Chapter 36 - Operating systems in detail

36.1 Features of Operating Systems

In an earlier chapter, we saw that an operating system is a piece of software that looks after the management of a CPU-based system. The actual facilities provided by an operating system will very much depend upon the design and type of facilities needed by a particular set-up. The needs of the following systems will all vary.

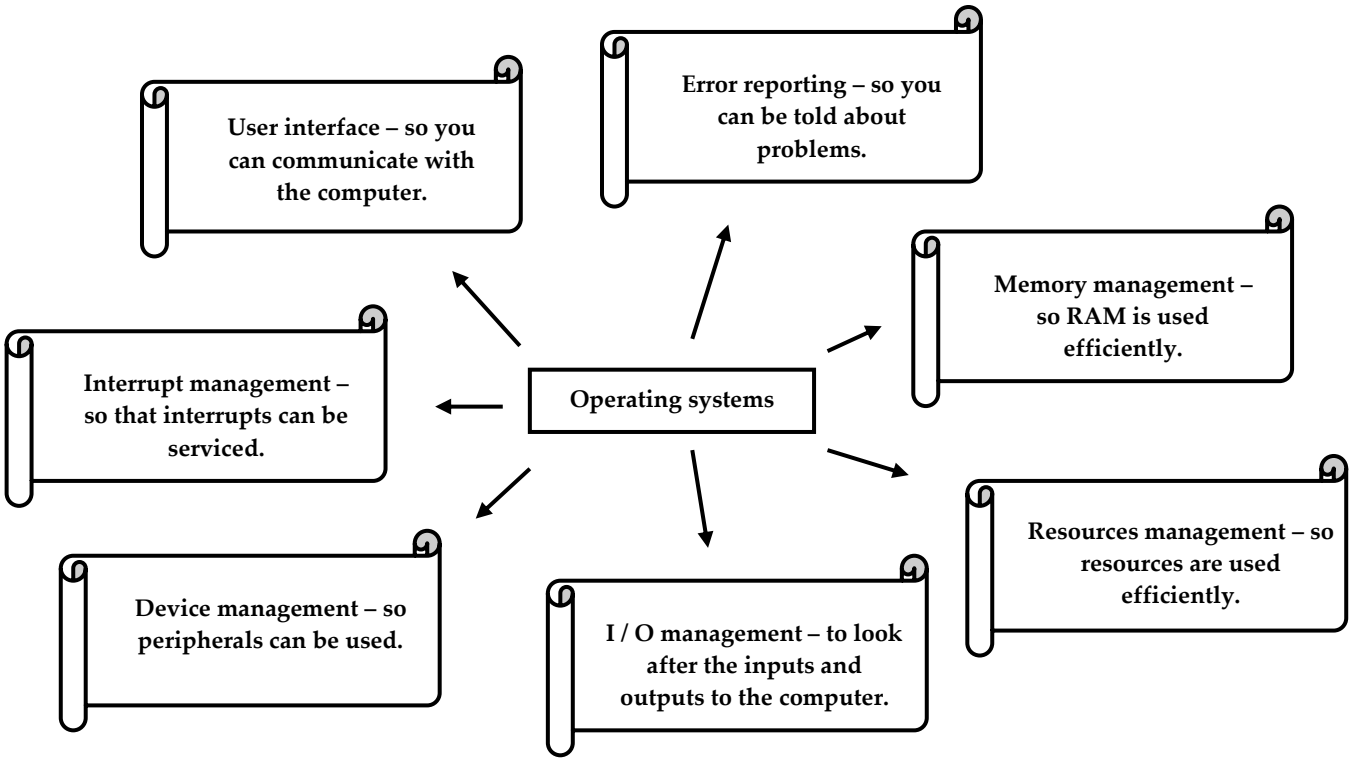
- A batch processing system such as a bank-statement producing system.
- A real-time system such as a system controlling a plane.
- A personal computer.
- An embedded system as found in, for example, a microwave or dishwasher.
- A network with many users, as found in many schools.

For example, a real-time operating system may not have any need for memory management routines because the program and data to run a washing machine are held in ROM. There is no need to transfer data between RAM and secondary storage.

36.2 The features of personal computer operating systems

A PC operating system is a collection of programs and utilities. These manage the computer's hardware and software. The operating system looks after the basic functions of the computer. A computer is made up of a number of different components that have been connected together. Once connected to make a 'computer', however, the components don't start working together until an operating system takes control.

The operating system, then, is a piece of software that manages the basic functions of the computer. The jobs done by an operating system can be summarised using the following diagram.



The basic functions of an operating system.

36.3 The basic functions of an operating system

- Making sure all the parts of the computer can work together.
- Making sure that the user can interact with the computer by providing an interface.
- Ensuring that any errors in the computer are reported to the user, usually with messages on the VDU.

- Managing files on the hard drive and in memory and allowing the user to manage files by providing a file management utility program.
- Making sure that data can be entered into the computer, displayed or saved by providing software that allows data input and output.
- Providing utility programs such as FORMAT DISK, FILE MANAGEMENT or DEFRAG.
- 'Scheduling'. The operating system provides programs that ensure the resources of the computer such as CPU time, are used as efficiently as possible by all the different users and the different jobs.

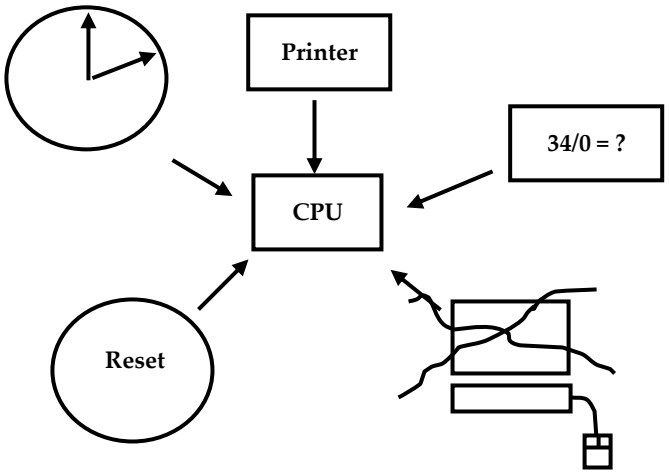
36.4 Applications of interrupts

Interrupts are signals generated by a hardware device or by a software application that tell the CPU it needs some CPU time. A CPU can only ever do one job at a time. If it is working on one job and another job needs some CPU processing time then that other job sends a signal, an interrupt, to the processor. When the processor receives the signal, it does one of a number of things.

- The CPU might stop what it is doing and start servicing the device or software that sent the signal.
- The CPU might carry on doing its current job until it is finished and then service the device or software that sent the signal.
- The CPU might carry on doing its current job and then service other interrupts that have happened. When it has done those, it then services the interrupt in our discussion!

To summarise, an interrupt is a signal sent to the CPU by a device or by a program that indicates to the CPU that a device or program needs some 'CPU time'. The CPU must stop what it is doing and run a program, a piece of software, for that particular type of interrupt, known as an **interrupt handling routine**. The first decision the CPU must make is when to run the interrupt handling routine - how urgent is it compared to what it is currently doing and compared to what else is waiting to be done!

36.4.1 Different kinds of interrupts



There are different kinds of interrupts.

1) **Timer interrupt.** This is an interrupt that occurs at fixed time intervals. There are circuits in a computer that can generate an interrupt signal every 1 ms, for example. This could then be used to signal to the CPU to refresh the VDU, for example.
2) **Program error interrupt.** When a program is written, it may need to do maths sums! If you divide any number by zero, the result is meaningless and the program you are running will get confused! The CPU needs to know about this so a 'program error' interrupt signal is sent to the CPU to tell it what has happened. The CPU can then take action, for example, by ending the program and reporting the error.
3) **Hardware error interrupt.** Your computer is made up of a lot of different hardware devices. Generally, they work well and do what they are supposed to do. But like any electrical or mechanical device they will one day fail. When that happens, the CPU needs to know about it. A device that has failed sends an interrupt signal to the CPU. The CPU can then, for example, put a message on the screen to notify the user of the problem.
4) **I/O interrupt.** When a file is sent to a printer, it will go to a buffer. The buffer will then communicate with the printer. The reason for this is that it frees up the CPU to do other jobs. Remember, the CPU can only do one job at a time. If it is managing the printing of a file, it can't be doing other things. This, however, isn't a very efficient use of CPU time because printing is slow compared to the speed CPUs work at. We have discussed this in great detail in a previous chapter.

Sometimes, the file that you want to print can't fit into the whole of the buffer. You can remind yourself about the transfer of data using buffers in another chapter. Another example of an I/O interrupt is the one generated every time you press a key on the keyboard. When you press a key, an interrupt is sent to the CPU. The CPU stops what it is doing and then manages the reading in of the key code into a buffer. Then it goes back to doing what it was doing before. At some point, the CPU will return to the keyboard buffer and retrieve the contents for use somewhere.

36.4.2 How does the Interrupt process work?

The CPU has a special register called the 'interrupt register'. You can imagine this, for example, as a two-byte number (in other words, 16 bits). When a particular interrupt happens, the signal sent causes the correct bit in the interrupt register to be made a 'one'. Here is an example.

1) This is the interrupt register: 0000 0000 0000 0000
2) The DVD drive fails, so a 'hardware failure' interrupt signal is sent to the interrupt register.
3) The bit that is used to signal hardware failure is bit 3 so that bit is set to one. (The bit on the right-most side, the least significant bit, is bit zero.)
4) The interrupt register now looks like this: 0000 0000 0000 1000

36.4.3 So what happens next?

- At the beginning of each FDE cycle, each bit in the interrupt register is checked in turn.
- If a bit has been set, that would indicate an interrupt has happened.
- The CPU decides whether to service the interrupt immediately, or leave it till later. For example, if 2 interrupts have happened at the same time, one of them has to wait! Which one? That depends upon which one is the least important! Some interrupts are more important than others and so need to be done before others. What about the situation where one interrupt is currently being serviced by the CPU and another happens? Again, it depends on how important the new interrupt is compared to the one already being done. If it is more important, then the CPU will want to service it immediately.
- When the CPU decides to service an interrupt, it stops processing the current job, 'pushing' the contents of its registers onto the stack. This would include, for example, the contents of the Program Counter and the Accumulator. The CPU is now free to work on another piece of software but can return to what it was doing after the interrupt has been serviced because it has saved where it was.
- It then transfers control to the interrupt handling software for that type of interrupt. You can read more about how it does that (using the Vectored Interrupt Mechanism) later in this chapter.
- When it has finished servicing the interrupt, the contents of the stack are 'popped' back into the appropriate registers and the CPU continues from where it left off before the interrupt happened.

36.5 More on the prioritisation of interrupts

A computer needs to make a decision! It has to decide what priority to give each of the different kinds of interrupts, in case it has to service more than one kind of interrupt simultaneously. One possible order is shown below. I have decided that the most important kind of interrupt is the hardware failure interrupt. If anything fails, I want the computer to stop what it is doing immediately and service it - attempt to sort out the problem and also tell me about it. I have equally decided that I/O requests are the least important interrupts and can be serviced last if a choice between which interrupt to service has to be made.

- Hardware failure: Priority 1 (most important)
- Reset interrupt: Priority 2
- Program error: Priority 3
- Timer: Priority 4
- I/O: Priority 5

36.5.1 The Vectored Interrupt Mechanism

When an interrupt happens, the CPU needs to jump to the relevant software routine. How does it know where it is? It can do this by using a technique known as 'indirect addressing'. This is fully explained in the chapter on low-level languages but briefly, when an interrupt happens, the CPU identifies what kind of interrupt it is. With every type of interrupt, there is an associated memory address, known as a **vector**. This memory address, or vector, holds the address of the relevant interrupt handling routine. So when an interrupt happens, the CPU jumps to the vector, gets the address of the start of the interrupt handling routine, loads it into the Program Counter and processing continues. For example:

- The CPU is working on a word processing application. The interrupt register is 0000 0000 0000 0000.

- A software error occurs that causes a program error interrupt to occur.
- Bit 6, used for program error interrupts, is set. The interrupt register is now 0000 0000 0100 0000
- At the start of the FDE cycle, the interrupt register is checked and the CPU sees that an error has occurred.
- It isn't currently servicing any other interrupt and no other higher priority interrupt has occurred so it decides to service this routine.
- It pushes the contents of the CPU's registers onto the stack.
- It jumps to the address (known as a vector) for that type of interrupt.
- At that vector address, it finds another address!
- It loads that address into the Program Counter.
- The interrupt routine completes.
- The contents of the stack are popped into the CPU's registers.
- The CPU continues doing whatever it was doing before the interrupt happened.

36.6 Resource allocation, multiprogramming and scheduling

CPUs can carry out instructions very quickly although it is important to recognise that they can only ever work on one task, or process, at a time. Consider a multi-user network where there are many users on 'dumb' workstations with one central file server (which has one CPU), or consider a single-user system (such as a home PC) where the user is multi-tasking. Each different job or application in use will regularly need some CPU time. But how is the CPU time to be organised, given that it can only do one thing at a time? The organisation of CPU time is another responsibility of the operating system and more specifically, a piece of software within the operating system known as the 'scheduler'. A computer that is capable of working on more than one application at apparently the same time is also known as a **multiprogramming environment**.

36.6.1 Scheduling

Scheduling is the name given to the process of managing CPU time. The aims of scheduling are to:

- make sure the CPU is working as hard and as efficiently as possible
- make sure the resources such as printing are used as efficiently as possible
- ensure that users on a network think they are the only ones on the system.

36.6.2 How does scheduling work?

A system will have many jobs (which we should more properly call '**processes**') that need CPU time. For example, in a multi-user environment, one user may need to print a document, another may need to read a file of data, another may need to do a series of calculations, and so on. Each job (or '**process**') needs some CPU time. In addition, each **process** may require some input or output from a peripheral such as a printer. Processes that require lots of processor time and very little input/output are known as '**processor-bound**' jobs. This might be a process, for example, that requires lots of calculations. A process that requires lots of input/output but very little actual CPU processing is known as an '**I/O-bound'** job. An example would be a process that needs to print a document or needs to read a lot of data from a file. Each process can be classified as a processor-bound job or an I/O-bound job and this is important because the scheduler looks at what resources are available when deciding what process will get CPU time.

The scheduler software, then, is responsible for looking at what resources are available (CPU time and peripheral devices), seeing what processes need to be serviced along with what resources they need and then making decisions about what order to put all the processes in, when to start any particular process, and when to finish it. The scheduler software can be broken down into three parts, the **high-level**, **medium-level** and the **low-level** scheduler.

