ECS404: Computer Systems and Networks

Prof E.P. Robinson and Dr A. Alomainy

April 5, 2018

Contents

Aims

The aim of this course is to provide students with a basic understanding of how a computer works, how programs are executed by the CPU at the machine level, and how computer networks function. They will gain this firstly by studying the major components of a computer, the interaction between them, and how computers communicate over networks. Secondly, they will learn how data is represented to be processed by computer. Thirdly, students will learn some assembly language and understand how high-level programming concepts are related to their machine language implementation. Finally, they will learn how computer networks function, and will understand how low-level network traffic implements communication between computers.

Summary

The course presents the concepts needed to understand typical computers at the level of their 'machine-code' instruction set, and to understand the basic concepts of computer networks.

The material covered includes

1. the major components of a computer, including CPU, memory, I/O and buses and the role of bandwidth, latency and power dissipation in determining the relationship between them.

2. the use of bits, bytes and data formats to represent numbers, text and programs

3. CPU structure and function: the conventional (von Neumann) computer architecture

4. data types, addressing modes and instruction sets

5. machine-level program structure and its correspondence to higher-level programs

6. the role of wired and wireless networks in modern computer systems

7. a basic understanding of typical network technologies, e.g. ethernet, wifi

8. the role of protocols such as ethernet in the implementation and use of network technology

Part I

Basic Computer Architecture

Chapter 0

Introduction

0.1 Aims

These are the overall aims of the course, with the aims dealt with by this part picked out in bold.

The aim of this course is to provide students with **a basic understanding of how a computer works,** how programs are executed by the CPU at the machine level, and how computer networks function. **They will gain this firstly by studying the major components of a computer, the interaction between them,** and how computers communicate over networks. Secondly, they will learn how data is represented to be processed by computer. Thirdly, students will learn some assembly language and understand how high- level programming concepts are related to their machine language implementation. Finally, they will learn how computer networks function, and will understand how low-level network traffic implements communication between computers.

In more detail: The aim of this part of the course is to familiarise you with the way a computer is built. There are three over-arching themes:

- **taxonomy:** an analysis of the different kinds of devices, so that you know the range of devices that your studies will cover

- **construction:** concentrating on how the devices are built, so that you understand the basic components of a typical computer, something of how the components are made, and a reasonable amount about how they are put together to make a working machine

- **history:** to give you a sense of how rapidly computers have developed and to give you a feel for the way they are likely to develop in the future (this is the least emphasised of the themes).

In addition there is a section on power, covering the energy use of computers and IT generally, with the aim of helping you to understand why the energy use of IT is technically, commercially, and ultimately socially important.

0.2 Summary

This is the overall summary of the course, with the items dealt with by this part picked out in bold.

The course presents the concepts needed to understand typical computers at the level of their 'machine-code' instruction set, and to understand the basic concepts of computer networks.

The material covered includes

1. **the major components of a computer, including CPU, memory, I/O and buses and the role of bandwidth, latency and power dissipation in determining the relationship between them.**

2. the use of bits, bytes and data formats to represent numbers, text and programs

3. **CPU structure and function: the conventional (von Neumann) computer architecture**

4. data types, addressing modes and instruction sets

5. machine-level program structure and its correspondence to higher-level programs

6. the role of wired and wireless networks in modern computer systems

7. a basic understanding of typical network technologies, e.g. ethernet, wifi

8. the role of protocols such as ethernet in the implementation and use of network technology

In more detail:

- **Different types of computer:** (taxonomy, link to history) an analysis of the different kinds of devices,

- **Basic device-level architecture:** looking at how the different kinds of devices are constucted

- **Integrated Circuits:** aka computer chips. An introduction to how they work leading up to the construction of adders and simple memories.

- **Memory:** the various forms of computer memory and how they are used.

- **Internal comms:** how the different parts of a computer talk to each other, the use of buses.

- **IO:** a short section on how IO is handled.

- **Power:** an increasingly important topic for small devices and for large server farms, we provide an introduction.

- **History:** this centres on Moore's Law and the way is has enabled an astonishing development in computer power that not only makes computers faster, but makes different forms of computation feasible to carry out on available devices.

Chapter 1

Different types of computer

Aims

The aim of this part of the course is to introduce the different kinds of computers you might encounter and at some point need to program.

Learning Objectives

By the end of this chapter you should:

☐ understand what different sorts of computers there are

☐ be able to define the concepts of **microprocessor** and **microcontroller** and understand what they are

☐ be able to identify whether an artefact contains a microprocessor or microcontroller with reasonable accuracy

☐ be able to identify when an artefact is not likely to contain either.

