Computer Hardware Module 2021 – 2022

Common First Year in Computing

(CW_KCCOM_C / CW_KCCSY_D / CW_KCCIT_B / CW_KCSOF_D / CW_KCSOF_B / CW_KCCYB_D / CW_KCCYB_B)

Course Details

Lectures (Theory):	1 hour per week throughout the year (Friday 11:15 in L117). Lecturer: David Kelly (kellyd@itcarlow.ie). Office: C202 (top floor, middle corridor, Science/Engineering)
Theory Assessment (50%):	Three one hour tests based mainly on lecture material, but some laboratory material will be assessed too.
Provisional timetable:	Assessment 1: Friday 12th November 2021 L117 09:00 Assessment 2: Friday 11th February 2022 L117 10:00 Assessment 3: Monday 25th April 2022 L117 10:00 **No big end of year exam.**
Laboratories (Practicals):	Two hour session per week per group (laboratory C305). A Group David Kelly Thursdays 15:00-16:45 B Group David Kelly Wednesdays 15:00-16:45 N.B. -> C Group David Kelly Thursdays 11:00-12:45 N.B. -> D Group David Kelly Wednesdays 11:00-12:45 E Group David Kelly Mondays 09:00-11:00 F Group David Kelly Wednesdays 09:00-11:00 Each laboratory practical will either have goals to be achieved or research/write-ups to be carried out, or a little of both
Lab. Assessment (50%):	All lab sheets will be corrected (depending on numbers), some in more detail than others. Labs chosen for more detailed correction are chosen at random and you will <u>not</u> be informed which is which. Therefore attendance at <u>all</u> labs is crucial. We will run "catch-up" labs again this year.
Repeat examinations:	Available in Autumn. Both a Theory and Laboratory Examination must be attended and these examinations are <u>very</u> difficult.
Notes:	Course material will be placed on your common drive.
Results:	Results will be disseminated through your common drive.

Suggested Textbooks

How Computers Work, 10th Ed., Ron White, 978-0-7897-4984-0, Que November 2014
Arduino Workshop, John Boxall, 978-1-59327-448-1, no starch press, 2013
Exploring Arduino, Jeremy Blum, 987-1-118-54936-0, Wiley, 2013
Adventures in Arduino, Becky Stewart, 987-1-118-94847-7, Wiley, 2015
PC Hardware in a Nutshell, 3e Thompson & Thompson 0-596-00513-X O'Reilly 2003
Inside the PC 8th ed Norton & Goodman, Sams
The Indispensable PC Hardware Book, Addison Wesley
Upgrading and Repairing PCs, Que

Punctuality, Attendance & Requisites

Please be on time for all lectures and laboratories. Punctuality in hardware laboratories is especially important. Most of these start with a brief introduction, in which important points and pitfalls are identified. If you are late, you may miss this information, and thus put yourself, your colleagues and the laboratory equipment at risk.

People who are consistently late will be refused admission.

Attendance is constantly monitored, as is punctuality.

Your attendance is closely considered by examination boards.

Many of our hardware laboratories will build upon material learned in previous laboratories. If you have not done the previous labs you will not be allowed to do any follow up labs. It is your responsibility to keep up to date.

Header information such as your name, login id, subgroup, HWL number etc. must be completed in full on every lab sheet, Lab sheets will not be corrected if any of this information is omitted

For reasons of safety and practicality, only one specific lab sheet may be done in any particular lab session.

Please switch off mobile phones and put them away. It is completely unacceptable, and extremely discourteous, to be taking or making calls in lectures and laboratories, and similarly for texts. The only time the use of a phone is acceptable in a laboratory is to photograph a hardware configuration that you are about to dismantle, to assist in its reconstruction. If, on a very rare occasion, internet servers are down, then access using a phone is acceptable. Using a calculator app is also acceptable.

Bring a pen, pencil and eraser into every laboratory session. A calculator might be useful from time to time. Every laboratory will be introduced to you by your supervisor. Pencil in any additional information to the relevant section of the lab sheet to remind you of these extra points/requirements when you get to that section. Erase them when you're done.

Always have a USB key with at least one Gigabyte of free space with you at every hardware practical. Many practicals will explicitly expect you to have your USB keys with you. (Deane's: 16GB for €10.00, 32GB for €14.00 approximately, September 2019).

Modern computer systems use colour coding quite extensively - ports and wiring are colour coded, for example. Coloured pens/pencils might be an asset. We'll also encounter colour coding when we introduce the Arduino microcontrollers – resistors, in particular, usually have their resistance values colour encoded.

Course Objectives

The principal objectives of your hardware course are:

- To understand the basic operation of a computer system;
- To be able to identify the significant components in a PC, and understand their function;
- To be able to dismantle and rebuild a PC, and other computing equipment;
- To be able to identify and remedy common problems with PC hardware;
- To be able to upgrade PC hardware;
- To understand the basics of assembly language on a PC system;
- To be able to build simple circuits and control them by programming Programmable Logic Controllers (e.g. Arduinos).

Experience has shown that a substantial number of our student intake are not good at following instructions and procedures.

This is not really surprising in the modern world. Most software is supplied without manuals, and their built-in help facilities are often not very good at all. Most hardware manuals are also quite poor. There is a disproportionate emphasis on safety, to the detriment of efficient operation of the hardware. Of course, safety is particularly important, but most of the safety instructions are motivated by avoidance of legal actions, more so than the safety of end users. Futhermore, many instruction manuals are often poor translations from other languages, and are frequently incomprehensible.

So understandably, most people tend to skip reading instructions altogether and just jump straight into using their new systems. They only refer to manuals when everything else fails. For this reason, and at the risk of being a little patronising (completely unintentional), we might add an additional objective:

- To be able to follow a set of instructions (or procedural steps) accurately and sequentially to achieve a particular goal;

If you expect to be able to tell a computer to carry out detailed sets of instructions to achieve some task (i.e. program it), it is not going to be very easy if you are incapable of doing the same thing yourself.

Plagiarism

The ethos of the common first year is to learn things through discovery. You will observe certain things as you work through your lab sheets. The lab sheets have been carefully crafted to facilitate this process and guide you in the right direction.

Every lab is designed to introduce a number of new topics and skills. Virtually everything you require is clearly specified in your lab sheets, and additional information will be presented at the start of most classes, which you can pencil into your lab sheets during the introduction.

So, in general, helping people in labs tends to be counterproductive. Of course, clarifying a point or two from time to time is fine, but if this happens too much then the course ethos is undermined.

Unfortunately, some of our students are happy (or even expect) to be given the answers and will not learn as well as they might. Some other students mistakenly think they are being helpful (or even important) by providing answers to these students.

Every student should follow the procedures laid out in the lab sheets and achieve the maximum intended benefit. Find your answers for yourself and don't undermine other student's learning process by supplying them with answers that they should have worked out for themselves.

Any work submitted must be your own, even if you are working in a team. If, for example, you take a program that has been written by somebody else and present it as your own achievement, then you are only fooling yourself.

In all cases of copying, both the person supplying the information and the person receiving it will be penalised – neither party will be informed. Copied work is very easy to identify, so don't be tempted. It is also possible that students involved in copying will be referred to their head of department and/or infringement panels, leading to severe sanctions.

Safety in Hardware Laboratories

When a computer or a computer peripheral is connected up to the mains electricity supply, there are potentially lethal voltages present inside the casing.

- Under no circumstances should a machine ever be opened while still connected to the mains;
- It should be noted that some hardware devices (such as an old CRT monitor) can store charges for many years after disconnection from the mains that may seriously injure, or even kill;

- Keep food and drink out of the laboratory and away from the computers. Furthermore, food and drink should not be brought into lecture theatres and classrooms.

If you are not used to dismantling and rebuilding mechanical and electronic equipment, then it can take a little time to learn how to do this - after a while you will get the knack of it. You'll discover that both sight and touch play important roles – your other senses are also important. In particular:

- Examine the equipment carefully before you start to dismantle it – determine which items need to be removed before other items can be extracted;
- Make copious notes of a device's initial configuration and record what connects to what;
- A phone's camera is useful for recording initial configurations, but don't be over reliant on it;
- As you dismantle lay items in a sequence on your work bench (including screws and other fixtures) – then everything you require to rebuild the equipment is queued correctly for you;
- As you rebuild things learn to align connecting parts correctly before joining them up– in general a minimal amount of pressure should ever need to be applied – if this isn't working for you then study the set up carefully – don't just apply more force (it usually destroys equipment). If you are having difficulties then ask your lab supervisor for assistance.

For two principal reasons, be very careful when handling components within the computer system.

- Many items have sharp edges or points that may pierce or slice your hands. Identify all hazards before doing any task. Safe handling will be demonstrated to you in the laboratory.
- Another reason is that our bodies often carry static charges, which, if allowed to discharge through components within the computer may destroy them. You will be shown how to discharge static charges safely whilst working on your computers. Learn to do this constantly.
- Never touch contact pins or strips in connectors or memory/expansion cards – no matter how clean your hands are, they leave a residue which corrodes metal and causes poor electrical contacts.

Keep your bags and other belongings that you don't need for the lab safely stowed away under the desks where you or your colleagues won't trip over them. Actually, it's preferable that you minimise the amount of baggage you bring into the laboratory – leave any excess in your locker. Don't allow chairs and other furniture to block access routes – park them out of the way when not using them.

Be careful with cables of all types. Ensure devices are never connected in such a way that their interconnecting cables pose a risk, such as a trip hazard or causing access difficulties to work areas.

Be careful when lifting heavy items such as the PC itself – keep your back straight. Also when placing equipment on to work surfaces let it down gently. Constant bumps to equipment reduces its working life, so should be avoided. Keep your work area neat and organised and learn to work safely and efficiently with your class colleagues.

Turn up for laboratory practicals on time so that you will hear pertinent introductory information needed to complete the practical safely and effectively. Students who turn up late will be refused access to the laboratory. Please be on time for lectures too, so as not to disrupt your colleagues.

Finally, never disconnect nor reorganise the wiring to the KVMs (apart, obviously, from the leads supplied for connecting up a second processor into the system).

Addendum for COVID-19:

You should use the blue tissue paper and disinfectant supplied in the hardware lab (C305) to clean your work area and any equipment you use during your lab classes. This should be done at the start of every lab class and at the end just before you leave. There are two bins in C305. Put the used blue tissue into the bins once you are finished with it. Hand sanitiser will also be supplied – use it constantly, particularly if you leave or return to the lab during a practical class. Maintain social distancing as best you can with everyone else in the laboratory. Face masks must be worn and must

be correctly fitted, i.e. covering your nose and mouth at all times. Do not share equipment with anyone else, be it tools supplied in the lab, or personal items such as pens, erasers, paper, etc.

What is a Computer?

- A machine for processing information.
- A computer takes *data* and processes it to produce *information:*

- *Data* is a collection of facts that is not immediately useful to us.
- *Information* is processed data that is useful to us.
- What constitutes data and information in one context might be totally different in another context – for example, the *payroll program* for IT Carlow takes items such as employee name, pay scale, overtime hours, tax table, etc. as input *data*, and produces *information* such as net pay, tax deducted, pension contribution, etc.

  A *tax reconciliation program*, on the other hand, would see the tax deducted for each employee as input *data* and produce the total tax payable to the revenue for all employees in the IT as *information*.
- Processing can be a very simple action such as sorting a list into alphabetic order to very complex such as predicting the weather.
- It is common practice to use the words *data* and *information* interchangeably, as most of the time we are not concerned with the strict distinction.
- A single item of *data* is called a *datum.*

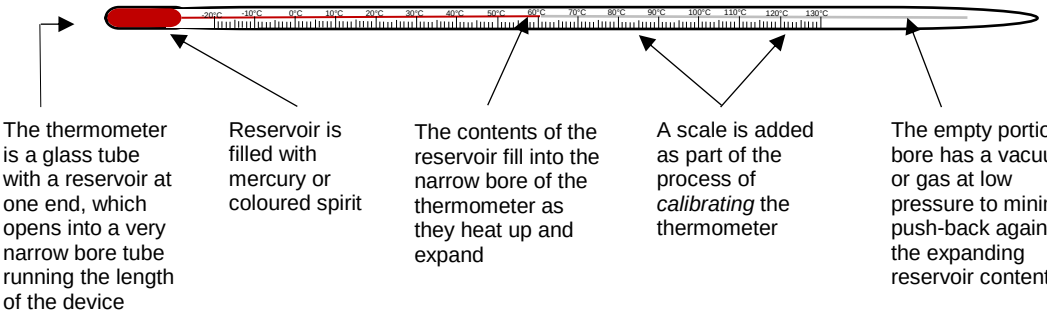
How Long Have Computers Existed For?

- When most people hear the term *computer* they think of a recent electronic invention (such as a PC) that processes data to produce information - such machines have been around for only 60 to 70 years.
- The term *computer* had been in use for a few hundred years as the job title of a person employed to do difficult calculations (using pen, paper, tables and brains).
- Machines that process data for us have been used for at least as long as history has been recorded (and probably much longer) - can you think of any examples ?
  - Sundial
  - Barometer
  - Sextant
  - Watch
  - PC
  - etc. etc.
  - Abacus
  - Mercury (or red spirit) Thermometer
  - Speedometer
  - Calculator
  - Transistor Radio
- Can you identify what would constitute *data* and what would constitute *information* in each of the above examples?
- The electronic device that processes data to produce information (that most people think of on hearing the term *computer*) is in fact the subject matter of this course (for example a PC) - so your initial thoughts would have been pretty close to the mark in the context of this subject.

Initial Classifications – Analogue, Digital and Hybrid.

Computer systems (in the wide context just discussed) broadly break down into two categories – *analogue* (*analog* if you're American) and *digital*. The distinction is usually obvious, but can sometimes be a little difficult to discern. We'll discuss both types by looking at examples. It is also possible to mix the two types, yielding a *hybrid*[1] computer.

A good example of an analogue computer is the mercury (or spirit) thermometer that we should all be familiar with.



This device contains a small amount of mercury (or coloured spirit) in an evacuated glass tube. As the temperature around the outside of the thermometer (*ambient* temperature) rises and falls the mercury inside expands and contracts proportionately, travelling up and down a narrow glass tube from the reservoir. The top or the *displacement* of the column of mercury can be used to interpret the ambient temperature. Thus, the displacement of the mercury is an *analogue* of the ambient temperature. We use the displacement of the mercury as an *analogy* for the temperature.

An analogue computer must be *calibrated*, that is, an association must be established between what is happening and what we wish to interpret from what is happening.

For example, with the mercury thermometer we need to put a scale on the device before it is of any use to us. We might note the displacement of the mercury when the reservoir is immersed in melting ice (0°C) and mark that point on the device. Similarly, we might also note the displacement of the mercury when the reservoir is immersed in boiling water (100°C) and mark that point on the device. We can then put 100 equal divisions between the two points and label them accordingly.

The range of values that may be represented is *continuous*. Given any two temperatures (displacements), no matter how close together they are, we can always represent another value in between. Indeed this is just another way of saying that there are an infinite number of possible values between any two points. A continuous range of values is the principal characteristic of an analogue computer system.

For a number of reasons, the values we read from an analogue computer may have inaccuracies.

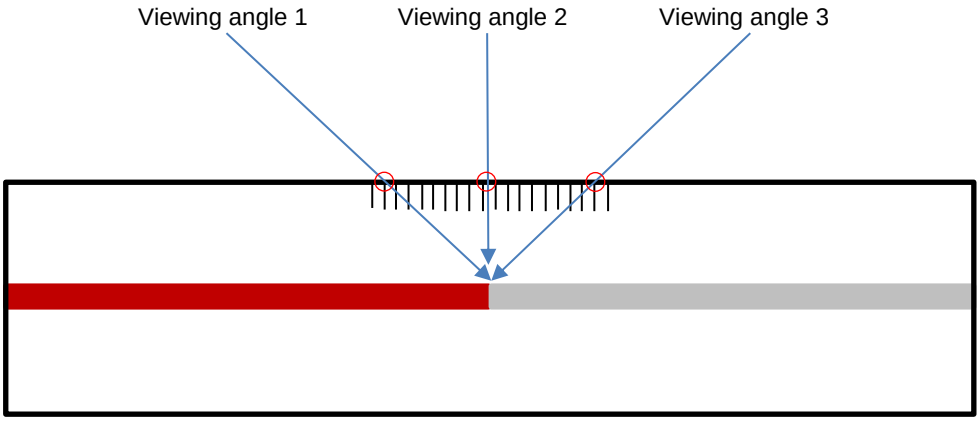
Firstly, we depend on how well the device was calibrated. If we were not at sea level, then water's melting and boiling points may differ slightly from the 0°C and 100°C we used. Then we have to consider how accurately the 0°C and 100°C marks were applied to the device. We also assumed that

[1] The term hybrid may be used whenever technology types are mixed, such as analogue and digital. A supercomputer that uses SIMD and MIMD technology simultaneously is also a hybrid computer, but is completely different to what we have been discussing here. Other hybrid mixes are also possible. Any time you encounter the term *hybrid*, be careful to distinguish what technologies are being mixed. Obviously, all hybrid computers are not the same.

the mercury expands linearly when we broke the scale up into 100 divisions – this assumption may not be correct. We also have to consider how accurately the subdivisions were made and applied.

Secondly, we depend on how well the device is read. We may, for example, get subtly different answers depending on the angle from which we read the device (*perception errors*).

For example, with a mercury thermometer, the small red circles in the diagram below show the point of the scale that is seen to coincide with the top of the mercury column when viewed from different angles.



So the values we interpret from an analogue computer are never 100% accurate. However, a well calibrated instrument in expert hands can still be extremely accurate.

There is a multitude of analogue computer examples such as barometers (measures atmospheric pressure), slide rules (an analogue calculator), thermometers, manometers (measures pressure – such as an old blood pressure monitor), sextants (measures latitude), sundials, pressure gauges, camera light meters, analogue clocks, speedometers, sound meters, distance measuring devices (from tape measures to ultrasonic technology), pitot tubes (aircraft speed), voltmeters, ammeters, wind socks, weather vanes etc. Consider what constitutes data and what constitutes information for each of these devices. Establish the analogy involved in each case. Consider if any of the devices employ more than one analogy.

An obvious example of a digital computer is a PC or a tablet computer. Values are represented using sequences of binary[2] digits (which are usually called *bits*). A bit can have one of two values, 0 and 1. Within the hardware of a digital computer system, a bit may be represented using a magnetic field (N-S or S-N field), an electric current (current flowing or current not flowing), an electric charge (a charge present at some point, or absent), or numerous other possibilities. We can use multiple bits to increase the range of values that may be represented. Two bits can represent four (2^2) possible values, three bits can represent eight (2^3) values, 32 bits can represent in excess of four billion (2^{32}) values and 64 bits can represent in excess of 18 quintillion (2^{64}) (18,446,744,407,000,000,000 approx.) values. The more bits that are used, the greater the range of values that can be represented. Increasing the number of bits for a quantity increases its *precision*.

The range of values that may be represented in all instances is ***discrete***. Given any two values, it is not always possible to represent a value in between (in particular, if the two values only differ at their least significant digit). The only way a digital computer could have a continuous range of values would be to allocate an infinite number of bits for each value. This is clearly impossible.

[2] it is possible to have a digital computer system that works in a different base – for example, one of the earliest digital computer systems, the ENIAC, worked in decimal – nowadays, however, binary is the norm.

Within a digital computer, *integer* values are input, processed and output with 100% accuracy (unlike an analogue system where the integer concept doesn't really mean anything). Errors arising from calibration and perception (reading) simply don't arise for digital systems. For *real* numbers (non-integer numbers or numbers with fractional components) small errors may occur during input, output and processing. For example, if we input the value 0.1_{10} as a real number to a computer system, it yields a binary number $0.0(0011)_2$ where the digits in brackets recur indefinitely. So to store this number in binary will require any digits after the number of bits of storage being used, being truncated, thus introducing an error. We simply increase the precision of the number (the number of bits used) until the error is acceptable for our purposes. We have precisely the same issues when processing real data manually.

Digital computer systems may also use components called *analogue-to-digital* and *digital-to-analogue* converters for interfacing with analogue systems. If this is done to a significant extent then we have a computer that part analogue and part digital, and so is called a *hybrid* system. Most digital computer systems will have some hybrid aspects to them.

If we record sound (an analogue waveform) using our computer's microphone (an analogue input device), it converts the sound waveform to an electrical waveform. This is then fed into an analogue-to-digital converter (on our *sound card*) which converts the waveform into a stream of numbers to be stored in an appropriate format, such as MP3, by our recording software.

To play the sound, the MP3 data is fed into a digital-to-analogue converter which converts the numbers back to an electrical waveform which is fed to a speaker system which generates sound once more.

Unless the amount of conversion is very significant, or the computer system is specifically dedicated to processing analogue data or producing analogue output, we will usually still consider the system to be 'digital with analogue or hybrid capabilities' rather than hybrid.

Sometimes it can be quite difficult to distinguish what type a computer is. For example, is the watch below analogue or digital?

It is actually a digital computer (it has a battery, a quartz oscilator and a digital integrated circuit inside the case) but it has an analogue face for output. Up to recently digital watches had the word *quartz* somewhere on the face, but as in the example above this is not always done now. The analogue face can cause confusion.

Is an abacus analogue or digital? (hint: is the range of representable values continuous or discrete?)

From this point on, we are only going to concern ourselves with *electronic stored-program general purpose digital computer systems*. These systems can perform a multitude of tasks simply by

running different programs as required and these programs can be stored within the computer system ready for use at any time. Processing is performed using digital electronics.

Notice that the computers mentioned up to now (particularly the analogue ones) are *programmable* to some extent, but aren't *general purpose* as they are not intended to be reprogrammed over and over again. For example, the process of calibrating a thermometer or installing a sun dial is essentially programming it – however this programming is restricted to the device's initial construction or installation, and they are generally not reprogrammable beyond this point.

We will attempt a further classification of our restricted class of computer systems a little later on, before we look at the architecture of these systems in detail. Before this, we are going to examine quantities in computer systems. It is important to develop an appreciation of the different magnitudes that we will encounter, so this is a good time to begin. You will cover this in more detail in your mathematics course.