36.6.3 The high-level scheduler (HLS)

The high-level scheduler is responsible for organising all of the processes that need servicing into an order. As each job enters the system, the high-level scheduler places it in the right place in a queue known as the **READY TO PROCESS** queue. The actual order is based on a 'scheduling algorithm' – in other words, some rules about how to order the processes. Some different scheduling algorithms are discussed later in this section. **So the high-level scheduler puts new processes in the READY TO PROCESS queue into the primary memory if it can.**

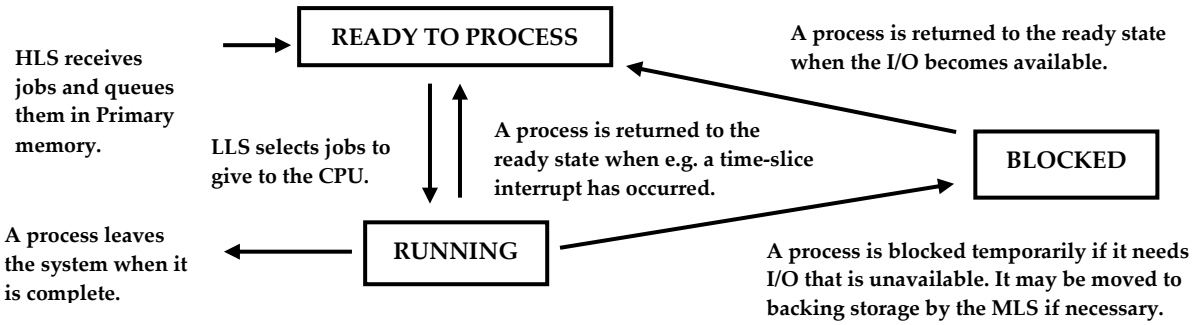
36.6.4 The medium-level scheduler (MLS)

If a process is in the **RUNNING** state (it is in the middle of being serviced by the CPU) and it suddenly needs some peripheral device usage, it is taken out of the **RUNNING** state and put into the **BLOCKED** queue. If that process stayed in the running state while waiting for I/O, the CPU would be sitting around wasting time, which isn't an efficient use of the CPU! A process put into the blocked state can be moved out of the IAS and into backing storage if necessary, to free up memory. When the

resources are available to service a process held in the backing store, it can be moved back into the IAS. **It is the job of the medium-level scheduler to move a process into and out of IAS as necessary.**

36.6.5 The low-level scheduler (LLS)

Once the processes are in the IAS, the low-level scheduler takes control. This piece of software is responsible for actually selecting and running a process. It looks at the order the high-level scheduler has put the processes in by looking at the **READY TO PROCESS** queue. It then checks that any necessary resources are available for the most important process in the queue and then runs it using the CPU - it takes the process from the **READY TO PROCESS** queue and puts it into the **RUNNING** state. It will stay in the running state until either an interrupt happens, that process finishes or the process needs resources that are unavailable, for example, a printer. **The low-level scheduler will select the next most appropriate process to give to the CPU.** You can summarise what happens with the following diagram.

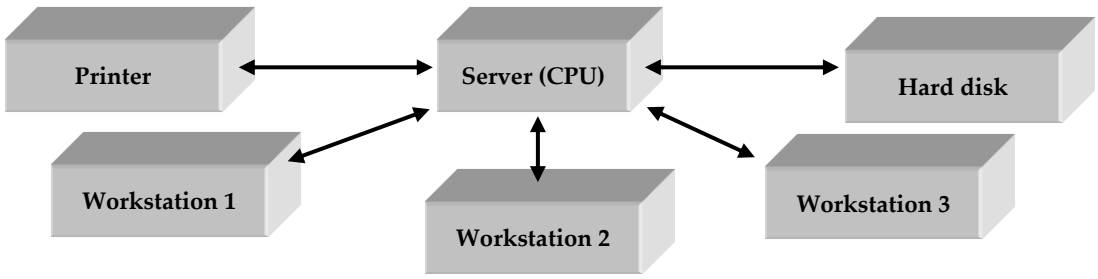


MLS swaps jobs into and out of the primary memory as necessary.

There are many algorithms that the scheduler can use to put the processes that need servicing into an order. Two of the most common ones are **Round Robin** (also known as **time-slicing**) and **Priorities**. These are discussed next.

36.7 A Typical problem

Consider a small network with some workstations and some resources. The network has a server. The server has a CPU. This has to service a number of workstations as well as all of the peripherals. We can represent this with the following diagram.



Scheduling ensures the CPU is working as hard as possible.

Consider the above example. Someone working at the first computer is typing in a letter. Even someone typing in at 100 words a minute is only working at a tiny percentage of the processor's speed. Suppose Workstation 2 needs to read a file from the hard drive. Again, this takes a long time compared to the operating speed of the processor. Perhaps workstation 3 is sending an email and requires some I/O to the modem. Even though there are three users on the system they all think that they have sole access to the network because that is what it appears as to them. But they haven't got sole use of the CPU and the associated resources. It's just that the system is **scheduling** the workload efficiently.

36.7.1 The Round Robin (time-slicing) system

The scheduler looks at the different programs from the different workstations. It puts them all in a queue. It then allocates each item in the queue a slice of CPU time. For example, it might allocate 1/100th second to each item. It then creates a **time-initiated interrupt** set for this time interval. Every 1/100th of a second, an interrupt is sent to the processor that tells the low-level scheduler to transfer control from the current program to the next program in the queue. The current program is sent to the back of the queue. Of course, at any time, another interrupt may happen that needs servicing or the process may finish.

From the discussions so far, you should know that a CPU would potentially be executing lots of different programs at 'the same time'. It will try to hold as much of all the programs as possible in primary memory but may need to use virtual memory if enough RAM isn't available. If two users are using the same program, for example, Word 2000, it will only keep one copy in RAM and simply swap information about each job that needs the program into and out of itself as each user is given a CPU time-slice.

36.7.2 The priorities system

Time-slicing isn't the only way that processes can be dealt with by the CPU. Processes can be given **priorities** and the scheduler can organise processes according to those priorities. For example, some jobs may be submitted in batch mode. These might be run at a time when the resources aren't being used such as at night. They could, however, be submitted in a multi-user environment at any time. In this case, batch jobs would be given a low priority, to be run in the background using the CPU and resources when they aren't being used to do more important jobs. Supervisory programs would be given the highest priority. It is also possible for users to be given priorities, so that one person's work is more important than another person's.

36.7.3 Other algorithms

There are a number of other algorithms that a scheduler might use. For example, jobs might be organised on a first-come-first-served basis. They could be organised on a 'Do the job that will be finished the quickest' basis.

36.8 Scheduling processes and the Process Control Block

When the CPU stops working on the current process, ready to start working on another program, it needs to record exactly where it is in the current process. This will enable it to pick up from where it left off next time it works on that process. How does it do that? Each program that could run (known as a **process**, as we've already discussed) is allocated a block of memory called a **process control block**. This will hold the following information:

- Process Identity Number.
- Current state of the job when the job was last left.
- The contents of each register when the job was last left.
- What priority the process has.
- Estimated time for the job to be finished.
- Its current status, whether it is waiting for I/O, whether it's waiting to use the CPU.
- Pointers pointing to areas in memory reserved for this process and resources that could be used.

When a process is stopped, the above details are loaded into the Process Control Block, ready for next time the process uses CPU time. When that time comes, the contents of the PCB are moved into the CPU and the process repeats itself!

36.9 Memory management

Your computer has some Immediate Access Store (known also as IAS but you may also know it as RAM – the terms are used interchangeably). This is a fixed size although you can of course go out and buy some more! For example, you may have 4 Gbytes of RAM and you could upgrade this to 8 Gbytes. IAS is used to store the programs and data that you *currently* need to use (as opposed to secondary storage devices such as the hard disk or CD-ROM, which are used to store applications and data that you may use in the future but don't need now).

When you want to use some applications or want to work on a file, you typically double-click on it. It is then transferred by the operating system from a secondary storage device into the IAS. The operating system has a piece of software called a '**loader**' that is responsible for:

- Transferring an application or file into an appropriate place in the primary memory.
- Adjusting any references to RAM addresses that have been made in a program. For example, if you have an instruction that says "Get data from RAM location 1000 000" and the operating system has used that location for another program instead of a data file, then the loader will have to adjust the reference so it is looking at a different (but correct) RAM location.

If you are multi-tasking, however, then you may have a number of applications and a number of files open (i.e. in the IAS) at the same time. There is a potential problem here for the loader. These applications and files may interfere with each other and the loader must ensure that they don't! They all need to be in RAM but each one is in danger of overwriting the RAM locations of other applications and files! For example, consider the following. A computer with 32 Mbytes of RAM has an operating system and a virus checker running (in RAM). Let us assume that the operating system is in memory locations 0 Mbytes up to address 20 Million. The virus checker starts at 20 Mbytes and goes up to address 25 Million. We can represent this as:

Operating system	**Virus checker**	**Free**	
O Mb	**20 Mb**	**25 Mb**	**32 Mb**

If the user then double clicks on some word processing software and opens some files (let's assume that this take 4 Mbytes of RAM), the application and the data must be put somewhere after 25 Mbytes. If they are put, e.g. from address 18 Million onwards then the Virus checker software would be overwritten (and wouldn't therefore work) and some of the operating system would be overwritten (which may cause the computer to crash). Deciding how the RAM is to be used for all of the different applications and data is known as memory management and is an important role of the operating system. Our RAM now looks like this:

Operating system	**Virus checker**	**Word processor**	**Free**	
O Mb	**20 Mb**	**25 Mb**	**29 Mb**	**32 Mb**

Now suppose the user opens another application as well as the word processor. Suppose the user opens a spreadsheet that needs 5 Mbytes. The operating system puts it at the start of the free space, at address 29 Million – but then realises that there isn't enough RAM to fit the whole application in!! You may think that the computer might crash or at the very least produce an error message. However, the computer has a memory management trick up its sleeve. It uses some 'pretend RAM' by 'converting' temporarily some of the unused space available on the hard disk. The proper name for hard disk space being used like RAM is '**virtual memory**'.

How can more than one program be running in memory at the same time, even though there isn't enough primary memory to run both programs? The operating system relies on 'virtual memory'. It relies on the fact that, even if you have 5 applications open at the same time, only one of them is 'active', or being serviced by the CPU, at any one time. In addition, you often don't actually need the entire code for an application in RAM at any one moment. For example, if you are not using some clipart that comes with a word processing application, then the clipart doesn't need to be in RAM, although if you do start using it then it must be moved into RAM.

1) All programs are split up into equal lengths by the operating system. These equal lengths are known as **pages**. A page might be 32 Kbytes long, for example.
2) The RAM is also split up into pages.
3) The computer keeps as many pages as it can in RAM, especially those needed to run the active application. In the above example, if the word processing software was active, then the operating system, the virus checker, all of the word processing application and 3 Mbytes of the spreadsheet software would be kept in RAM.
4) The remaining 2 Mbytes of the spreadsheet software can't fit into RAM and would be stored, in page-size blocks, in virtual memory, on backing storage (the hard disk) in an area called the **page file**. In Windows, this is also known as the **swap file**.
5) When you switch from the word processor to the spreadsheet, so that the spreadsheet is the active application, what happens?
6) The CPU moves out of RAM and on to the hard disk some of the pages that it doesn't need for a while, (in this case some of the word processing pages) and then moves from the backing storage into RAM those spreadsheet pages which it now does need!
7) This takes a little bit of time - the hard disk is very slow compared to the CPU. Have you ever noticed when you are multitasking that it can take a little while (sometimes ages) before you can actually access your computer when you switch between applications?
8) When the pages have been swapped, you can then start using the spreadsheet.
9) If you swap back again, the above process is repeated.

What you are doing here is to use a little part of your hard disk as a sort of (slow) RAM. Because this part of your hard disk is being used like real memory, but isn't actually memory, it is known as 'virtual memory'.

36.9.1 Problems and solutions

Virtual memory allows a user to have and use apparently more RAM than is actually available. There are problems, however.

1) As has already been highlighted, swapping pages in and out of the hard disk takes time! '**disk threshing**' (or thrashing your hard disk) as it is known, causes the computer to work slowly. The more memory you have, however, the more pages you can keep in it and the less swapping you need to do! Ideally, you want to have enough RAM to run applications without using virtual memory. In the example above, we only needed 3 Mbytes of virtual memory, so it would have been unlikely to slow down the computer too much. Imagine if we had 200 Mbytes of applications and only 32 Mbytes of RAM!! You would recommend a user in this situation to go out and buy some more RAM, quickly!
2) You should run a 'defragmentation utility' regularly on your computer's hard disk. As you use a hard disk, files get split up and stored increasingly all over the hard disk. This might eventually mean that pages used in virtual memory can't be stored together. To access pages that are all over a hard disk is slower than accessing pages that are all together. Running Defrag regularly will keep your hard disk working at its optimum performance.

36.9.2 Page tables

The operating system manages the primary memory in a computer and one very important part of memory management is the management of virtual memory. This was described in the previous example. All the operating system has to do is keep track of all of the pages. Each application will have associated with it a page table. This will identify each page for each application and will contain information about where to find that page if it is on the hard disk in virtual memory. If a page is in RAM then the entry for that particular page will be blank. When a page is needed, the operating system looks up the page reference in the page table, finds out where it is in virtual memory and then transfers it from the hard disk to the primary memory.

This, of course, all takes time and contributes to the computer slowing down. This kind of tracking system for pages is relatively simple to implement but there are some disadvantages.

1) Paging is a simple system to implement but may not be the most efficient. This is because the pages are of a **fixed size**. A page may only have a small amount of data in it but will require a whole page to store it in.
2) Pages are fixed sizes. They will contain code that may not be a logical group of code (for example, a logical block of code such as a function might run across 6 different pages). This means that to load that particular function will require 6 pages to be loaded up, which is not as efficient as loading e.g. one page for the whole function.
3) Page tables are typically large and accessing them slows down the computer.

36.9.3 Segmentation

An alternative approach is to use **segmentation**. Instead of pages being a fixed size, they are a variable size. Because their size can vary, code can be logically grouped into one segment. You could have a segment per procedure, for example, and this would reduce the size of the tables needed to store the segment information. The downside to segmentation is that retrieving a segment is more complicated. You need to store both the start address of a segment and the size of the segment. Start addresses have to be calculated with care because the sizes of each segment varies. With paging, it is entirely predictable where each page starts and finishes.

36.10 Spooling

A 'print spooler' is a piece of software that intercepts and stores on a storage device files that have been sent to a printer. It then takes over the management of the transfer of the files from the computer to the printer. The computer (a very fast device) is therefore freed up from the task of printing (a very slow operation) so you don't suffer from the problem of 'speed mismatch', where you have to work at the speed of the slowest device. The print spooler can receive lots of jobs at the same time from lots of different applications.

On a network, you might be waiting a long time while other print jobs are being serviced and the printer becomes free. The print spooler collects and stores on a storage device all the jobs sent to the printer, queues them all up by creating a queue of references to each file that needs to be printed (and prioritises them if necessary) and then sends them to the printer as and when the printer is free. This is known as '**de-spooling**'. Everyone can send their printing jobs at any time from any application and their application will be immediately freed up.

36.10.1 Print management software

The network manager can configure the print spooler software on a network. That means, for example, that she could set up the spooler software to make some people's print jobs more important than others. When they send a print job, their work jumps

straight to the front of the print queue! Sometimes, organisations buy a further piece of software, known as 'print management software' to provide the network manager with even more print management features.

- Printing is costly. It costs a lot of money to buy paper and ink cartridges.
- If one person is printing a huge amount of work that is unfair to people who are waiting, perhaps to print small jobs. If this keeps happening people will get annoyed! Print management software can allow organisations to identify people who are printing huge amounts and to take appropriate action if necessary.
- Organisations need to budget for resources. Print management software helps them to do this by providing information about how much is being printed and by whom and then controlling it.
- Costs can be allocated to printouts so that printing costs can be charged to projects.

