# Accelerated Learning Series

## (www.ALearnOnline.com – A site dedicated to education)

# Modules in Computer Science & Engineering

## General Purpose Computer Architecture

***A set of notes on the architecture of the general purpose processor and peripherals.***

*Author: SKG.*
*August 2006.*

# Revision History

## Version 1.0 (August 2006)

- First version created

**Table of Contents.**

# Prerequisites

- Application Specific Integrated Circuit Design (ALS notes in Hardware Engineering)

# Preface

Prior modules in the Hardware Engineering series, discussed the physics of semiconductors and the design of digital modules for specific applications. Through these modules the student has gained an appreciation for the complexity, cost, and other time and effort related constraints in designing application specific circuitry.

This background naturally leads us to the concept of a “General Purpose Processor”. Such a processor is designed to load instructions from memory and execute them as opposed to being **hard-wired** to perform one and only one function. The instructions in memory can be altered at will, without any need to change the underlying hardware and thus allowing one to completely change the application for the processor by changing the instruction bits stored in memory.

I start this module by discussing the layout of the circuit board for a modern general purpose computer (also known as the **motherboard**). A general purpose computer is comprised of a general purpose processor (also known as the **microprocessor or Central Processing Unit - CPU**) and peripheral modules that are designed to work in concert toward storing, loading and executing instructions in memory.

This will be followed by a discussion on each of the modules on the motherboard from a functional perspective. The student is encouraged to think about how these individual modules may be implemented in hardware, based on the prior exposure to digital design.

Finally I will conclude by briefly introducing the **Computer Operating System** by discussing the purpose that it serves and some of its more common components.

# Acknowledgements

I am grateful to A. Walker for taking the time to review this document and for providing very valuable and constructive feedback. I am also indebted to M.K. Achuthan for providing valuable input and feedback at various times during the writing of this set of notes.

# 1.0 – Computer System Fundamentals

At the heart of every computer is a **Central Processing Unit**, also known as the **CPU**. It is responsible for **fetching** instructions from **memory** and **executing** them. These instructions are invariably mathematical operations on one or more pieces of data. Whether a computer is used to solve a complex mathematical problem such as mapping the trajectory of a rocket that needs to go from the earth to the moon, or for a more mundane operation like allowing me to write this document, the instructions that need to be executed by the CPU are operations in logic that are implemented as a sequence of mathematical instructions.

A computer programmer translates a real-world problem into set of logical operations and then uses a tool (known as a **compiler** or **assembler**) to convert these logical operations into mathematical instructions that a CPU can execute. Note that the compiler/assembler are themselves a set of mathematical instructions that allow the CPU to translate logical statements into mathematical instructions.

Once an **application** is written (real-world problem translated into logical statements and then converted to mathematical instructions), the application is stored in some form of memory. When the CPU is asked to execute this application, it has to go to the location in memory where the application is stored and then fetch each mathematical instruction in the application and execute them in sequence until it completes the last instruction in the application.

In this section, we will study in some detail the various subsystems that make up a computer.

# *1.1 – General Purpose Mother-Board*

Fig 1.1.1 shows a generic layout for a modern general purpose motherboard. We have a **Central Processing Unit** (**CPU**) with direct access to a **Memory Hub**. For historical reasons the memry hub is also known as the NorthBridge. The Memory Hub has three other direct links.

- The first link is to a **Memory Bus** that hooks the memory hub to a bank of **Random Access Memory** (**RAM**).
- The second is to an **Accelerated Graphics Port Bus** that hooks the memory hub to an **Accelerated Graphics Port** (**AGP**). The AGP serves as a fast visual interface into the computer.
- The third connection into the Memory Hub is yet another hub called the **Input/Output Hub** or **I/O Hub**. For historical reasons, the I/O hub is also known as the SouthBridge. The I/O Hub hosts all the other input and output interfaces into the computer.

The **Universal Serial Bus** (**USB**) is the modern way to interface external peripherals onto the motherboard. It goes through the I/O Hub to access the rest of the system.

Similarly the **Peripheral Component Interconnect** (**PCI**) bus is yet another way to interface external peripherals onto the motherboard. It also goes through the I/O Hub to access the rest of the system.