Quantities in computer systems

The smallest amount of information possible on a computer system is called a *bit* (BInary digiT). A bit can have either a zero (0) or a one (1) value.

We mentioned earlier that a bit may be represented in a number of ways. It may be represented magnetically (as a N-S or as a S-N oriented magnet). It may be represented by a current being able to flow through a transistor or by the current being blocked by the transistor. It may be represented by the reflectivity or non-reflectivity of the surface of a CD or DVD. It may be represented by the presence or absence of a mark (e.g. a Lotto play slip) or a hole (e.g. old punch cards) on a piece of paper or card. Many other representations are possible.

All of these examples exhibit *bistable behavior* – that is to say they are unambiguously in one state or another – so they can clearly represent a zero or a one value. They are clearly digital and clearly not analogue[3]. It is easy to find materials or construct gadgets that exhibit such behavior. It is considerably more difficult to achieve this for ten stable states. This is the principal reason that all information within a computer system is in binary rather than in decimal (actually, the very first electronic computer was a decimal machine – pretty much all since then have been binary).

It is only possible to operate on individual bits within the CPU. If we need to work on a bit elsewhere, (e.g. in memory) we need to extract it from a collection of bits called a *byte*. A byte consists of a collection of eight bits. A byte can have 2^8 (or 256) distinct values.

- A byte is the smallest amount of information that we can move from one part of the computer system to another.
- Each storage address or location in the main memory of modern computer systems holds a single byte – it is said to be *byte-addressable*.
- A byte is the amount of storage used to store an individual character in the ASCII and EBCDIC character sets.

Historically, some prototype computers used a smaller collection of four bits called a *nybble*, e.g. the first true microprocessor, the Intel 4004.

[3] Actually transistors are fundamentally analogue – we restrict both the input voltages or currents, and the outputs, to two distinct bands each (corresponding to 0 and 1 values) to make it behave as a digital device. We separate the bands with wide areas of "no man's land" to ensure there is no ambiguity between bit values. If a transistor switches value we must allow its outputs to cross "no man's land" before using its result, so "switching speed" becomes a serious issue.

Computers process vast numbers of bytes all the time. Just as we have convenient units for handling lengths or distances (e.g. micrometer, millimeter, meter, kilometer, etc), so too have we convenient measures for bytes. As these units are based on binary numbers, they may look strange to us, but a binary computer is perfectly at ease with them. Furthermore, as they are close in value to well known decimal measures, we can use these measures as close approximations unless high accuracy is required.

A kilobyte (1K or 1KB) consists of precisely 2^{10} = 1,024 bytes.
One thousand (1,000) bytes is an acceptably close approximation for most purposes.

A megabyte (1M or 1MB) consists of precisely 2^{20} = 1,048,576 bytes.
One million (1,000,000) bytes is an acceptably close approximation for most purposes.

A gigabyte (1G or 1GB) consists of precisely 2^{30} = 1,073,741,824 bytes.
One billion (1,000,000,000) bytes is an acceptably close approximation for most purposes.

A terabyte (1T or 1TB) consists of precisely 2^{40} = 1,099,511,627,776 bytes.
One trillion (1,000,000,000,000) bytes is an acceptably close approximation for most purposes.
Terabytes have come into common usage in the last few years (typical hard disk capacity).

Note that the above units are increasing by a factor of 1024 (approximately 1000) in each step.
The next units (extraordinarily huge quantities), following the same logic, are:

Petabytes (1P or 1PB)	$2^{50} \approx$ 1 quadrillion (1,000,000,000,000,000) bytes.
Exabytes (1E or 1EB)	$2^{60} \approx$ 1 quintillion (1,000,000,000,000,000,000) bytes.
Zettabytes (1Z or 1ZB)	$2^{70} \approx$ 1 sextillion (1,000,000,000,000,000,000,000) bytes.
Yottabytes (1Y or 1YB).	$2^{80} \approx$ 1 septillion (1,000,000,000,000,000,000,000,000) bytes.

Petabytes are beginning to move into common computer terminology and will be well established within the next few years. Storage capacities in *data centres* would be measured in Exabytes.

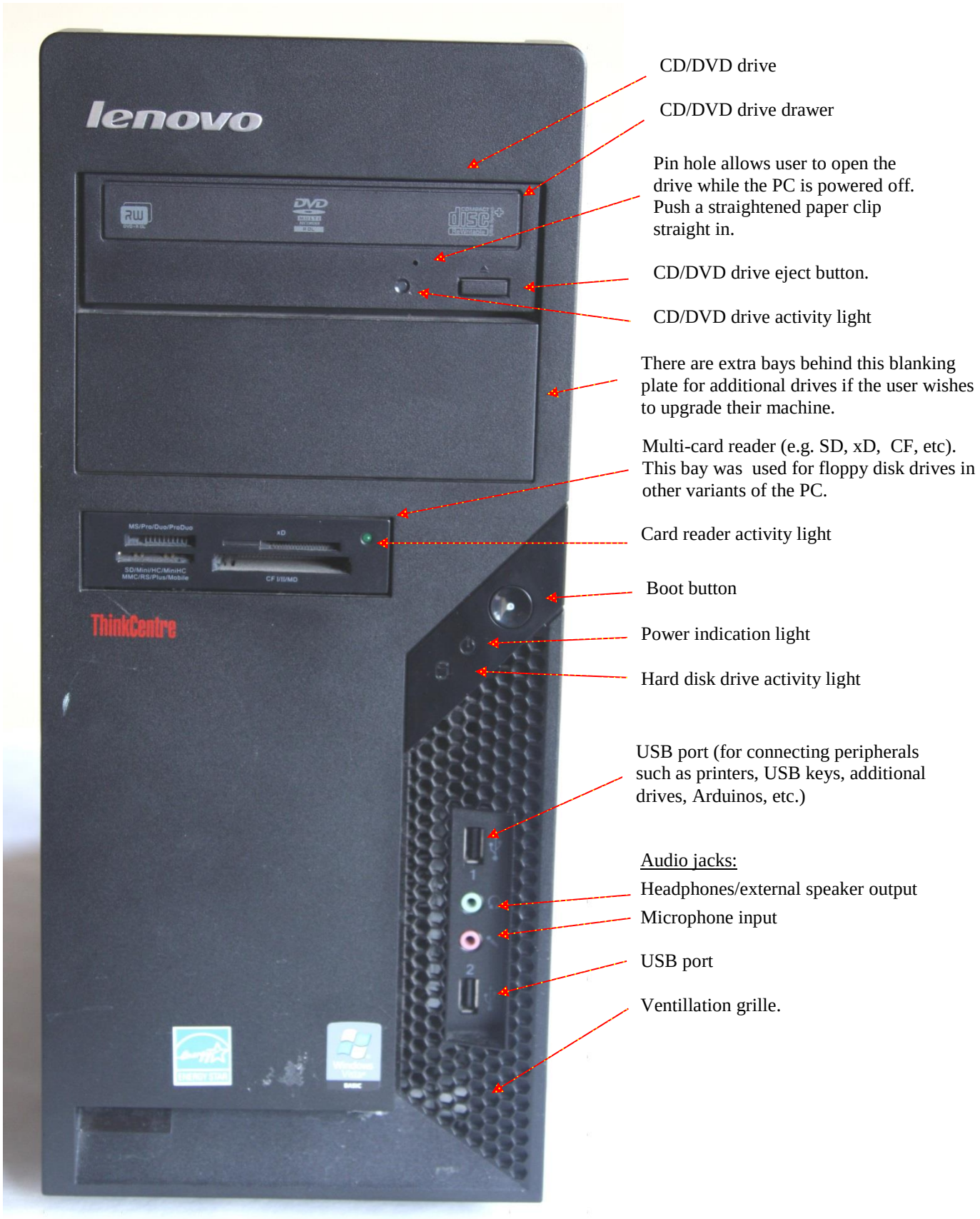
Due to the inconsistent use of the prefixes kilo-, mega-, giga-, tera-, peta-, exa-, zetta- and yotta- in the ‘normal’ world (e.g. for measuring distances, weights, etc., where each term is a factor of 1000 greater than its predecessor) and the computing world (for measuring memory storage capacities where they increase by a factor of 2^{10} or 1024), new standard prefixes were introduced for the latter (JEDEC / IEC). These were kibi- (Ki), mebi- (Mi), gibi- (Gi), tebi- (Ti), pebi- (Pi), exbi- (Ei), zebi- (Zi) and yobi- (Yi) respectively. These standards have not been universally adopted; the old SI system is still used extensively, despite its inconsistencies. The ‘bi’ in the prefixes means ‘binary’.

An Overview of a Typical PC’s Hardware

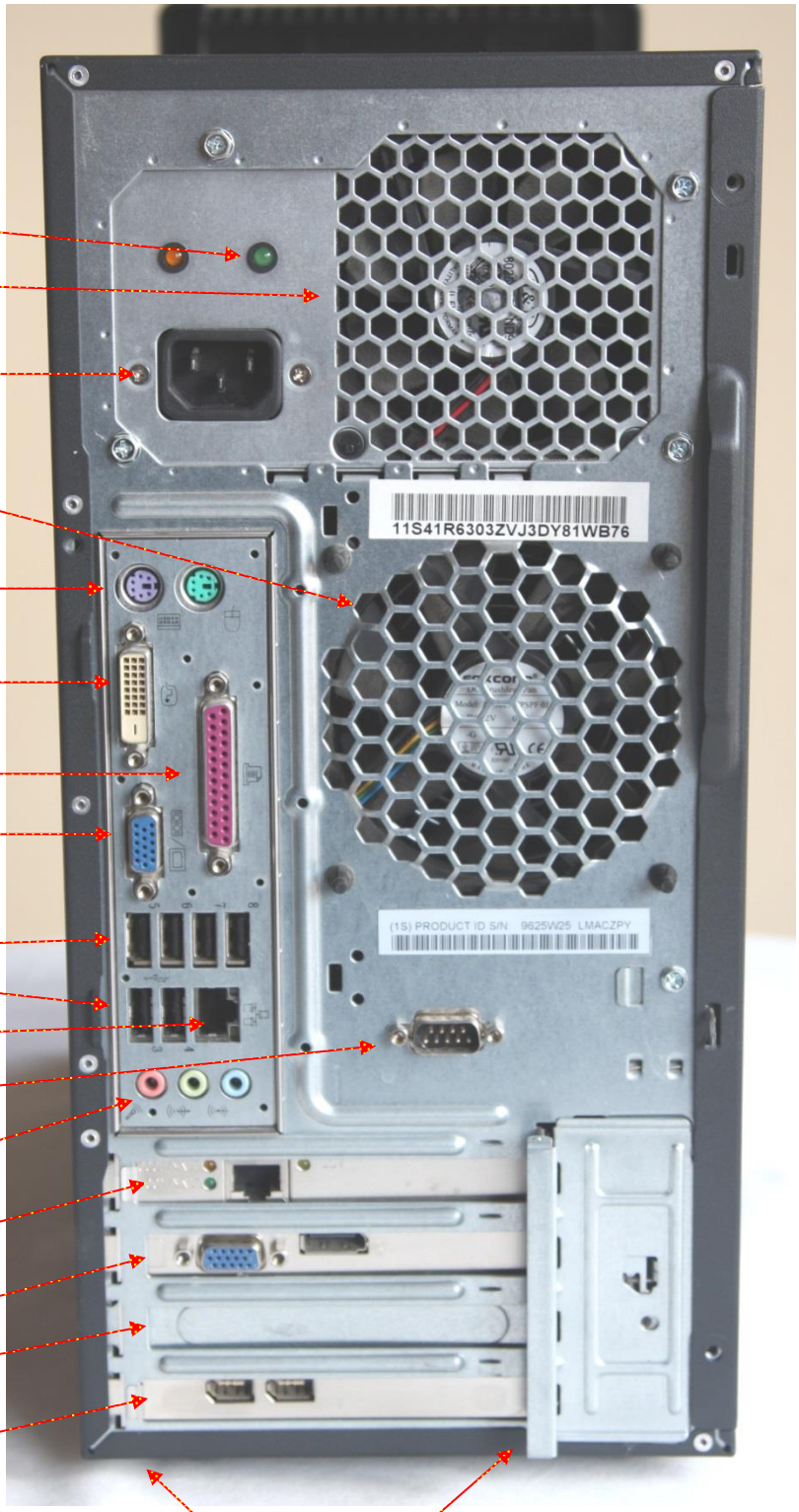
PC’s have evolved quite dramatically in the 30 to 40 years that they have been in existence. This section identifies the components you would expect to find in a reasonably up to date machine.

Many of the components are fundamentally the same as the earliest machines. Floppy disk drives have been replaced by CD and DVD drives (which are themselves disappearing), monochrome text based monitors have been replaced by full colour graphics monitors (assisted greatly by the ubiquitous mouse), matrix printers have been replaced by inkjet and laser printers and serial and parallel ports have been supplanted with USB ports, amongst other changes. The diagrams that follow show a variety of views of a PC from one of our own laboratories (room C305).

The first diagram shows the front panel.



The next picture shows the rear the PC (this one has a three expansion cards installed):



Mains indicator LED

Cooling fan for PC power supply

Mains electrical power connector

Motherboard cooling fan (and CPU cooling fan, indirectly))

Keyboard (left/purple) and mouse (right/green) use mini-DIN/ PS2 ports.

DVI (Digital Video Interface) video port (for monitor)

Parallel port (socket or *female* connector)

VGA video port (for monitor)

6 USB (universal serial bus) ports (4 above, 2 below)

Ethernet port

Serial port (plug or *male* connector)

Audio connectors (microphone left (pink),headphone/speaker middle (green) and line-in right (blue))

Ethernet expansion card

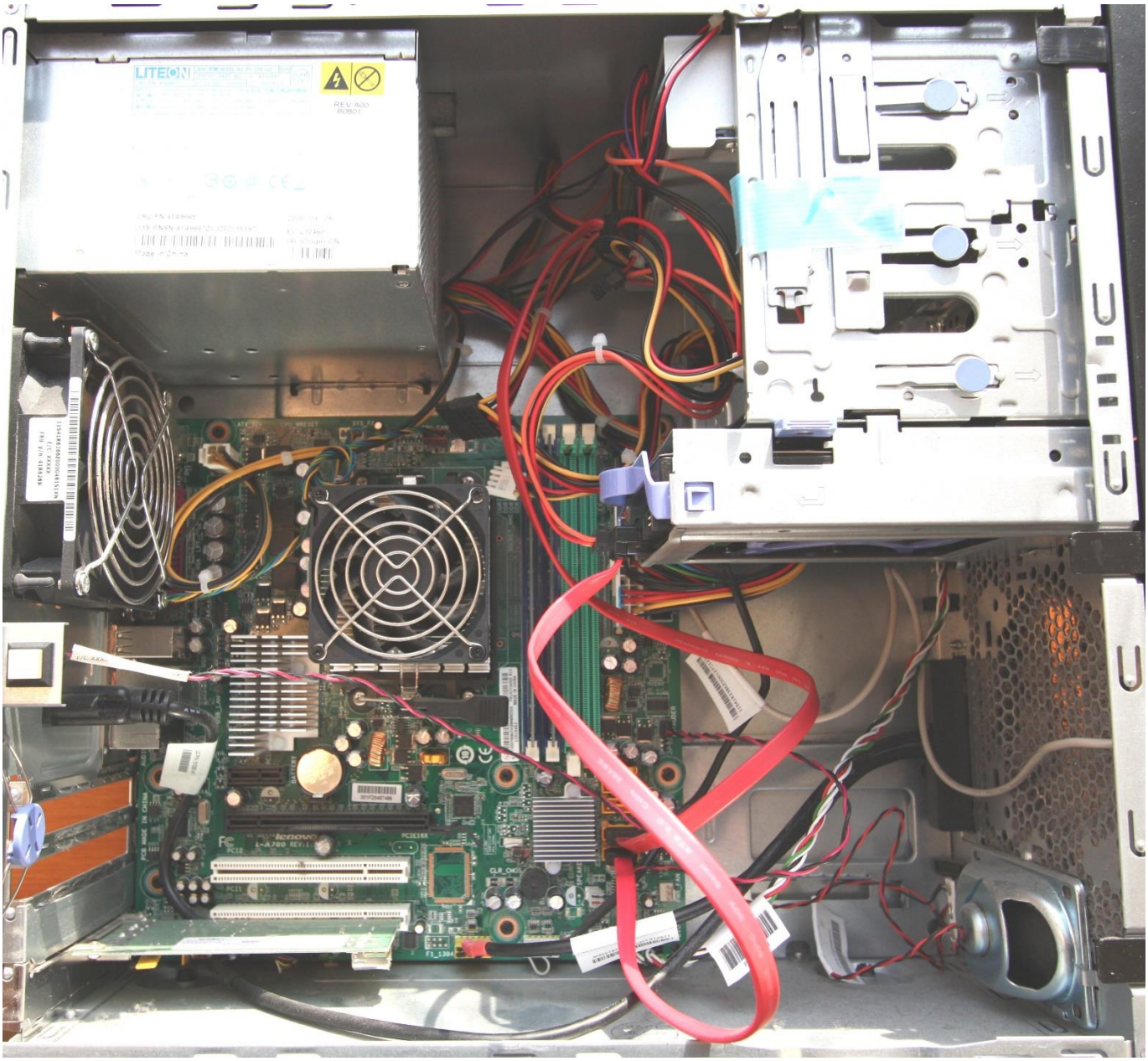
Video (VGA) expansion card

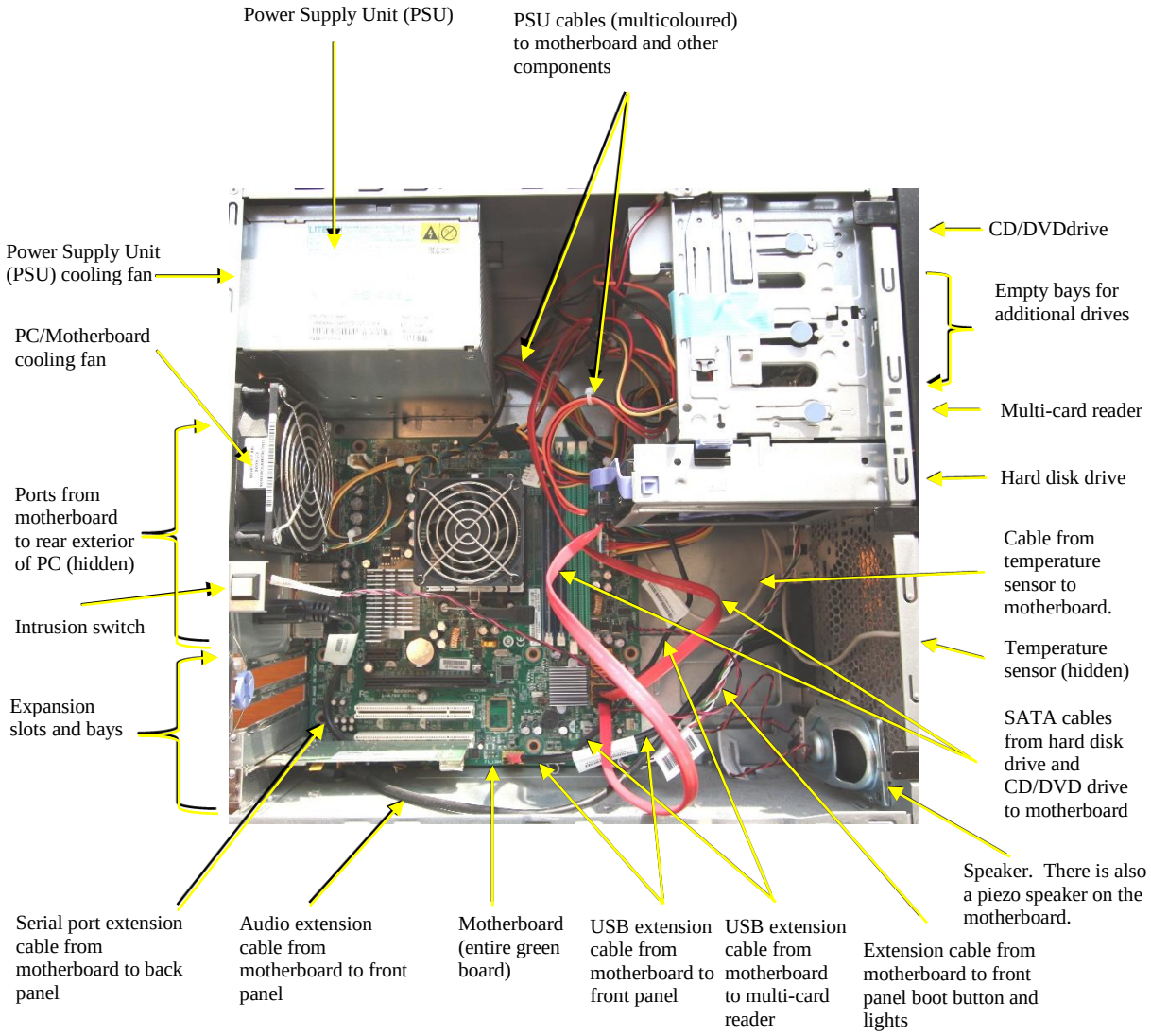
Vacant expansion slot

Firewire expansion card (obsolete)

4 expansion slots for additional expansion cards (3 occupied).

The next two photographs show the whole inside of the PC when the side panel has been removed. Note that the Video and Ethernet expansion cards have been removed to give a better view. Two versions of the photograph are given – one without labels - the other with labels and significantly reduced to fit those labels into the diagram.