With print management software, you could, for example,

- Produce reports showing how many copies were printed by each user on the network.
- Automatically generate charges for individuals who over-print.
- Set limits on how much a person can print in any particular time period.
- Prevent certain people accessing certain (expensive) resources, for example, colour printers.
- Prevent people from printing at all, or allowing them to print only at certain times.

36.11 Main components of a PC operating system

Operating systems are very important to computers! They look after many basic functions that allow people to use computers. There are quite a few operating systems about. Some you will have heard of and some you may not have. Widely used operating systems include Windows, DOS and Linux, for example. One of the first jobs a personal computer has when it is switched on is to get the operating system loaded up into RAM from the hard disk. The operating system can then start managing the computer, the applications and the data files.

36.11.1 Windows

The Windows operating system dominates the PC market. It has an interface made up of a **window** for each open application. You can open many applications at the same time although only one of them will ever be 'active'. In other words, you can only ever work on one of the applications at a time although you can easily switch between them to make a different one active. The Windows operating system also uses drop-down **menus** and pop-up menus. These allow many choices to be presented to users for selection but they take up little space on the screen until the menu is actually selected. Clicking on an **icon** (or small computer picture) is far quicker and often more intuitive than typing in instructions to tell the computer to do something and is another feature of Windows. Windows also comes with **pointing devices** such as mice, to allow quick and accurate selection compared to typing in selections. For the above reasons, we say that Windows has a **WIMP** interface (Windows, Icons, Menus and Pointers).

36.11.2 MS-DOS

Command line operating systems such as MS-DOS have been largely side-lined for standalone PCs, although there are still notable examples of this type of operating system in use. UNIX, which is widely used on multi-user and networked systems, is one such example. Command line interfaces require you to type in instructions rather than click on windows, icons and menus.

36.11.3 A GUI verses a command line interface

The reasons why GUIs dominate command line interfaces vary but the following points can be made.

- It takes more time to learn a command line interface than a GUI because you must learn the syntax of instructions rather than clicking on an intuitive WIMP GUI.
- You can 'multitask' in a GUI i.e. run more than one application at (apparently) the same time but you can't in some operating systems that use a command line interface, such as DOS.
- Memory is managed more efficiently in a GUI compared to a command line interface.

Of course, you need more RAM to run a GUI than a command line interface - those windows, icons, menus and pointers and all those utility programs take a lot of programming! You've probably noticed how long Windows takes to load up when you boot-up a computer. DOS takes very little time to load up compared to Windows.

36.11.4 What do operating systems do?

Both the DOS and Windows operating systems (O/S) allow you to work on the computer without needing to know how the computer does what it does! To summarise what we have said before, an operating system:

1) decides where in RAM to put applications and data you use
2) manages file management, both organising the storing and retrieving of files
3) provides utilities for basic PC management such as de-fragmenting disks or formatting disks
4) reports errors
5) provides an interface so a user can access the PC
6) provides the management of I/O, for example, sending a file to the printer
7) schedules different jobs, so that the CPU and other PC resources are used efficiently.

36.12 What happens when you boot-up a computer?

Lots of things happen to a computer from the moment you turn the power on to the moment you start using it.

1) When you power up a computer, a little program of instructions (known as the **BIOS** or Basic Input Output System) held in the ROM chip starts running.
2) The job of the BIOS is to check to see if the computer has the basic, error-free hardware needed to get the PC up and running. It runs what is known as a **POST** routine, or 'Power On Self Test'. Specifically, this small program checks to see if there is a keyboard, a hard disk, a storage drive and some memory (RAM). If it finds any of these absent or if it finds an error with any of them, then it displays an error message (and beeps)!
3) Some of the BIOS is held in ROM. Some of it, however, is held in battery-backed RAM (known as CMOS) and you the user can configure it! Don't play with the BIOS settings on your computer unless you know what you are doing! You can create a lot of serious problems!
4) Once the basic hardware check is complete, the BIOS then looks in some pre-set places for a special program called the '**bootstrap program**'. Often, the first place it looks in is in the first sector (known as the '**boot sector**') of the hard drive and then if it can't find the bootstrap there, it looks in the first sector of any CD in the CD drive or possibly on any pen drive. We said above, however, that you could configure certain settings in the battery-backed part of the BIOS. One thing the user can configure is the order that the BIOS looks at devices when it is looking for the bootstrap. Once again, I don't recommend you play with the BIOS, but your teacher may show you the BIOS if you ask nicely! One thing the user might change is to get the computer to always look for the bootstrap program on a specific storage device first.
5) When the BIOS has located the bootstrap program, wherever it is, it loads it into RAM and runs.
6) The bootstrap program then takes over control of loading whatever operating system you are using. It first of all locates the operating system on the hard drive and then transfers it from the hard disk into RAM and runs it.
7) Once your operating system has loaded, the computer automatically looks for '**boot files**'. These are special files that the user can play with to set up the computer in exactly the way that the user wants.

36.13 More about boot files

Once the operating system has been loaded, users will often want to customise their system. For example, they might want a virus-checking program to run automatically and for the computer to open Word. To customise a system, users can add their own commands to certain files known as 'boot files' (e.g. command.com and autoexec.bat in Windows). These files are looked at automatically after the operating system has been loaded. If present, the instructions in them are carried out. The user then uses the computer. Boot files allow a user to set up (or 'configure') their PC in exactly the way they want.

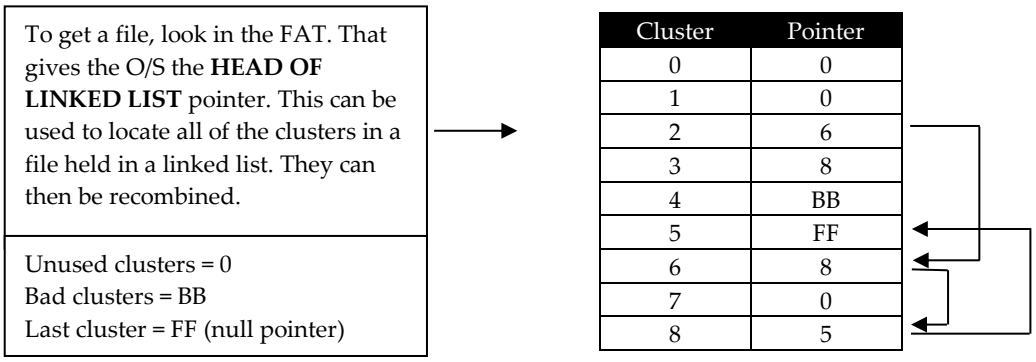
36.14 Managing files on a PC

Hard drives typically store all applications you have loaded up, all of your data files and the operating systems you have available, whether they are currently in use or not. To understand how a PC actually manages files and applications, you need to understand how the hard drive works, how information is stored on it and what the File Allocation Table (FAT) does.

36.14.1 How a hard drive works

A hard drive is made up of a set of magnetic disks. Each one is divided up into circular areas called **TRACKS**. The tracks are divided into **SECTORS**. Sectors can be grouped together into **CLUSTERS**. To read or write to the hard drive, the hard drive is first spun very quickly. A read-write head then moves in over the disk. When the head is above a track, it can read or write data to it. Actually, the head is going to be above several tracks at the same time so it can read and write data to several tracks when it is in one position. Since a hard drive is made up of a number of magnetic disks on top of each other, all the tracks together will form a cylinder. Note that there may be several thousand tracks on a disk. On a track, there may be e.g. 120 sectors (SCSI drive). Each sector has an address, made up of which track it is on and which sector on the track it is. Each sector, remember, is

a fixed size - it can only hold e.g. 512 bytes. So what happens if you want to save a file that is 1 Mbyte in size? You will need to use lots of sectors (or clusters). You will end up with bits of your file saved all over the hard disk! No problem. That's where the FAT comes in.



The File Allocation Table (FAT).

36.14.2 The File Allocation System (FAT)

The FAT is a database that keeps a log of every file on the hard drive. It is created by the operating system when you **format the disk** and it is stored on **track zero** of the disk itself. Your hard disk is divided into sectors, each one being able to store a fixed number of bytes. These sectors are grouped into **clusters**.

Each file is held in a **linked list of clusters.** The FAT holds the start address of the first cluster in the linked list. When you want a particular file, the operating system looks up the address of the first node in the FAT. It then follows this address and retrieves the first cluster as well as the address of the next node. It then follows that address to the next node, recombining the clusters into a file, until the end-of-file marker is reached (the null pointer).

36.14.3 More on the FAT

The FAT is used to keep track of the storage space on a storage device. Entries in this table can indicate bad clusters, free clusters, clusters in use by a file or the end of a file. The smallest unit of space that can be allocated to a file is not necessarily a sector. When DOS was first designed it was decided to allocate space on a disk in units called 'clusters'. The number of sectors in each cluster depends on the actual capacity (the 'format') of the disk in question. Clusters rather than sectors were used because the size of the table needed to keep track of cluster usage was smaller than the one that would be needed for sector usage. The single-sided 40 track disk formats had a cluster size of one sector but the double-sided 40 track formats used two sectors per cluster. Hard drives many more sectors per cluster. What this means is that the smallest amount of disk space that can be allocated to a file when it is written to a disk is the size of the cluster. This size can be calculated by the number of sectors per cluster multiplied by 512 bytes per sector. When a small file is saved onto a large capacity hard drive, the amount of disk space taken up by that file is 4096 bytes, even if the file is 10 bytes long. This wastage is because of the way DOS keeps track of where it puts files onto disks. This is one reason why data compression is sometimes able to shrink files by a large amount.

36.15 Network operating systems

A Network Operating System (NOS) is an operating system that gives computers the ability to communicate with other computers on a network. NOS software often includes extra facilities to allow backups and improve security, for example. Some widely used NOS software includes Novell and UNIX.

36.15.1 Peer-to-peer network operating systems

A peer-to-peer network is a number (more than one) of computers that have been connected together so that their resources can be shared. The computers on this type of network have equal status (they are all 'peers') and can all be used as workstations; any of them, at the same time, could also manage the printing (act as a print server) or look after sharing files (act as a file server) or cache web pages or look after email. These kinds of networks are very simple compared to 'client-server' networks. Once you have connected all your computers together, you can configure the network and then away you go! The network is very simple. It will control the communication between workstations and allow you to e.g. control the transfer of files from one machine to another, printing and using workstations on the Internet at the same time with only one modem

36.15.2 Client-server network operating systems

A client-server network is made up of computers and servers. The computers are known as ‘clients’. These are connected up to each other as before but at least one of the computers acts only as a resource manager. The computers on the network then make

use of the resources available at any of the resource managers. The resource managers are more properly known as 'servers'. You can have a range of different kinds of servers on a typical client-server network. These might include a file server, a print server, a web server and an email server, for example. Client-server networks are more complicated than peer-to-peer networks and need to have a Network Operating System (NOS). A Network Operating System typically:

- controls communication between workstations themselves and between workstations and servers
- manages security and user accounts using logins and passwords
- allows and manages file sharing
- allows and manages printing and other resources
- controls Internet access amongst stations
- manages automated system backups
- manages and reports on the use of the system's resources, such as each user's hard disk allocation and how many prints they are allowed to do.

36.15.3 File sharing

The NOS must stop two people writing to the same file at the same time! When someone opens a file, other users get **locked out** until the file is finished with. In addition, the NOS can give each user **rights**. These give users the right to read files or to change files or to add and delete files, for example.

36.15.4 Printing

When files are sent for printing on a network, a program known as the 'spooler' intercepts it. A spooler program is usually part of the NOS program although spooler programs will also come with dedicated print servers.

36.15.5 Security

Access to the resources on the network must be controlled. Information is valuable, time on the network is valuable, resources must be accounted for and privacy issues must be respected - when a user saves their files onto the server, they must not be accessible to anyone else! Network Operating Systems use a system of 'accounts' to help achieve this. Users are given a User ID along with a password. These must be used to gain access to the network.

Many things that have security implications can be controlled by the NOS. You can force the user to change their password after fixed amounts of time. You can allow access to only certain applications and equipment, from certain machines and at certain times of the day. The amount of disk space allocated to each user can be controlled so that one user doesn't hog a fixed resource such as disk space. Some of you may have had messages in the past whilst working on a school computer saying that you are getting close to using up your allocation of disk space so delete some old files or that your print credits are low and need to be topped up. These came from the NOS! In addition, everything that users do can be logged and reported!

36.15.6 Accounting

As has already been mentioned, the resources on a network are often of a fixed size. Examples include the size of the hard disk, the number of licences for an application, the cost of printing, the use of computer time and time spent on the Internet. Sometimes, you just want to try to be fair to everyone. Sometimes you might want to charge for certain services such as printing or Internet access or you might want to reserve applications for which you have limited licences. To help you do this, the NOS gives you tools to control the resources and reporting facilities to enable you to account for them and review or charge for them if necessary. A network manager can, of course, buy additional software to help her. For example, she may buy some print management software, as discussed earlier.

Q1. Define an 'operating system'.
Q2. List six typical functions of a personal computer's operating system.
Q3. Define an 'interrupt'.
Q4. Describe how an operating system handles two different kinds of interrupts occurring simultaneously.
Q5. Describe the difference between a 'processor-bound' job and an 'I/O-bound' job.
Q6. What are the aims of scheduling?
Q7. Describe the role of the process control block in scheduling.
Q8. Describe the role of 'pages' and the 'page file' in memory management.
Q9. What is meant by 'disk threshing'?
Q10. Describe how files are stored on a hard drive using the File Allocation Table.

Chapter 37 - Translation in detail

37.1 Introduction

Programmers write programs in source code. This might be using PYTHON, BASIC or C++, for example. However, this code cannot be executed directly on a computer. First, the source code must be translated into a form that the computer can understand. This can be done using interpreters and compilers.

37.2 Compilers and interpreters

This type of translator takes the whole source code and compiles it into object code. The object code (sometimes called **machine code**) can then be run. Pascal is an example of a programming language that uses compilation. An Interpreter on the other hand takes the source code and translates the first line of the program and then executes it. It then does the second line, and the third line, until it gets to the end of the code. BASIC, LISP, PROLOG and APL are programming languages that use interpretation.

37.3 Compilers plus interpreters

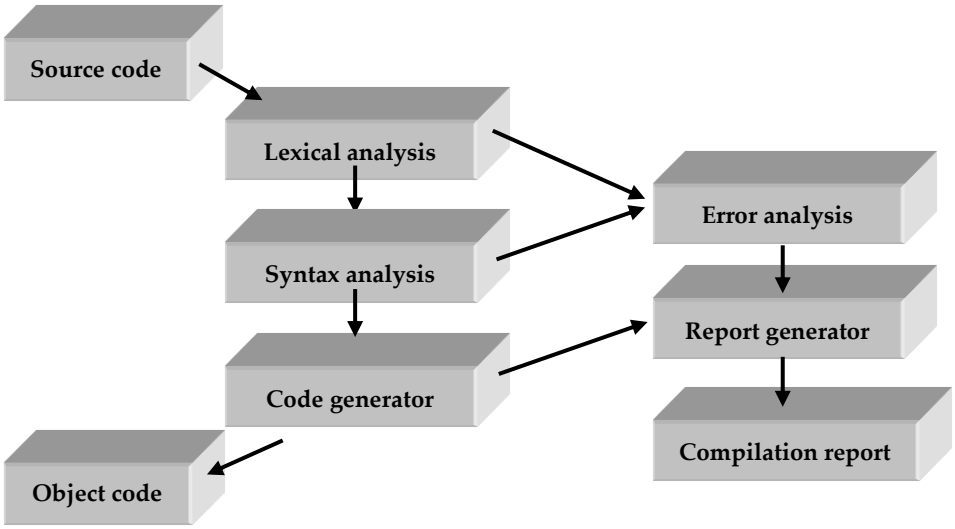
Some programs written in languages such as JAVA are both compiled and interpreted! A program is firstly compiled into an intermediate code known as **bytecode**. It is then distributed to users who use a wide range of computers such as Mac or PCs. These computers then run their own 'interpreter' to convert the bytecode into a code they can use. Languages such as JAVA are said to be 'platform-independent', because any program written in that language can run on any machine. These types of languages are ideal for use on the Internet; you don't need to know anything about the PC that will be running your code!