Check these off when you are confident you have achieved them.

1.1 What makes something a computer?

There are lots of computers about. Some of them are obvious and others more hidden, either as server farms hidden away in big warehouses, or as small special-purpose devices hidden away inside things you don't think of as computers. Still others are devices that are structured and built as computers but which you might think of as something else. At the top end there are questions about whether a warehouse full of machines is a collection of many different computers or a single massively parallel computational device. At the bottom end, you can ask whether a basic microcontroller is really a computer or just a control device.

The view we take is this: complex concepts have fuzzy definitions. There are things on the boundaries where you can argue either way. Having a precise definition will not be important to us. What will be be important is understanding the rough location of the boundary and what might put a device on one side or the other.

1.1.1 Some basic examples

Here is a list of devices to get things going. We'll expand it later.
Computers obvious to everybody:

- Mainframe servers
- Desktop PC's (personal computers)
- Laptops

Computers marketed as general-purpose devices:

- Smart phones
- Tablets

Computers hiding in plain sight disguised as special-purpose devices:

- Games consoles
- Routers

Anything that contains a microprocessor

- Consumer devices (mircrowaves, ovens, burglar alarms,...)
- Cars

1.1.2 Making things a bit more precise

We're going to focus on **programmmable digital computers.** Unpack this.

Definition [Computer]: A device or machine for performing or facilitating calculation; An electronic device (or system of devices) which is used to store, manipulate, and communicate information, perform complex calculations, or control or regulate other devices or machines, and is capable of receiving information (data) and of processing it in accordance with variable procedural instructions (programs or software) (Oxford English Dictionary)

This definition is in partly in terms of **function**, i.e. what it does:
performing or facilitating calculation and **store, manipulate, and communicate information, perform complex calculations, or control or regulate other devices or machines, and is capable of receiving information (data) and of processing it in accordance with variable procedural instructions (programs or software)**

It is also partly in terms of **implementation**, i.e. how it is constructed: **An electronic device (or system of devices)**.

The distinction between **function** and **implementation** is at the heart of component-based design, hardware and software.

According to the OED, one of the things a computer does is run **programs** (see later).

Definition [Digital]:

1. (of signals or data) expressed as series of the digits 0 and 1, typically represented by values of a physical quantity such as voltage or magnetic polarization. Often contrasted with analogue. (OED)

2. Property of representing values as discrete, usually binary, numbers rather than a continuous spectrum. (Wiktionary)

Digital refers to the representation system used by the computer. All computers have to represent things in the real world in some way. Digital computers use a representation system based on some fixed finite alphabet. We use a digital representation system when we write down numbers: the alphabet is the decimal digits 0..9 plus a few additional signs (-, ., e).

Digital contrasts with **analogue**.

Definition [Analogue]: of a device or system) in which the value of a data item (such as time) is represented by a continuous(ly) variable physical quantity that can be measured (such as the shadow of a sundial) (Wiktionary)

Definition [Analogue Computer]: A computer that performs computations (such as summation, multiplication, integration, and other operations) by manipulating continuous physical variables that are analogues of the quantities being subjected to computation. The most commonly used physical variables are voltage and time. Some analogue computers use mechanical components: the physical variables become, for example, angular rotations and linear displacements. (OED CS)

Example: Before electronic calculators and digital computers, engineers used simple analogue calculators called slide rules.

Definition [Program]: A set of statements that (after translation from programming-language form into executable form ...) can be executed by a computer in order to produce a desired behaviour from the computer. (OED Computer Science)

The point here is that the hardware is multi-purpose. What it does can be changed by giving it different instructions, a different program.

Some other definitions are useful.

Definition [Electronic]: (of a device) having or operating with components such as microchips and transistors that control and direct electric currents. (OED)

Definition [Microprocessor]: A semiconductor chip, or chip set, that implements the central processor of a computer. Microprocessors consist of, at a minimum, an ALU and a control unit.

Definition [Microcontroller]: A single-chip computer designed specifically for use in device control, communication control, or process-control applications. It typically comprises a microprocessor, memory, and input/output ports. A typical microcontroller might have a relatively short word length, a rich set of bit-manipulation instructions, and lack certain arithmetic and string operations found on general-purpose microprocessors. (OED Computer Science)

1.2 Self-test

Check these questions off when you are confident you have answered them correctly. Be warned, sometimes the questions can be argued either way. A good answer might do that:

☐ 1. Which of these devices use **analogue** representations, and which use **digital**?