LITEON
Lenovo
L-A780 REV:1.0



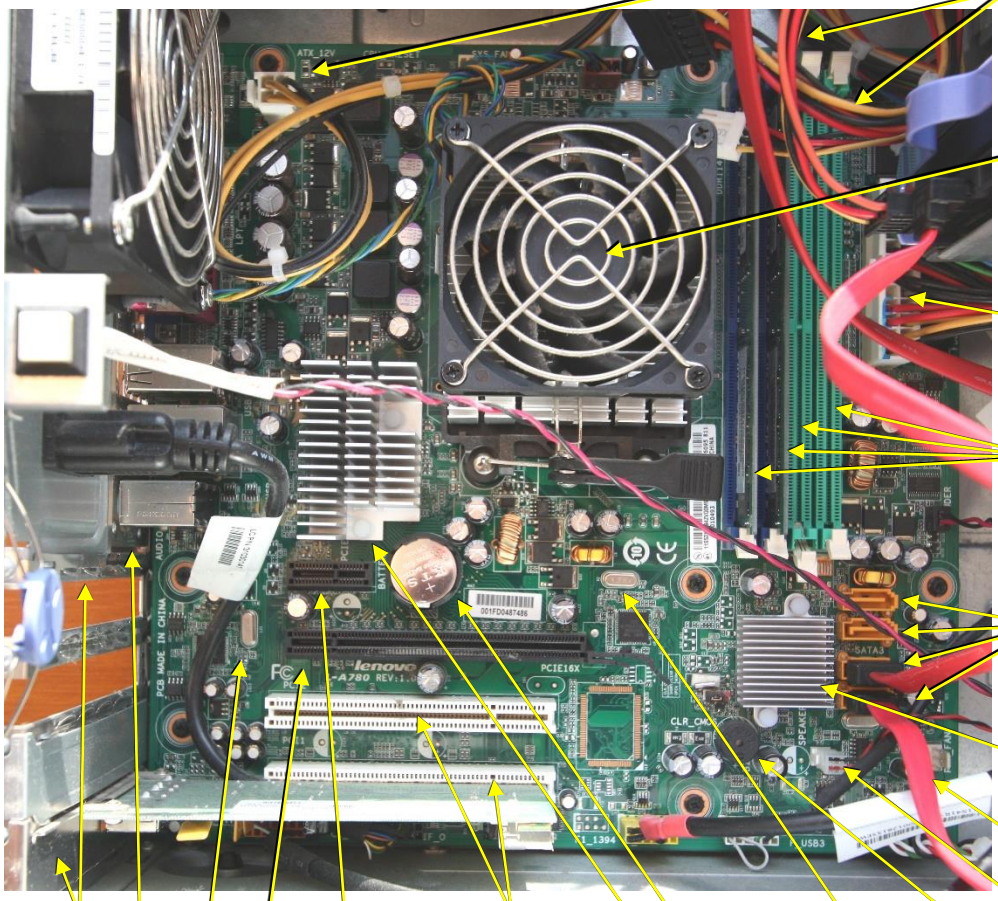
The final photograph shows the motherboard circuitry of the PC. The motherboard is the principal circuit board of the computer system (from which it derives its name) and contains most of the electronic logic of your PC system. It is augmented with memory cards (SIMMS, DIMMS, RIMMS, or whatever your particular system uses) and expansion cards which are used to expand the functionality of most systems.

PSU

Location of Power Supply Unit (not in photo).

CPU PSU connector

Output cables from power supply – these connect to the motherboard and to the drives which are to the right of the photo.



CPU (AMD Athlon64) connected to motherboard, with a heat sink (cooling fins) sitting on top, and a cooling fan on top of the heatsink

PSU connector block to motherboard

4 RAM slots – leftmost two are occupied with 2GB 667MHz DDR2 SDRAM cards – the others are empty

4 SATA slots – lower two are occupied (to HD and CD/DVD) – the upper two are not in use here

Southbridge component of chipset (+ heatsink)

SATA cable (to HD)

Front panel speaker connector

Piezo electric speaker

external expansion card access slots

Rear panel ports

Quartz oscillator

PCI Express x16 graphics adapter expansion slot

PCI Express x1expansion slot

PCI (peripheral component interconnect) expansion slots (cream colour) – for expansion cards

northbridge component of chipset (aka video graphics controller) + heatsink

Quartz oscillator

Lithium button cell to hold CMOS data

The Power supply transforms 220/240Volt AC (alternating current) to much lower DC (direct current) voltages. Different voltage output cables are usually colour coded. For example, a typical PC's power supply will produce +3.3V (orange cable), +5V (red), +12V (yellow) and 12V (blue) DC, as well as a common line (effectively 0V or electrical earth, cable colour black). The power supply is housed in a metal box for good electrical safety. The power supply provides a power source for all components within the computer processor's casing (and some external components via USB cables).

The case of the computer system is usually metal and / or plastic, with a metal chassis to support the installed components. The case, as shown in the previous four illustrations, is often called a processor box or CPU, which is a slight misnomer - the processor or CPU is only one of a multitude of components installed within the case.

Secondary storage devices can typically be classified into a number of distinct categories:

- disks;
- tapes;
- semiconductor type storage.

Disks themselves fall into three main categories; magnetic, solid state and optical. Magnetic disks, as their name suggests, store and retrieve information magnetically. They include:

- hard disks - most common and effective form of secondary storage;
- floppy disks - obsolete, but formerly extremely important;
- zip disks - obsolete with the advent of flash memory or pen drives;
- etc.

Solid state drives:

- SSDs - semiconductor replacements for hard disks. Currently they are significantly faster, much more expensive and lower capacity.

Optical disks store and retrieve information using finely focused laser beams. They include:

- CD-ROM - read-only - popular media for distributing software and documentation;
- CD-R - can be written to once by the user - extremely useful for personal back-ups;
- CD-RW - can be written to over and over again;
- DVD-R - once writeable;
- DVD+R - once writeable;
- DVD RAM
- DVD-RW - rewritable;
- DVD+RW - rewritable;
- etc.

Magnetic disks of all types are structured and operate in much the same way. They differ considerably with optical disks. Optical disks also have many similarities across the different types.

Disks of all types have a most useful characteristic that they support *direct* or *random access* - this means that to access an item of data (e.g. a file or program) we simply move the read heads of the disk to the area where the file is stored and read it immediately - we don't have to scan through data that logically occurs before the data we require, in order to access it (compare to tapes next).

Tapes are not generally used in home computers - they are however used extensively for backing up *servers* in *local area networks*. Tapes have the serious drawback that to access some data on it, we

need to scan through all the data that precedes it to get at it (think of accessing your favourite song on an audio cassette tape, and compare this to CD). This is called *serial* or *sequential access*. It is very inconvenient for normal usage, but is not a issue for backing up data (or indeed retrieving backed up data in the event of a data loss - most people are extremely happy to suffer a minor inconvenience to retrieve lost or corrupted data). Tapes on modern systems are generally implemented in convenient cartridges for easy and efficient usage

A recent innovation in recent years is semiconductor memory. This can take a number of forms, though the basic principal is much the same in each case:

- SSDs are becoming very prevalent and popular.
- USB keys/pen drives/flash memory - very popular due to small size, speed and convenience;
- SD (secure digital) cards - used extensively with tablet computers, phones, digital cameras and games consoles;
- MMC (multi media cards);
- MD (IBM micro drive);
- xD;
- etc.

USB keys are constantly dropping in price and increasing in capacity and have rendered the floppy disk, in particular, and also the zip disk, obsolete.

With all forms of secondary storage we have to draw a distinction between a device's *drive* and a device's *media*. A floppy disk illustrates this point very well. The floppy disk itself is the media (strictly speaking, actually, the media is within the rigid plastic case of the floppy disk) - the drive is the device within the PC case that we insert the floppy disk into in order to read or write information. This distinction between a drive and its media is quite obvious for devices such as CDs, DVDs, zip disks and tapes. We say such drives have *removable media*.

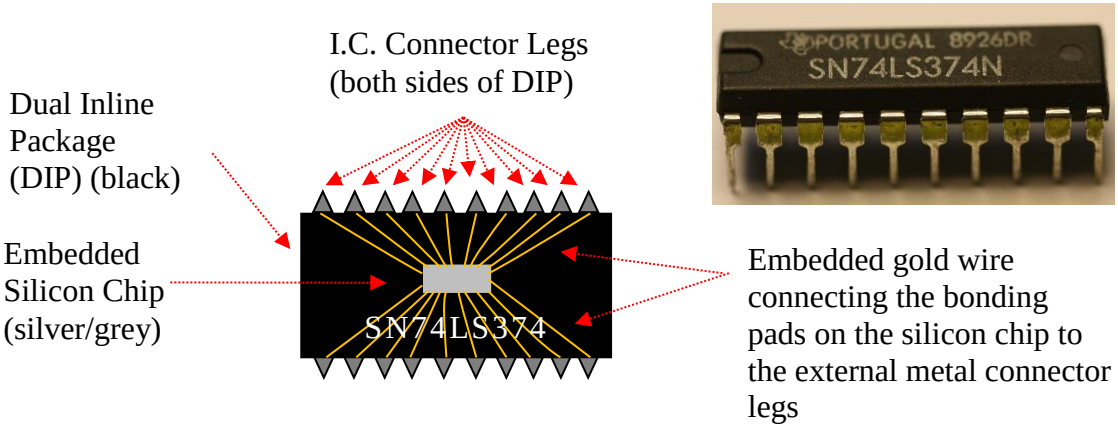
Hard disks and USB keys do not have removable media – they have *fixed media*. The media is an integral part of a hard disk system, and cannot generally be physically separated from the drive. In the case of a USB key and a removable hard disk drive (as in the previous four illustrations), the drive and integral media is designed to be easily removable from the computer system - these are examples of *removable drives*.

The microprocessor on a PC is typically an Intel Pentium, an Intel Celeron, an AMD (advanced micro devices) Athlon or AMD Duron. More up to date machines use Intel dual core and quad core processors (or AMD equivalents).

We need first of all to describe what a *microprocessor* is.

An *integrated circuit* (IC) is an electronic circuit that has been fabricated on a tiny sliver of pure crystalline silicon. We call this sliver of silicon a *chip*. The chip is encased in a block of plastic or ceramic material and connected to the circuit board via metal legs on the packaging (which are in turn internally bonded to special connection points on the silicon chips surface within). It is common practice to refer to the chip and the packaging it is contained in, as a chip (but this is not strictly correct).

A microprocessor is a CPU implemented on a single chip. All computers have CPUs, but not all necessarily have microprocessors (particularly big old computer systems). PCs have, and have always had, microprocessor based CPUs.



A DIP (viewed from above) showing its embedded chip and the wiring that connect it to the outer system. The inset is a photo of the same IC viewed from the side.

Each transistor in a modern microprocessor gives off an infinitesimally small amount of heat. However, as there are millions of these transistors present, the amount of heat produced can be quite significant, particularly for PC processors. As microprocessors are produced to work faster and faster as time goes by, the amount of heat generated increases. This heat has to be drawn away or the microprocessor will get hotter and hotter and quickly malfunction (if not self-destruct!).

For this reason, most microprocessors have *heat sinks* attached. Heat sinks are typically flat chunks of aluminium with a multitude of fins attached, and work rather like ordinary radiators (though their purpose is to draw heat away from an object rather than heat the surrounding environment). The heat sink absorbs the heat from the microprocessor and spreads it into the fins. Air is constantly drawn through the fins by an electric fan nearby, thus drawing the heat away from the microprocessor. It is obviously most important to keep the cooling system clean and functioning correctly to safeguard your PC. Never operate your system with the air conduits removed, or indeed with the computer's side panels removed for prolonged periods of time.

The area between the CPU chip and the heat sink is inevitably filled with *thermal paste*. This allows for efficent heat conduction between the CPU and the heat sink, helping to protect the CPU from overheating.

The BIOS (basic I/O system) is an area of ROM (Read Only Memory) (probably embedded in another IC) that contains the ROM bootstrap loader and a number of primitive device drivers that allow the PC to power up and load its operating system. The BIOS has to be closely tailored to the hardware of your system - hopefully the reason for this is obvious.

The CMOS is a special type of memory chip that holds the configuration of your system, particularly when the system is powered down. The CMOS require a constant power source to retain its data - this is supplied typically by a small button cell (lithium battery) mounted on the motherboard. When the cell's power is exhausted, or the battery is removed from the motherboard, the CMOS configuration is lost. It is possible to backup the CMOS data to a file on the hard disk. This is always a wise thing to do. Most systems nowadays can automatically update the CMOS, however it significantly slows down the boot process.

The *quartz oscillator* generates a stream of very rapid electrical pulses. These electrical pulses are fed into a timer chip which delivers one clock pulse to all the electronic components in the computer system every time it counts up a fixed number of quartz oscillator pulses. The number of clock pulses output by the timer chip corresponds to the PC clock speed (though as a variety of different devices operate at different speeds, then the timer chip has to deliver a number of different

frequencies - e.g. the CPU might operate at 3 gigahertz (3 billion operations per second) while RAM might operate at 330megahertz (330 million operations per second) - all of these clock frequencies have to be generated).

We've discussed RAM previously. RAM is volatile. It is implemented as a number of small memory chips mounted on mini-circuit boards. The design and functioning of these boards constantly changes (e.g. SIMMs, DIMMs, RIMMs, etc.). A simple, cheap and effective upgrade to most systems is the addition of more memory modules.

USB (*universal serial bus*) ports have been present on PC systems for many years now. They provide a very neat, consistent and easy way to add a huge variety of devices to your system. They avail of *plug-and-play* technology where newly connected devices are detected, identified and appropriate drivers are automatically installed into the operating system with the minimum of fuss. Initially USB was introduced with the USB 1.1 standard - that has been largely replaced by the USB 2.0 standard which is considerably faster. USB was challenged by *firewire* technology, much faster again, but as USB is free for all to use, unlike firewire, the outcome is that USB prevailed. USB 3.0 is replacing USB 2.0 currently.

With the introduction of USB, older ports such as mini-DIN keyboard ports, PS2 mouse ports, parallel ports and indeed older serial ports are being used less and less, and will probably become redundant as time passes.

PCI (*peripheral component interconnect*) slots are used for adding expansion cards (such as sound cards, modem cards, etc.) to increase the functionality of your PC. Like USB, these connectors avail of *plug-and play* technology.

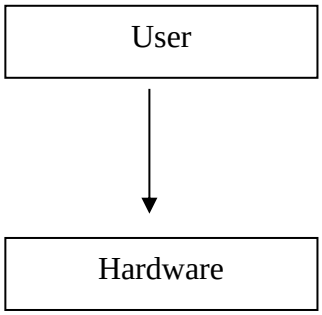
IDE (*integrated drive electronics*) connectors are principally used for connecting hard disks to the motherboard via ribbon cables. They also handle floppy and optical drives. IDE *controllers* are usually built into the motherboard to control the interfaces.

AGP (*accelerated graphics port*) is an expansion slot (differs from PCI) that allows a graphics card to be added to your system. The port facilitates direct dedicated access between the graphics card and RAM for better screen graphics handling - particularly good for handling 3-D graphic output.

A network connector allows your PC to hook up to local area networks using Ethernet protocols over UTP (unshielded twisted pair) cabling and RJ45 connectors.

Computer Hardware, Computer Software (and Computer Users)

Ultimately, a computer system is a collection of circuits and devices that allow us to carry out complex operations to transform data into information. It is capable of performing a bewildering variety of different tasks. At the simplest level, every computer system has two fundamental components - *hardware* and *software* (we'll discuss these in a little more detail later).

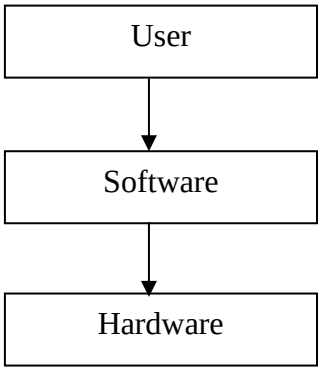


Hardware is the physical part of the computer system - you can touch it, lift it, move it about, etc. On its own, hardware is virtually useless to us as it has no intelligence or decision making capability, and it is almost impossible for users to communicate with it.

The hardware has to be instructed to do everything we want it to do - this has to spelt out to it in painfully minute and detailed steps. If we could instruct it so, then the computer would perform our instructions at lightning fast speed, would never make any mistakes (well, almost never) and would never tire or complain. To achieve this however, we would need to send it extremely long sequences of instructions and data encoded in binary (ones and zeros). While the computer can carry out our instructions effortlessly and accurately, we would make loads of mistakes trying to instruct it due to the tedious, repetitive and never ending task that would face us – in reality the computer would prove to be far more of a hindrance than a help. The very earliest computer professionals had to deal with computers on such a basis (actually, the first electronic programmable computer was programmed by soldering lengths of wire into new configurations). Fortunately, things are much better nowadays.

Humans and computers complement each other very well - each has distinctly different skill sets - humans tire easily, dislike very repetitive tasks, and make lots of mistakes, particularly when they are bored. On the other hand, most humans are good at assessing new and complex situations and making appropriate decisions. They display intuition and imagination, qualities that are virtually impossible to instil into a computer system or a computer program.

To overcome the communication difficulties between users and hardware, we include an in-between layer called *software* that will help us to control the hardware much more easily and efficiently. The term software is really just another word for programs - that is, a set of instructions that tell the hardware what to do.



The user gives instructions to the software which interprets them and supplies the appropriate stream of instructions to the hardware to make it carry out the desired task. The instructions we supply to the software can be much closer in nature to the everyday instructions we would normally give and get, thus making our control of the hardware much simpler and natural. The software layer effectively masks the intricacies and tedium of the hardware from us. Software exists in many different forms; operating systems, program development systems (compilers, etc.), applications, utilities, systems software, user programs and device drivers are but a few examples.

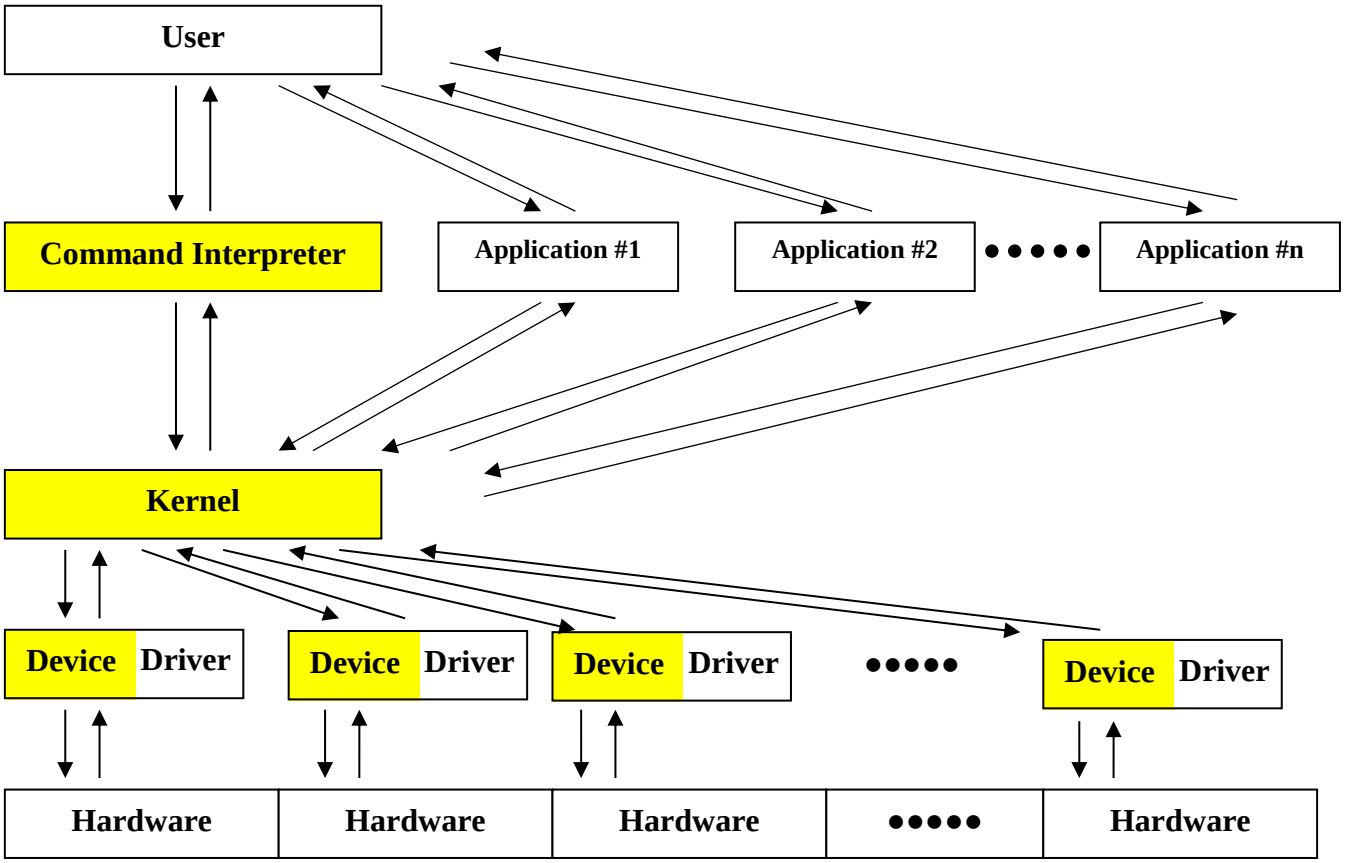
One extremely important piece of software on every computer systems is the *operating system*. Currently most PC systems use Microsoft Windows 7 or Windows 10 as their operating system. PCs can use other operating systems too - Linux and UBUNTU (PC versions of UNIX) are popular alternatives, for example. It is also possible to have a choice of operating systems available on a single PC (though we can generally only use one of them at a time).

An operating system is an extremely complex program. It is the default program on our computer system - when we are not running any particular program, then we are almost certainly communicating with the operating system (the *desktop*, in Windows, for example). The operating system provides us with a nice friendly environment to work in and shields us from the complexity and awkwardness of the hardware. The operating system provides us with an environment to run any programs we might want to use and helps us to keep our computer system nicely organised (for example, managing files and subdirectories for us). The operating system also protects the computer - it prevents us (to some extent at least) from doing serious damage to the system. It also allows our computer to communicate readily with other computer systems (networking). An operating system typically has a number of distinct parts. These are shown in the shaded boxes below. This diagram represents a *single-user multi-programmed* (*multi-tasking*) operating system (such as Windows).