37.4 Compilation and interpretation compared

- Compilation is much faster than interpretation. Once the compilation process has been completed, the object code will run faster than the same interpreted code. And you only have to compile the code once.
- Object code does not need the compiler to actually run it, only to convert it from source code to object code. Therefore, the object code produced by a compiler can be distributed to other computers without the compiler. You could write a program and then sell the object code on CD to anyone who wants to run the application.
- Object code is difficult to 'read' so that it can be distributed without actually revealing the code. This helps protect your intellectual rights.
- Source code can be compiled in sections to produce object code in sections. You can't do this with interpreters. If you have a lack of free space in RAM, compilation may be more useful.
- If there is an error in interpretation, the program will run successfully up to the point where the error occurs. Then it will stop. All of the variables are available for inspection at that point. Once the error has been corrected, the program doesn't need to be recompiled - it's just run again using the interpreter. Because of this, debugging and program development are easier and quicker using interpretation compared to compilation.
- Interpreters are far simpler to write than compilers. This is why JAVA works - because it is a relatively easy job to write an interpreter for a particular computer or application compared to writing a compiler for the machine.

37.5 How is source code actually translated?

A translator's job is to turn source code into object code. It must be able to translate good code that follows all the rules of the programming language as well as signal problems with bad code. There are three distinct steps to producing object code: **lexical analysis**, **syntactic analysis** and **code generation**.



Three steps to producing object code.

During the lexical and syntactical analysis stages, any code that cannot be analysed will be passed to the error analyser software routine. If errors are found, the report generator will display error messages and object code will not be produced. If the code produced by the lexical and syntactical analysis stages produce no errors, then the object code is generated and the report generator will say as much. It will report on other details, such as how much space the object code takes up in memory.

37.6 The lexical analysis stage

The lexical analysis stage is the first stage of program translation. A number of things happen in this stage.

37.6.1 The actions during lexical analysis

1) The source code is looked at. Any unnecessary parts of it are removed. This would include comments and spaces.
2) Keywords and symbols are then replaced with 'tokens'. A token is a **generic description** of the type of symbol. For example, if you had a constant called 'months_in_year', this would be replaced by the token **CONSTANT**. If you had variables called 'speed' and 'distance' then these would be replaced with the token **VARIABLE**. If you had a keyword 'IF', this would get replaced with the selection token for 'IF'. If you had a maths operator such as '>', this would get replaced with the token **OPERATOR**. By replacing symbols, variables, keywords and so on with generic tokens, you can turn a program into a set of 'patterns' of instructions. It is then a relatively easy job for the compiler in the syntax stage to check each pattern against the allowable ones. Tokens replace:

    i. a keyword like IF, FOR, PRINT, THEN and so on
    ii. a symbol that has got a fixed meaning, such as +, *
    iii. numeric constants
    iv. user-defined variable names.

For example, if you had the following line in a program: **IF x > 10 THEN** this might be changed into this pattern of tokens:

SELECTION/IF-VARIABLE-OPERATOR-CONSTANT-SELECTION/THEN

Notice that the spaces have been removed and generic descriptions (tokens) have replaced keywords, variables, symbols and numeric constants.

3) A look-up table is created. This stores the values of all constants used along with their data type.
4) Variable names are also entered into the look-up table.

Once you have lexically analysed your program, you end up with a string of tokens. This is passed to the syntax analysis stage.

37.7 Syntax analysis

Once the source code has been passed through the lexical analysis program, the tokens generated are passed to the syntax analysis program for the next stage translation. This stage checks to see if the program makes sense - the program **semantics**.

37.7.1 Analysis of the token stream from lexical analysis

Following lexical analysis, the syntax of the program is checked. The first job is to take the long stream of tokens from the lexical analysis stage and to split them up into **phrases**. Each phrase can then be checked against the rules of the programming language (the syntax of the language). Sentences in English must be constructed according to rules. So for example,

"**The computer helps me run my business.**" is okay but you cannot write,
"**Computer business run the me helps my.**" because it breaks the syntax rules of English.

Here is another example of what this means during compilation. Consider the instruction:

IF|SizeOfEngine|>|2000|THEN|Payment|:=|Mileage|*|40 During lexical analysis it is converted to:

TOKEN/IF|VARIABLE|RELATION|CONSTANT|TOKEN/IF|VARIABLE|TOKEN/ASSIGN|VARIABLE|TOKEN/MATHS|CONSTANT

When this is checked against the allowable patterns in syntax analysis, it is found to be fine. Now consider the same instruction that has been written incorrectly, not following the syntax rules.

THEN |SizeOfEngine|>|2000| IF|PaymentMileage|*|40|:=

This would generate an unacceptable pattern of tokens during lexical analysis. During the syntax analysis, therefore, an error would be generated because the pattern of tokens did not match any in the compiler's database of allowable patterns.

It is a relatively easy job for a compiler to keep a record of allowable patterns for a high-level language. One method for describing each pattern is known as **Backus-Naur form, or BNF**. This is dealt with later. When the syntax analysis stage is being carried out, each phrase of code is checked against the allowable patterns. This is known as **parsing the code**. If a match cannot be made, then the report generator generates an error.

37.8 Code Generation

Source code is first passed through a lexical analysis program and then a syntax analysis program. It is then given to the code generation program to actually produce the object code.

37.8.1 Code generation

A high level language is first LEXICALLY analysed. Then its SYNTAX is analysed. During syntax analysis, the SEMANTICS of the code is checked to see if it makes sense. If any errors are found in these stages the REPORT GENERATOR program springs into action and displays helpful (or not so helpful) error messages. The programmer uses these to aid debugging the program. This is known as 'translator diagnostics'. When the program can be lexically and syntactically analysed without errors, the object code can be generated. Consider the line:

LENGTH := 2 * (FIRST - SECOND) + 4 * (THIRD - FOURTH)

During lexical analysis, a look-up table will have been created of all the variables found in the source code, their data type and where in memory the variable value can be found. For example;

Name	Type	Address
LENGTH	INT	FFB2
FIRST	INT	1445
SECOND	INT	A4A3
THIRD	INT	FF21
FOURTH	INT	C411

Also, library routines may need to be called up and referenced. In Pascal, for example, you would call the library known as CRT to enable you to use the 'Clear the screen' routine.

37.9 Code optimisation

The generation of code proceeds along the lines described above for each instruction until the whole program has been turned into object code. It can then be run on the computer, or distributed to users to use on their computers. Code generated using this method may not be the most efficient code that could be generated. For example, when a tree is produced, you may get:

VALUE := RESULT*1 or you may get another example: **X : = Y + 0**

Clearly, **VALUE := RESULT*1** is the same as **VALUE := RESULT,** and **X := Y + 0** is the same as **X := Y**

These examples, then, show that the code generation program could produce inefficient code! The code must be optimised! When the compiler has generated the object code, it runs routines that do just this. It tries to make the code as fast and as efficient as possible by finding and removing unnecessary code.

37.10 Evaluating algebraic expressions

Computers need to evaluate expressions like X = P + Q. Because we are human, we understand the logic of saying P + Q. This kind of notation is called **infix notation**, because the + operator goes between the variables we want to add together. An alternative to saying 'P plus Q' is to say 'Add P to Q'. We could write this as +PQ and is known as **prefix notation**, because the operator goes before the variables you want to add together. Finally, we could also say get P and Q and add them together. We could write this as PQ+ and is known as **postfix notation**, because the operator goes after the variables you want to add together. Postfix notation is also known as **reverse Polish notation**. So A + B becomes AB+ in reverse Polish notation. X * Y becomes XY* in reverse Polish notation.

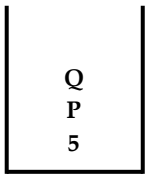
37.10.1 Converting between Reverse Polish notation and infix notation

Let's look at some more examples of reverse Polish notation.

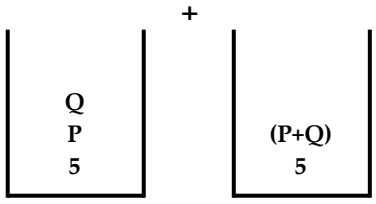
Consider 5 * (P + Q). This becomes 5 * (PQ+) because you do what is in the brackets first. This then becomes 5(PQ+)* and finally, we remove any brackets. This gives us 5PQ+*

Reverse Polish notation doesn't require any brackets. There is no confusion about what has to be done first. Reverse Polish notation uses a stack to evaluate expressions. The rule is that all variables and constants go into the stack but an operator works on only the two top things in the stack, with the result of this operation being put back into the stack.

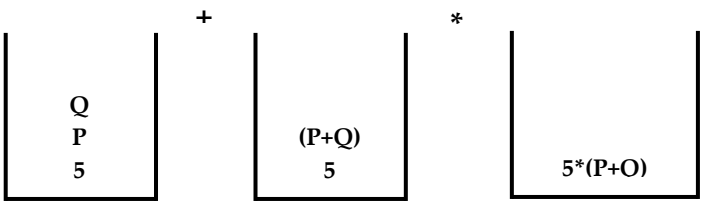
Consider 5PQ+* First, we put 5 into the stack. Then P. Then Q.



Next, we have an 'add' operator. This adds the top two items in the stack and puts the result back in the stack.



Finally, we have a 'multiply' operator. This multiplies the top two items in the stack and puts the result back in the stack.



So 5PQ+* is the same as 5 * (P+Q)

37.11 Loaders and Linkers

Two programs that are very important to program translation are **linkers** and **loaders**. These are discussed below.

37.11.1 Linkers

A linker is a program that allows a user to link library programs or separate modules of code into their own programs. It is used to combine different modules of object code into one single executable code program. This may involve combining a program with library programs, or involve recombining blocks of object code from the same program, or a mixture of both.

- A program may call library modules.
- Library modules will use data stored in 'relative addresses'. See the section below on loaders for more about relative addressing.
- The program itself is compiled.
- The library modules are compiled.
- The linker program is run.
- The compiled library modules are linked to (connected to) the compiled program.
- Base addresses used in the library modules are adjusted so that the calls in them to data work from within the main program.
- One executable program is produced.

Linkers are also used when a user has to compile a big program in sections, perhaps because there is a shortage of RAM. The user's program is split up into sections and held on backing storage. Each section is then brought into RAM, one section at a time, from backing storage and compiled into object code. Each block of object code is then saved back to the backing storage. When the whole program is compiled, the compiler can be removed from RAM and all of the blocks of object code can then be brought into RAM because there is enough space without the compiler program. The linker program is used to recombine the blocks of object code in RAM to get a working full program.

37.11.2 Loaders

A loader is a piece of software that chooses exactly where to put object code in RAM, ready for it to be run. It also adjusts the memory references in programs. These pieces of software are explained in more detail below.

The job of a piece of software known as a loader is to take the object code generated by compilation and to find a 'good' place for it in RAM, where it can then be executed. Imagine that a software house has written a program to sell to the public. They wrote the source code, compiled it so that they then had some object code, and then copied the object code onto a CD, ready to sell. They wouldn't distribute the source code because they would want to keep the actual program code secret from competitors - it is practically impossible to turn object code back into source code. Besides, your customers may not have the necessary compilers on their machines to convert the source code into object code. In addition, it would be a little inconvenient if your customers had to compile every program they wanted to use before they actually used it!

If you bought the CD and wanted to run the program, you might double click on the .exe file. The loader would then copy the object code from the CD into your RAM and run it from there. But where in RAM would the loader put it? You have other applications running in RAM, for example, the operating system and a virus checker. You may also be multi-tasking, with various programs and data in RAM. If the loader is not careful, it will load a program in a place in RAM that interferes with other programs. Your loader program, then, must decide where to put the object code in RAM so that it doesn't interfere with other programs and data. This is the first main job of the loader program.

The second main job involves adjusting references that are used within a program. Programs can be written by programmers using either '**absolute addressing**' or '**relative addressing'**. Relative addressing is more common because then the loader can put the program anywhere in RAM - absolute addressing isn't flexible.

37.11.2.1 Direct addressing (also known as 'absolute addressing')

Consider this program pseudo-code that uses direct addressing.

- Get data from memory location 2001 and store in 2010
- Get data from memory location 2002 and store in 2011
- Add data in location 2010 to data in 2011 and store the result in 2012.

To run this code, it needs to be compiled and placed in RAM. Look at the instructions and look at the data. Have you noticed that specific memory addresses have been used, for example, "Get data from memory location 2000 and store in 2010"? When

specific memory locations are used, this is known as 'direct addressing'. It is important to recognise the consequences of direct addressing. It means that the code, once compiled, will have to be put in a certain part of memory to work. What would happen, for example, if another program were already occupying memory locations 2000 - 2012? Your program wouldn't be able to look up the correct data in the data file!

37.11.2.2 Relative addressing

Now consider the above pseudo-code, this time using relative addressing instead.

- Set the base address is 2000
- Get data from memory location +1 and store in +10
- Get data from memory location +2 and store in +11
- Add data in location +10 to data in +11 and store the result in +12.

In this example, all data items and all references in the code are made relative to a base address. In fact, the base address is the only address where a specific memory address is mentioned. If, for example, you want to get the data from +1, you get the base address, add 1 to it and go to that address. The loader can now put the compiled code for the above anywhere it likes, anywhere that is free! All it needs to do is simply change the base address. This is the second main job of the loader program - to adjust any base addresses as appropriate.

37.12 What is a Dynamically Linked Library (DLL)?

A DLL is a collection of executable programs. These provide functions that can be called and used by Windows-based applications. An application that wants to use a particular function needs to create a link to the DLL. If the DLL is missing, then a message will pop up saying that it 'cannot find XXX.dll' or 'call to undefined dynalink'. This can also happen if a program is trying to call a DLL that is not compatible with the operating system. This might happen because the DLL being called is too old for that latest operating system you have recently upgraded your computer to!! If this happens, you will need to locate the correct DLL on the Internet and install it. There are lots of sites that you can go to and download the DLL you need.

Device drivers are sometimes packaged as DLL files. (A computer needs a Device driver if it wants to communicate with a peripheral such as a printer or a scanner). The system of having Dynamically Linked Libraries stored on the hard drive means that they do not have to be loaded into RAM until the point that they are needed. If you wanted to print a document, for example, then your application, for example, Word, would load the DLL only when you actually wanted to print something out. This saves space in RAM. Not only that but it also means that any Windows-based program can simply call the DLL. The developers of a new Windows-based application don't have to re-write the DLL or some other code for printing, for example. By using existing DLLs, the developers are:

- Saving space on the hard drive.
- Ensuring RAM is used as efficiently as possible.
- Saving time in development costs.

The ability to reuse modules of code is one of the benefits of modular programming. You can find out a lot more about DLL files by searching for **What is a DLL** in Google.