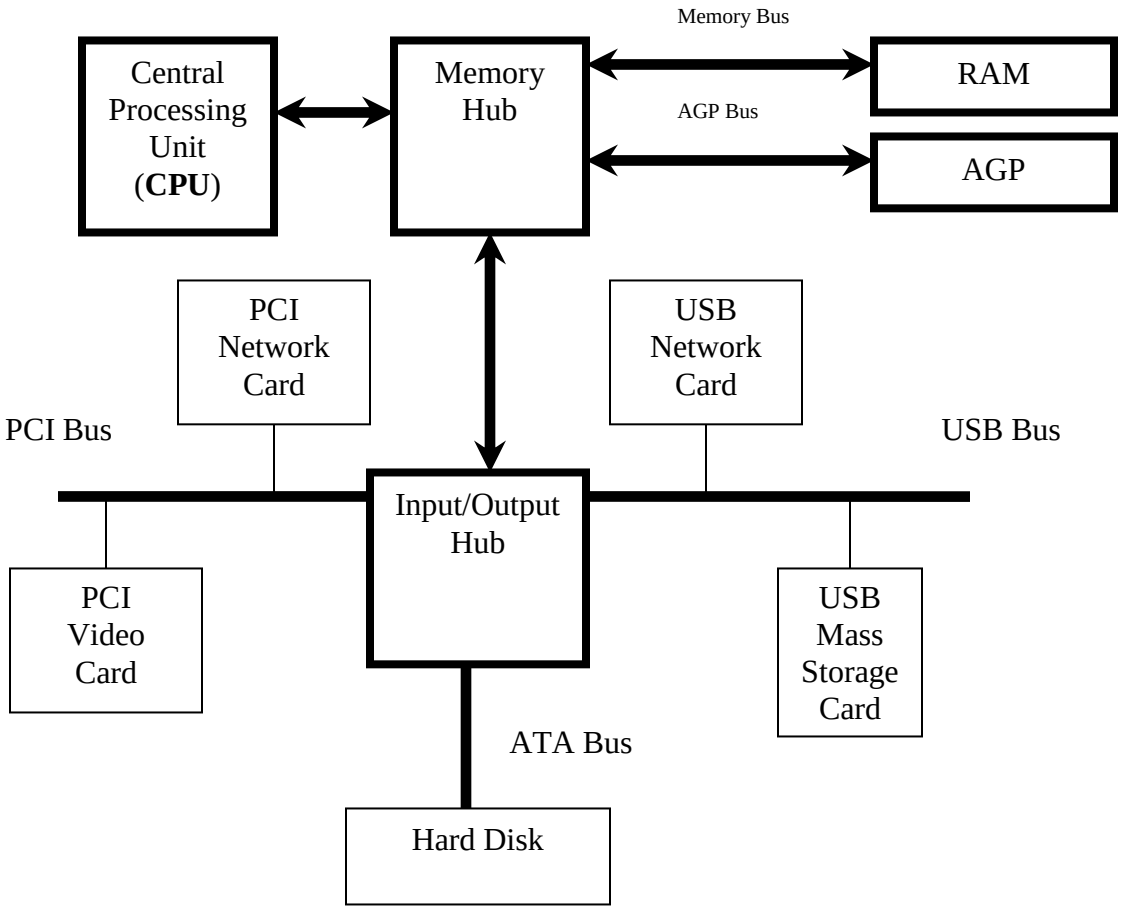


**Fig 1.1.1: General Purpose Motherboard**

Yet another common I/O interface that is attached to the I/O hub is the **AT Attachement** (**ATA**). AT identifies a storage part used in the original IBM Personal Computer. This interface falls in a category known as **mass storage devices** because it allows storing large amounts of data.

The peripherals attached to the I/O hub generally fall under three categories – **Non-Plug-&-Play**, **Plug-&-Play** and **Hot-Swapping**.

- Non-Plug-&-Play refers to peripherals that needs to be setup (using switches) to specify various settings before they can be accessed over the I/O hub (eg. ATA bus).

- Plug-&-Play refers to peripherals that can be simply plugged into the bus and powered up (eg. PCI bus).

- Hot-Swapping refers to peripherals that can be plugged in while the computer system is already powered up (eg. USB bus).

A key thing to note in this layout is that data can flow from one peripheral to another peripheral over either of the hubs without dependency on any other component including the CPU. For example it is common to plug in a USB memory device on the USB bus and transfer data to the RAM directly. While this is happening the CPU is free to continue processing unrelated instructions. This paradigm is commonly referred to as **Direct Memory Access** (**DMA**).

In the following sections we will study each of the modules in Fig 1.1.1 from a functional perspective. We will not cover details on the bus interfaces as these are fairly intricate and are covered by interface standards that are publicly available (on the internet) so that peripheral manufacturers can provide standard and compatible interfaces to motherboard manufacturers.

## *1.2 – The CPU*

Fig 1.2.1 shows a block diagram of the internal workings of the Central Processing Unit (CPU). Fundamentally the CPU is responsible for **fetching** instructions from memory and **executing** them.

The task of executing instructions is done by the **Arithmetic and Logic Unit** (**ALU**). The ALU relies on very fast memory that is physically resident inside the CPU for input and output to the ALU. These fast memory modules are referred to as **Registers**. The ALU performs very simple arithmetic and logical operations on data in the registers such as addition, subtraction, multiplication, division, bit-shifting, bit-AND, bit-OR, bit-exclusive-OR etc.

The Memory interface is responsible for interfacing the CPU to the memory bus on the mother board. The **Fetch module** will transfer instructions from memory to the registers in the CPU with the aid of the **memory interface** and the **decoding modules**.