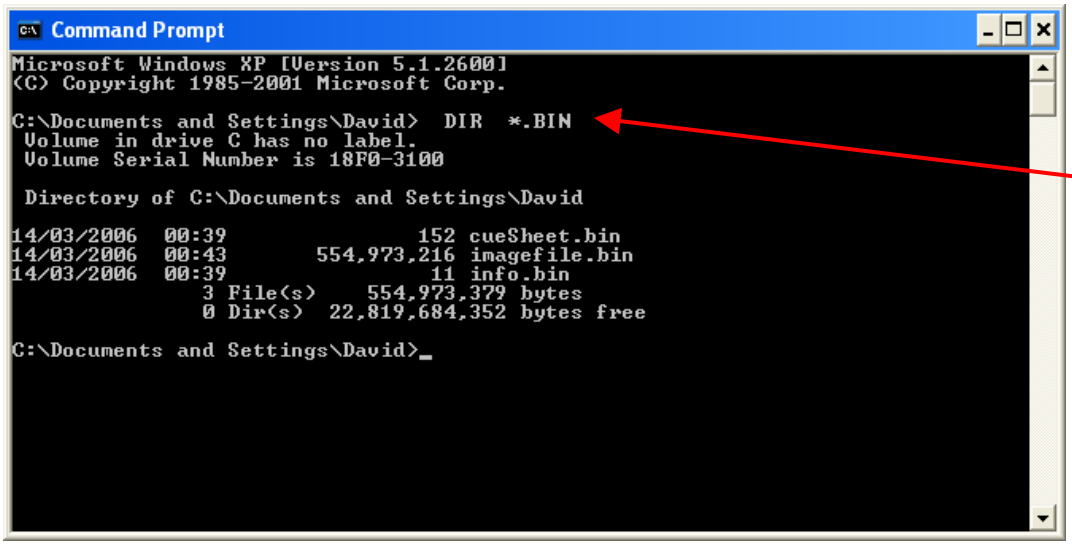
The command interpreter is the part of the operating system that we are most familiar with, though it is really only a relatively small part of it. It typically takes one of two forms:

- A command line interpreter or
- A GUI (graphic user interface) based command interpreter

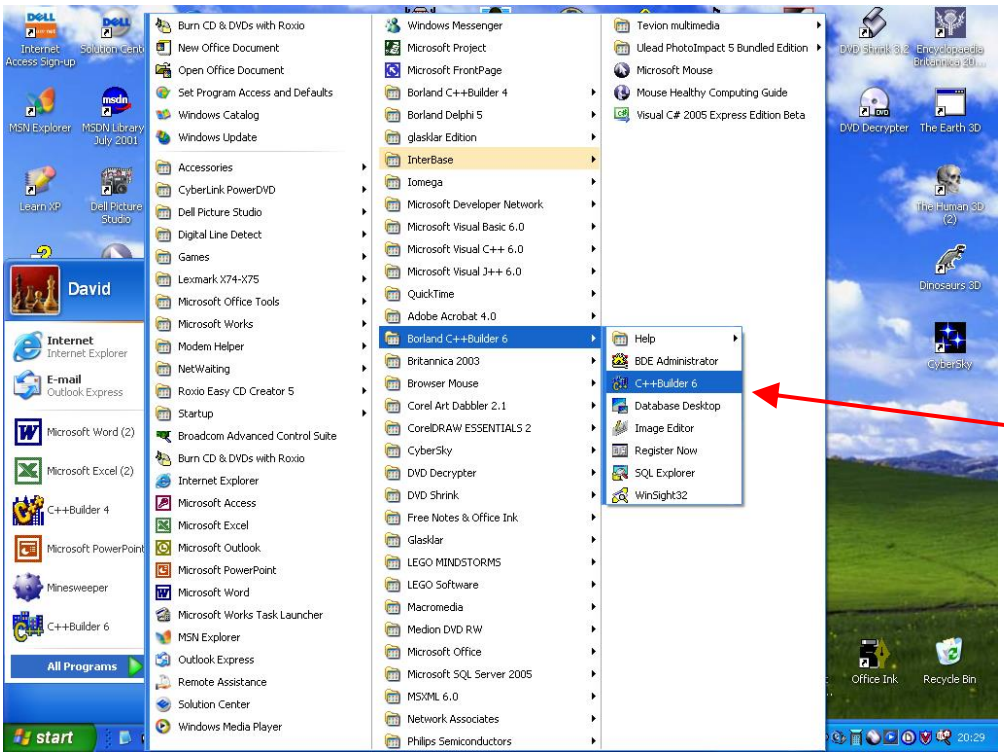
With a command line interpreter we type the name of the program we wish to run after a *prompt* on a *command line*. We may supply *parameters* after the program name to indicate data sources or other entities that we wish our program to use as it executes. We execute it by typing the enter key.



Examples of command line based operating systems would be Linux (a dialect of UNIX) and MS-DOS (the predecessor of Windows). The example below is actually the *Command Prompt* accessory from Windows XP, but looks essentially the same as an old implementation of MS-DOS.



Windows XP, Windows 7 or Windows 10 are typical examples of operating systems that use a GUI (graphic user interface) based command interpreter.

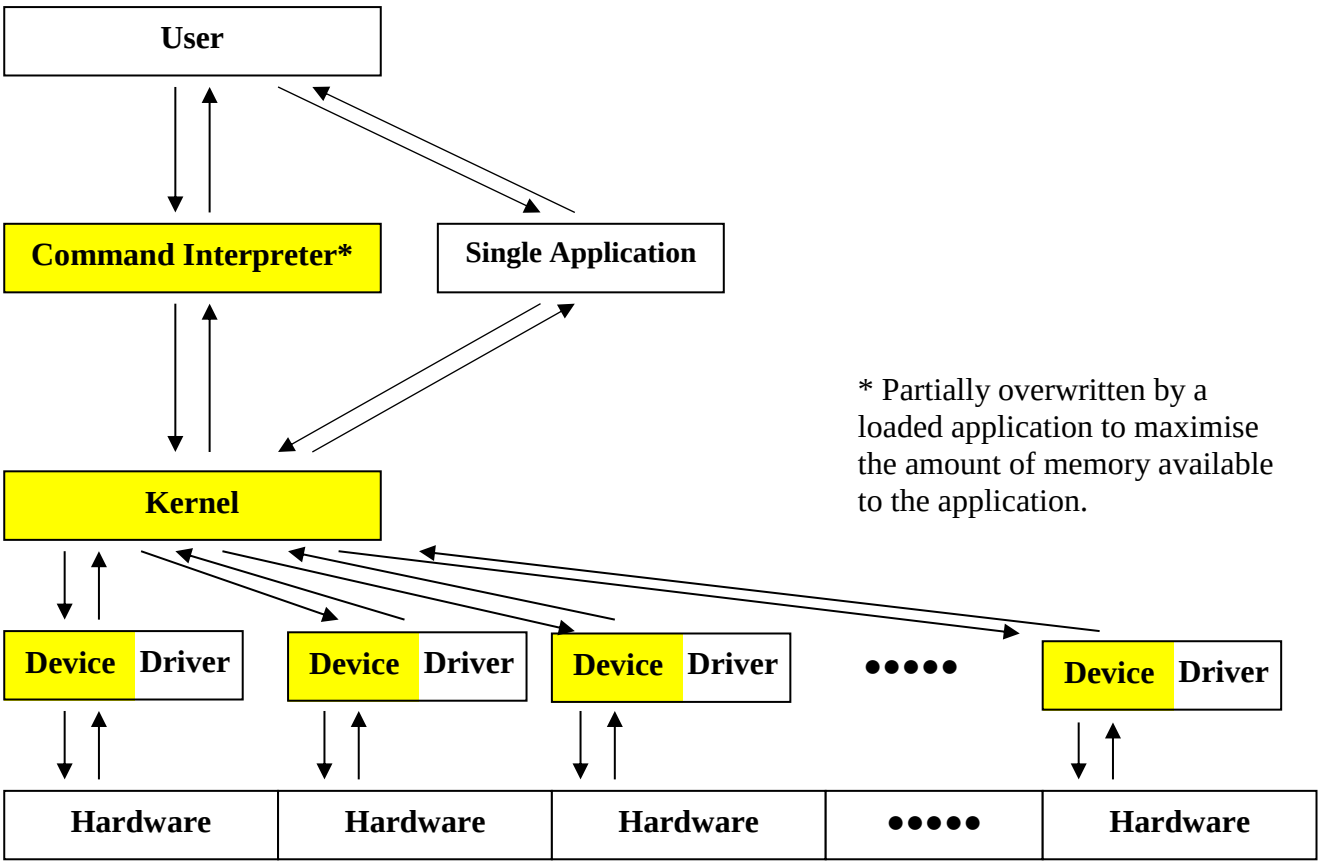


We run programs on such a system by double clicking on them using a mouse, or selecting them from a menu like the *start menu*. Such a system is considerably more pleasant to use than a command line system, but on the odd occasion can be somewhat less flexible.

Because we communicate with the command interpreter when we use an operating system, we tend to think of it as the entire operating system. In fact it is only a small program running under the operating system. Indeed, operating systems such as UNIX allow us to write our own command interpreters and thus tailor the system to our exact requirements. The command interpreter is often defined as being the "user's interface to the Operating System".

The main purpose of the command interpreter is to launch other programs, or applications. Its task is to provide the user with an environment to request that applications be executed - it then finds those applications, loads them into main memory and executes them by simply passing control over to them (see earlier diagram). When the application is about to terminate it passes control back to the command interpreter. With some operating systems, much, if not all, of the command interpreter may be overwritten by a running application and must be reloaded as the application terminates (e.g. MS-DOS). Other command interpreters remain present, but temporarily dormant, in memory while the application executes (e.g. UNIX). In the diagram above, note that the command interpreter is portrayed much like any other application. Observe that we can use the hardware of the computer system by communicating either through the command interpreter or through an application, and our operating system should allow us to switch between different applications and the command interpreter.

Some operating systems (e.g. MS-DOS) will only allow one application to be run at a time. MS-DOS was a *single-user, single tasking* operating system:



Other operating systems allow multiple applications to run simultaneously, and allow us to hop from one application to another as suits our needs. The original diagram on page 10 depicts such a system. If the system allows multiple programs to run then we have a *multi-programmed* or *multi-tasking* operating system[4]. Windows XP and UNIX are *multi-tasking* operating systems. UNIX in particular, allows multiple applications to interact and cooperate with one another - it also allows applications to run actively in the background - Windows XP facilitates this to a lesser extent (background processes tend to be dormant). UNIX uses *pre-emptive multi-tasking* (the operating

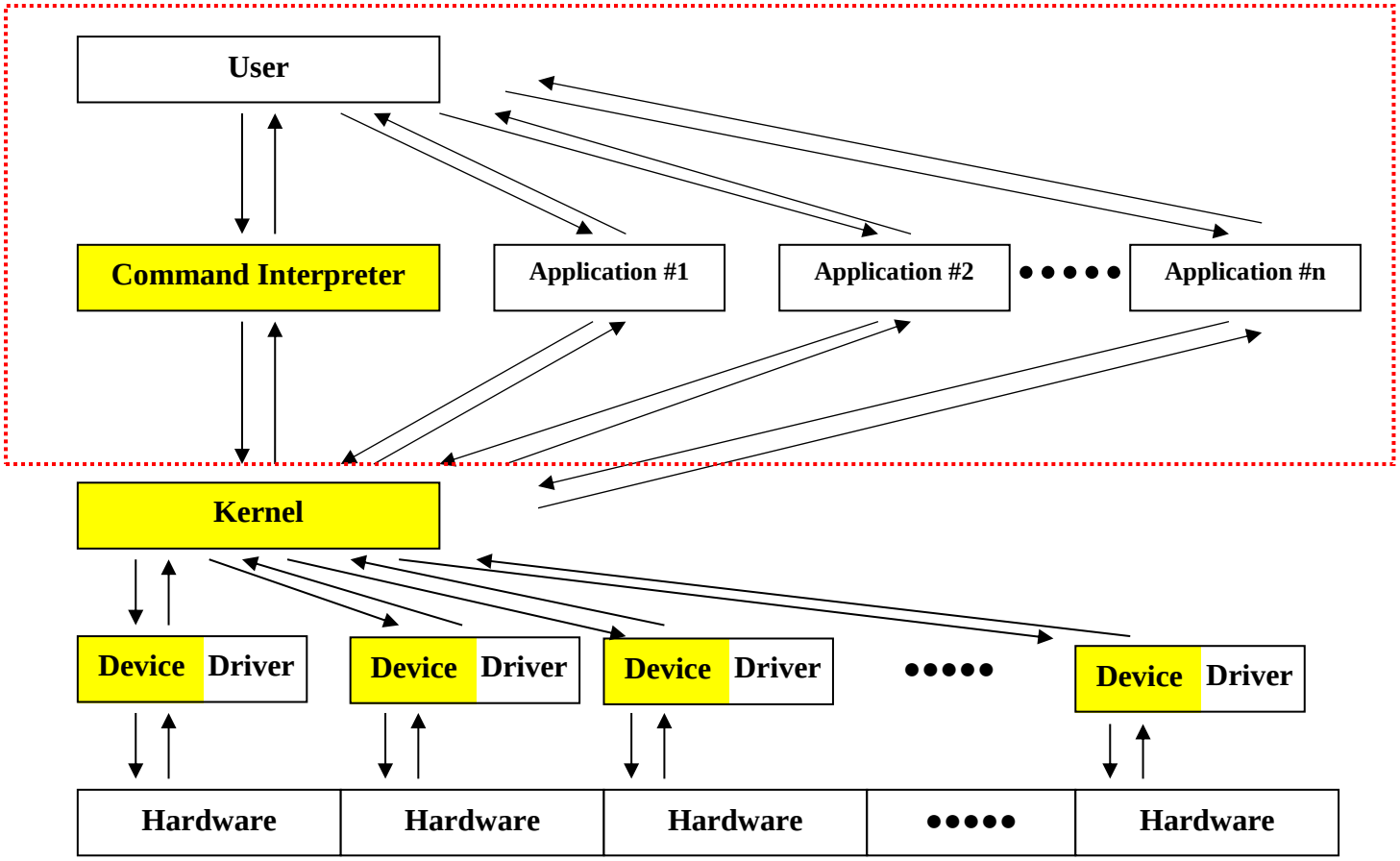
[4] Sometimes a multi-tasking system is distinguished from a multi-programmed system by allowing programs to interact with one another (which is more sophisticated than a multi-programmed system) – but this distinction is not commonly made now

system causes execution to hop from job to job), whereas Windows uses *co-operative multi-tasking* (the applications themselves manage execution-switching between jobs).

Both Windows and MS-DOS are examples of *single user* operating systems (though Windows facilitates different user profiles to be available on a single PC, in practice only a single user can be using the PC at any moment in time). UNIX on the other hand is a *multi-user* operating system as it can have a number of users using the machine simultaneously. The following diagram reflects a *multi-user, multi tasking* operating system such as UNIX. For each user on the system, the contents of the red dashed box are duplicated – all user applications (and command interpreters) interact with a single kernel. Furthermore, each UNIX user may run more than one command interpreter at a time – these are called *shells* in UNIX.

Note that communication between the user and the command interpreter or application is two way - the user sends instructions to the command interpreter or application and it responds accordingly to the user. Indeed, observe that all of the parts of our user-to-hardware scenario above can communicate bi-directionally with the previous and next layers.

The biggest and most significant part of any operating system is its *kernel*. The kernel is often defined as being a "program's interface to the Operating System" as should be evident from the previous diagrams. As the user never interacts with the kernel directly, then they tend to be oblivious to its presence or functionality.



Essentially, the kernel is a large collection of software routines that allow all aspects of the computer system to be used. Programs that are running constantly request the kernel to do tasks (typically I/O) for them. The kernel also implements the main services of the operating system such as file management, memory management, process management to name but a few.

Typically, most computer systems can use a very diverse range of hardware devices such as monitors, printers, scanners, hard disks and so on. Even monitors, a standard hardware device on every PC, come in a variety of forms. All of these devices expect to be controlled in ways that are different to other devices - that is they expect very specific signals to be sent to them to control what they do. It is as if they all speak different languages. It is unreasonable to expect an operating system to be able to communicate with every possible device type. For this reason, we have an interface layer between the kernel and the hardware devices called *device drivers*. These are effectively translators that allow the kernel to control the device and for the device to respond back to the kernel.

Windows is supplied with hundreds of device drivers for a range of more common devices that may be attached to a PC, however there are a vast multitude of devices not covered. Typically, new devices that we might want to install on our computer system are supplied with copies of their own device drivers that are usually included with their installation disks. When a new device is installed on a PC, Windows will first check if it has an appropriate driver for it - if it has it will stitch it into the system's present configuration - if not, it will ask for the devices installation disk and load the driver from there. Every time the system is booted after that, the new driver will automatically be loaded and made available (a copy of it is stored with the operating system files on hard disk). The device driver box in the previous diagram is partially shaded - some drivers are supplied with the operating system's installation disks - others are supplied with the devices installation disk.

At this stage it should be apparent, that a user's instructions have to burrow through many layers of software before hardware responds to them. This system makes the user's control of hardware much easier. The software also provides many safety features that help prevent the user damaging the hardware or data on the system (if the user is determined to do damage, or is not very clued in, they can still manoeuvre about these safeguards and wreak havoc). Finally, the many layers of intercommunicating software will inevitably slow the system down a little - the benefits far outweigh this disadvantage, however.

On a multi-programmed or multi-tasking system many programs can be executing simultaneously, as suggested by the previous diagrams. However, the CPU (Central Processing Unit) of a system can only execute a single instruction at a time, and this instruction has to obviously come from a single program. That is, only one program can actually be executing at any specific instance in time (although all of the other programs will be somewhere between their start and end points).

On a pre-emptive multi-tasking system (e.g. UNIX), the CPU is made to hop constantly from one program to another, to another , to another , to another … by the operating system. It can do this so quickly (maybe visiting each program 50 or so times per second) that the user gets the impression that he or she has the processor's undivided attention. The process of dividing the processor's time between processes is called *time-slicing*. Only a single program is ever executing at a particular instance in time, but every program is at some intermediate stage of execution, giving the impression that every program is running at the same time. This phenomenon is called *concurrency*.

If a system has more than one CPU (including dual-core, quad-core, etc., processors!) then it is possible that more than one program is truly executing at any instance in time. This phenomenon is called *parallel processing*.

A similar situation holds for a co-operative multi-tasking system (e.g. Windows), however the time division is managed by the applications themselves. This has the disadvantage that a program can "hog" the processor and not allow other programs to share CPU time. This is a significant disadvantage and renders co-operative multi-tasking less suitable for multi-user systems.

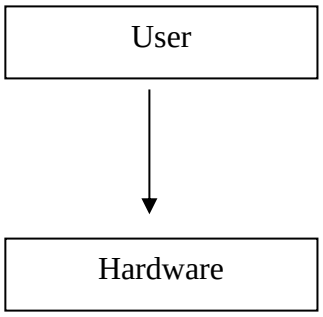
What is Hardware?

- Hardware is the *physical* part of a computer system such as:
  - Processor Box
  - Keyboard
  - Screen
  - Mouse
  - Mother Board (circuitry!)
  - Disk Drives
  - Cables
  - Expansion Cards
  - Printers
  - Modems
  - etc.

- Effectively, if you can touch or lift it, then its most likely to be hardware.

- The earliest computer systems required their users to have a detailed understanding of their hardware (indeed, very early electronic computers were programmed using a soldering iron, a reel of cable and a wire clippers – i.e. circuits had to be reconfigured to perform specific tasks – computer professionals were more likely to be very specialised scientists or engineers in the early days of computing compared to what they are today)

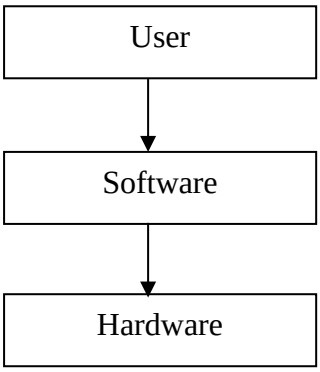
- As we've discussed in the previous section, taken in isolation, the hardware of any computer system is of little use to us - in general we cannot easily interact (or communicate) directly with it to gain any of its potential benefits



- Hardware, though generally complex, can only obey a limited range of very primitive instructions:
  - These instructions must be presented in binary (a sequence of 0 and 1 values), or as simple electrical signals (which is really only another form of binary)
  - It usually takes a large sequence of binary instructions to accomplish even a simple task
  - So direct communication with the hardware is a rather complicated, tedious and error prone activity, requiring extensive knowledge and experience.

- The hardware of a computer system has no intelligence or decision making capacity of its own:
  - All hardware can do is obey sequences of simple (binary) instructions (i.e. execute programs) - but it does this very well
  - Hardware will execute all of the instructions given to it tirelessly and accurately

- It is virtually impossible for a person the control hardware directly as it calls for unerring accuracy and patience, and is extremely tedious.
- To gain the potential benefits that hardware offers, we need a layer between us and the hardware - this layer is called *software*



  - In general, the software layer between us and the hardware:
    - allows us much simpler and more convenient access to the functionalities of the hardware
    - provides a layer of security to protect the hardware (and us) from accidental or intentional damage

What is Software?

- Software is a general name for any of the programs that run on the computer (again, we'll discuss what a program is shortly).
- When we communicate with the computer system, we are in fact communicating with software, not hardware (see previous diagrams).
- Software transforms the commands or instructions we give it into sequences of simple signals that the hardware can understand and carry out
- Software can take many forms:
    - operating systems (such as Windows XP,7,8.., UNIX, MacOS, etc.)
    - compilers (for Java, C++, Pascal, COBOL, etc.)
    - networking software
    - operating system or other utilities
    - applications (such as MS Word, MS Excel, Oracle, etc.)
    - device drivers
    - user programs
    - etc. etc.
- The intelligence that most of us perceive a computer to possess is actually contained in the software
  - The software in turn is a set of commands that tells the hardware what to do
  - The software is written by a programmer – so software is only a reflection of a programmer's intelligence.
  - Hardware is virtually devoid of any abilities that we could consider to be intelligence or initiative – it depends exclusively on software for all of the decisions it has to make – software in turn is only a representation of a programmer's wishes.
  - Computers have no intelligence of their own.

What is Firmware?