Q1. Why do programs that people write need to be translated into object code?
Q2. What are the three main steps in translating some source code?
Q3. Describe what happens in the lexical analysis stage of translation.
Q4. What is meant by the 'syntax' of a program instruction?
Q5. Describe what happens in the syntax analysis stage of translation.
Q6. Why does the code generated during translation need to be optimised?
Q7. Show how the reverse Polish notation AB+CD*+ is calculated using the stack.
Q8. What is a linker program used for?
Q9. What does a loader program do?
Q10. What is meant by 'relative addressing'?

Chapter 38 - Computer architecture and the FDE cycle

38.1 Von Neumann architecture

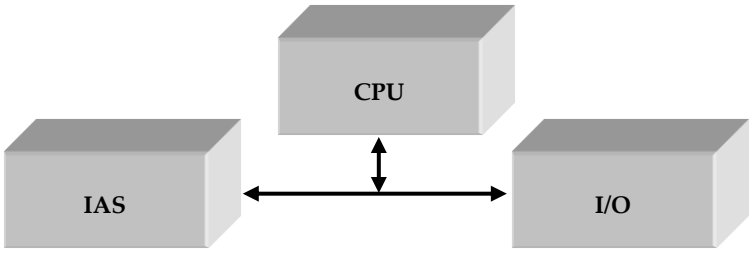
In the 1940s, a mathematician called John Von Neumann described the basic arrangement (or architecture) of a computer. Most computers today follow the concept that he described although there are other types of architecture. When we talk about the Von Neumann architecture, we are actually talking about the relationship between the hardware that makes up a Von Neumann-based computer.

38.2 A Von Neumann-based computer is a computer that:

- Uses a single processor.
- Uses one memory for both instructions and data. A von Neumann computer cannot distinguish between data and instructions in a memory location! It 'knows' only because of the ***location*** of a particular bit pattern in RAM.
- Executes programs by doing one instruction after the next using a fetch-decode-execute cycle.

A Von Neumann CPU is made up of four important components. These are the , the registers, the control unit and the IAS (Immediate Access Store, otherwise known as the RAM or main memory).

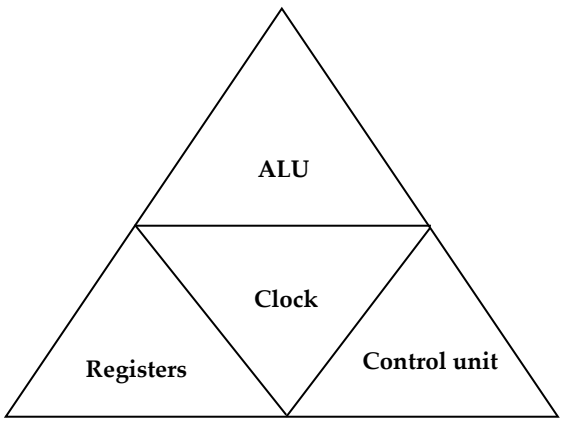
Because the IAS is so important, we are definitely going to move it to its own section in our model of a computer. (We discussed this previously). We need to get data into and out of the computer so we will include this as a separate section as well. We will also introduce the idea of a clock and clock cycles in the CPU. Our new model of a computer now looks like this:



A model of a Von Neumann computer system.

38.2.1 Von Neumann Component 1 - The CPU

The CPU, or Central Processing Unit, is the name given to the component that controls the computer and works on the data. It can be split up into four sub-components:



A diagrammatic representation of the 4 main parts of a CPU.

We know a few things from before about the Von Neumann CPU. A Von Neumann CPU has an ALU. This is the part of the CPU that performs arithmetic and logic operations on data and acts as the revolving for the CPU, letting data enter and leave the CPU. We also know that CPUs have a 'word size'. This is the number of bits that can be added, for example, in one go. The bigger a CPU's word size, the more bits it can work on in one clock cycle and the more work you can get done. A Von Neumann

CPU has a control unit. The control unit is in charge of 'fetching' each instruction that needs to be executed in a program by issuing control signals to the hardware. It then decodes the instruction and finally issues more control signals to the hardware to actually execute it. A Von Neumann CPU has registers. These are very fast memory circuits. They hold information such as the address of the next instruction (Program Counter), the current instruction being executed (Current Instruction Register), the data being worked on and the results of arithmetic and logical operations (Accumulators), information about the last operation (Status Register) and whether an interrupt has happened (Interrupt Register). Registers are covered in a lot more detail later in this chapter. Instructions are carried out to the beat of the clock! Some instructions take one beat and others more than one beat. Very roughly speaking, the faster the clock, the more clock beats you have per second so the more instructions per section you can do and the faster your computer will go.

38.2.2 Von Neumann Component 2 - IAS

We also know that the Von Neumann computer has an IAS, or Immediate Access Store, where it puts both programs and data. We often commonly refer to this memory as RAM. RAM is made up of lots of boxes that can store a bit pattern. Each box has a unique address. A memory address might store an instruction (which is made up of an operator and an operand) or it might store just a piece of data. A Von Neumann computer can't tell the difference between the bit patterns as such, but 'knows' indirectly because of *where* the bit pattern is stored in RAM. Pre-Von Neumann computers used to split up memory into program memory and data memory and this made computers relatively complex. Von Neumann was the first to realise that there was actually no difference between the nature of an instruction and the nature of a piece of data. One important function of an operating system is to manage memory and to keep track of the RAM addresses of applications as well as any data.

We also know that computers have an address bus, so that the CPU can address each individual memory location in the IAS, for example, when it wants to store a piece of data or retrieve a piece of data. The data itself is moved about between devices on a data bus. There is also a control bus, to generate signals to manage the whole process.

38.2.3 Von Neumann Component 3 - I/O

A computer needs peripherals for inputting and outputting data. It needs to be able to read data into itself and send data out. It reads data in and sends data out through its I/O ports. A port is simply a gateway, like a port used for shipping. Just like every port used for ships needs its own harbour master, so every I/O port needs to be managed. An I/O controller is the term used to describe the I/O port along with the circuits that manage data into and out of the port. It allows you to connect up any I/O device to the PC and transfer data in to or out of the computer. You wouldn't want to connect an I/O device directly to a CPU because you would have to redesign the CPU every time a new type of device came along. Besides, a new type of device might need different voltages and control signals from the CPU, again necessitating a CPU redesign. The I/O controller acts as an **interface** to overcome these problems.

38.3 CISC and RISC

There are different types of CPUs in existence. For example, there are the Intel 80x86 processors upon which today's home computers are based (also known as **IBM compatibles** and variations thereof). With these types of processors, you can load up operating systems such as Windows and Linux. Then there are the 680x0 processors upon which the Apple Macs are based. Each type of CPU has its own set of machine code instructions that it can work with. All of the instructions together for any particular CPU are known as the CPU's '**instruction set**'. One type of CPU will not be able to use the same instruction set as a completely different CPU.

The above types of processors are sometimes referred to as CISC, or **Complex Instruction Set Computers**. These processors have complex instructions which can be carried out in just one single 'fetch decode execute' cycle (see the next section). They have many different addressing modes and a wide range of instructions that can be used

There are also CPUs that are known as RISC (pronounced 'risk'), or **Reduced Instruction Set Computers**. RISC processors such as ultra-SPARC and ALPHA use a much smaller, simpler set of instructions than CISC processors and so to carry out any particular programming task may take many 'fetch decode execute' cycles. RISC processors, however, are much more efficient at processing huge blocks of data than CISC.

38.4 The fetch-decode-execute cycle (FDE cycle)

You want to run a program that you have on your hard disk. You double-click on it in your file manager. The loader program (part of the operating system) goes to the hard disk and copies it to a suitable place in the IAS. It adjusts any memory references in the program as appropriate and then loads the address of the first instruction of the program into the Program Counter (PC). It then signals to the CPU that it has done this. The CPU wants to run the program and gives control to the control unit. The control unit fetches the first instruction from the IAS. It then decodes it and executes it. Once executed, the cycle is reset and the control unit fetches, decodes and executes the next instruction in the program, then resets the cycle again. This process is

repeated until the whole program has been 'run'. At all stages in the FDE instruction, the CPU will need to use the registers. These are discussed in detail below.

38.4.1 The fetch part of the cycle

Every program has a start address. The start address is placed in the Program Counter (PC). The contents of Program Counter is copied to the Memory Address Register (MAR). The Control Unit goes to that address in memory and copies the bit pattern it finds there to the Memory Data Register (MDR), where it is then copied to the Current Instruction Register (CIR). While it is doing this, the PC is automatically incremented, so it is pointing to the memory address of the next instruction. The following describes what happens in the fetch part of the FDE cycle.

MAR ← **PC**

PC ← **PC + 1**

MDR ← **(MAR)**

CIR ← **MDR**

The address in the program counter is copied to the memory address register. The program counter can then be incremented so that it points at the next instruction to be done. The contents of the memory address pointed to by the memory address register are then copied to the memory data register. The contents of the memory data register are then copied to the current instruction register.

The registers are used in this way for the fetch part of **EVERY** instruction, regardless of what the instruction actually is.

38.4.2 Decoding an instruction

Once an instruction has been fetched, it needs to be decoded by the control unit. This simply means that the control unit works out what the instruction is and what signals it has to send out to the various parts of the CPU to make the instruction happen – to 'execute' it.

38.4.3 Executing an instruction

To execute an instruction, we need to use the registers again. How the registers are used depends on exactly what the instruction we want to execute is. Generally however, once we have fetched an instruction and have got it into the CIR, it is split into two parts:

1) the actual instruction we want to execute
2) the actual address of the data we want to use with the instruction.

Central to every operation that is fetched, decoded and executed is the Accumulator. You do any actual mathematical or logical operations in this register. That means that before you can work on any data, you must put it into the Accumulator first. This is straightforward. Simply put the operand of the instruction on the MAR and then move the contents of the MDR to the Accumulator. To store the Accumulator's contents, simply put the address where you want to store the contents on the MAR and then move the Accumulator to the MDR. Also note ADD (x) instructions usually take the form of 'ADD the contents of a location given by the operand x to the contents of the Accumulator and store the result in the Accumulator'. SUB (x) means 'subtract the contents of the location given by x from whatever is in the Accumulator and store the result in the Accumulator'.

Suppose you wanted to add two numbers together, held in locations 2000 and 3000 and store the result in 4000. The instructions might be:

LOAD (2000) //This loads the contents of memory location 2000 into the accumulator.
ADD (3000) //This adds the contents of location 3000 to the accumulator, storing the result in the accumulator.
STORE (4000) //This instruction takes whatever is in the accumulator and stores it in location 4000.

MAR	←	**PC**	**Fetch the instruction pointed to by the PC.**
PC	←	**PC + 1**	
MDR	←	**(MAR)**	
CIR	←	**MDR**	
MAR	←	**2000**	**It was decoded and found to be LOAD (2000). Get the contents of 2000**
MDR	←	**(MAR)**	**into the ACC.**
ACC	←	**MDR**	

MAR	←	**PC**	**Fetch the next instruction.**
PC	←	**PC + 1**	
MDR	←	**(MAR)**	
CIR	←	**MDR**	
MAR	←	**3000**	**It was decoded and found to be ADD (3000). Add the contents of 3000 to**
MDR	←	**(MAR)**	**the Acc. Store the result in the ACC by overwriting contents of the ACC.**
ACC	←	**ACC + MDR**	
MAR	←	**PC**	**Fetch the next instruction.**
PC	←	**PC + 1**	
MDR	←	**(MAR)**	
CIR	←	**MDR**	
MAR	←	**4000**	**It was decoded - STORE (4000). Copy the contents from the ACC to 4000.**
MDR	←	**ACC**	
(MAR)	←	**MDR**	

The use of the registers can be summarised as follows:

1) **The PC or Program Counter.** This holds the address of the next instruction in the Fetch-Decode-Execute cycle.
2) **The MAR or Memory Address Register.** The contents of the PC are loaded into the MAR. This allows the PC to be incremented.
3) **The MDR or Memory Data Register.** This register holds the bit pattern held in the address pointed to by the address in the MAR.
4) **The CIR or Current Instruction Register.** This register holds a copy of the bit pattern in the MDR, thus freeing up the MDR.
5) **The Accumulator.** All logical and arithmetic operations take place in the accumulator. Sometimes there is more than one accumulator and these might be referred to as accumulator A, accumulator B etc.
6) **The Interrupt Register.** This holds information about what interrupts have happened. It will be checked after each Fetch-Decode-Execute cycle.
7) **The Status Register.** This register gets updated after every ALU operation. It holds information about the result of the last operation that was carried out. It will tell you whether the last operation produced a carry bit, whether the result was zero, or negative, or there was an underflow, or an overflow, for example.
8) **The Input Register.** This register is used to transfer information from an input device into the CPU.
9) **The Output Register.** This register is used to transfer information from the CPU to an output device.

38.4.4 Dealing with a jump instruction

If an instruction is fetched and decoded and turns out to be a jump instruction, we can execute it in the following way:

IF the instruction turns out to be a **CONDITIONAL JUMP** instruction, **THEN**

- The Status Register is examined.
- If a jump is required, then the operand associated with the jump instruction is copied to the PC and the next fetch, decode, execute cycle is done. If a jump isn't required, then the next fetch, decode, execute cycle is done anyway.

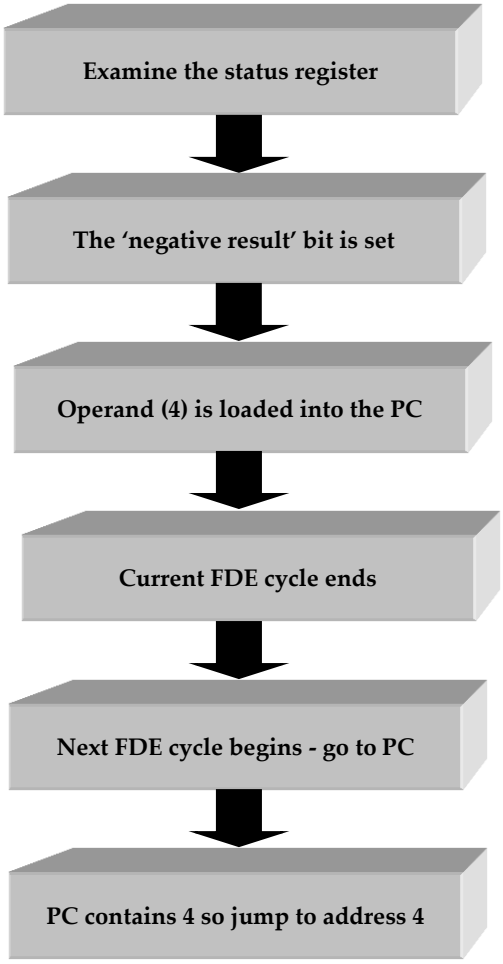
ELSE IF the instruction is an **UNCONDITIONAL JUMP** instruction, **THEN**

- Copy the operand associated with the unconditional jump to the PC.
- Do the next fetch, decode, execute cycle.