In most cases, the fetch module can continue to fetch the next instruction from memory while the CPU is executing the previously fetched instruction. This paradigm is referred to as **pipelining** and it is a means of increasing the efficiency (speed of execution) of the CPU. The drawback to pipelining is that, if the execution of an instruction requires a jump to a location other than the next instruction in sequence, then the pipeline needs to be flushed causing wasted clock cycles in the ALU while the fetch module picks instructions from the new location.

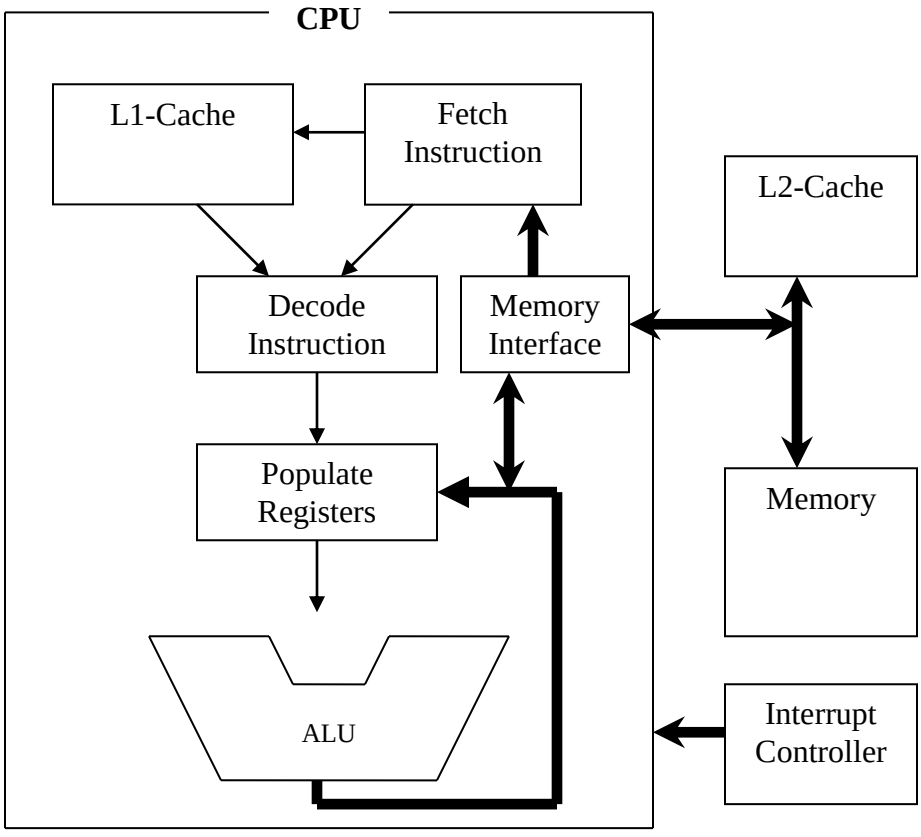


**Fig 1.2.1: CPU Architecture**

Often times, recently executed instructions or data will be needed again. The **L1-Cache** is fast memory located in the CPU where this information is preserved so as to avoid unnecessary delays by having to go back to external memory over the memory bus.

The **L2-Cache** serves a similar purpose to the L1-Cache but is **not** physically located inside the CPU and hence is slower to access than the L1-cache, but is still faster than accessing the memory.

Peripherals that need **time critical** access to the CPU, will be wired into an "**Interrupt Controller**" on the motherboard. The interrupt controller can force the CPU to stop whatever it is currently doing, save its current state and attend to an urgent request from a peripheral. Once the urgent request is serviced, the CPU restores its previously saved state and continues from where it left off, prior to the interrupt.

# *1.3 – Primary Memory Subsystem*

In computer parlance, when people refer to the memory subsystem, they are usually referring to the **primary memory that is directly tied to the CPU over the memory bus and hub**.

This primary storage memory is different from the registers and the CPU cache that we discussed in the previous section. T**he CPU has to go over the memory bus to get to the primary memory** and hence does not have direct access to it.

The primary storage memory is also different from the larger secondary storage memory modules that are linked to the CPU over the I/O Hub, in that it is **byte addressable** (can access individual bytes) whereas secondary storage modules are usually **block addressable** (a block is a group of bytes).

The primary memory is usually **slower than the registers and the cache memory**, but **larger in the amount of data that it can store**. In general the larger capacity memory types are slower in terms of access times.

The primary memory on a motherboard is generally of two types – **Read Only Memory** (**ROM**) or **Random Access Memory** (**RAM**). ROM memory is usually written to only once and then available for reading at anytime after that. RAM memory can be read from or written to at anytime.

Both ROM and RAM memory can generally be addressed **randomly** at any location within the memory without having to read sequentially from the beginning of the memory to the address where access is required.

