol	*Usage*	*Salient Points*
PCI Express	*motherboard bus*	*high speed asynchronous bus, supports multiple lanes*
SATA	*storage devices*	*half duplex, asynchronous bus, supports low level commands on storage devices*
SAS	*storage devices*	*full duplex, asynchronous bus, backward compatible with SATA, extensive SCSI command set*
USB	*peripherals*	*single bit, half duplex, asynchronous bus. Extensive support for all kinds of traffic (bulk, interrupt, and isochronous).*
FireWire	*peripherals*	*full duplex, data strobe encoding. Peripherals organized as a computer network with a leader node.*

9. *Storage Devices: Hard Disk*

   (a) In a hard disk we encode data by changing the relative orientations of tiny magnets on the surface of the platter.

   (b) We group a set of typically 512 bytes into a sector. On a platter, sectors are arranged in concentric circles known as tracks.

   (c) The head of a hard disk first needs to move to the right track (seek time), then wait for the correct sector to arrive under the head (rotational latency). Finally, we need to transfer the data to the processor after post processing (transfer latency).

10. *Storage Devices: Optical disc*

    (a) The surface of an optical disc contains flat region (lands), and depressions (pits). The pits have lower reflectivity.

(b) *Optical discs rotate on a spindle similar to a platter in a hard disk. The optical head focuses a laser light on the surface of the disc, and then an array of photodetectors analyzes the reflected light. A transition between a pit and a land (or vice versa) indicates a logical 1. Otherwise, we read a logical 0.*

(c) *CDs (compact discs) are first generation optical discs, DVDs are second generation optical discs, and Blu-Ray discs are third generation optical discs.*

11. *Storage Devices: Flash Memory*

(a) *Flash memory contains a floating gate transistor that has two gates – control and floating. If the floating gate has accumulated electrons then the transistor stores a logical 0 (else it stores a logical 1).*

(b) *We program (set to 0) a floating gate transistor by applying a high positive voltage pulse to the control gate. Likewise, we erase the value when we apply a pulse with the opposite polarity.*

(c) *Floating gate transistors can be connected in the NAND and NOR configurations. The NAND configuration has much higher density and is thus more commonly used.*

(d) *While designing flash devices we need to perform wear leveling, and take the phenomenon of read disturbance into account.*

13.9.2 Further Reading

For the latest designs of motherboards, and chipsets, the most accurate and up-to-date source of information is the vendor's website. Most vendors such as Intel and AMD post the details and configurations of their motherboards and chipsets after they are released. However, they typically do not post a lot of details about the architecture of the chips. The reader can refer to research papers for the architectural details of the AMD Opteron North Bridge chip [Conway and Hughes, 2007], Intel Blackford North Bridge chip [Radhakrishnan et al., 2007], and AMD North Bridge chip [Owen and Steinman, 2008] for the Griffin processor family. The book by Dally and Poulton [Dally and Poulton, 1998] is one of the best sources for information on the physical layer, and is a source of a lot of information presented in this book. Other books on digital communication [Proakis and Salehi, 2007, Sklar, 2001] are also excellent resources for further reading. Error control codes are mostly taught in courses on coding and information theory. Hence, for a deeper understanding of error control codes, the reader is referred to [Ling and Xing, 2004, Cover and Thomas, 2013]. For a detailed description of the Intel's I/O architecture and I/O ports, we shall point the reader to Intel's software developer manuals at [int,]. The best sources for the I/O protocols are their official specifications – PCI Express [pci,], SATA [sat,], SCSI and SAS [scs,], USB [usb,], and FireWire [fir,]. The book on memory systems by Jacob, Ng, and Wang [Jacob et al., 2007] is one of the best resources for additional information on hard disks, and DRAM memories. They explain the structure of a hard disk, and its internals in great detail. The official standards for compact discs are documented in the rainbow books. Each book in this collection contains the specifications of a certain type

of optical disc. One of the earliest and most influential books in this collection is the Red Book [red,] that contains the specifications for audio CDs. The latest DVD standards are available from `http://www.dvdforum.org`, and the latest Blu-Ray standards are available at the official website of the Blu-Ray Disc Association (`http://www.blu-raydisc.com/en`).

Exercises

Overview of the I/O System

Ex. 1 — What are the roles of the North Bridge and South Bridge chips?

Ex. 2 — What is the role of the chipset in a motherboard?

Ex. 3 — Describe the four layers in the I/O system.

Ex. 4 — Why is it a good idea to design a complex system as a sequence of layers?

Physical Layer

Ex. 5 — What is the advantage of LVDS signaling?