ENDIF

For example, the control unit fetched an instruction using the registers. It was decoded by the instruction decoder and found to be JMZ 4. JMZ means 'jump to the address given in the operand if the result of the last instruction produced a negative result'. The **operator** is JMZ and the **operand** is 4. We carry it out as follows:

Dealing with jump instructions - JMZ 4



- Because this is a 'conditional jump' instruction, the Status Register is examined.
- The status register holds information about the last instruction that was done. It can be thought of as a byte, with 8 bits, each bit indicating a status – one bit would indicate if the last instruction done produced a carry, or an underflow, or an overflow, or if the result of the last instruction produced a negative number.
- The bit associated with negative numbers is examined.
- Assuming that the last instruction did indeed produce a negative result, this bit is found to be set.
- The control unit therefore loads the operand associated with JMZ into the PC (in other words 4 is loaded into the PC) and the current FDE cycle ends.
- This FDE cycle has now finished so the next instruction must be done. The address of the next instruction to do is held in the PC.
- The control unit goes to the address held in the PC and fetches it – it goes to memory address 4 and gets the instruction it finds there.
- The program has successfully branched off to a different part of the program (beginning at address 4) than it would have done. A jump to a subroutine has been achieved!

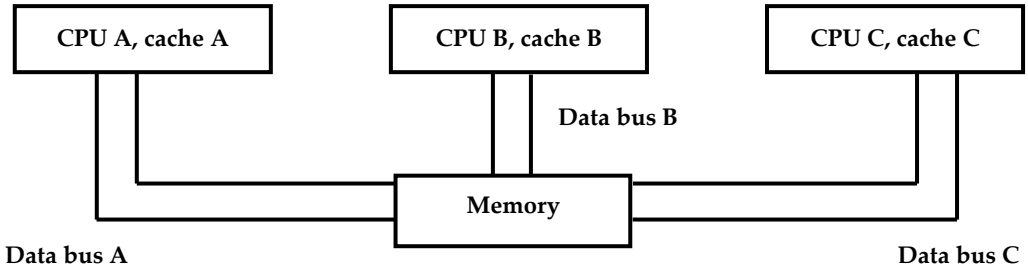
38.5 Von Neumann bottleneck

Whatever you do to improve performance, you cannot get away from the fact that instructions can only be done one at a time and can only be carried out sequentially. Both of these factors hold back the efficiency of the CPU. This is commonly referred to as the '**Von Neumann bottleneck**'. You can provide a Von Neumann processor with more RAM, more cache or faster components but if real gains are to be made in CPU performance then a major review needs to take place of CPU design.

38.6 Parallel processing - a different processor architecture

Rather than have one processor executing programs, you could have more than one! This is sometimes referred to as a **Supercomputer**! This would greatly increase the speed that programs could be executed – a major advantage of this kind of processing. **Different parts** of the **same program** could be executed **simultaneously**, each processor dealing with a particular task. Of course, this assumes that the original program was designed to have parts of it executed at the same time. Most programs are not designed in this way and this adds an extra degree of complexity to writing software – a disadvantage of parallel processing! When programs are written to use parallel processing, however, great improvements in the speed of

processing are seen, for example, in weather forecasting because this requires a lot of mathematical processing. One possible layout for a parallel processing system is shown in the next diagram.



Parallel processing in action.

In this system, there are a number of CPUs working together, sharing the same memory. The processing that all these CPUs do will greatly increase the demands on the data bus - each CPU needs to get data to and from memory and only one CPU can use the data bus at any one time. To help reduce the time one processor waits for access to the data bus, each CPU has its own cache. Data and instructions can then be transferred to it in one go and the CPU can get on with processing it without constantly requiring memory access. Now a CPU will be waiting much less than before. The provision of a data bus for each CPU will further improve performance because the CPUs will not have to wait for access to the only available bus.

38.7 Pipeline processors - another different processor architecture

In the conventional Von Neumann architecture, an instruction is fetched, then decoded and finally executed. This happens in sequence. It would be nice if an instruction could be fetched and then, while it is being decoded, another instruction is fetched, and while the decoded instruction is being executed, other instructions are being fetched and others still decoded - all at the same time. A processor that is designed to do just this is known as a 'pipeline processor'. As with parallel processing, programs need to be written to take full account of this capability.

38.8 Array processors

An array processor is designed to allow a single machine instruction to operate on lots of different data sets at the same time. If you have lots of data that all have to be processed in the same way, than this can be a very fast way of carrying out processing. An array processor is known as a Single Instruction Multiple Data computer (SIMD). Weather forecasting and airflow simulations around planes, for example, are very good examples where array processing might be useful.

38.9 Maths co-processors

Another kind of processor worth mentioning is the Maths co-processor. This is a specialised processor that works in addition to the main processor. It has large registers and specialist functions that can manipulate big floating-point numbers easily. Whilst it is doing this, the main CPU is free to work on other things. Maths co-processors reduce the time that CPUs have to spend working on floating-point numbers, a task which they can do but not in one go – they are not specifically designed for that.

Q1. What are the characteristics of a Von Neumann processor?
Q2. State three busses in the CPU.
Q3. What does an I/O controller do?
Q4. Define RISC and CISC computers and make one comparative statement about them.
Q5. What does the Program Register do?
Q6. How are the registers used to fetch an instruction from the main memory?
Q7. How does a conditional jump instruction know whether to jump to a different part of a program or not?
Q8. What is a 'Von Neumann bottleneck'?
Q9. Describe briefly how parallel processing works.
Q10. Describe how a pipeline processor works.

Chapter 39 - Floating-point numbers

39.1 A reminder of some binary systems

Before you begin this section, you should completely review the work done in previous chapters relating to binary numbers and binary arithmetic. Following a review of these chapters, you should know that when you want to select a numbering system you need to ask yourself a number of questions.

- What range of numbers do you need to represent?
- Do you need to represent negative numbers?
- Do you need to do any arithmetic on the numbers?

The **main** point about any numbering system that we are interested in, however, is that we can define each of the bits in any byte in any way at all that we want to!

39.2 Fixed-point representation

We could define a byte so that we can represent fractions! A very simple way of achieving this is to fix the decimal point in a particular position. Let's set up our byte so we can do this. Some bits will represent whole numbers and some bits will represent fractions. Also notice that the decimal point isn't actually stored; it doesn't take up any of the bit positions. We 'know' where the decimal point is only because we know how we've defined each bit position.

16	8	4	2	1	1/2	1/4	1/8

This system is known as the 'Fixed-point' system. How is it used? Here are some examples.

- 0110 1001 is 8 + 4 + 1 + 1/8 = 13 $^1/_8$
- 0001 0110 is 2 + 1/2 + 1/4 = 2 $^3/_4$
- 1000 1010 is 8 + 1 + 1/4 = 9 $^1/_4$

This system is fine if you know in advance that all of your data is a fixed size, within a fixed range. If a group of data is not like that then you will have great trouble deciding where to put the decimal point! In fact, you will need a different system!

If you fully understand two's complement and fixed-point numbers and especially the point made that we can do what we like with our binary bits then you are ready to tackle floating-point numbers!

39.3 Floating-point numbers

So far, and staying with our byte, we know that in two's complement form, we can represent integers (whole numbers) from +127 to -128. We can't represent real numbers using this system. We can represent real numbers using fixed-point representation but it is only useful if the data is predictable and within a certain range. What we need is to be able to represent a wide range of real numbers.

Switching to decimal for a moment, you can represent these real decimal numbers in a different way. Look at these examples.

- 325.5 can be represented as 0.3255×10^3
- 1050.23 can be represented as 0.1050×10^4
- 478934.52 can be represented as 0.47893452×10^6
- 0.005 can be represented as 0.5×10^{-2}
- 0.000421 can be represent as 0.421×10^{-3}
- 0.00005 can be represented as 0.5×10^{-4}

So you can have a number like 5000.23 and represent it as 0.500023×10^4. The way of representing numbers in different forms involves moving (or 'floating') the decimal point to a new position.

The number part e.g. 0.500023 in the above example is called the '**mantissa**' while the number above the 10, the 4 in this case, is called the '**exponent**'. So, to represent a real number in this form, you need to know 2 numbers - the mantissa and the exponent.

Note that all these numbers are positive. With positive numbers, we always get the mantissa so that it lies between zero and one. In fact, you can represent a number in all sorts of ways. For example, 3000 could be represented as 0.03×10^5, or 0.3×10^4, or 3×10^3 and so on. For reasons that will be explained later, always get the mantissa of denary positive numbers to lie between 0 and 1. The mantissa of positive binary numbers should always begin with '01'.

In the following examples, we are not using two's complement (yet). Study the following:

- 1001 can be represented as 0.1001×2^4 ----- This is the same as 0.1001×2^{100}
- 101111 can be represented as 0.101111×2^6 ----- This is the same as is 0.101111×2^{110}
- 1101111 can be represented as 0.1101111×2^7 ----- This is the same as is 0.1101111×2^{111}
- 0.00111 can be represented as 0.111×2^{-2} ----- This is the same as is 0.111×2^{-10}
- 0.000111 can be represented as 0.111×2^{-3} ----- This is the same as is 0.111×2^{-11}

Note that all these numbers are positive. With positive numbers, we always get the mantissa to start in the form "01". Interestingly, when we look at negative real numbers, we always get them to begin with "10". When we get a number into one of these forms (01 or 10), we say that the number is in its 'normalised form'. As already said, this is explained later.

39.4 Positive binary floating-point number representation using one byte

We will now look at some examples of binary floating-point numbers. Remember three things:

1) Both the mantissa and exponent are **ALWAYS** two's complement numbers.
2) The decimal point is there in the mantissa, but not actually represented using a bit.
3) The decimal point **ALWAYS** goes between the first two left hand digits!

In our one byte, we will use 5 bits for the mantissa and 3 for the exponent.

Example 1. Convert this binary floating-point number into decimal: 01101010

1) Write down the mantissa. It's 01101
2) Insert the decimal point between the first 2 digits. That gives us 0.1101
3) This is a positive normalised number because the left-most 2 bits are 01.
4) The exponent is 010
5) This is positive because the left-most bit is a zero.
6) The exponent equals the denary value +2.
7) We must move the decimal point in the mantissa 2 places to the right. We go from 0.1101 to 011.01
8) This is a fixed-point binary number. The digits on the left of the decimal point give us the whole number part whilst the digits on the right of the decimal point gives us the fraction part. We can get rid of redundant zeros, to give us 11.01
9) Converting this fixed-point binary number into a denary number gives us 3.25

Example 2. Convert this binary floating-point number into decimal: 01010101

1) Write down the mantissa. It's 01010
2) Inserting the decimal point gives us a mantissa of 0.1010
3) We know it's a positive normalised number because the two digits on the left are 01
4) The exponent is 101
5) This is a negative number because the left hand digit is a one.
6) Converting this two's complement number into a negative binary number gives us -(011) or -3 in denary. If you are not sure about this step, refer back to the examples given in the section 'Getting back to a negative denary number'. The decimal point in the mantissa now needs to be moved three places to the *left*. (Move left because the exponent is negative.)
7) The mantissa goes from 0.1010 to 0.0001010
8) Removing redundant zeros gives us 0.000101
9) Converting this number gives us 1/16 + 1/64, or 0.078125

39.5 A note on dropping zeros

Think of the denary number 248.65 If you wrote this down, or put it into a calculator as 000248.65000, the unnecessary zeros would be dropped. That means that the zeros in front of the left-most non-zero digit on the left of the decimal point would go.

In the above example, the 000248 would become 248. It also means that any zeros after the last non-zero digit on the right of the decimal place would go. In the above example, the .65000 would become .65

- The number 034 is exactly the same as 34
- The number 0004563 is exactly the same as 4563
- 0.500000 is the same as 0.5
- -0.00600 is the same as -0.006
- 000248.65000 is exactly the same as 248.65

It is the same in binary.

- -(0110011.100) is the same as -(110011.1)
- 0001.1000 is the same as 1.1
- 1000110.000010000 is the same as 1000110.00001
- -(001.0100) is the same as -(1.01)

TASK 1 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number 01100011 into decimal by filling in the following table:

STEP 1	This number is positive. I know that because:
STEP 2	The mantissa (with the decimal place inserted) is:
STEP 3	The exponent is 011 which is positive because:
STEP 4	The denary equivalent of 011 is:
STEP 5	The decimal place in the mantissa must be moved LEFT/RIGHT because:
STEP 6	The mantissa now looks like this:
STEP 7	Removing redundant zeros, the mantissa now looks like this:
STEP 8	The final denary answer is:

TASK 2 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number 01110111 into decimal by filling in the following table:

STEP 1	This number is positive. I know that because:
STEP 2	The mantissa (with the decimal place inserted) is:
STEP 3	The exponent is 111 which is negative because:
STEP 4	The denary equivalent of 111 is:
STEP 5	The decimal place in the mantissa must be moved LEFT/RIGHT because:
STEP 6	The mantissa now looks like this:
STEP 7	Removing redundant zeros, the mantissa now looks like this:
STEP 8	The final denary answer is:

Converting denary numbers into normalised binary floating-point numbers is a reverse of what we have just been doing.

Example 3. Convert 5.5 into a one byte normalised floating-point number, 5 places for the mantissa and 3 for the exponent.

1) Convert 5.5 into a fixed-point number. You get 101.1
2) This is a positive number. It must begin with 0.1 so we must move the decimal point to the left until we get 0.1
3) Using 5 places for the mantissa, we now have 0.1011 (NOTE: If you now had a mantissa that was *not* five digits long, simply add extra zeros to the end of the number!)
4) How many places would you need to move the decimal point to get back to 101.1? The answer is 3 places to the right! So the exponent, using 3 digits, is 011
5) Putting mantissa and exponent together, the answer is 01011011

Example 4. Convert 3.5 into a one byte normalised floating-point number, 5 places for the mantissa and 3 for the exponent.

1) Convert 3.5 into a fixed-point number. You get 11.1
2) This is a positive number. It must begin with 0.1 so we must move the decimal point to the left until we get 0.1
3) Using 5 places for the mantissa, we now have 0.111 There are only four digits here and our mantissa needs 5! Add a zero to the end of the number to give 0.1110
4) How many places would you need to move the decimal point to get back to 11.1? The answer is 2 places to the right! So the exponent, using 3 digits, is 010
5) Putting mantissa and exponent together, the answer is 01110010

TASK 3 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number 2.25 into a floating-point number using the following table to guide you:

STEP 1	Converting 2.25 into a fixed-point number gives me:
STEP 2	The mantissa (with the decimal place inserted) is 5 digits long and must begin 0.1 The mantissa is:
STEP 3	To get the mantissa back to the original fixed-point number, the decimal place must be moved ____ places to the LEFT / RIGHT.
STEP 4	The exponent, which is 3 digits long, is:
STEP 5	Putting the mantissa and exponent together, the final answer is:

TASK 4 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number 1.5 into a floating-point number using the following table to guide you:

STEP 1	Converting 1.5 into a fixed-point number gives me:
STEP 2	The mantissa (with the decimal place inserted) is 5 digits long and must begin 0.1 The mantissa is:
STEP 3	To get the mantissa back to the original fixed-point number, the decimal place must be moved ____ places to the LEFT / RIGHT.
STEP 4	The exponent, which is 3 digits long, is:
STEP 5	Putting the mantissa and exponent together, the final answer is:

39.6 Negative binary floating-point number representation using one byte

Now let's look at some negative numbers. These are some more steps in dealing with negative numbers than in dealing with positive numbers but it is still a mechanical process. Work through the following examples carefully, taking note of each step.

Remember! A normalised negative binary floating-point number always begins 10

Example 5. Convert 10010010 (Using 5 bits for the mantissa and 3 for the exponent).