ROM is generally a form of **non-volatile** memory, meaning it does not require power to maintain its contents. The most popular form of ROM modules is the **Erasable-Programmable-Read-Only-Memory** or **EPROMs**. These contents of these modules can be erased and reprogrammed multiple times using specialized hardware. Once programmed, the CPU can only read its contents but cannot change (write to) it.

The most prevalent form of RAM is Dynamic RAM or **DRAM**. This is a high density form of memory where a bit of information is stored with 1 transistor and 1 capacitor. But the drawback to this type of RAM is that it needs to be periodically "refreshed" (data recharged to counteract drain) in addition to having a constant power supply.

Static RAM or **SRAM** is a less dense form of memory but does not have the refresh constraint imposed by DRAM. Nevertheless, SRAM is also a form of **volatile memory** (like DRAM) and hence requires a constant power supply.

**SDRAM** is a variation of DRAM that has a synchronous interface (tied to a clock edge for latching). By default, DRAM uses an asynchronous interface (change occurs as soon as physically possible).

There is another category of memory modules referred to as **Flash** memory. Generally this category will fall under secondary memory, but I will make mention of it this section on primary memory, because there is a particular variety of flash memory that is byte addressable and is gaining popularity in embedded systems.

Unlike RAM, flash memory offers **non-volatile storage**. Flash, in a way, is the evolution of EPROMs. There are two main types of flash memory – **NOR-Flash** and **NAND-Flash**. **The difference between NOR and NAND flash is that NOR is byte accessible** although it takes longer to write to NOR. This characteristic makes NOR flash useful for storing application code, since the CPU will only fetch (read) code and not have to change it (write to that location). When used in this fashion, the code is said to be **executed-in-place** or **XIP**, in that it does not have to be loaded into RAM before the CPU can fetch instructions from it.

M-Systems have come up with a combination of NOR and NAND such that a boot section appears like NOR and the rest of the flash appears like NAND. This storage module is referred to as **Disk-On-Chip**.

Fig 1.3.1 shows the physical layout of primary memory. The memory bus is made up of "**Address lines**", "**Data lines**" and "**Control lines**". When the CPU wants to access memory at a particular address, it puts the address value on the address bus and tells the Memory module that it is interested in the data at that

location with the help of the control lines. In response to the request by the CPU, the memory module will set the Data lines to reflect the value of the data at that address location and use the control lines to indicate that it has completed the operation. At this point, the CPU will read the Data lines and get the information at the address location in question. This process is referred to as a "**Fetch cycle**".

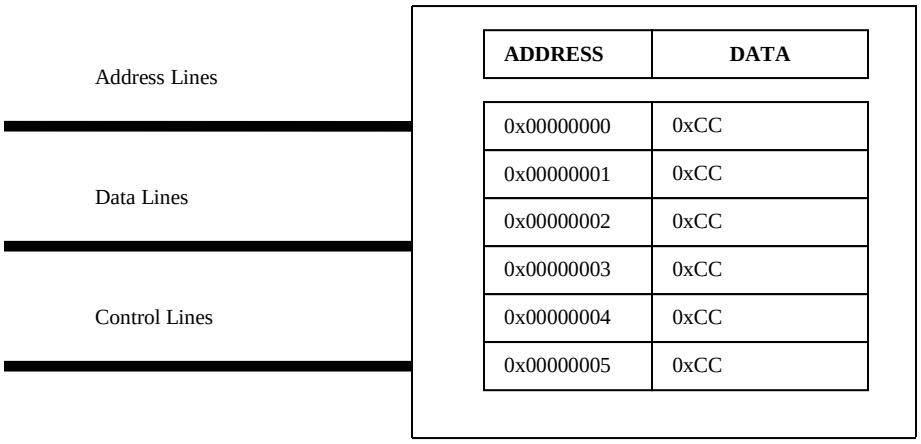


**Fig 1.3.1: Physical Memory Layout**

The number of Address lines will depend on the system architecture. In Fig 1.3.1, I have used 32 bit addresses and so this reflects a 32-bit architecture. Most systems will employ various optimization techniques to read more then 1 byte of data in a fetch cycle.

# *1.4 – Secondary Memory Subsystem*

The Secondary Memory subsystem covers all storage devices hanging off the I/O hub. This category of storage devices almost exclusively represents **non-volatile** memory.

The most common form of secondary storage in a general purpose computer is a **Hard Disk** that is tied to the **ATA bus** (also historically known as the **IDE** bus). Until recently, one would be forgiven for assuming that the hard disk was the sole secondary storage mechanism on a general purpose computer.

The popularity of **Flash memory** and the increased storage capacity of removable storage media like the **USB Mass Storage**, have introduced more players into the secondary memory subsystem.

From a technological perspective the main difference between the hard disk and flash memory is that the **hard disk relies on magnetic media and a revolving head** (mechanical moving parts) which makes it unsuitable for certain environments. **Flash memory on the other hand, is based on solid-state electronics and hence does not have any moving parts**.