Ex. 6 — Draw the circuit diagram of a LVDS transmitter and receiver.

Ex. 7 — Assume that we are transmitting the bit sequence: 01101110001101. Show the voltage on the bus as a function of time for the following protocols: RZ, NRZ, Manchester, NRZI.

Ex. 8 — What is the advantage of the NRZI protocol over the NRZ protocol?

* **Ex. 9** — Draw the circuit diagram of the receiver of a plesiochronous bus.

* **Ex. 10** — What are the advantages of a source synchronous bus?

* **Ex. 11** — Why is it necessary to avoid transitions in the keep-out region?

Ex. 12 — Differentiate between a 2-phase handshake, and a 4-phase handshake?

* **Ex. 13** — Why do we set the strobe after the data is stable on the bus?

** **Ex. 14** — Design the circuit for a tunable delay element.

Data Link, Network, and Protocol Layer

Ex. 15 — What are the different methods for demarcating frames?

Ex. 16 — Consider an 8-5 SECDED code. Encode the message: 10011001.

** **Ex. 17** — Construct a code that can detect 3 bit errors.

** **Ex. 18** — Construct a fully distributed arbiter. It should not have any central node that schedules requests.

Ex. 19 — What is the advantage of split transaction buses?

Ex. 20 — How do we access I/O ports?

Ex. 21 — What is the benefit of memory-mapped I/O?

Ex. 22 — What are the various methods of communication between an I/O device and the processor? Order them in the increasing order of processor utilization.

Ex. 23 — Assume that for a single polling operation, a processor running at 1 MHz takes 200 cycles. A processor polls a printer 1000 times per minute. What percentage of time does the processor spend in polling?

Ex. 24 — When is polling more preferable than interrupts?

Ex. 25 — When are interrupts more preferable than polling?

Ex. 26 — Explain the operation of the DMA controller.

Hard Disks

Ex. 27 — What is the advantage of zoned recording?

Ex. 28 — Describe the operation of a hard disk.

Ex. 29 — We have a hard disk with the following parameters:

Seek Time	50 ms
Rotational Speed	600 RPM
Bandwidth	100 MB/s

(a) How long will it take to read 25 MB on an average if 25 MB can be read in one pass.

(b) Assume that we can only read 5 MB in one pass. Then, we need to wait for the platter to rotate by 360° such that the same sector comes under the head again. Now, we can read the next chunk of 5 MB. In this case, how long will it take to read the entire 25 MB chunk?

* **Ex. 30** — Typically, in hard disks, all the heads do not read data in parallel. Why is this the case?

Ex. 31 — Let us assume that we need to read or write long sequences of data. What is the best way of arranging the sectors on a hard disk? Assume that we ideally do not want to change tracks, and all the tracks in a cylinder are aligned.

* **Ex. 32** — Now, let us change the assumptions in the previous exercise. Assume that it is faster to move to the next track on the same recording surface, than starting to read from another track in the same cylinder. With these assumptions, how should we arrange the sectors? [NOTE: The question is not as easy as it sounds.]

Ex. 33 — Explain the operation of RAID 0,1,2,3,4,5, and 6.

Ex. 34 — Consider 4 disks D0, D1, D2, D3 and a parity disk P using RAID 4. The following table shows the contents of the disks for a given sector address, S, which is the same across all the disks. Assume the size of a block to be 16 bits.

D0	D1	D2	D3
0xFF00	0x3421	0x32FF	0x98AB

Compute the value of the parity block? Now the contents of D0 are changed to 0xABD1. What is the new value of the parity block?

Ex. 35 — Assume that we want to read a block as fast as possible, and there are no parallel accesses. Which RAID technology should we choose?

Ex. 36 — What is the advantage of RAID 5 over RAID 4?

Optical and Flash Drives

Ex. 37 — What is the main difference between a CD and DVD?

Ex. 38 — How do we use the NRZI encoding in optical drives?

* **Ex. 39** — What is the advantage of running a drive at constant linear velocity over running it at constant angular velocity?

* **Ex. 40** — For an optical drive that runs at constant linear velocity, what is the relationship between the angular velocity, and the position of the head?

Ex. 41 — What are the basic differences between NAND and NOR flash?

Ex. 42 — Explain *wear leveling*, and *read disturbance*? How are these issues typically handled in modern flash drives?

Design Problems

Ex. 43 — Read more about the Hypertransport™, Intel Quickpath, Infiniband™, and Myrinet® protocols? Try to divide their functionality into layers as we have presented in this chapter.