- *Firmware* is a sort of hybrid between hardware and software.
- It is generally considered to be software implemented directly upon hardware.
- An example you will have seen in your laboratory practicals is the PC's POST (Power On Self Test) and its BIOS (Basic Input Output System).
- The BIOS is a set of very primitive device drivers and a bootstrap programs (program loaders) that are permanently stored on a ROM (Read Only Memory) chip installed on the motherboard of the computer system.
- Every computer will have a small amount of firmware present – it plays a very small role in the normal running of the computer system, but is nevertheless essential to its operation.
- Note that applications installed on hard disk (or indeed the operating system) might also be considered to be *software implemented directly upon hardware* - but they are not considered to be firmware. They are not an intrinsic part of the computer hardware and the computer will be able to function to some degree without them (for example, we should be able to install the operating system for the first time). The computer will be effectively unusable without firmware, and this can only be remedied by making a hardware modification (e.g. installing a new ROM BIOS on the motherboard).
- In recent times the term *firmware* has been significantly misused. For example the operating system on many mobile phones is often referred to as firmware. This misuse has become so common that the meaning of the term is now changing.

Computer Classifications

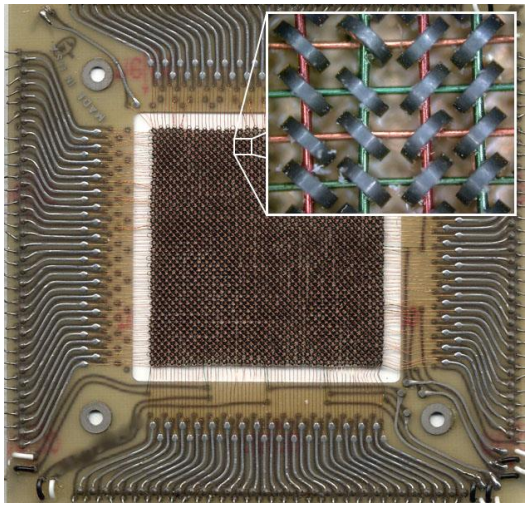
In this section we are going to try to classify the different types of electronic computer systems that are available in the world today. This is not a particularly easy task as computers have gone through many dramatic changes in their short but very eventful evolution. Perhaps we might start by studying older classifications and then try to bring matters into a more up to date perspective. Traditionally, the classifications were:

- Microcomputers;
- Minicomputers;
- Mainframes;
- Supercomputers.

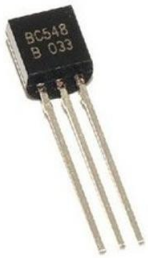
At the dawn of computing, all computers would have fallen into the *mainframe* category – very large permanent installations occupying a substantial specialised room. The principal component in these earliest systems would have been the *vacuum tube* (or *valve*) – a very large, power hungry device with a relatively short life span.

Four vacuum tubes (valves) from a very early computer system

With the advent of the *core memory* (a type of magnetic memory) and the *transistor*, which replaced valves, it became possible to greatly reduce the size of systems, allowing *minicomputers* to evolve.



Core memory from an early computer system (Image from Wikipedia)



A single discrete (not integrated) transistor

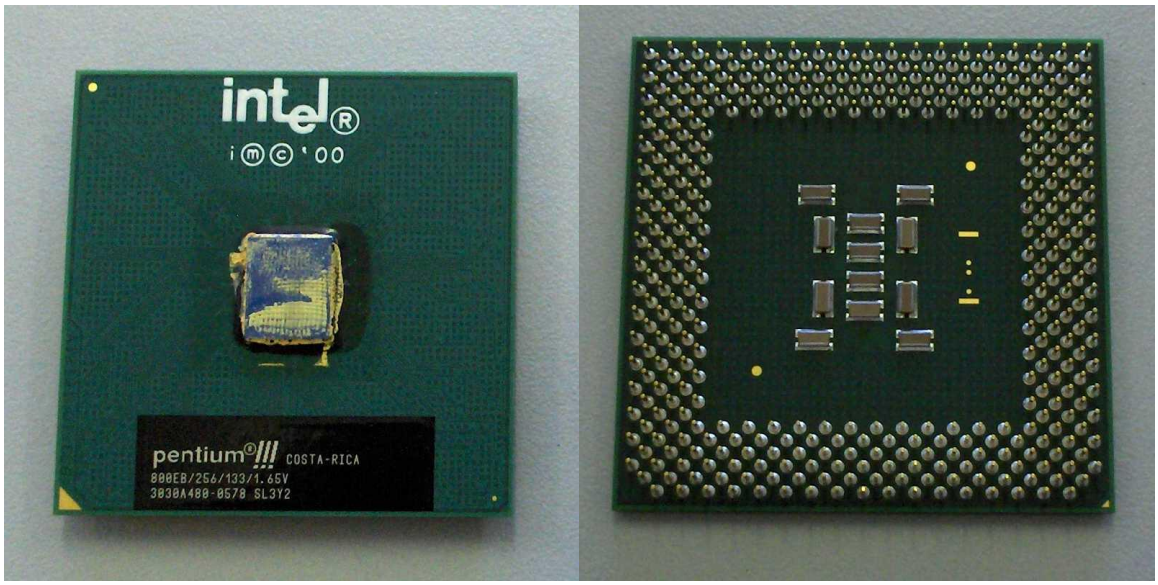
Designers and manufacturers who built their systems to be as fast and as powerful as the technology of the day would allow would have been pushing into the *supercomputer* category.

Cray-1 supercomputer circa. 1976 (image from Wikipedia)

Finally, the advent of integrated electronics opened the door to the *microcomputer* category.

Up until recently almost all computer systems fell pretty much unambiguously into just one of these four categories.

Microcomputers



An Intel Pentium III microprocessor viewed from above (left) and below (right).

A microcomputer is a computer built around a *microprocessor*. A microprocessor (above), is a central processing unit (CPU) implemented as a single[5] *integrated circuit* (IC). An IC is a circuit fabricated on a sliver of pure crystalline silicon. This is embedded in special packaging that facilitates connections from the silicon circuitry to the other circuitry in the computer system.

[5] Some authors maintain that a microcomputer may have one, or a few, ICs making up its CPU

Strictly speaking the circuit implemented on the silicon is the IC or *chip*. However it is quite usual to name the silicon circuit and its packaging using the same terms.

Packaging comes in many forms. The Pentium III previously is implemented using an SPGA (staggered pin grid array) – it is staggered due to the diamond, rather than square, pin layout. Types of IC packaging

The actual chip and the wires connecting it to the outer circuit are embedded in the packaging material, and consequently not visible. The packaging material is usually black plastic or dark coloured ceramic. The packaging in the Pentium III above is much the same as the fibreboard used to make printed circuit boards.

Note that while a microprocessor, by definition, must be an IC, not all ICs are microprocessors. For example, a memory chip is clearly not a microprocessor.

We are all familiar with microcomputers. The IBM personal computer (PC) and the Apple Macintosh have for a long time dominated the sector. The IBM PC gets its name because it was invented by IBM and launched in 1981. Most PCs nowadays are manufactured by other manufacturers (DELL, HP, Compaq, Lenovo, Siemens, etc.) but these essentially conform to the design criteria initially laid down by IBM.

Many people believe that IBM invented the microcomputer, but this is not the case. Many types of microcomputers were available for many years before IBM entered the market (List of early microcomputers , Museum of old computers). Examples include the Commodore Pet, Radio Shack's TRS-80 and the Apple II. These were essentially used for home computing. Early commercial microcomputers were made by Digital, ICL, Olivetti, Altos, Rair, etc.

These very early microcomputers were based on a number of different microprocessors, notably the MOS Technology 6502, the Motorola 6800, the Intel 8080 and the Zilog Z80. Curiously enough, though many of these processors were first introduced nearly forty years ago, they are still readily available from electronic suppliers (such as Farnell and RS) as there is still an appreciable demand for them.

Early microcomputers used various operating systems, CP/M and MP/M probably being the best known. These operating systems were fairly primitive by today's standards, but still impressive for their time.

The IBM PC is based on the Intel 8086 family of microprocessors. This family started with the Intel 8086 itself and continued with the 80186, the 80286, the 80386, the 80486 (or simply the 486), the Pentium I, II, III, IV, and continues with the dual core and multi core processors of today. Minor variants of the family include the Intel Celeron and Intel Centrino processors.

Many manufacturers, such as Cyrix (earlier) and AMD (latterly), created *clones* of the Intel chips – these were completely separate designs to the Intel family, but could execute 8086 machine code, so could therefore be used to build clones of IBM PCs. They were *functionally identical* (well, almost!) even though their actual implementations were significantly different.

As the family of 8086 microprocessors developed, they always maintained *backward compatibility*. This means that new versions of the processor can always execute programs developed for earlier generations. In principle we should expect code from an original 1981 PC to be able to run on the latest PC model. Backward compatibility was an extremely important marketing strategy. It encouraged PC users to upgrade to newer PCs as they would still be able to use the programs they had bought or written for the old machines, and copy the data they had created from the old machines to the new machines, with minimal difficulties. Jumping to different architectures was considerably more costly and problematic.

The IBM PC used the PC-DOS operating system (i.e. Microsoft MS-DOS for the IBM PC) initially, which was replaced by successive releases of DOS and then the Windows operating system. More recently, other choices have become available such as Linux and Ubuntu.

The Apple II microcomputer was based on the MOS Technology 6502 microprocessor. Apple Macintosh microcomputers were based on the Motorola 68000 family of microprocessors (also backward compatible). The family started with the 68000 and continued with the 68010, 68020, 68030, 68040, etc. Some time ago, Apple moved from the CISC based 68000 architecture and adopted the RISC based PowerPC processors (from an alliance of Apple, IBM and Motorola). More recently still, Apple have moved to the Intel 8086 line of processors. Early Apple Macintosh computers used an operating system called *Finder* and *MultiFinder*. Currently Macintoshes use *OS X*, which is a UNIX based platform.

Microcomputers dominate the world computer market today, being represented by desktop models, laptops, tablets, pads, handhelds, gaming consoles, phones, smart watches, etc. They have infiltrated almost every aspect of modern life, from the domestic environment to business, government, education, research, leisure and the military to name but a few.

Wikipedia on microcomputers Wikipedia on microprocessors

Minicomputers

The minicomputer was a computer designed for modest to large sized businesses, third level educational establishments and similar institutions. They were typically around half to double the size of a standard sized filing cabinet and could operate in normal office type accommodation (preferably a separate room, but nothing more demanding).

Minicomputers could handle 10 to 50 or 60 users working on them at the same time. Users accessed the systems via *dumb terminals*. Dumb terminals were essentially keyboards and display screens with virtually no processing capability of their own. Unlike modern PC monitors, whose video memory is located in the PC processor box rather than within the monitor itself, dumb terminals had their own video memory. A consequence of this was that the terminal displayed the most recently updated screenful of information even when the minicomputer itself was switched off – compare this to your PC.

Many of the more famous computers of the past were minicomputers, notably the PDP-8, the PDP-11, the VAX series (all made by Digital Equipment Corporation or DEC) as well as the IBM AS400 series, the Data General Nova, the Hewlett Packard HP 3000 series and many others from manufacturers such as CDC and Wang. The term minicomputer is effectively obsolete today, having been replaced by the term *midrange computer* system. A midrange computer is conceptually somewhat different to a minicomputer, however.

Wikipedia on minicomputers

Mainframe Computers

Mainframe computers have existed throughout the age of electronic computing. They are always characterised by being very large computer systems with enormous processing capabilities for their time. At the outset, they were typically installed in large air conditioned rooms (for clean and cool air) upon raised false floors. Different parts of the system were placed in different parts of the room and all of the interconnecting cable-work ran under the floors to keep the working environment tidy and clean. Each component in the system was around the size of a filing cabinet (e.g. the CPU, the memory, the hard disk drives, the tape drives, etc.).

Initially, the programs run on these machines were *non-interactive*. This means that the user did not feed input into, and receive output from their programs as they ran. Rather, the user had to anticipate what data was required by the program and queue it all along with the code. A program and its data was called a *job*. Jobs were submitted for execution and the turn around time could vary from an hour to a few days. This was known as *batch processing*. While this may seem very primitive by today's standards, it actually made computer programmers and users very disciplined and efficient. Programmers learned to get their programs right first time, every time. This skill is not so evident in today's programmers.

Mainframe computer systems ran 24 hours a day, 365 days a year and were managed by teams of *operators* who worked in shifts. Their principal role was to nurse the maximum throughput from the system so as to justify its huge cost. The operator's task was to make sure that as many jobs as possible were run quickly and efficiently through the system. They did this by ensuring that all the resources that jobs needed, such as hard disks, tapes and pre-printed stationery were loaded and ready for the job before or as it needed them. This ensured that jobs were not stalled, thus denying other jobs access to the system. Operators would study the progress of particular jobs on a *system console* and expedite their progress through the system, where possible. Operators also spent a lot of time keeping the systems clean, as early systems were very susceptible to damage from dust and dirt particles.

Mainframe systems evolved into *time shared interactive systems* with the advent of the dumb terminal. Users could now interact with their programs and submit them for execution as and when they needed to. The CPU would give each user the impression that they had sole access to the system, as the system always seemed to respond immediately to them. The systems achieved this through *time-slicing* which meant that the CPU would hop from user to user (or job to job) constantly, visiting each one perhaps 50 to 60 times a second and doing some work on it. Mainframe systems still carried out a large amount of batch processing, particularly at off peak times.

A particular advantage that interactive mainframe systems had over modern networked PC systems is that when the system wasn't busy then all of its resources were effectively available to the users on the system. On a networked PC system the processing capability has been shifted from a centralised location and shared out to each of the users. The dumb terminal has effectively become a very intelligent terminal (your PC), with very little processing done at the centre of the system any more (your servers). Each user gets a fixed small percentage of the overall system's processing power. They cannot, however, avail of any unused processing capacity from the other idle PCs on the system.

Mainframe systems are not as plentiful as they were many years ago, having been replaced to a large extent by server based PC networks. They are, however, still very much in use in large companies such as banks and utility companies (for doing jobs such as billing, accounting, maintaining centralised databases, airline reservation systems, to name but a few) and large institutions such as hospitals and the military. They are still characterised by having enormous processing capacity and enormous storage capacity, and are very expensive.

Mainframes generally are able to provide the user with a choice of operating systems, often at the same time.

Probably the most famous mainframe computer was the IBM360, launched in 1964. It implemented a huge number of new design concepts which influence computer design to the present day. The IBM360 series was followed by the IBM370 series, which was also a very influential design.

Historically, mainframe computer systems were manufactured by Burroughs, Control Data Corporation (CDC), General Electric (GE), Honeywell, International Business Machines (IBM), National Cash Register (NCR), Radio Corporation of America (RCA) and UNIVAC in America – known as *IBM and the seven dwarfs* - Siemens and Telefunken in Germany, ICL in the UK, Olivetti in Italy, and Fujitsu, Hitachi, NEC and Oki in Japan. This list is not exhaustive.

Currently IBM dominates the mainframe market with about 90% of market share. Other manufacturers currently include Groupe Bull (who bought over Honeywell information systems), Fujitsu (absorbed Amdahl and ICL (80%) and collaborates with Siemens), Hewlett Packard, Hitachi and Unisys (evolved from RCA, UNIVAC, Sperry and Burroughs).

Wikipedia on mainframe computers

Supercomputers

In the three categories of computers described so far, the ability to reorganise data is probably at least as important as the ability to do computations. Much of these computers' time is spent doing the electronic equivalent of paper shuffling (that isn't intended to sound in any way facetious). Many of the computations that are done are relatively simple arithmetic operations such as totalling and averaging, for example. Many of the operations are integer based.

There are, however, many activities that require the computer to perform hugely complex mathematical computations (using *floating point numbers*) rather than just moving data and doing simple calculations. Such activity is known as *number crunching*. Big number crunchers are needed by people working in the fields of engineering, science and mathematics. The biggest and best of these machines are known as *supercomputers* and are designed principally for their ability to process *floating point numbers*.

For example, weather forecasters take a constant stream of data such as temperature, wind speed and direction, relative humidity, and so on, from a multitude of locations all over some geographic area and have to process all of this data on an ongoing basis to establish what is going on with the weather systems in that area. Then they have to process this data further to try to predict as far in advance as possible, what is going to happen with the weather in the future. This requires a massive amount of processing power that even a big mainframe will be hard pressed to cope with.

The power of conventional computers is generally measured in *millions of instructions per second* (MIPS). Supercomputer performance, however, is measured in floating point operations per second (FLOPS), used with an SI prefix such as Mega-, Giga-, etc.

The currently fastest supercomputer, the Sunway TaihuLight, in mainland China can deliver 93.01 PetaFLOPS. This is 93,010,000,000,000,000 or 93.01 quadrillion floating point operations per second. To put this into perspective the Intel core i750 microprocessor (Oct. 2012) can achieve up to 7 GigaFLOPS without being over-clocked. So, the Sunway TaihuLight has the equivalent processing power of a staggering thirteen and a quarter million Intel core i750 processors. The Sunway TaihuLight is also very significant insofar as most of its components are of Chinese origin, rather than being sourced from the West.

(Update: since June 2018 the US has taken over the lead with a supercomputer called the Summit clocking in with an awesome 122.3 PFLOPS, equivalent to seventeen and a half million Intel core i750 processors).

(Further update: Since June 2020, the Japanese Fugaku is the world's most powerful supercomputer, initially reaching 415.53 petaFLOPS, but after an update in November 2020 it now clocks in at 442.01 petaFlops – equivalent of 63 million i750 processors)

Supercomputers use a multitude of techniques to achieve their processing capabilities. We'll look at a few of these now.

A very common approach is to employ parallel processing. Conventional computers have a single CPU and thus can only execute a single instruction from a single program at any instance in time. Parallel processors can process more than one instruction at a time. This can be achieved in essentially two ways.

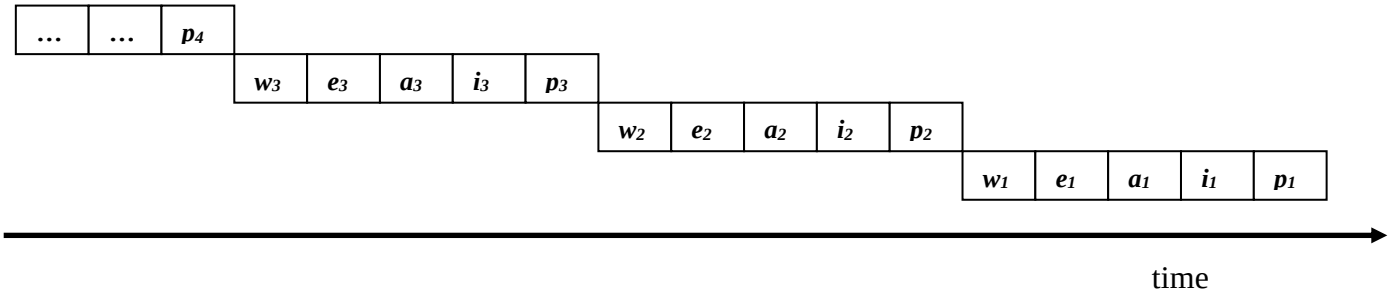
The first way is called *Single Instruction Multiple Data* (SIMD) parallelism. In this approach we have a single CPU executing the same instruction on multiple streams of data simultaneously. Such a processor is very suitable for doing operations like adding vectors or matrices – the same operation is done on multiple items of data at the same time. For this reason SIMD computers are often called *vector* processors. Their CPUs are often viewed as having a single *control unit* (which processes program instructions) and multiple *ALUs*, one for each stream of data. The ALUs are said to operate in *lockstep*, meaning they are all doing the same operation at the same time, but on their own stream of data. Programming languages for SIMD processors typically support vector and

matrix data types as well as conventional scalar data types. Because SIMD parallelism occurs down at the instruction / data stream level, it is also known as *fine grain* parallelism.

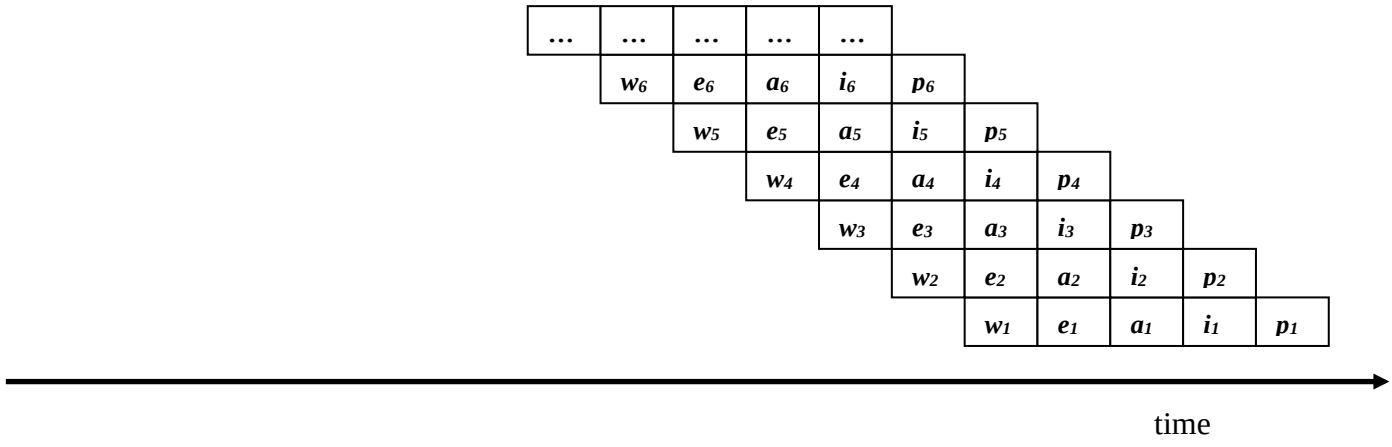
The second way is called *Multiple Instruction Multiple Data* (MIMD) parallelism. In this approach the computer has multiple CPUs operating simultaneously and can truly execute multiple programs all at the same time (or possibly different parts of the same program, where the execution order of the parts doesn't matter – we say that these parts are *mutually exclusive*). Compare this type of parallel execution to *concurrent* execution which we introduced earlier. A MIMD computer might be viewed as multiple computers sharing resources such as memory and secondary storage very tightly. Because MIMD parallelism occurs up at the program / program section level, it is also known as *coarse grain* parallelism.