Data on a hard disk is stored on magnetic **tracks** in a cylindrical platter. Corresponding tracks on each platter make up a **cylinder**. The tracks are divided into **sectors which represent the smallest addressable data block on the disk**. A **sector** by default represents **512 bytes**, although this can be modified. Each face of the cylindrical magnetic platter has a **head**. A sector is addressed by the **cylinder**, **head** and **sector**. There is often a **logical sector** number that is a sequential numbering system from 0 to n, where 'n' represent the total number of sectors on the disk. For each logical sector, there will be a physical sector number defined by the cylinder, head and sector.

If RAM is volatile and Hard Disk is non-volatile, one may ask why we don't replace the RAM with a Hard Disk. There are two main reasons why this is not possible – **speed and byte addressability**.

Generally speaking, accessing a Hard Disk is about 10,000 times slower than accessing RAM. Note that from a CPU perspective, even accessing the RAM can be a bottleneck. This is the reason why the CPU maintains the L1 and L2 caches. Waiting for the hard disk can be prohibitively slow for the CPU. Generally applications will save data in RAM and periodically flush the data in "bursts" onto the hard disk.

The second reason why a hard disk will not replace RAM is because, as we have already discussed, the smallest block of data that can be addressed by a hard disk is a sector of 512 bytes. The CPU would require access to individual bytes or words of data.

A hard disk always maintains a **Table of Contents** at a fixed location. The location and format of this Table of contents will be a function of the file system that the hard disk supports. But generally the format of Table of contents will specify the different sectors where a file resides. Note that a file does not have to occupy contiguous sectors (and seldom does). As the file grows on the hard disk, it will use up sectors in different regions of the hard disk.

Fig 1.4.1 shows an example of a how a table of contents can keep track of the sectors where a file resides. In this example there are 8 entries in the Table of Contents.

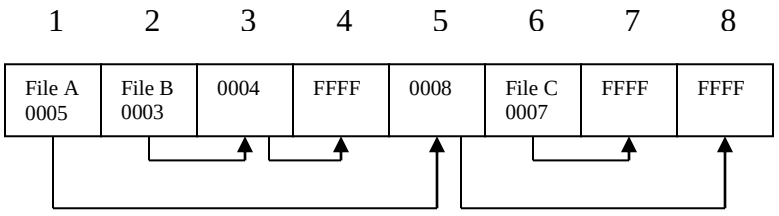


**Fig 1.4.1: Hard Disk Table of Contents**

First we created “File A” and put it in sector 1.

Then we created File B and put it in sectors 2, 3 and 4. Note the “FFFF” marker in the forth Table of Contents entry indicates that it is the last sector for File B.

In the meantime, “File A” outgrew sector 1 and next free sector was sector 5. We put a pointer to the 5th Table of Contents entry in the 1st Table of Contents entry to indicate that the balance of “File A” is now in sector 5.

Next we create “File C” that takes up sectors 6 and 7.

“File A” outgrows sector 5 and we extend it to the next free sector, which happens to be sector 8.

Note that the sector entries in the Table of Contents refer to “logical” sectors.

When a file is spread over non-contiguous sectors like ‘File A’ in the above example, it is said to be **fragmented**.

Over extended use, a hard disk can get so fragmented that accessing any single file will require access to sectors in non-contiguous locations on the disk. This can impact performance as well as the life of the disk because it requires a lot of mechanical movement of the head on the disk.

To circumvent fragmentation, computer users periodically run a utility to **de-fragment** their hard disk. A de-fragmentation utility will try and collate sectors such that files occupy contiguous sectors.

# *1.5 – Networking Subsystem*

With the advent of the internet, the ability of a computer to talk to other computers around the world has become just as important as its ability to load instructions from memory and execute them. The networking subsystem is responsible for interfacing a computer to the outside world.

A group of computers that are able to talk to each other form a computer **network**. Each computer in a network is also referred to as a **node** on that network.

Most computers come equipped with a **Network Interface Card** (**NIC**) that hangs off the PCI bus on the motherboard.

A NIC is responsible for accepting data from applications, formatting it and transmitting this data on a network medium. This out-going direction of data transfer is often referred to as the **Transmit Path** (**Tx Path**) in networking terminology.

A NIC is also responsible for receiving data from a network media and forwarding it to applications that are interested in this data. This in-coming direction of data transfer is referred to as the **Receive Path** (**Rx Path**).

**Networking media** represent the physical connection between computers. A **wired** networking medium implies some sort of an electrical wire connecting computers. A **wireless** networking medium represents computers connected by **radio** devices.

The data from and to an application is referred to as the "**pay-load**" because this is the information that the user is willing to pay for in terms of transmission costs. On the Tx path, the NIC will pad additional data to the pay-load to account for control information overhead required for transmission of the pay-load. On the Rx path, the NIC will strip additional overhead data, before forwarding the pay-load to the applications.