1) The mantissa is a normalised negative number because the left-most bits are 10
2) Write down the mantissa with the decimal point between the first two digits: 1.0010
3) Because it's negative, there is an extra step. You must convert it into a negative binary number that is *not* in two's complement form. 1.0010 therefore becomes -(0.1110) Notice that the decimal place stays where it is for the moment. Refer back to the 'Getting back to a negative denary number' section if you need help.
4) The exponent is 010, which is the same as +2.
5) We therefore need to move the decimal point two places to the right.
6) The mantissa goes from -(0.1110) to -(011.10)
7) Getting rid of redundant zeros gives us -(11.1)
8) Converting this fixed-point number gives us -(3.5) or simply -3.5

Example 6. Convert 10111110 (Using 5 bits for the mantissa and 3 for the exponent).

1) The mantissa is a normalised negative number because the left-most bits are 10
2) Write down the mantissa with the decimal point between the first two digits: 1.0111
3) Because it's negative, there is an extra step. You must convert it into a negative binary number that is *not* in two's complement form. 1.0111 becomes -(0.1001) Notice that the decimal place stays where it is for the moment.
4) The exponent is 110, which is the same as -2.
5) We therefore need to move the decimal point two places to the *left*.
6) The mantissa goes from -(0.1001) to -(0.001001)
7) There aren't any redundant zeros in this case.
8) Converting this fixed-point number gives us -(1/8 + 1/64) or simply -0.140625

TASK 5 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number 10111010 into decimal using the following table to guide you:

STEP 1	This number is negative. I know that because:
STEP 2	The mantissa (with the decimal place inserted) is:
STEP 3	The mantissa converted into a negative binary number that is not in two's complement form is:
STEP 4	The exponent is:
STEP 5	The decimal point in the mantissa must now be moved ___ places to the LEFT / RIGHT.
STEP 6	The mantissa now becomes:
STEP 7	Removing unnecessary zeros gives:
STEP 8	Converting from a negative fixed-point binary number into denary gives the final answer as:

TASK 6 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number 10011111 into decimal using the following table to guide you:

STEP 1	This number is negative. I know that because:
STEP 2	The mantissa (with the decimal place inserted) is:
STEP 3	The mantissa converted into a negative binary number that is not in two's complement form is:
STEP 4	The exponent is:
STEP 5	The decimal point in the mantissa must now be moved ___ places to the LEFT / RIGHT.
STEP 6	The mantissa now becomes:
STEP 7	Removing unnecessary zeros gives:
STEP 8	Converting from a negative fixed-point binary number into denary gives the final answer as:

Converting negative denary numbers into normalised binary floating-point numbers is a reverse of what we've been doing.

Example 7. Convert -5.5 into a normalised binary floating-point number using one byte, with 5 bits for the mantissa and 3 bits for the exponent.

1) This is a negative number. Convert -5.5 into a fixed-point number in two's complement form.
2) Now +5.5 is 101.1
3) Pack the *front* of this number with excess zeros to make the mantissa the correct size. You now get 0101.1
4) The two's complement form of this is 1010.1
5) The normalised form of 1010.1 is 1.0101. (Just move the decimal point left to the left-most 10 pair, and put the decimal place between the one and the zero).
6) To get the decimal place of the normalised form back to its pre-normalised form, the decimal point needs to be moved 3 places to the right. The exponent therefore is 011
7) Putting all this together gives 10101011

Now try reconverting this normalised binary floating-point number using the method you saw in examples 16 and 17 and see if you can get -5.5

Example 8. Convert -1.25 into a normalised binary floating-point number using one byte, with 5 bits for the mantissa and 3 bits for the exponent.

1) This is a negative number. Convert -1.25 into a fixed-point number in two's complement form.
2) Now +1.25 is 1.01
3) Pack the front of this number with excess zeros to make the mantissa the correct size. You now get 001.01
4) The two's complement form of this is 110.11
5) The normalised form of 110.11 is 11.011 (Just move the decimal point left to the left-most 10 pair, and put the decimal place between the one and the zero).
6) Now there is another step here. We need to discard the excess 'ones' in *front* of the 1.0 pair, just like we would discard excess 'zeros' in positive numbers. Excess zeros are then added to the *end* of the number to make the mantissa the correct size. In this example, 11.011 becomes 1.0110
7) To get the decimal place of the *normalised form* back to its pre-normalised form, the decimal point needs to be moved 1 place to the right. The exponent therefore is 001
8) Putting all this together gives 10110001

Now reconvert this normalised binary floating-point number using the method discussed earlier and see if you can get -1.25

TASK 7 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number -1.5 into a floating-point number using the following table to guide you:

STEP 1	+1.5 in binary is:
STEP 2	Packing the front of the binary answer in STEP 1 with zeros to get the mantissa to the correct size (5 digits) gives:
STEP 3	The two's complement form of the answer in STEP 2 is:
STEP 4	Moving the decimal point in the answer to STEP 3 to the correct place for the normalised form gives us:
STEP 5	Removing excess 1s from the front of the number and adding extra zeros at the back of the number to get the normalised form gives us:
STEP 6	To get the decimal place of the normalised form answer in STEP 5 back to its pre-normalised position, the decimal point needs to be moved __ places to the LEFT / RIGHT.
STEP 7	The exponent is therefore:
STEP 8	Putting the mantissa and the exponent together gives the final answer of:

TASK 8 - A one-byte floating-point numbering system uses 5 places for the mantissa and three places for the exponent. Convert the number -4.0 into a floating-point number using the following table to guide you:

STEP 1	+4.0 in binary is:
STEP 2	Packing the front of the binary answer in STEP 1 with zeros to get the mantissa to the correct size (5 digits) gives:
STEP 3	The two's complement form of the answer in STEP 2 is:
STEP 4	Moving the decimal point in the answer to STEP 3 to the correct place for the normalised form gives us:
STEP 5	Removing excess 1s from the front of the number and adding extra zeros at the back of the number to get the normalised form gives us:
STEP 6	To get the decimal place of the normalised form answer in STEP 5 back to its pre-normalised position, the decimal point needs to be moved __ places to the LEFT / RIGHT.
STEP 7	The exponent is therefore:
STEP 8	Putting the mantissa and the exponent together gives the final answer of:

39.7 Further notes on normalisation

Numbers can be represented in different ways using floating-point notation. To illustrate this point using decimal, suppose you have this number: 45379510 How could this be represented using the floating-point system?

453795×10^2 or 4537×10^4 or 45×10^6 or $0.0045379510 \times 10^{10}$ and so on.

Look at this example, which uses 10 places after the decimal point: $0.0045379510 \times 10^{10}$ You could write it like this $0.00453795 \times 10^{10}$ if you only had 8 places after the decimal point. The problem with this is that you have lost some of the accuracy of the number. If you only had 8 places after the decimal point, you might prefer this: 0.45379510×10^8 You have kept the accuracy and still used only 8 places after the decimal point. Also, by having a smaller number of places after the decimal point, you have increased the number of places you can use for the exponent, thereby increasing the range of numbers you can represent.

How did we keep the accuracy for the same number of mantissa places? We did it by getting rid of 'leading zeros'. In other words, we didn't write down 0.00453 etc, we wrote down 0.453 etc. We ended up with the same number, improved accuracy

and still used only 8 places for the mantissa. The process of ensuring the maximum accuracy of a number for a given mantissa size is known as **'normalisation'**.

- Normalisation ensures that maximum accuracy of a number for a given number of bits.
- It also ensures that each number has only one possible bit pattern to represent it!

Some examples:

- 3004×10^4 normalised is 0.3004×10^8 (Check to see if these numbers are the same.)
- 0.0005×10^2 normalised is 0.5×10^{-1} (Check to see if these numbers are the same.)

39.7.1 Normalisation of binary numbers

Converting binary numbers into their normalised form is a similar process.

Consider the binary number 00001000 where there are 5 bits for the mantissa and 3 for the exponent. We have to normalise this number.

- The mantissa is 00001 (Think of it as 0.0001)
- We move the decimal 3 places to the right so it begins 0.1
- This gives us 01000 if we remove the decimal point and leading zeros and add following zeros.
- The exponent must be -3, to get the decimal point back to where it was to start with.
- -3 is 101 as a two's complement number if you have 3 digits for the exponent.
- The normalised version of 00001000 is therefore 01000101

Consider the binary number 00010011 where there are 5 bits for the mantissa and 3 for the exponent. We have to normalise this number.

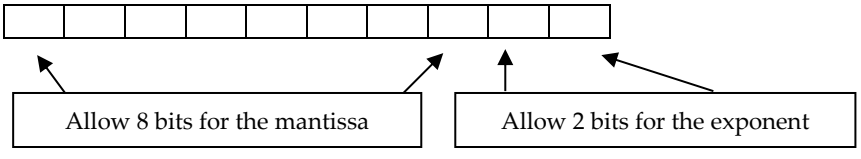
- The mantissa is 00010 (Think of it as 0.0010)
- We move the decimal 2 places to the right so it begins 0.1
- This gives us 01000 if we remove the decimal point and leading zeros and add following zeros.
- The exponent was 011 (move 3 to the right).
- 3 places to the right and 2 places to the left means overall, the exponent should be one place to the right. This is represented as 001
- The normalised version of 00010011 is therefore 01000001

Consider normalising the binary number 0000010100 000010 There are 10 bits for the mantissa and 6 for the exponent.

- The mantissa is 0000010100 (Think of it as 0.000010100)
- We move the decimal point 4 places to the right so it begins 0.1 and to get it back to where it was, we must move it 4 places to the left.
- This gives us 0101000000 if we remove the decimal point and leading zeros and add following zeros.
- The exponent was 000010 (move 2 places to the right).
- 2 places to the right and 4 places to the left means overall, the exponent should move 2 places to the left.
- -2 is 111110 as a two's complement number.
- The normalised version of 0000010100 000010 is therefore 0101000000 111110

39.8 Floating-point numbers - accuracy verses range

Imagine that you have ten boxes in which to represent a denary number in scientific notation. You have to decide how many places to allow for the sign, how many for the mantissa and how many for the exponent. The choices that you make will affect the range and accuracy of the numbers you can hold. Let's look at some examples to see what we mean. Let us invent a system for storing the digits of a scientific number. Suppose we decide on the following, where each box can hold one number:



39.8.1 What is the biggest number it can hold?

0	9	9	9	9	9	9	9	9	9

39.8.2 What happens if you want to represent a really massive number with an exponent of 345?

You can't, because you have only allowed two places for the exponent. By allowing only two places for the exponent, we are limiting the size of the numbers we can represent.

39.8.3 What happens if you want to represent a number with a mantissa of 0.99999994?

You can't, using our current system because we have only allowed 8 places for the mantissa. By doing this, we have limited the accuracy of the numbers we can hold. We would have to round any numbers that need a mantissa of more than 8 places.

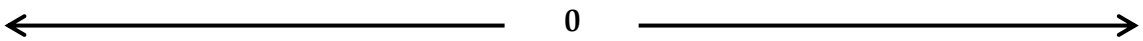
Of course, you can decide to allow 3 places for the exponent. That means you will only have 7 places for the mantissa. What is the result of this? It means we can represent much bigger numbers (up to an exponent of 999) but less accurate numbers (up to 0.999999 this time). Conversely, we could allow just one place for the exponent and 9 places for the mantissa. That means we are really limiting the range of numbers we can represent (an exponent of only 10^9 (instead of 10^{99}) but now we can represent numbers with a much greater accuracy, to 0.99999999

The same argument can be applied to binary numbers. You will always have a fixed number of bits, perhaps 2 bytes (16 bits) for example. You need to make a decision about what the range of numbers is going to be. What will be the smallest number you need to represent and what will be the largest? This will guide you in deciding how many bits you need for the exponent.
You also need to decide how accurate you need the number you're storing. In other words, how many decimal places you want for the number. This will help you decide how many bits to allow for the mantissa.

39.9 The largest and smallest values

Imagine you have a byte used to hold a floating-point number. In our byte, we will allow 5 bits for the mantissa and 3 bits for the exponent. Remember, both the mantissa and the exponent are always held as two's complement numbers.

When thinking about the largest positive number, the smallest positive number, the largest negative number and the smallest negative number, it is often useful to think about a number line.



Bigger negative numbers Bigger positive numbers

39.9.1 The largest positive number

- The largest number that can be held in our byte is when both the mantissa and the exponent are the largest positive numbers that can be held.
- The largest mantissa that can be held is 01111. (Note that you must assume the mantissa is normalised, and so positive numbers always begin with 01).
- The largest exponent that can be held is 011. (Remember, the zero here represents -4)
- The largest number that can be held is therefore 0.1111×2^{011}
- This is the same as 111.1, or 7.5 in decimal.

39.9.2 The smallest positive number

- The smallest positive number that can be held is when the mantissa is the smallest positive number and the exponent is the largest negative number.
- The smallest positive mantissa is 01000.
- The largest negative exponent is 100
- The smallest positive number that can be held is therefore 0.1000×2^{100}
- This is the same as 0.00001, or 1/32, or 0.03125

39.9.3 The largest negative number

- The largest negative number that can be held (the negative number that is furthest from zero) is when the mantissa is the biggest negative number that can be held and the exponent is the largest positive number possible.
- The biggest negative number you can hold in the mantissa is 10000 (Remember all negative numbers begin with 10).
- The biggest positive number the exponent can hold is 011.
- The largest negative number is therefore 1.0000×2^{011}
- This is the same as -1.0000×2^{3} or -1000, or -8.

39.9.4 The smallest negative number

- The smallest negative number that can be held (the negative number that is closest to zero) is when the mantissa holds the smallest negative number and the exponent holds the largest negative number.
- The smallest negative number the mantissa can hold is 10111 (Remember that the mantissa must begin with 10 for normalised negative numbers).
- The largest negative number the exponent can hold is 100.
- The smallest negative number that this byte can hold therefore is 1.0111×2^{100}
- This is the same as -0.1001×2^{-4} or -0.00001001, or -0.03515625

39.9.5 To summarise

You have a fixed number of bits. You have to decide how many bits to use for the mantissa and the exponent.

- The more bits you allow for the exponent, the greater the range of numbers you can represent - the down side being that the accuracy of the mantissa is reduced.
- The fewer bits you allow for the exponent, the smaller the range of numbers you can represent, but the good news is you have more bits available for the mantissa - so the number you put in it can be more accurate.

Q1. Describe using a suitable example how binary numbers can be represented using fixed point notation.
Q2. What is meant by a mantissa?
Q3. What is meant by an exponent?
Q4. Convert 01000010 into decimal. This normalised floating-point number uses five bits for the mantissa and three for the exponent.
Q5. Convert 01100001 into decimal. This normalised floating-point number uses five bits for the mantissa and three for the exponent.
Q6. Convert 4.5 into a normalised floating-point number using five bits for the mantissa and three for the exponent.
Q7. What is the largest number that can be held in a normalised binary floating-point byte that uses four bits for the mantissa and four bits for the exponent?
Q8. How many bits are there is two bytes?
Q9. Give two reasons why binary floating-point numbers are normalised.
Q10. Explain with an example the trade-off between accuracy and range in a binary floating-point number.

Chapter 40 - More on linked lists

40.1 Static verses dynamic implementation of data structures

We have already seen that the method used to structure data in a computer's memory can be classified in one of two ways, either as a static data structure or a dynamic data structure.