If we employ both techniques simultaneously to create a hybrid[6] system, it is termed an MSIMD processor.

Yet another approach to parallelism is called pipelining. Each instruction that the CPU executes involves a number of distinct phases. For example, the Intel 486 CPU took five *ticks* to execute an instruction, one tick for each phase - *Prefetch, Instruction decode, Address generate, Execute* and *Write back*. If we have a sequence of instructions I_1, I_2, I_3, I_4, etc., … from a program that has to be processed by the CPU where the phases of I_1 are denoted $\boldsymbol{p_1}$, $\boldsymbol{i_1}$, $\boldsymbol{a_1}$, $\boldsymbol{e_1}$ and $\boldsymbol{w_1}$, (and similarly for the other instructions in the sequence), then the CPU's activity over time will be as follows:



The circuitry that performs the prefetch phase for an instruction is idle while other phases are happening - and likewise for the other phases. Pipelining exploits this redundancy by starting the prefetch phase of the second instruction while the instruction decode phase for the first instruction is taking place. Then, it starts the prefetch phase of the third instruction while the instruction decode phase of the second instruction and the address generate phase of the first instruction are happening - and so on for subsequent instructions. At any particular moment in time, five different phases from a sequence of five different instructions may be taking place (*in the pipeline*), as shown below:



[6] Remember from earlier that a computer that is part analogue and part digital is also called a hybrid system. The two types of hybrid systems encountered so far are completely unrelated. So, be very careful when you see the term *hybrid*.

This gives a pipelined processor with a five phase instruction cycle an immediate speed up approaching fivefold compared to its non-pipelined equivalent. Pipelined processors are very advanced architectures - they require that the phases of the instruction cycle are mutually independent (i.e. they can occur simultaneously) and also that the duration of each phase is the same. It should be also apparent that the more phases an architecture has for its instruction execution, then the more it can benefit from a pipelining approach.

A *superscalar* architecture is where multiple pipelines are implemented allowing instructions to be executed in parallel (i.e. multiple instructions per tick).

Current PC processors employ all of these techniques to some extent (e.g. SIMD pipelines within the processors, and multiple cores) but don't come anywhere near to qualifying as supercomputers. However, many current supercomputers are effectively huge clusters of PC (or other) processors combined together into a single machine.

Wikipedia on supercomputers

Ten of the coolest and most powerful supercomputers of all time

Updated Computer Classification

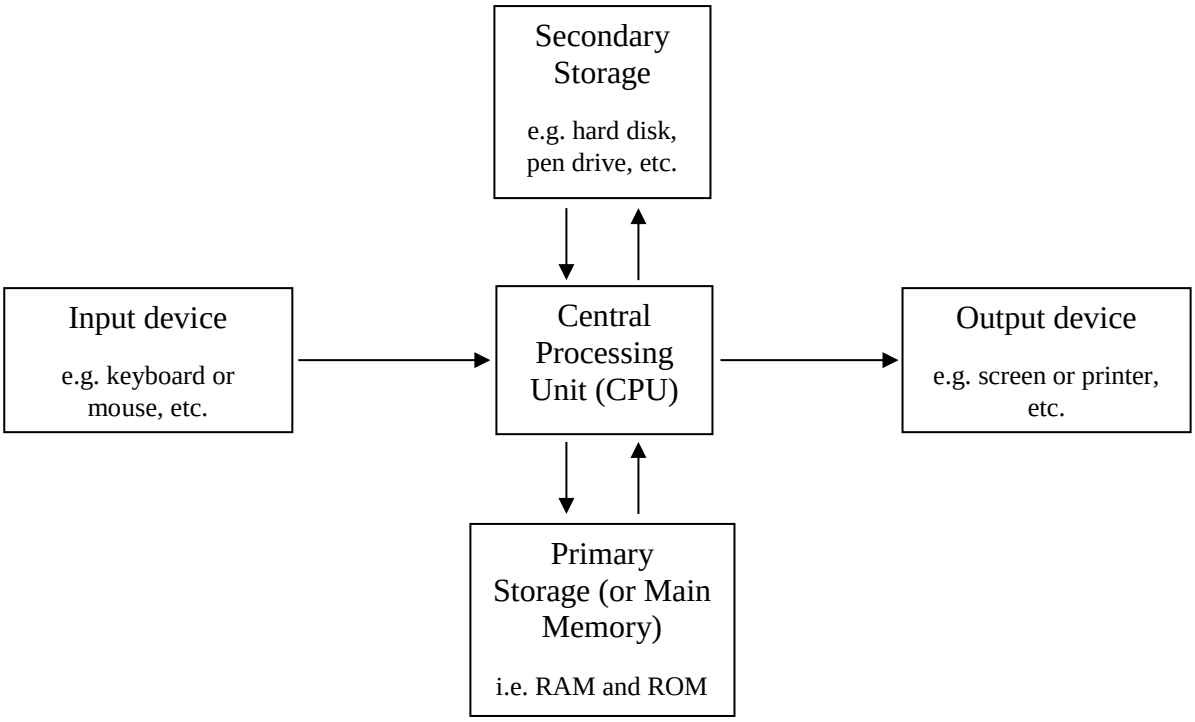
- Supercomputers;
- Mainframes – still tend to be large commercial or institutional machines with very significant processing capability. Less dependent on specialised environments to operate in. In a lot of instances where mainframes would have been used in the past, they have been replaced by server based PC networks;
- Minicomputers – now replaced by midrange computer systems / midrange servers;
- Microcomputers – this is the area where most diversification has taken place. The classification might now include:
    - Servers – High end microcomputers engineered to provide services to other networked computers;
    - Workstations – essentially high end desktop computers with powerful processors and lots of memory capacity – often used for specialised purposed such as 3D graphics, animation, game development, etc.;
    - Desktop computers – the 'classic' microcomputer example - much less portability but more versatility than their portable equivalents;
    - Laptops and Notebooks – high portability but less versatility than desktops – equipped with displays, keyboards and pointing facilities such as mice or touchpads – battery powered – Hard disk or Solid State Drives (SSD), or cloud based;
    - Tablets – similar to laptops, but using a virtual keyboard and pointing device via a touch sensitive screen – flash memory storage (or cloud based) rather than hard disk;
    - Palmtops or Personal Digital Assistants (PDAs) – these are roughly placed between tablets and mobile phones and are being squeezed out by them – stylus based touch screens - virtually obsolete;

- Mobile phones – initially simply mobile telephones, but have become small tablet computers with phone capabilities;
- Smart watches - It's probably too soon to say if this technology will become popular or simply disappear ;
- Game consoles – specialised microcomputers for entertainment purposes;
- Embedded systems – very specialised computer systems built into other devices. Usually simpler processors are used with non volatile memory preloaded with software that is intended to run indefinitely. Examples would include controllers for microwave cookers, washing machines, DVD players, engine management systems for cars, digital cameras, etc ;
- Wearable technology – microcomputer systems that can be incorporated into clothing or jewellery – e.g. Arduino Lilypad;
- Disposable technology – as electronic technology advances and becomes cheaper it is only a matter of time before it becomes disposable. For example shrink wrapped plastic screens on bottled products advertising their contents have been speculated upon. The screen technology and circuitry technology pretty much exist already – the main problem is to develop flexible batteries - only time will tell.

Computer Generations

	Era (approx.)	Switching Element	Memory	Input	Output	Storage	Interface & Programming	Examples
1st	1940 to 1956	Vacuum Tube	Drum Memory	Paper Tape & Punch Card	Printouts	Paper Tape and Punch cards	Machine Code – one problem at a time	ENIAC UNIVAC
2nd	1956 to 1963	Discrete Transistor	Core Memory	Punch Card	Printouts	Magnetic Tape	Assembly & early Fortran & COBOL – Batch Processing	IBM 7090
3rd	1964 to 1971	Integrated Transistor (SSI & MSI ICs)	Core Memory to RAM / ROM	Dumb Terminals & Keyboards (Interactive Systems)	Dumb Terminals and Printouts	Hard Disk	Fortran and COBOL – Multi-Program O.S.	IBM 360 PDP 8
4th	1971 to Present	Integrated Transistor (LSI & VLSI ICs) - Microprocessors	RAM / ROM	Networking – Internet - Mouse	Networking - Internet	Hard Disk etc.	C, C++, etc. GUI OS Voice Recognition	PDP 11
5th	Present onwards						AI, Parallel Processing, Superconductors, Quantum, Molecular, Nano, DNA, etc.	

What is the basic structure of a computer system?



- A *computer* (as far as we are currently concerned), such as a PC, is an electronic machine that processes data according to a set of *instructions* to produce information
    - The set of instructions is called a *program* (note the spelling - not the same as a *programme* on television or a *programme* of events at a concert!) - we'll talk about programs in a short while.
    - A program can be used, changed or replaced as needed - we say such a computer is *programmable*
    - Such a computer is a *stored program electronic information processor* (such as a PC, Macintosh, UNIX workstation, etc).
    - In our very general list of example computers (apart from the PC) on the first page of these notes, most were not *programmable*, that is to say we cannot easily modify their behaviour to do new tasks for us. On a programmable computer we can install new programs that instantly increase the functionality of the machine. All of the other devices are programmed typically during manufacture to do a single or a small number of tasks. For example, with a sundial we would carefully need to shape and position both the triangular fin that casts the shadow and the curved dial that the shadow falls upon, then we would need to position the whole assembly carefully in its final resting place relative to the movements of the sun - once this initial "programming" is done the devices functionality is set and will not generally be subsequently altered.

- The Processor, or CPU (Central Processing Unit), is a complex electronic circuit that carries out the instructions present in a program to transform raw data into information. Programs (and indeed data) must always be *loaded* into primary memory (RAM and ROM on a modern system) before the CPU can process them.

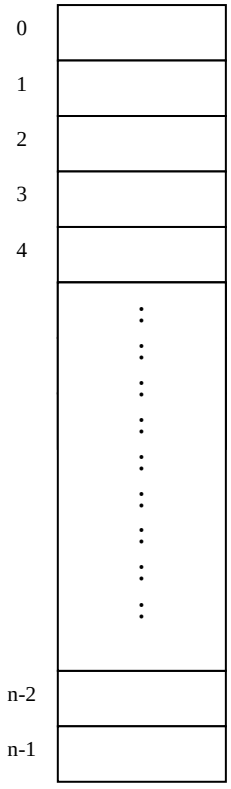
- Data has to be fed to the CPU by the user of the program - this is the purpose of an input device. Input devices can take many forms.

    - A PC will have the following input devices as a standard minimum configuration:
        - Keyboard
        - Mouse
    - Many other input devices are possible:
        - Touch Sensitive Screens (these are also output devices)
        - Microphone
        - Document Scanners
        - Digitising Pad
        - Bar-code Scanners
        - Card Reader (e.g. ATM cards, Phone Cards, Student cards, etc)
        - etc.

- Information has to be returned by the program to the user of the program - this is the purpose of an output device. Output devices can take many forms.
    - A PC will have the following output devices as a standard minimum configuration:
        - Screen (or Monitor)
    - Many other output devices are possible:
        - Printers (inkjet, laser, thermal, etc.)
        - Speakers
        - Data Projectors
        - Touch Sensitive Screens (these are also input devices)
        - etc.

- Note that input devices can be used to get programs into the system as well as data.

- We've seen that the CPU's activities are controlled by a set of instructions called a *program*. While the term program can mean many different things (which we'll explore shortly), the programs that the CPU executes are typically very long lists of very simple instructions expressed in binary and termed *executable programs* (to distinguish them from other manifestations). Binary code is made up of sequences of *bits* (Binary digITS). A bit can only have one of two possible values - zero or one.



- The instructions in the program must be readily and rapidly accessible by the CPU. For this reason active (i.e. executing) programs are stored in *Primary Storage* (also called *Main Memory*) which is made up of RAM (random access memory) and ROM (read only memory).

- The CPU, as it is being instructed by a program on how to transform data into information, also needs to be able to rapidly access the data and store the resulting information. Primary storage is also used to temporarily hold data after it has been input and awaiting processing, and it holds the resulting information after processing before it is dispatched to an output device.

Main memory or *primary memory* consists of a long series of storage locations that are used for holding program instructions and data (completely interchangeably). Modern computer systems allow a single *byte* of information (i.e. 8 bits) to be stored at each location. This is the

smallest amount of information that can be moved either to or from memory at any one instance.

Every location has a unique *address* associated with it. Typically, if we have *n* bytes of memory, then its addresses will run from *0* through to *n-1*, as shown in the accompanying diagram. Note that an address is not stored anywhere – only the data at that address is stored.

Memory is used to hold programs, and the data associated with these programs, as they are being executed. To execute a program requires the computer system to *load* its code and data from secondary storage (such as hard-disk) - and data from input devices - into main memory. As the CPU processes the program, each instruction and its operands are loaded from main memory into the CPU for processing in turn.

To move data to memory (called *writing* memory) or from memory (called *reading* memory) requires that the CPU first tell memory which particular location is to be used to effect the transfer. This is known as *addressing memory*.

- Two types of memory are used on modern computer systems, RAM and ROM (since around the 1970's). They are both used for holding programs and data for processing by the CPU.

    - RAM is an abbreviation for *Random Access Memory*.
    - RAM has the three principal properties:
        - It is possible to *write* information into the storage area of RAM. We may also have information that has already been stored in RAM's storage area *read* back and made available to the rest of the computer system. We say that RAM is a *read/write* storage medium.
        - Access to information in RAM is by *direct* (or *random*) access (which is where RAM derives its name). This means that we can access a piece of information stored in any part of RAM with exactly the same effort as if it were stored in some other part of the RAM (compare this to information on a rewound magnetic tape cassette - access to information at the end of the tape is considerably more difficult than information near the start of the tape - magnetic tape is an example of a *serial access* storage medium).
        - Finally, RAM has one serious disadvantage in that it loses all of its information when we turn off the power supply that controls it. We say that RAM is *volatile*. A consequence of this is that the contents of RAM will be completely spurious immediately after powering up the computer system – computer professionals will often say that memory contains *rubbish* or *garbage* in such an instance.
    - Most of primary memory is implemented as RAM
    - The CPU of a computer system can only do one thing - execute binary instructions that are part of a program. Programs are executed from RAM. When we power up our computer system, the CPU has no code that it can execute (as RAM will have lost its previous contents and only contain garbage). Unless it can execute some code, the CPU will stall before it even gets started. For this reason, the primary memory of every computer system will contain a small (but vitally essential) proportion of *ROM*.

    - ROM is an abbreviation for *Read Only Memory*.
    - ROM has the three principal properties:

        - It is generally not possible to write information into the storage area of ROM. It is, however possible to read information from it (hence its name). We say that ROM is a *read-only* storage medium. (Actually, if we can read information from ROM, then it must have been written there some time previously - this is done during manufacture only - in general it is not possible to write to ROM once it is installed into a normal working computer system)
        - Access to information is ROM is by *direct* (or *random*) access, exactly as for RAM. So naming RAM as it is to distinguish it from ROM is a bit confusing, given that ROM also facilitates random access.
        - Finally, ROM doesn't lose any of its information when we turn off the power supply that controls it. We say that ROM is non-*volatile*. Code stored in ROM will be immediately available for execution on power-up. CPUs are generally designed to automatically execute code from the ROM portion of primary memory at start up time. Typically, code from ROM will be immediately copied to RAM for execution, as ROM based code can't support program variables (remember we cannot write to ROM, so we cannot assign values to variables mapped into ROM storage).
    - Only a small proportion of primary memory is implemented as ROM

Summary of characteristics:

RAM	**ROM**
Read/Write capability	Read Only capability
Direct or Random Access	Direct or Random Access
Volatile	Non-Volatile

- So far, the computer system we have described will be very unwieldy and awkward to use because we will have to enter the code for every program we wish to execute each time we wish to execute it, as well as the current data it has to process, through an input device.

- Input (and indeed output) devices tend to be rather slow at passing data / information into (and out of) the system compared to the speeds that the CPU can process the information at.

- Some form of semi-permanent storage within the computer system is very desirable. This would allow us to store programs that we use a lot within the system instead of having to read them in from an input device every time we need them. It would also allow us to store data/information that we need to regularly update. Therefore virtually every computer system has some form of *secondary storage*.

- Secondary storage might include the following devices:
    - floppy disks (low capacity and extremely slow - effectively obsolete)
    - hard disks - high capacity and high speed (though still slowish compared to processor speeds) – hard disks are used extensively in the computing world – they are now also becoming commonplace outside of the immediate computing world – for example, they are now being used in audio systems

(e.g. Macintosh iPod) and video systems (Sky+ uses hard disks to record satellite television broadcasts)

- ZIP disks ("superfloppies" - rendered obsolete with the advent of memory sticks). These require a ZIP drive to be installed on your computer to read or write the disks. Few computer systems have such drives.
- CD (compact disk) - very popular for data, programs, audio, etc. - most computer systems are now equipped with CD writers as standard – these are very useful for a variety of purposes including easy back ups of important information.
- DVD (digital versatile disk) - principally for digital video, but many other applications are possible too.
- Magnetic tape - mainly used for backing up servers on networked systems - not particular popular for home computing
- Flash memory / USB pen drives / memory sticks - a really good alternative to CD and zip drives - available with capacities from about 32 Mbytes to 2 Gbytes - relatively inexpensive - extremely compact - no special drive required on your computer to utilise these devices.

I/O devices

A computer system would be seriously restricted if it were unable to accept stimuli from the outside world (i.e. receive input) and / or return feedback to the outside world (i.e. send output). For this reason, our model of a computer system includes I/O devices. I/O devices take many shapes and forms, some being exclusively input, some being exclusively output, some sharing dual roles.

Examples of input devices would include keyboards, mice, trackballs, scanners, bar code readers, smart card readers, microphones, graphics pads, games joysticks, CD readers, DVD readers, digital cameras, etc.

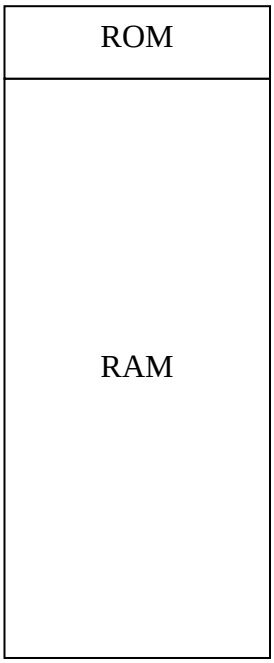
Examples of output devices would include screens (flat panel and CRT types), printers (inkjet, laser, etc.), plotters, speakers, etc.

Examples of devices sharing input and output roles might include disks (floppy-disks, hard-disks, ZIP disks, etc.), tapes, flash memory (pen drives), CD burners, DVD burners, touch sensitive screens, etc., as well as network connections.

Composition of Memory

Primary memory on modern computer systems consists of two particular types of storage, RAM and ROM, whose characteristics were described earlier.

As you can see from the diagram below, most of a computer system's primary memory is implemented using RAM and a relative small amount is implemented as ROM. Why do we need both? To explain this we need to explore the *bootstrap* process, i.e. what happens in our computer system in the time interval between powering up and the moment that we can actually do some productive work? Bootstrapping is the process of transforming a piece of hardware into a fully functioning computer by loading and initiating its *operating system*.



The Bootstrap Process

When we power up our computer we have a number of dilemmas facing us that have to be overcome. To begin:

- When we power up a computer, RAM contains a random pattern of binary digits (remember that RAM is *volatile*) – i.e. it contains *garbage;*
- The CPU can only do one thing – execute binary instructions – all of its capacity to make decisions comes from these instructions – it is effectively helpless without them;
- So on power up the CPU is effectively deprived of instructions - it will most likely get into an endless cycle of executing meaningless instructions (i.e. trying to interpret the garbage as valid instructions).

To overcome these problems, the CPU is designed to look at a specific address in memory (address $0FFFF0_{16}$ on the Intel 80x86 family of microprocessors) and execute the instruction it finds there. Computer designers ensure that this address is always implemented in the ROM portion of the computer's memory map.

On a PC, a machine code instruction stored at this address is executed and causes control to jump to the power-on self test (POST) which is also situated in ROM. The POST loads instructions into RAM and executes them. These instructions tell the computer to do a quick diagnostic check of a number of items of core hardware on the motherboard of the computer (e.g. CPU, memory, etc.).

The POST next checks that the system set-up is still valid. This set-up (or configuration) is stored on a complimentary metal oxide semiconductor (CMOS) chip that is powered by a small battery such as a Lithium cell (that many watches might use). CMOS technology is used because it has a very low power consumption. The system set-up includes information such as date and time settings, installed floppy drive and hard drive configurations, and any other information that might differ from one particular computer to another. Assuming that the POST finds no problems, its task is now complete.

Control passes to the ROM Bootstrap Loader which, as its name suggests, is located in ROM. The task of this code is to find the disk bootstrap routine (operating system loader) on disk, load it into memory and run it. We have another problem here which is also solved by the presence of ROM on the system. To utilize any of the devices on a system we require software routines called device drivers. These are located on disk, but we need those same drivers to access them. We used the analogy of locking your keys in your car to describe this situation.

Once again ROM comes to the rescue. Another part of ROM contains the BIOS (basic input out put system). The BIOS essentially contains a number of very primitive device drivers that the computer needs to get started but before it has had a chance load the necessary full device drivers from the boot disk. For example, the BIOS disk driver is only capable of reading the boot record from the boot sector (typically the outermost track) of the boot disk – but this is sufficient to get started. The BIOS has primitive drivers for at least the following:

- Floppy disk
- Hard disk
- CD drive
- Monitor
- Keyboard (lights flicker immediately after power up)
- Mouse
- Printer (activity noticed immediately after power up)
- etc.

The ROM bootstrap loader begins by looking at the A: drive for a bootable floppy. If none is found it then looks at the computer's hard disk. On more recent systems it will also look at the CD drive.

Either way, it loads the disk bootstrap routine from the first sector (also called the boot sector) of the system start up disk into an arbitrary location in RAM and then transfers control to it. Also found on the boot sector is a table of information about the disk format.

The disk bootstrap routine checks to see if the disk contains a copy of the operating system files by reading the first sector of the root directory and checking if the files are present. If they cannot be located the user is typically prompted to change disks and to hit any key to resume.