Various **Media Access Control (MAC) Protocols** define when and how data should be formatted and transmitted on a particular media. In general data is broken into small packets called "**datagrams"** and each datagram is addressed to a particular destination before transmission.

The most prevalent wired MAC protocol is **Ethernet**. This is a network where every node on the network is free to transmit data at anytime. However the transmitting node should also confirm that its transmission did not **collide** with the transmission from another node at the same time. If such a collision is detected, the transmitting node waits a certain amount of time and tries again. The amount of time that a node waits is random to reduce the chances of multiple collisions.

Fig 1.5.1 shows two computer networks connected together by a device known as a **router.** The router bridges the two networks so that computers on one of the networks can communicate with computers on the other network as if they were on the same network.

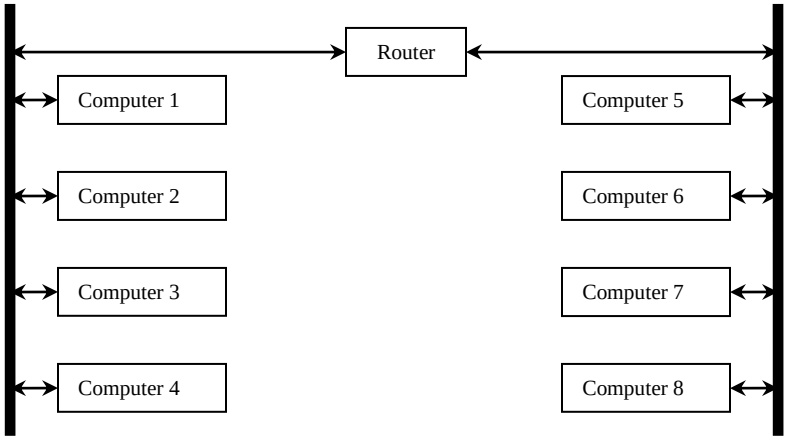


**Fig 1.5.1: Two Computer Networks connected by a Router**

Computers 1 through 4 are in one network and Computers 5 through 8 are on another network.

The **internet** is actually several smaller networks all over the world that are interconnected together in a similar way, such that any computer is able to talk to another computer.

# 2.0 – Operating System (OS) Fundamentals

In the previous section we studied the various hardware subsystems in a computer. Ultimately, these varied subsystems aid in storing, loading and executing software applications and in communication between other computers.

As you can imagine, even the most rudimentary tasks such as storing an application in a hard disk without over-writing other applications that are already in the hard disk and then asking the CPU to load an application from a particular location on the hard disk into RAM can be extremely cumbersome and error prone. For example you would probably prefer to identify your application by a name as opposed to the starting sector of its location in the hard disk. The starting address may change if you move your application from one hard disk into another. These sorts of difficulties beg for a management utility that allows a level of translation between what you wish to do and how it needs to be done. A **Computer Operating system** is effectively such a manager.

In one sense, the Operating System is just another application which has the responsibility to ensure that all other applications have regulated and user friendly access to components on the motherboard. In this section, we will very briefly discuss some of the components and functions of an operating system. We will defer the more intricate details of an operating system to the ALS module on the Real Time Operating System.

# *2.1 – The OS Kernel*

The **Kernel** is the core of an Operation System. In this section we will walk through some of the common tasks performed by a kernel. It must be emphasized that there are several nuances that distinguish the kernels of different operating systems and that the discussion below is of a generic nature to give the student an appreciation for the considerations involved in kernel design and an introduction to terminologies.

When a general purpose computer is powered up, the CPU is usually hard-wired to load instructions at a particular memory address, also referred to as the **Reset Vector**. Usually this address corresponds to some form of non-volatile Read-Only-Memory (ROM). The motherboard manufacturer would have placed a special program known as a **Boot-Loader** at this ROM location.

The boot loader is responsible for locating the kernel components of the Operating System in the Secondary Storage (usually the Hard Disk or Flash memory) and loading it into the Primary memory (RAM). Once the kernel is loaded into memory, the boot loader asks the kernel to continue with system initialization. From this point onwards, the kernel is the manager of the computer system. Different kernels will operate in different ways, but the general responsibilities of a kernel are much the same.

One of the first things the kernel has to do is to load and initialize the display, keyboard and mouse **Drivers**. A driver is a specialized piece of software that is designed to interface with peripheral hardware. The display, keyboard and mouse hardware are tied to one of the two hubs on the computer motherboard. Once the kernel initializes these peripherals, it is able to interact with the user.

The next thing the kernel usually does is to load a "**Device Manager**" that is responsible for identifying all the peripheral devices and buses on the computer system. The kernel then allows the device manager to use the CPU to execute its instructions.