40.1.1 Static data structures

A static data structure is one whose size is fixed. An example of a static data structure is an array. To use an array, the programmer needs to state a number of things:

- the array a name
- the data type the array will hold
- the size of the array, state how many data items it can hold.

It is relatively easy (compared to dynamic data structures) to write the code that sets up an array. This type of structure allows fast access to the data held in it because you can use random access (hashing algorithms) or index sequential algorithms with arrays. In addition, when an array is defined, space is reserved in memory for it when the program is compiled - space for the data in the array will therefore always be available.

- The size of the array is fixed before it is used. If the array has no data in it, it will still be taking up the space in memory that was allocated to it when it was compiled. This is not a very efficient use of memory.
- Sometimes, estimating the array size needed is difficult.
- Once the array is full, that is it! You can't put any more data into it.

40.1.2 Dynamic data structures

A dynamic data structure (such as linked lists, stacks, queues and trees) is one whose size can vary, depending upon the data storage needs of the program. With these types of structure, you don't need to fix the size in advance. The available RAM is used very efficiently because only the required space for actual data in the structure is used while the rest of the RAM is available for other applications.

- These structures are more complicated to program than arrays.
- Some structures can be slow. For example, linked lists only allow serial access not random access to data. If you had to search serially through one million records to find Mr Zebb's record, it would take a long time!

40.2 Linked lists

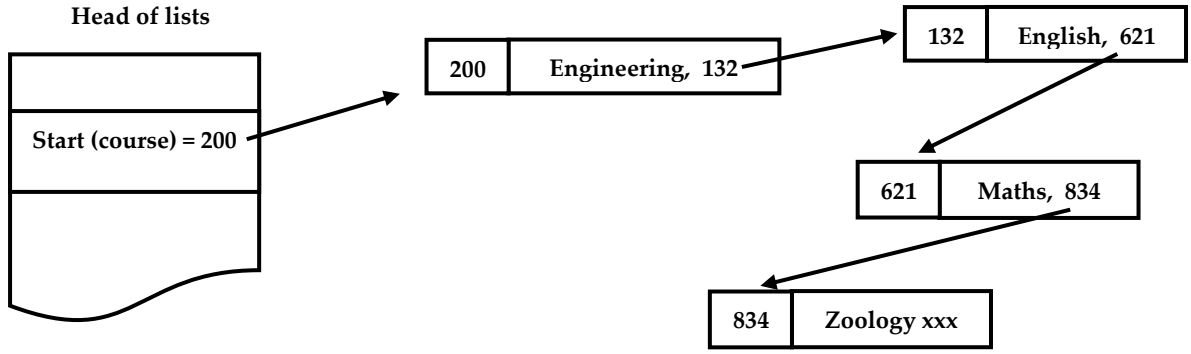
We introduced this dynamic data structure in a previous chapter. A linked list is a way a computer can keep a set of data in a predefined sequence. It is usually given as an example of a 'dynamic data structure' because its size can shrink and grow, unlike a static data structure such as an array. However, linked lists can also be implemented within an array structure. A truly dynamic linked list data structure would use pointers to an area of free space in the main memory called known as the 'heap'. This is a record of free memory locations that can be used by any program as they need them. When a new location is required, an address can be fetched from the heap. When an item is deleted from a list, the memory location can be returned to the heap for use by other applications. We have discussed the heap in detail in a previous chapter.

As mentioned before, we will look at algorithms that use a linked list in an array. It is a dynamic data structure, in that it can grow and shrink, but is not truly dynamic because the structure is held in a table (an array), so its size is predefined - there are limits to how much it can grow by.

Just to remind you of what we said in an earlier chapter, the key features of linked lists are:

- the head of each sequence of data is held in a 'Head of lists' area of memory
- the head points to a node, where the first piece of data (assuming there is one) can be found.
- each node is made up of a piece of data and a pointer
- the pointer points to the memory location where the next node can be found
- the end of a list is signalled by a 'null pointer'. (We've used xxx.)

Do note that the data in a linked list can be stored all over the memory. They don't have to be stored together in one block of memory. In the next example, memory locations 200, 132, 621 and 834 have been used. This leads to a very efficient use of memory because you can use up all of the gaps between other applications and data files! The downside of this, however, is that they can be slow structures to search through, not only because the data isn't together but also because you have to serially search through a linked list. That means you have to start at the beginning of the list and work your way through the nodes to find any particular item. In a static structure such as an array, you can access data items directly, without going through all of the other items. For example, here is a linked list of degree courses held alphabetically.



A linked list of degree courses held alphabetically.

40.3 Creating a new linked list from an old one

So far, we know that

- linked lists are **dynamic** data structures
- each item in the list is known as a **node**
- each node has a data part and a **pointer** part to it
- linked Lists must have a start pointer held in the **Head of Lists**
- linked lists must have a **Null pointer**
- the **Null pointer** signals the end of the list.

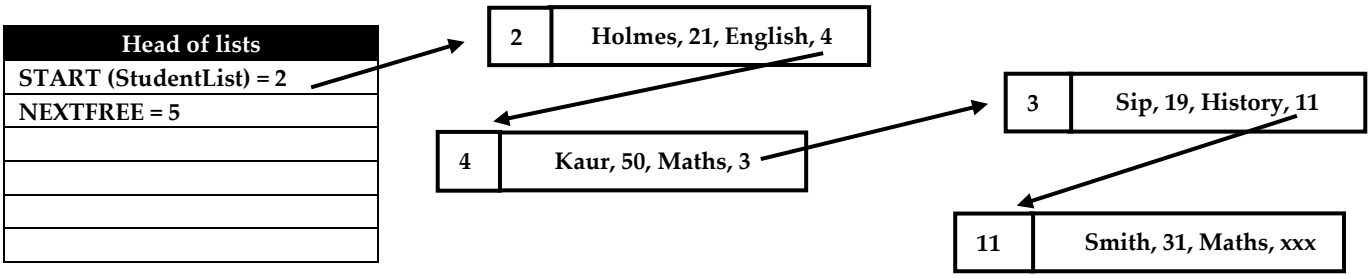
A Linked List is a data structure. You set up the structure by writing programming code. As with any data structure in software, there will be times when you want to do some operations on it. These can include finding a piece of data in the structure, inserting a piece of data, deleting it, printing out all or some of the data items in a structure, re-organising the structure to create new structures and so on.

Consider this example. Suppose I want to keep a Linked List of these students (Name, Age, Course) in ascending alphabetical (StudentList) order: Smith, 31, Maths; Holmes, 21, English; Sip, 19, History; Kaur, 50, Maths. Here is one way of representing what I want to do (Note that I have picked any 'free' memory addresses that were available to use, even though they weren't next to each other):

Memory address	Name	Age	Subject	Pointer
1				5
2	Holmes	21	English	4
3	Sip	19	History	11
4	Kaur	50	Maths	3
5				10
10				xxx
11	Smith	31	Maths	xxx

START=2 NEXTFREE=1

Another (easier?) way of showing this is diagrammatically:



Notice the introduction of a new pointer - the NEXTFREE pointer. It simply points to the next available node, the next memory location that could be used to store a data item if I needed to add an item to the linked list. If you start deleting nodes, the NEXTFREE pointer will become important because it will allow you to reuse deleted nodes, rather than leaving unrecoverable space in the data structure. In fact, it is possible to have a linked list of all the free spaces available! So you actually have two linked lists in a data structure - one for the data and one for the free nodes. If I wanted to set up a linked list of free space, I would have the start of the free space pointer in the head of lists pointing to memory location 1. Memory location 1 would point to 5. 5 would point to the next free space, and so on until the null point is reached. We'll see this again later in this chapter.

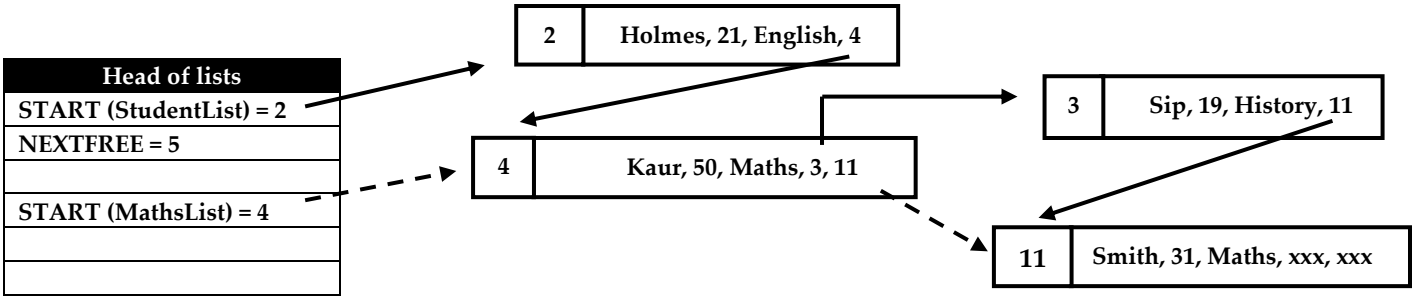
In the 'linked list of students' example, I could print out quickly a list of the students in alphabetical order. I can do that because I can follow an alphabetical path. However, I can't easily and quickly get back all the pupils who are doing a Maths course. I don't have a path to follow for that. I don't have a linked list of Maths pupils. I need to create one from my existing list! How do you describe how this can be set up in software, in pseudo-code or otherwise? The idea is that you create a new head of list in your 'Main Index' table and then you create a new set of pointers to follow!

- You get the pointer from the head of the linked list you want to create a new one from.
- You check the list isn't empty by seeing if it is the null pointer.
- You follow the pointer to the first node.
- You test the data in that node.
- If it needs to be part of the new list, then you add it to the new list, adjusting pointers as necessary.
- Else you follow that node's pointer to the next node, and repeat, until you come to the end of the list.
- The final job is to give the last node in the new list the null pointer.

A pseudo-code way of writing this might be as follows:

```
BEGIN
NULLPOINTER=FALSE
CREATE new entry in Head of Lists for 'new list'
GET POINTER to first node of 'existing list' from Head of Lists
IF POINTER=XXX
        NULLPOINTER=TRUE
        REPORT "LIST EMPTY"
ENDIF
WHILE (NULLPOINTER is FALSE) DO
        IF (data at node) from 'existing list' belongs in 'new list' THEN
                ADD it to 'new list'
                ADJUST 'new list' pointers
                FOLLOW pointer to next node in 'existing list'
                GET data
        ENDIF
        IF POINTER at next node = XXX
                NULLPOINTER=TRUE
        ENDIF
ENDWHILE
SET POINTER of last node in 'new list' to Null.
END.
```

A diagrammatic representation of the second list would look like this:



Creating a new list.

If you want a list of all pupils, you get the START (StudentList) and follow the **first** pointers. If you want a list of all maths pupils, you get the START (MathsList) and follow the **second** pointers.

40.4 Adding an item to a linked list

Imagine we have a linked list of car manufacturers that we want held in alphabetical order. Imagine that we have decided to keep this information in a linked list. So far, we have entered 4 items. Our linked list looks like this:

Address	Name	Pointer
1	BMW	4
2	Skoda	xxx
3	Ford	2
4	Fiat	3
5		
6		
7		

START = 1

Suppose we want to delete the item Ford, how could we do it? We would simply need to move pointers around. The actual data item Ford will still be in the table! However, no other data item will actually point to it! Refer back to chapter 18 if you are stuck!

40.5 More on managing free space

In the previous example, Ford was deleted. That made memory location 3 free. The question now is how to reclaim that memory location. How can we signal to the computer that memory location 3 is now available to use? Managing free space in a linked list is important. We need to keep track of which memory locations are free, or become free, so that we can add new data items to the linked list. If a data item is removed from a memory location in a linked list then we must not only remove it from the linked list of data but we must also *add* it to a linked list of free space! If we didn't do this, memory locations would be lost to the computer! This would not be a very efficient use of our RAM.

40.5.1 An example of managing free space

We will use the example of car manufacturers in alphabetical order. The empty linked list looks like this:

Address	Name	Pointer
1		2
2		3
3		4
4		5
5		6
6		7
7		xxx

START = null
NEXTFREE = 1

Notice that even though there is nothing in the linked list, a linked list of all the available free space exists! After inserting 4 car manufacturers, we have:

Address	Name	Pointer
1	BMW	4
2	Skoda	xxx
3	Ford	2
4	Fiat	3
5		6
6		7
7		xxx

START = 1
NEXTFREE = 5

Can you see that there are two linked lists, one for the data items and one for the free space? Notice that there is a null pointer for the end of the linked list of data items and also a null pointer for the end of the linked list of available free space!

40.6 Inserting data

Now we understand how free space is managed, we are in a position to write some algorithms to insert data. We will also need to consider **special cases** in the algorithm. For example what happens when there is no more free space available, what happens if the item is the first one in the list to be inserted and what happens if the linked list is completely empty, waiting for the first item to be inserted. Suppose we need now to insert Saab into the car manufacturer's list. The basic algorithm would be:

- Put Saab in the position pointed to by NEXTFREE.
- Change NEXTFREE so that it contains the next value in the linked list of free items.
- Starting at the head of lists, determine where Saab should fit in to the linked list of data.
- Change Saab's pointer so that it points to Skoda.
- Change Peugeot's pointer so that it points to Saab.

40.7 Special cases

With any algorithm you write, you should always ask yourself what the special cases are and then add pseudo-code to your algorithm to take these into account.

1) What happens if there is no free space available? At the beginning of the algorithm, check that the free linked list is not empty. If it is, report an error message and then stop.
2) What happens if the item to be inserted is the first one in the list? Before you check where a new piece of data should go, check to see if the pointer at the HEAD OF LIST is the null pointer. If it is, then set the pointer of NEWDATA to be the null pointer and set the pointer at the HEAD OF LIST to point to NEWDATA, then stop.
3) What happens if an item to be inserted needs to go in the first position? Put the NEWDATA into the position pointed to by NEXTFREE. Change the HEAD OF LIST to point to NEWDATA. Make NEWDATA point to the second item.

The trick with remembering all of this is **not to try and remember algorithms!** You should draw yourself a little linked list, with a few data items in it. Then run through the process of inserting a new item, writing down each step in your algorithm as you go.

40.8 Deleting and amending a data item

Address	Name	Pointer
1	BMW	4
2	Skoda	xxx
3	Ford	2
4	Fiat	3
5		6
6		7
7		xxx

START = 1
NEXTFREE = 5

Suppose you want to delete Ford from the list of car manufacturers. The algorithm might look something like this:

- Follow the pointers until you find Ford.
- Change Fiat's pointer to point at Skoda.
- Change Ford's pointer to point to NEXTFREE.
- Change NEXTFREE to point to Ford.

40.9 Amending Data

The algorithm is:

```
IF list empty THEN
        report ERROR and STOP
ELSE
        FOLLOW links UNTIL item found
        AMEND data {don't change any pointers}
STOP
```

Q1. What is meant by a static data structure?
Q2. What pieces of information do you need to declare an array?
Q3. What is meant by a dynamic data structure?
Q4. Why is a linked list an example of a serial access data structure?
Q5. What is meant by a node in a linked list?
Q6. What is meant by the Head of Lists in a linked list?
Q7. What is meant by a null point?
Q8. What is an algorithm?
Q9. Apart from linked lists, state one other dynamic data structure.
Q10. Describe a special case you should consider when writing algorithms for linked lists.