If the files are found, they are loaded into memory and executed. Initially they will configure the operating system for general unspecialised operation. Next, it will load the particular preferences for the system (for example, the driver needed for a scanner present on this system) and configure the operating system appropriately. Finally, it will load the particular preferences for the current user and then provide the user with their normal working environment – i.e. the system is fully booted and the user can begin to work on it.

One final point – programs that execute directly from ROM are necessarily very limited in what thay can do – the reason for this is that they cannot have variables (as to assign a value to a variable requires us to write a value to memory – remember that ROM is read only). Two approaches can be

used – the first is to copy the program to RAM and execute it from there – the second is to implement the ROM based program's variables in RAM.

Memory

To begin with, memory (in the broadest sense) might be considered to exist in a number of forms in a computer system. These can normally be organised into three distinct levels:

- CPU registers (as described above);
- Main memory (RAM and ROM);
- Secondary (or Backing or Auxiliary) Storage (e.g. Disks, Tapes, etc);

We also have a type of memory called *cache memory* – we would logically place this between the first and second levels listed above. For the present, however, we'll not consider cache memory until later on.

The following table compares the three levels above under the following headings:

i) alternative names and examples of each type;
ii) quantity available on a typical system
iii) relative cost per byte;
iv) relative access speed;
v) the typical content you would expect to find at each level.

	Level 1 (top)	Level 2 (middle)	Level 3 (bottom)
Names (e.g. main memory): Examples (e.g. disks):	Registers / CPU Registers / Accumulators In the Intel 80x86 archictecture these would include AX, BX, CX, DX, SI, DI, etc.	Main memory (or store) primary memory, first level store, etc RAM and ROM (Older systems: Drum Memory, Valve Memory, Core storage, etc)	Secondary, auxiliary, backing, 2nd level) store / memory etc. Floppy disks, hard disks, CDs, DVDs, Zip Disks, Tapes, USB drives, SD cards, etc.
Typical quantity found on a computer system:	Typically 10 – 100s of bytes	Currently around 1 - 4 Gigabytes	Currently around 200 Gigabytes to 2 Terabytes
Relative cost per byte:	Expensive (essentially requires a full CPU redesign)	Cheap (≈ €20 for 1 Gigabyte)	Very cheap (≈ €65 for 1 Terabyte)
Relative speed of access:	Very fast (they're within the CPU so are accessed directly - no addressing has to be done - etc)	Fast (outside of the CPU so have to be addressed and transferred to CPU for processing, then results are returned)	Slow to very slow (devices are often mechanical)
Type of program and data contents you would expect to find present.	The current instruction and its operands from the currently active program	The currently active program and its data (loaded into memory for execution)	Dormant programs and data files - can be accessed and loaded relatively quickly.

Memory in the current context refers to main or primary memory only (RAM and ROM).

What is a Program?

The term *program* can mean different things in different contexts. A program can exist in a number of different forms which we will examine in a moment. Whatever their form, however, all programs have a number of common qualities.

- A *program* is a sequence of instructions that tells a computer system precisely what to do in order to carry out some task for a user working on that system.
- All decisions that the CPU has to make are actually determined by the program.
- By default, each instruction in a program will be carried out in a very strict sequence. The first instruction will be carried out first, the second instruction will be carried out second, the third instruction will be carried out next, and so on until all of the instructions have been carried out.
- Some instructions will explicitly change the *one-after-another* sequence of instruction execution – this is used to implement sections of conditional code and loops.

An *executable program* or *executable image* contains instructions expressed as *binary* numbers. A binary number consists of a sequence of zero and one values (also called *bits*, an abbreviation for "BInary digiTs"). In essence, each instruction that the computer can execute has a corresponding binary value. An executable image is the only manifestation of a program that the CPU can recognise and execute. Each instruction does a very simple task (such as moving a value from one place to another, or adding two values, or comparing two values, etc.). We say that such instructions have a *low level of functionality*. In general it takes a significant number of binary instructions to accomplish any meaningful task because of their low level of functionality.

On a PC system, a file containing such a program has a .EXE file extension (an abbreviation for "executable"), or occasionally a .COM file extension. A human cannot generally read or understand the contents of such a file, and if typed out to a screen or printer, the file will produce strange symbols and effects. We refer to such a file as a *binary* file. We also refer to the instructions in an executable program as *machine code*, as they are designed to be understood and executed by the CPU, but are not intended to be comprehensible to us.

```
1010110101011010
1010101010101111
0101000100100101
1101010001001001
0011011111101111
0000001110101010
1010101100001110
0101011100100110
0010100110010010
0010110001010110
0100001010110010
1010101010101111
```

To execute a program, it must be *loaded* (copied from its executable file on secondary storage into primary memory (RAM)), from which the central processing unit (CPU) will carry out its instructions. Loading is automatically done by the operating system when we launch or invoke a program. We invoke a program by typing its name at the operating system's command prompt or double clicking its icon on the screen.

Every design of computer system has its own specific type of machine code. Code for one type of computer cannot be executed on a different type of machine. For example, we cannot expect to be able to take a program from a PC system and execute it on a Macintosh or a Sun Workstation, or vice-versa. We say that machine code is not *portable*.

Machine code should, however, be able to run on different computer systems within the same family. For example, we can expect a program that runs on an old 486 based PC to also run on a PC with a Pentium, a Pentium II, a Pentium III or a Pentium IV processor, as all of these machines belong to the same family, that is, they all have fundamentally the same *architecture*. Often, however, programs written for more recent members of a family will not be able to execute on older members of that same family as new instructions may have been made available on later models.

Assembly Language is another form that a program can exist in. An assembly language program uses recognisable symbols (called *mnemonics*) that allow a programmer to express binary instructions in an English-like form. For example ADD AX,10 might represent some sequence of binary digits. Assembly language is just a convenient mechanism for programmers to write primitive machine code instructions and as a result is termed a *low level language*. A programmer needs to write many assembly language instructions to carry out even relatively simple tasks. Very little program code is written in assembly languages nowadays, and usually only by very specialised programmers for very special reasons (usually where speed or storage utilization is a contentious issue). The programmer requires a detailed understanding of the architecture of his target computer system if he or she is to program it using assembly language.

Assembly language files on PC systems usually have a .ASM file extension. Such a file can be typed to the computer's screen and read by the programmer (unlike an executable image). We refer to such a file as a *text file* or occasionally as an *ASCII file*. The computer cannot execute assembly language programs. They must be translated into machine code by a special program called an *assembler* before they can be executed. It is possible to partially translate machine code back into assembly language code using a program called a *disassembler*, though this is not done very much. As assembly language code is intimately associated with machine code and consequently with the architecture of the target computer, it too is not portable to other types of computer system outside of the target computer's family.

Usually one assembly language instruction corresponds to one machine code instruction (occasionally more). They have the same low level of functionality. The following assembly language program outputs the message "Hello, World" to the user. By the way, most of the work for this program is done by the Operating System, not the program itself.

```
        .MODEL small
        .STACK 100h
        .DATA
HeloMsg DB      'Hello, World',13,10,'$'
        .CODE
        mov     ax,@data
        mov     ds,ax                   ;set DS to point to data segment
        mov     ah,9                    ;DOS string print function
        mov     dx,OFFSET HeloMsg       ;point to "Hello, World"
        int     21h                     ;ask OS to display "Hello, World"
        mov     ah,4ch                  ;DOS terminate program function
        int     21h                     ;terminate the program
        END
```

Most program code is written using *high-level languages*. Examples of high-level languages include JAVA, C, C++, Visual C++, Pascal, Delphi, Visual BASIC, COBOL, FORTRAN to name but a few. A program will usually be stored in a file whose file extension indicates the language used (e.g. .JAVA, .C, .CPP, .PAS, .VBA, .COB, .FOR, etc.). Each language is usually tailored for a particular type of task. For example, a visual language will allow you to write applications (programs!) that will run on Windows based computer systems such as PCs. COBOL (COmmon Business Oriented Language) has been used for many years for writing business programs (COBOL programs are the main source of the millennium bug!). FORTRAN (FORmuch mula TRANslation) was used extensively for scientific programming, though is not really used any more, having been replaced by C, C++ and other more up to date languages. JAVA is used for writing code for the World Wide Web (amongst other things). High-level language source code files can be easily read and printed as they are stored in text format rather than binary.

Each instruction in a high-level language allows the programmer to express very involved procedures in a convenient and readily comprehensible manner. High level language instructions are much more complex than low level language instructions. That is, a single high level language instruction will achieve the same amount of work as possibly dozens of assembly or machine code instructions. We say that high level language instructions have a *high level of functionality* compared to assembly or machine code instructions. Consequently, high level programs tend to be much shorter than their low level equivalents.

Like assembly language code, high level language code cannot be executed by a computer. It too must be translated to machine code. This is achieved using a program called a *compiler*. A compiler is an extremely complex program (as high level language code is very complex and very functional). High level languages are far removed from the architecture of the target computer system, that is, a programmer does not have to be particularly knowledgeable about its architecture. For this reason, high level languages tend to be portable, that is, the code (if the programmer takes due care) can often be run on more than one type of computer. It will need to be recompiled for each machine, however, as a compiler's output (machine code) is specific to particular machine types. Each target machine will require its own compiler for each high level language it needs to translate.

The following Java program outputs the message "Hello, World" to the user.

```
class HelloWorld {

    public static void main(String[] args) {
      System.out.println("Hello World");
    }
}
```

Finally, here is the C equivalent:

```
#include <stdio.h>

void main(void) {
  printf("Hello World\n");
}
```

The term program may have other meanings in some other circumstances. From now on in this hardware / computer architecture module, we'll use the term to mean *binary* or *machine code* that is

ready for immediate execution by the CPU, as described in the second instance above. You will adopt a different understanding in your Java programming course.

Normally programs reside on some form of secondary storage (such as hard-disk). All of the programs on secondary storage are *dormant* or inactive. When we wish to execute a program, we must first *load* it into main memory (i.e. make a copy of it into main memory) and then set the CPU to work on it. The CPU reads the program's instructions (one by one), as well as any data the instruction must operate on, into its registers, executes them and, if necessary, writes results back to memory before going on to the next instruction that needs to be processed.

In general, machine code instructions are carried out one after another in sequence (akin to high level language statements). Carrying instructions out one by one in a strict sequence is too restrictive, and sometimes we may prefer to take different routes through the code; for example to implement a loop (similar to a high level language's while statement) or a conditional statement (similar to a high level language's if…then…else statement). So some instructions will direct the CPU to take alternative routes through the code. It is the first function of the CPU's control unit to ensure that all instructions are executed in the correct sequence.

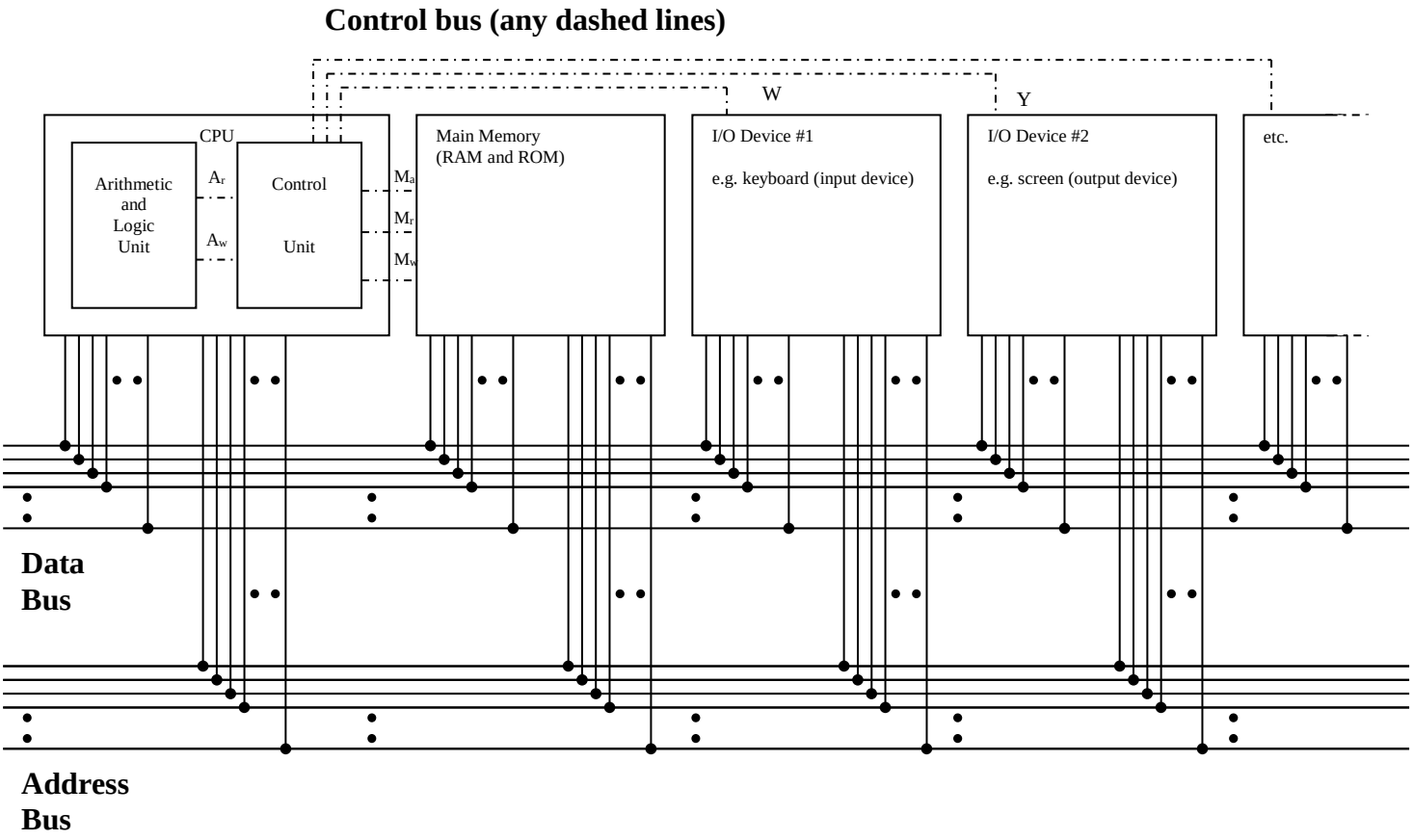
Most instructions will require the use of other components. For example, the instruction:

```
add  ax,value1     ; this is an assembly language equivalent of a machine code instruction
```

will add the contents of a variable value1 (implemented in main memory) to the contents of the CPU register ax and put the result in ax. The CPU's control unit will delegate responsibilities to other system components to achieve this task. In this case, the CPU will require the cooperation of main memory, the ALU and the CPU register ax to carry out the instruction. It is the second function of the CPU's control unit to manage the other components in the computer system, delegating tasks to them to assist it in the execution of program instructions.

Basic Computer Architecture

The diagram below shows a generalised architecture of a typical computer system. Observe firstly that the system has a number of distinct components. The most important of these are the CPU, main memory, and a variety of input and output devices. We also observe that the components are connected together in various ways. We'll begin by describing the components first.



The CPU

The CPU is the most important unit in any computer system. It has three principal parts (only two of these are shown in the diagram above):

1. The Arithmetic and Logic Unit (ALU). This part of the CPU contains circuitry for doing *arithmetic* operations (such as addition and multiplication) and *logical* operations (such as AND, OR, NOT, and so on). For each operation (e.g. addition) there will be a multitude of circuits for doing addition on all of the possible different data types supported by the machine's architecture. So there will be separate circuits for performing addition on low precision unsigned integers, low precision signed integers, medium precision unsigned integers, medium precision signed integers, high precision unsigned integers, high precision signed integers, all of the variants of floating point types, and any other data types for which addition is defined.

2. The Control Unit – this has two principal roles:
    a. It manages the execution of programs (executing the correct instructions in their correct sequence);

b. It manages every other component in the computer system and delegates tasks to them to achieve what is needed for the execution of instructions.

3. Registers – these are temporary storage areas within the CPU for holding instructions and data that the CPU is currently processing. In general these instructions and data items are read in from main memory, processed, and results are written back to main memory (if necessary) to make space for other items that are awaiting processing. There are only a few registers and they hold very little information, so are used for only very brief periods of time for each item the CPU has to process. Registers are sometimes considered part of the ALU.

   When we considered the three levels of memory found in a typical computer system previously, these CPU registers were the topmost level of the hierarchy.

   Each register has a name which we use explicitly in our assembly language programs, e.g. the instruction **ADD VAR1 , BX** will add the contents of the CPU register BX and the memory variable VAR1 and store the result in VAR1. Compare this to main memory where each location has a unique address (location number) rather than a name. When the above code is assembled into machine code VAR1 will be translated to the address in memory that VAR1 is mapped onto,

The CPU has only one source of information about what to do when it is executing a program - the instructions in the program itself. We've pointed out already that the CPU has no intelligence or intuition itself. Any apparent intuition in a computer system comes solely from the programs it executes - these in turn are written by programmers and thus reflect their thinking at the time of writing the program.

We've seen that the CPU manages the sequencing and execution of instructions in a program. To do this it goes through a cycle of activities for each instruction in the program that it executes. This cycle is known as the *instruction execution cycle*. It has essentially five phases:

- Fetch
  - Get the next instruction from main memory into the CPU - its address is in a CPU register called the program counter (PC).

- Decode
  - Analyze this instruction and set the CPU up to execute it.

- Retrieve Operands
  - If a required operand is in main memory, get it into the CPU.

- Execute
  - execute the instruction
  - write results back to memory, if required
  - update the status register SR with the outcome of the current instruction.

- Update PC
  - Update the program counter so that it contains the address of the next instruction.

When we study the internals of our model CPU, we will deal with the instruction execution cycle in considerably more detail.

Buses and Information Transfers

We mentioned above that the CPU a) manages the execution of the instructions in a program in their correct sequence and b) delegates tasks to the other components in the system to achieve this goal. To do this the CPU has to be able to communicate tasks to the other system components, and also it has to be possible for information (data, instructions and addresses) to be transferred between components. Computer systems are equipped with three *buses* for these purposes. These are:

i) the data bus — for transferring data and program instructions between components ;
ii) the address bus — for transferring addresses between components (principally between the CPU and main memory);
iii) the control bus — for sending control signals from the CPU to other components (and also for receiving status information back from components).

You should observe from the diagram above that the data and address buses appear to have a very regular structure and they connect consistently to every component in the system. The control bus, on the other hand, lacks the regularity of the other two buses. We'll concentrate on the data and address buses initially.

A bus (data or address) is a series of parallel connectors linking all of the different components in a computer system together (e.g. the brass traces you can see all over the motherboard). Each parallel connector in a bus is capable of transporting a single bit (0 or 1 value). The width of a bus, therefore, is the number of bits the bus can transport simultaneously; for example a 16-bit computer will typically have sixteen lines in its data bus.

Only two components can use a bus to transfer information at any one time – the *sender* and the *receiver* (technically speaking, every component will be able to *listen*, but this is not an issue). The CPU will determine which component is the sender and which component is the receiver. The CPU will know which components to designate for these roles, because the instruction that it is currently executing will tell it. The CPU will send a signal to the sender over the control bus to tell it to send – and likewise for the receiver. The sender will respond to its control signal by placing data on the data bus – the receiver will respond to its control signal by taking data from the data bus.

The procedure, therefore, for effecting a transfer between two components follows. Note that if memory is either sender or receiver then a slightly more complicated procedure has to be followed - the rationale and details for this will be explained after we describe *the four-step model for data transfers not involving memory*:

1) The CPU asserts (puts a signal/voltage on) a control line (on the control bus) to the sender to tell it to place data on the data bus;

2) The sender responds by putting data on the data bus;

3) The CPU asserts a control line (on the control bus) to the receiver to tell it to take data from the data bus;

4) The receiver responds by taking data from the data bus and storing it internally

In practice, for reasons of speed and efficiency, steps 1) and 3) will take place simultaneously, followed by steps 2) and 4) which will also take place simultaneously. To keep things simple, however, we will assume the four events happen in the sequence detailed above.

When we wish to make a transfer that does involve memory (i.e. memory is either the sender or receiver – by the way it cannot be both at one time) then there is an additional complication. We have seen that memory consists of a large number of different storage locations –which one do we use for the transfer? The CPU decides (again because the current instruction is telling it which one to use!). The CPU must first advise memory which location is to be used before attempting to involve memory in a transfer (either a read or write operation). This is known as addressing memory. So now we describe *the seven-step model for data transfers involving memory (as <u>either</u> a sender <u>or</u> receiver)* – the additional three steps at the beginning constitute *addressing memory*:

A) The CPU computes the address of some memory location that is to be either read from or written to, and places that address on the address bus;

B) The CPU asserts the control line M_a;

C) Memory responds to this control signal by taking the address in from the address bus, decoding it, and selecting the appropriate location (for a subsequent read or write operation)

1) The CPU asserts a control line to the sender to tell it to place data on the data bus (if memory is the sender then the CPU will assert the control line M_r) ;

2) The sender responds by putting data on the data bus (if memory is the sender, then the data will be copied from the addressed / selected location);

3) The CPU asserts a control line to the receiver to tell it to take data from the data bus (if memory is the receiver then the CPU will assert the control line M_w) ;

4) The receiver responds by taking data from the data bus (if memory is the receiver, then the data will be copied into the addressed / selected location)

- Once again, for speed and efficiency reasons, in practice steps 1) and 3) will take place simultaneously, followed by steps 2) and 4) which will also take place simultaneously. Once again, to keep things simple, we'll assume that events happen as detailed above.

- Note that as well as <u>data</u> transfers between memory and other components, program <u>instructions</u> from memory to the CPU will also follow this model.

- In step A) above, we say that the CPU <u>computes</u> an address before placing it on the address bus. Every computer system will have a number of quite complex methods of getting data and instructions from main memory – each method, which we'll introduce later, is called an addressing mode. Addressing modes regularly require some arithmetic calculation (a computation!) to produce an address in memory.

Examples of Transfers within a Computer System

Some simple applications of the four and seven step models follow (these are the same as the set of notes on *A Simple Computer Architecture*).