The device manager will then load other operating system components based on the particular hardware peripherals that are available on a given motherboard. For example, file storage and communications are some of the most common tasks for a computer. Most motherboards will have dedicated hardware peripherals for these purposes. The device manager will detect these peripherals and load the corresponding operating system modules that are responsible for managing these peripherals.

In the case of the various buses on the motherboard, the device manager will load the corresponding "**Bus Drivers**". The bus drivers are responsible for **enumerating** (identifying who is on the bus) the devices on their respective buses and loading corresponding operating system components for each of the devices that is on the bus. In effect, the bus driver becomes the device manager for devices on its bus.

Once the device manager has loaded all the components necessary for a particular computer system, the kernel then allows each of the loaded operating system components to use the CPU. The process where the kernel allows a component to use the CPU is referred to as **scheduling**.

Finally the kernel loads the **Shell** and schedules it to run (in other words, allows it to use the CPU). The shell is the **User Interface** to the computer system. It is the face of the computer. Think of the Shell as a special operating system component that is responsible for interacting with the end user.

A software component is usually a set of tasks. For example, the shell is a software component that is responsible for interacting with the user. At a minimum it will have two tasks – the first to accept input from the kernel and relay it to the user and the second will be to accept input from the user and relay it to the kernel. A kernel that allows multiple tasks to co-exist is referred to as a **multitasking kernel**.

All the software components that are loaded need to access the CPU periodically. The Kernel usually gives each component a certain amount of time of CPU use and then gives the CPU to another component. The process where the kernel forces a task to relinquish the use of the CPU is referred to as **preemption** and a kernel that operates in this fashion is called a **preemptive kernel**.

A multitasking kernel is responsible for keeping a list of all the **tasks** that are running on the system and **scheduling** each task as and when required. Some operating systems refer to these tasks as **processes** or **threads**.

Another responsibility of the kernel is to manage the primary memory. Usually the kernel delegates this to a component called the **memory manager** that is responsible for allocating and freeing memory that is required by each task. Most memory managers refer to memory using "**virtual**" addresses. These addresses map to physical memory address based on a table known as a "**Page Table**". The use of virtual addresses allows the memory manager to assign more memory to tasks than is physically available in primary memory. It also allows the using of contiguous virtual addresses even when the physical locations may not be contiguous. When the memory manager detects that it has run out of physical memory, it will copy some of the least frequently accessed physical pages into secondary memory and then re-assign those physical locations to virtual addresses of tasks that are in immediate need of primary memory. This process by which the memory manager saves the contents of physical memory into secondary memory is referred to as "**Paging-out**" memory. The inverse operation where the memory manager copies contents from secondary memory to primary memory is referred to as "**Paging-in**" memory. The files in secondary memory used by the memory manager for this purpose are called the **Page Files**.

Yet another important task performed by the kernel is **Interrupt Servicing**. When a CPU is interrupted by the interrupt controller, the CPU will load instructions from a predefined memory location known as the **Interrupt Vector**. The Kernel usually loads the instructions at these interrupt vector locations and hence controls what happens when a peripheral signals an interrupt.

To summarize, one can think of the Kernel as an operating system component that arbitrates access to the CPU and the primary memory on the motherboard such that all other software components can concentrate on their specific tasks without stepping over other tasks that also require access to the CPU and primary memory.

# *2.2 – The OS File System*

Most operating system components are designed in a hierarchical architecture. The layers at the bottom of the hierarchy are responsible for interacting with the hardware while the layers at the top are responsible for providing a user friendly interface to access the layers at the bottom. The file system architecture in most operating systems will follow such a scheme.

Fig 2.2.1 shows a generic file system architecture. Note that the file system is generally associated with secondary (non-volatile) memory. As discussed previously, most of these types of memory are block addressable. Hence, the lowest layer in such a file system is a **Block Device Driver**. Such a driver is responsible for reading from and writing to individual sectors on a hard-disk or to blocks in a flash memory device. The driver is not responsible for keeping track of all the sectors used by a particular file. That information is maintained at higher layers. The block driver allows higher layers to store and retrieve information at a certain location in the secondary storage device, remaining mostly oblivious to the greater relevance of the information being stored or retrieved.

Most block devices accommodate the notion of **Partitions** within a block device. Partitions are mechanisms to divide a large block device into smaller, more manageable sections. These sections may be used for specific purposes. The Partition Manager layer is responsible for providing a layer of abstraction between higher layers that would refer to data in a particular partition and the block driver that does not necessarily view the block device as divided into partitions.

Within each partition, data will be stored in a particular format. This format dictates the type of **File System Driver** that is used to manage the data in that partition. One of the most common file systems is the **File Allocation Table** (**FAT**) file system. It is at the file system driver layer that the file names and locations of each file's contents are maintained.

The topmost layer is the **File Manager**. This layer is a presentation layer. It allows users to view all the available files on the computer without distinguishing the files based on the file system, partition or block device that is used to store the contents of a file.

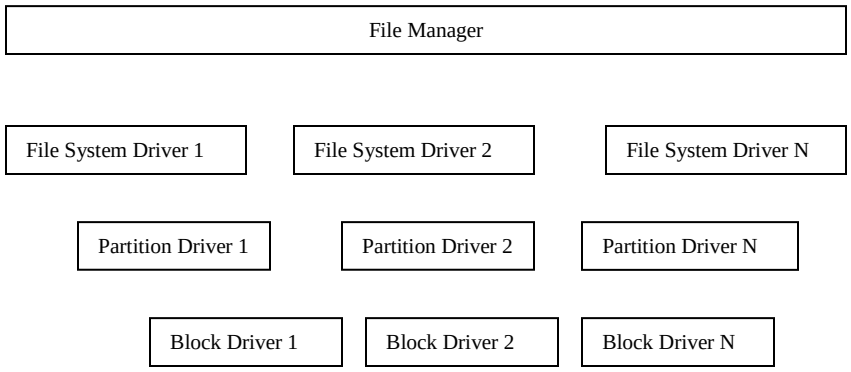


**Fig 2.2.1: File System Hierarchy**

# *2.3 – The OS Networking Subsystem*

The Networking subsystem is perhaps one of the most hierarchical and standardized components of any operating system. The most compelling incentive for this is that to be able to communicate with other computers, there must be an agreement among operating system vendors as to the format of the **control** and **data packets** that will be exchanged between computers. These agreements are often referred to as **Communication Protocols** and are published standards that are available to all operating system vendors. The networking subsystem in a general purpose operating system is almost always designed with the intension of accommodating multiple communication protocols.

The **Open System Interconnect** (**OSI**) **model** is a recommendation by the International Organization for Standardization (ISO) on how to architect the layers in a networking subsystem. It discusses a 7-layer model as shown in Fig 2.3.1.

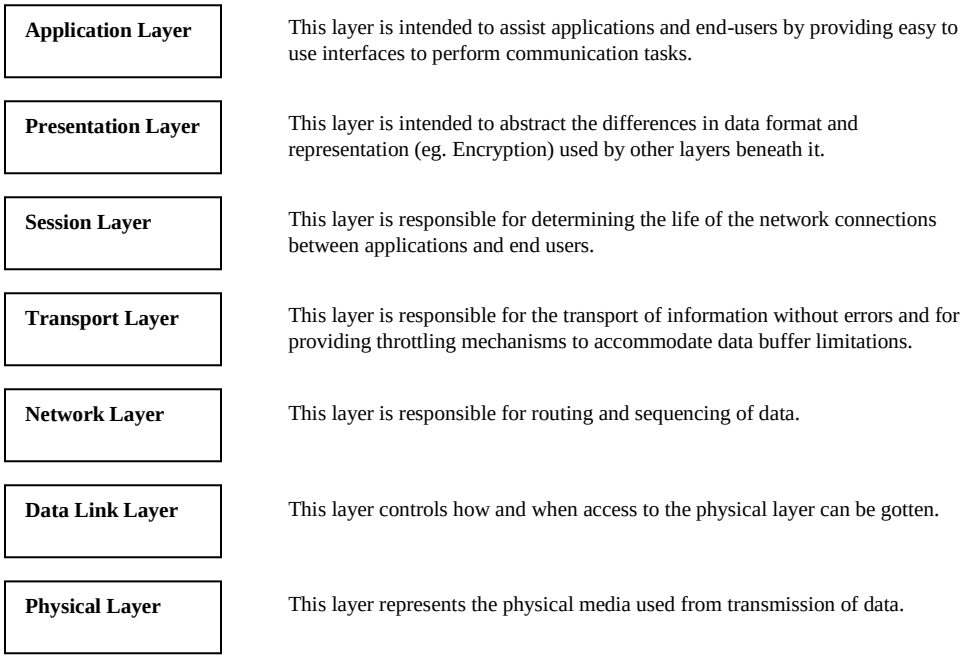


**Fig 2.3.1: 7-Layer OSI Model**

Each of the layers in the OSI model is governed by the various communication protocols available at that layer. For example the most common Data Link Layer protocol in use today is the **Ethernet** protocol. The most common Network layer protocol is the **Internet Protocol** (**IP**) and the most common Transport layer protocol is the **Transport Control Protocol** (**TCP**).

The layers above the Transport layer are less rigidly adhered to by operating system vendors. This is partly because the OSI model was introduced long after the TCP/IP protocol became ubiquitous and already existing installations never re-architected their networking subsystem to conform to the OSI model.

The general thrust of the 7-layer model is that changes at the lower layers do not have to impact end users and applications dependent on them, because end users and applications don't talk to them directly.

# 3.0 – Conclusion

In this set of notes on the General Purpose Computer Architecture, the student has been introduced to basic general purpose hardware and operating system, concepts and terminologies. This will serve as an adequate prerequisite for future modules on firmware and software engineering.