<u>Example 1 – To effect a transfer from the input device (i.e. from the outside world) to the ALU</u>

1) The CPU asserts control line W to the input device to tell it to put data on data bus;

2) The input device responds by putting data on the data bus;

3) CPU asserts control line A_w to the ALU to tell it to take data from the data bus;

4) The ALU responds by taking data from the data bus and storing it internally (in a register)

Example 2 – To effect a transfer from memory to the ALU

A) The CPU computes the address of some memory location that is to be either read from or written to, and places that address on the address bus;

B) The CPU asserts the control line M_a;

C) Memory responds to this control signal by taking the address in from the address bus, decoding it, and selecting the appropriate location (for a subsequent read or write operation)

1) The CPU asserts a control line M_r to memory to tell it to place data on the data bus ;

2) Memory responds by putting data on the data bus from the addressed location;

3) CPU asserts control line A_w to the ALU to tell it to take data from the data bus;

4) The ALU responds by taking data from the data bus and storing it internally (in a register)

Example 3 – To effect a transfer from the input device to memory

A) The CPU computes the address of some memory location that is to be either read from or written to, and places that address on the address bus;

B) The CPU asserts the control line M_a;

C) Memory responds to this control signal by taking the address in from the address bus, decoding it, and selecting the appropriate location (for a subsequent read or write operation)

1) The CPU asserts control line W to the input device to tell it to put data on data bus;

2) The input device responds by putting data on the data bus;

3) The CPU asserts the control line M_w to memory to tell it to take data from the data bus;

4) Memory responds by taking data from the data bus and copying into the addressed / selected location)

Example 4 – To effect a transfer from memory to the output device (to the outside world):

A) The CPU computes the address of some memory location that is to be either read from or written to, and places that address on the address bus;

B) The CPU asserts the control line M_a;

C) Memory responds to this control signal by taking the address in from the address bus, decoding it, and selecting the appropriate location (for a subsequent read or write operation)

1) The CPU asserts a control line M_r to memory to tell it to place data on the data bus;

2) Memory responds by putting data on the data bus from the addressed location;

3) The CPU asserts a control line to the output device to tell it to take data from the data bus;

4) The output device responds by taking data from the data bus and outputting it.

Consequences of Different Data and Address Bus Widths in Computer Systems

The number of connectors in the data bus influences the overall processing power of a computer system.

If the width of the data bus is, for example, 32 bits then it means that data can be transferred around the system in multiples of 32 bits at a time. We refer to such a computer as a 32-bit computer (by virtue of its data bus width). Given that all other things are equal (e.g. processor clock speed, etc.), a 32-bit computer is faster than a 16-bit machine, which in turn is faster than an 8-bit machine.

If a 32-bit computer wants to process a 32-bit item of data, then it can access it from memory in one memory read operation. To do the same task on a 16-bit computer system requires two memory accesses, first for the high order 16 bits of the 32 bit operand, followed afterwards by the low order 16 bits of the 32 bit operand. To effect the same operation on an 8-bit system means that four memory accesses have to be done in sequence to get the operand into the CPU for processing. Note, by the way, that if we are doing a lot of processing of 8-bit operands in some instance, then having a 32-bit system is not of particular benefit to us.

All of the registers interfacing to the data bus in all of the system's components need to have the same width as the data bus. Similarly, CPU registers need to be a minimum of the data bus width in size, or a multiple thereof. Furthermore the ALU would be expected to be able to process operands of the same width as its CPU's registers.

For example, a computer with an 8-bit data bus might have a CPU with 16-bit registers and an ALU capable of doing 8 and 16-bit operations. 8 bit operations would require one memory access; 16 bit operations would require two memory accesses.

The number of connectors in the address bus influences the maximum amount of memory that may be installed in a computer system (the amount installed is frequently less than the maximum amount possible).

If the computer has, for example, 20 address lines then the maximum amount of memory it can support is 1 megabyte. This is because with 20 connectors in the address bus the smallest address that can be carried is 00000000000000000000_2 while the largest address is 11111111111111111111_2. In terms of the address bus width, this is a range of values (addresses)

from 0 to 2^{20}-1. This is 1048576 different addresses (just over one million of them) - corresponding to a 1 megabyte address space.

If the address bus has 32 connectors, that is, the computer has 32-bit addresses, then the system has an address space of 4 gigabytes maximum (addresses running from 0 to 2^{32}-1). Of course, it doesn't necessarily need to have all of this memory installed (and usually doesn't). Currently, PCs are manufactured with 32-bit address buses but come equipped with between 256 megabytes and 1 gigabyte of memory as standard (i.e. from $1/16^{th}$ to 1/4 of maximum capacity). It is likely that the limit of a maximum of 4 gigabytes of address space with the current address bus width of 32 bits will very soon become problematic for PC systems (particularly as memory has become so cheap in recent years) - it is likely that address bus widths will have to increase in near future revisions of PC architecture.

The CPU (in more detail)

We have already seen that the CPU in a computer system has three fundamental parts:

- Control Unit;
- ALU (arithmetic and logic unit);
- Registers.

In this section we will study the CPU in a little more detail. We'll begin by recapping what we already know:

The control unit has two principal purposes:

- it ensures that the instructions of any program that is loaded into memory are executed in the correct sequence;
- it manages all of the other components in the computer system to achieve what is required by each instruction in the program.

The ALU is essentially a collection of circuits for doing arithmetic operations (addition, subtraction, multiplication, division, modulus, etc.) and logical operations (not, and, or, etc.).

Most instructions in a program require the CPU to manipulate data in some form or other - such data has to be temporarily stored in the CPU. If it is in memory, it must be read into the CPU prior to processing it - the CPU cannot process data directly in memory. For this purpose the CPU must have temporary storage areas which are called *registers*. There are only a few registers present, they allow data to be processed extremely quickly but they are very expensive to implement (compared to other forms of storage such as primary memory (RAM and ROM) or secondary memory (hard disks, optical disks, floppy disks, magnetic tapes, etc.). Registers can hold many different forms of information such as data, addresses, program instructions, etc., depending on the particular register in question and what it is being used for at any particular moment in time.

The CPU goes through a number of well defined and distinct *phases* in order to handle each instruction it must execute. The set of five phases are termed the *instruction execution cycle*:

- Fetch phase
    - Get the next instruction from main memory into the CPU. The CPU has a special register called the Program Counter (PC). It contains "the address in memory of the next instruction that should be executed from the program" (note that this assumes one instruction has just been completed and that we are ready to commence the next instruction). This involves addressing and reading memory (seven step model). Remember that what we are fetching from memory is just a binary number.

- Decode phase
    - Analyze the instruction just fetched from memory to find out what needs to be done. The initial part of the instruction is the "opcode" which specifies what operation is to be done. The remainder of the instruction specifies operands for that operation. For example, the binary equivalent of the number 14 as the opcode might indicate that an addition needs to be done. Activate an appropriate circuit in the ALU to perform the task. Instructions generally operate on operands, so we also need to ascertain how to get these operands (typically they will already be in some CPU registers, or they may need to be read from memory into the CPU).

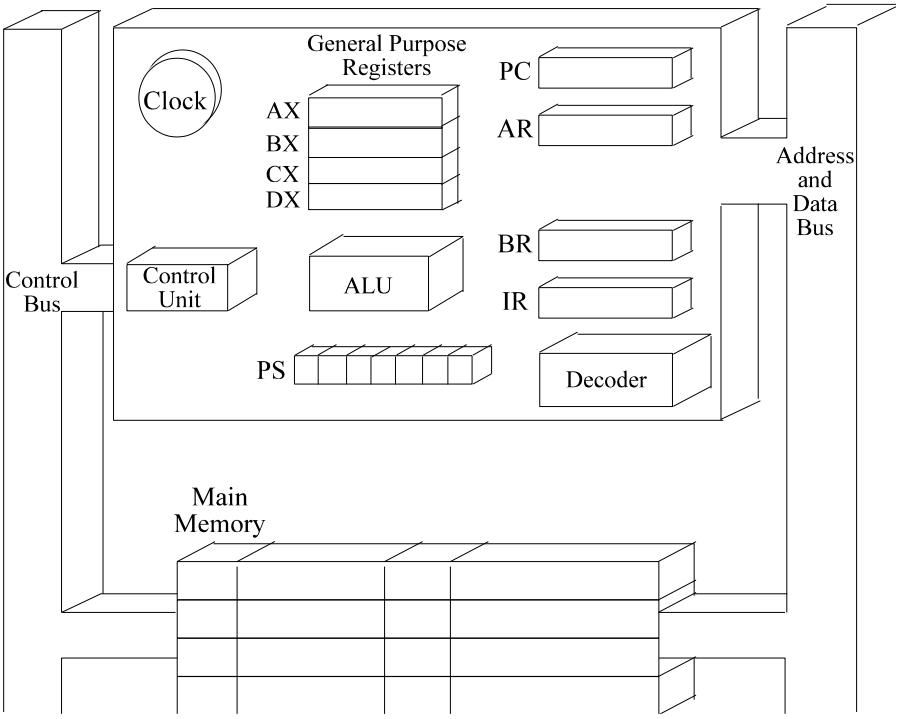
- Retrieve operands phase
    - If an operand is in memory, get it into the CPU (using the seven step model again).

- Execute - We've selected a circuit in the ALU to do the instruction, we've lined up the operands that the instruction will act upon in the CPU, so we're ready to go. We do three things in this phase:
    - execute – feed the operands into the selected ALU circuit – a result will emerge from the circuit's outputs
    - write results – these may need to be placed in a CPU register or may need to be written to memory (it depends on the precise instruction)
    - update the status register SR (this keeps a short term record of the outcome of the current instruction (e.g. did it produce a zero or non-zero result – was the result positive or negative, etc.) – this is available to the next instruction only which may take alternative courses of action depending on what it finds.

- Update PC
    - The program counter contains the address of the current instruction – update it so that it contains the address of the next instruction that should be executed, before repeating the cycle for that next instruction.

We pointed out before that the CPU has no intelligence or decision making capacity of its own - all decisions are actually built into the instructions in the program in memory - these instructions were written by a programmer, so are only a reflection of that programmer's intelligence.

The following is a more detailed model of the CPU (shown along with the connecting buses, and also showing main memory):



General-Purpose Registers

The general-purpose registers (AX, BX, CX, DX, etc. - not all are shown - there are a few more) are the CPU's local storage area. Operands or operand addresses are stored here temporarily, where they can be accessed quickly by other parts of the CPU. Since they provide a very limited amount of local storage, the general-purpose registers have to be used very selectively.

Processor Status Register (PS or SR)

This register stores information about the results of the last operation in a set of bits called condition codes. The condition code bits are set or cleared to indicate whether the result of the last operation was positive, negative, non-zero or zero, for example. Subsequent instructions can examine the status register to determine the outcome of the previous instruction and determine what they should do in this instance (this usually means either continuing on to the next instruction in the program or alternatively branching to some other part of the program). For example:

```
CMP AX,BX ; subtract BX from AX and update SR to show if the
          ; outcome was 0 or not, etc. CMP => Compare
JEQ LOOP  ; if they were the same branch to the instruction
          ; labeled LOOP, otherwise do the following instruction
INC CX    ; only done if AX and BX were not equal
     :
     :
     :
     :
LOOP: INC DX ; resume execution here if AX and BX were equal
```

Program Counter (PC)

The CPU must have some way of keeping its place in the sequence of instructions that it executes. The PC fulfils this role; it stores the address of the next instruction to be executed (assuming one instruction has completed and another one is about to begin).

Arithmetic Logic Unit (ALU)

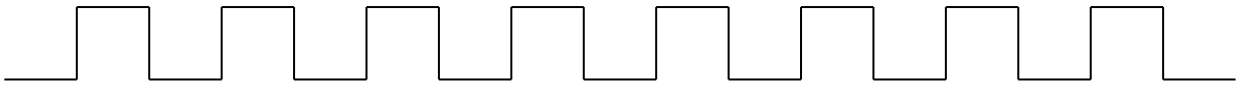
The ALU contains the circuits which perform arithmetic and logical operations. Arithmetic operations calculate numeric results. Typical arithmetic operations include addition, subtraction, division, and multiplication. Some common logical operations are AND, OR, and NOT.

Timing and Control Circuits

The timing and control circuits establish electrical pathways that run throughout the CPU. They are responsible for moving data into the right place at the right time. It should be noted that the CPU has its own internal bus system for moving stuff around within its confines, quite similar to the bus system outside the CPU, as described earlier.

Clock

The *clock* is responsible for the timing issues. Everything in the computer is controlled by the CPU (which is following the directions in a program) and everything it does is initiated using a constant stream of clock pulses that are distributed to all the electronic circuits within the computer system.



Each activity starts with a clock pulse and must be complete before the next clock pulse arrives. If the clock pulses are fed to the circuits faster than the computer's designer had intended (called *over-clocking* the CPU) then activities may commence before the previous ones have had time to complete, thus resulting in chaos in the system.

Control Unit

The *control unit* is responsible for sending control signals to all the components within the CPU as well as to the components outside the CPU which the CPU has to manage. The control unit is connected to all the internal CPU components via an internal CPU control bus and to all the devices connected to the processor via the *control bus*.

Address Register (AR)

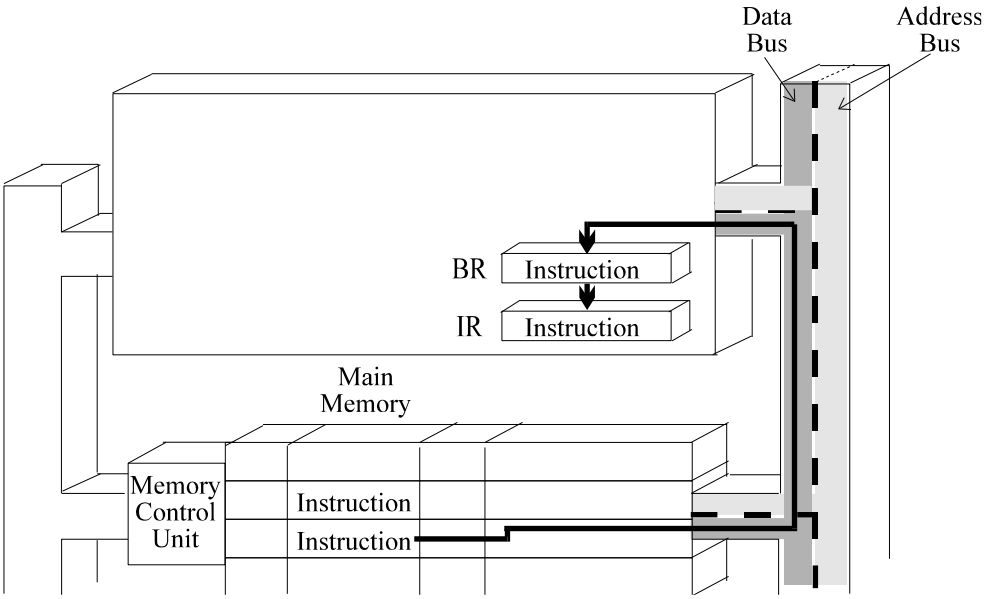
The address register is a register within the CPU that interfaces to the address bus. It temporarily holds the address of the memory location being referenced (read from or written to) by the CPU. The address register compensates for any differences in operating speed between the CPU and main memory. It holds the memory address until it can be accepted by main memory.

Buffer Register (BR)

The buffer register is a register within the CPU that interfaces to the data bus. It temporarily holds data that the CPU retrieves from, or sends to, main memory. Also, instructions are held here temporarily when they are fetched from main memory.

Instruction Register (IR)

The instruction register is used to hold each instruction while it is being executed by the CPU. As mentioned earlier, an instruction that is fetched from main memory temporarily waits in the buffer register BR (it is read into the CPU from main memory using our seven step model). Then, as it is an instruction, is transferred along the pathway established by the timing and control circuits to the instruction register IR. This frees up the buffer register BR to allow an operand to be read from memory for the current instruction. The contents of the instruction register IR are used as input to the instruction decoder.

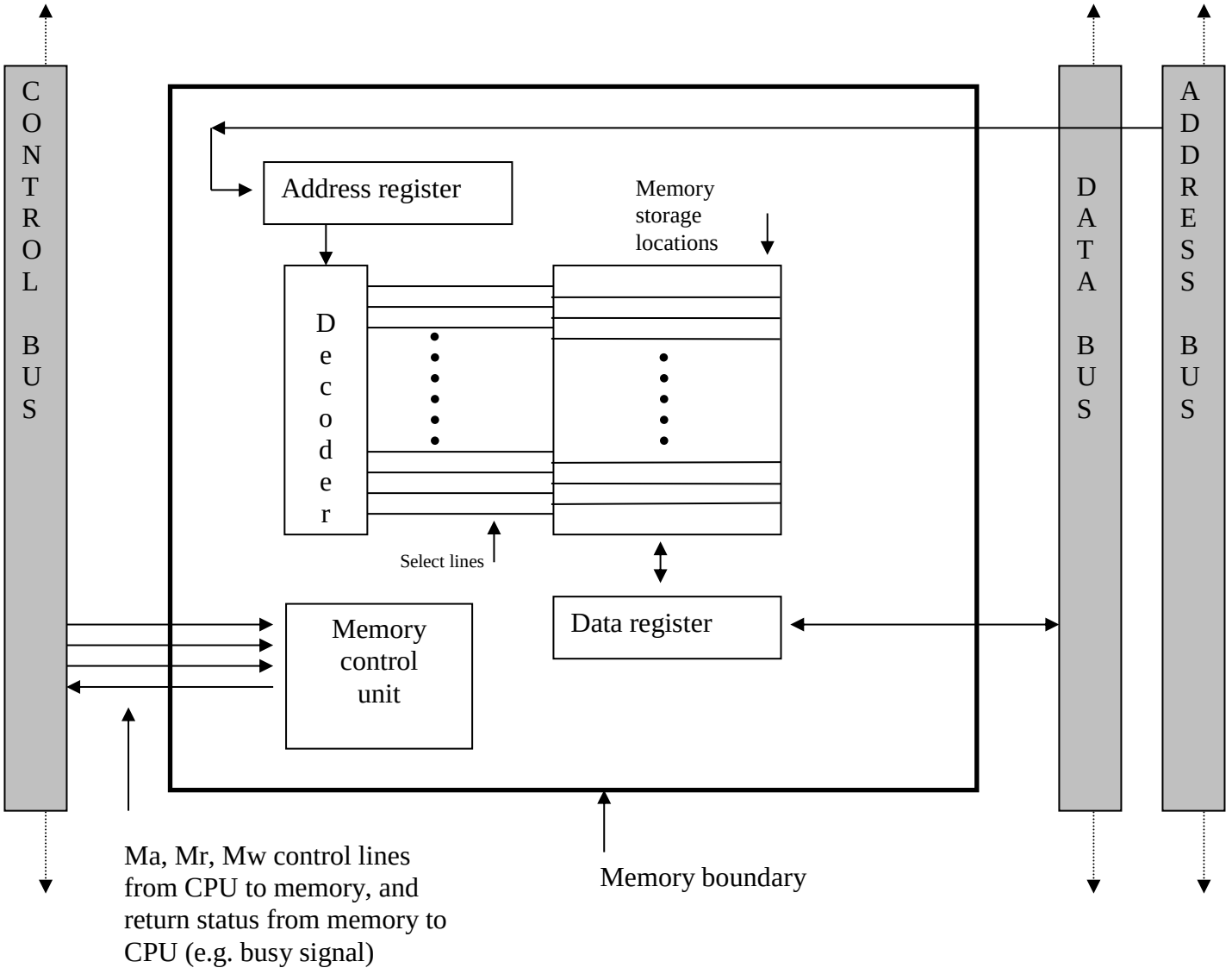


Movement of an Instruction into the Instruction Register

Instruction Decoder

Instructions are binary combinations subdivided into operation codes and operand fields. In order to execute any instruction, the binary op code must be decoded so that the desired operation may be identified. The instruction decoder performs that task. The instruction decoder translates the binary op code into a unique signal that prepares circuits in the ALU to execute that instruction. The instruction decoder also examines the operand field in the instruction and generates signals which select the register and addressing mode needed to locate the operand(s).

Basic Organisation and Operation of Memory



Random access memory is organised as shown within the heavy black boundary in the above diagram. The easiest way to describe the operation of this memory element is using our 7-step model described a little earlier.

Memory consists of a large number of different storage locations, each identified by a unique address. To effect a transfer to (write) or from (read) memory, we must first tell memory which particular location we wish to involve in the transfer. Thus we must *address* memory before reading from it or writing to it. This involves the three initial steps in our 7-step model.

A) The CPU computes the address of some memory location that is to be either read from or written to, and places that address on the address bus;

B) The CPU asserts the control line M_a;

C) Memory responds to this control signal by taking the address in from the address bus into the address register, decoding it, and selecting the appropriate location (for a subsequent read or write operation)

The address register is simply a temporary holding area for addresses coming from the address bus as they await decoding. The incoming address is a binary number. It is passed into the address decoder which interprets the binary number and activates a single select line corresponding to the decoded number – for example, if the binary number 1010_2 (= 10 in decimal) was passed in as an address then the decoder would put a signal on the tenth select line. This activates the storage cell at address 10 and a subsequent read or write operation with memory will involve this storage cell only – all the other storage cells in the memory are dormant as they are not currently selected.

Note that RAM employs *random* or *direct access* – it takes precisely the same effort to access information at one address as it does for any other address (compare this to a *serial access* medium such as magnetic tape which has to work harder to access information that is further away)

Now that a location has been selected in memory, we either read from it or write to it. In the former case memory is the *sender* in the transfer, in the latter case memory is the *receiver* (note that it cannot be both at one time). We assume that some other component in the computer system, known to the CPU, is at the other end of the transfer. The final four steps of the 7-step model are now used:

1) The CPU asserts a control line to the sender to tell it to place data on the data bus (if memory is the sender then the CPU will assert the control line M_r) ;

2) The sender responds by putting data on the data bus (if memory is the sender, then the data will be copied from the addressed / selected location into its data register, and from there out onto the data bus);

3) The CPU asserts a control line to the receiver to tell it to take data from the data bus (if memory is the receiver then the CPU will assert the control line M_w) ;

4) The receiver responds by taking data from the data bus (if memory is the receiver, then the data will be copied into its data register, and from there into the addressed / selected location)