- ☐ *a*) an ordinary tape measure
- ☐ *b*) a car speedometer
- ☐ *c*) a car odometer (measures the mileage)
- ☐ *d*) a clock

☐ 2. Which of these devices are **electronic**, and which are simply **electrical**?

- ☐ *a*) a vacuum cleaner
- ☐ *b*) a radio
- ☐ *c*) a cement mixer
- ☐ *d*) a pair of standard headphones

☐ 3. Identify at least two devices from your home that you are confident contain **microcontrollers**, and explain why you believe they do. (See later if you are unsure).

1.3 Different types of computer

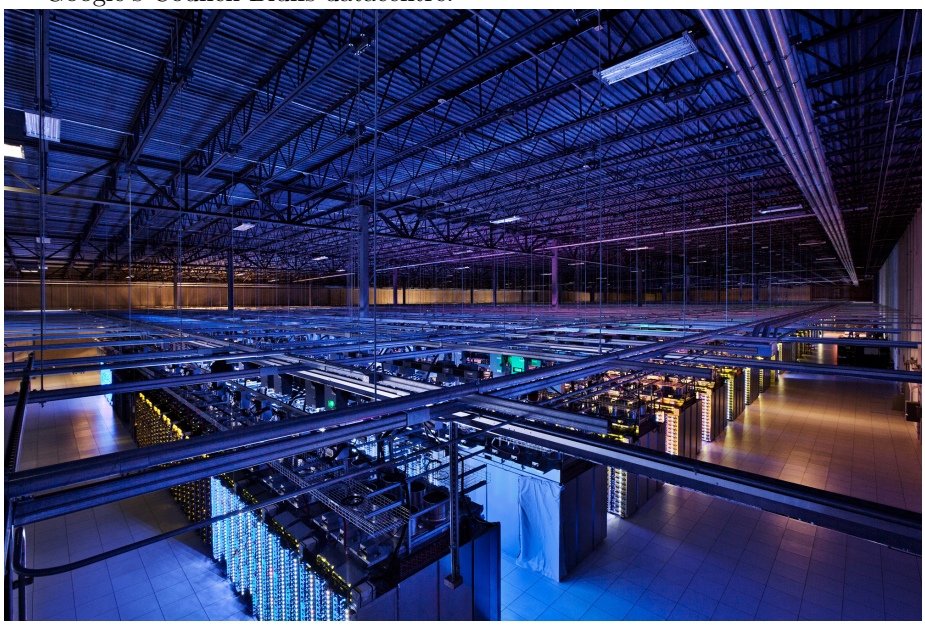
In this section we give a rough taxonomy (or classification) of the different kinds of computing devices around. It is arranged in approximate order of computing power and (generally) price.

1.3.1 Servers and server farms

Companies like Google, Amazon and Facebook (as examples, there are also many many others) use massive amounts of computing power. That computing power takes the form of server farms. A typical server farm consists of a room containing racks of high-end computers all networked together. The size of room can range from being about the size of a normal room in a house to a large shed covering the area of a football pitch. These facilities use a lot of power, and cooling is a major issue. The annual cost of power is a significant fraction of the build cost, and the cost of providing cooling can be about as much as that of the computers themselves.

For more information about how these datacentres are built, and matters such as energy costs, see: `https://www.google.com/about/datacenters/` and for some great pictures: `https://www.google.com/about/datacenters/gallery/#/`

Google's Council Bluffs datacentre:



A standard rack is 19in (480mm) wide, and units are in multiples of 1.75in (44.5mm) high (a bit less to allow removal). Rack-mounted servers are often flat boxes. Sometimes they are vertical boxes intended to be mounted in a chassis that allows them to share things like power supplies and network connections. This is an HP Proliant BL660c (reference `https://www.hpe.com/h20195/v2/GetDocument.aspx?docname=c04543743`). This particular server can have up to four cpu chips, each with up to about 20 cores, up to 2TB of RAM and four hard disks.

1.3.2 Personal computers and laptops

These are the computers we see at home and in offices. There are quite a few different sorts:

- old-style tower PC's with a separate box for cpu, RAM memory and disk, built out of standardised components
- higher-end laptops that follow standard PC architectures
- all-in-one PC's, basically a laptop (minus battery, keyboard and touchpad) packed into a screen
- low-end laptops, basically a tablet with keyboard and touchpad.

1.3.3 Mobile phones and tablets

The apps on smartphones and tablets are programs. An Android or iOS smartphone or tablet is basically a small computer running a form of the Unix operating system, and an app is a program running on it (technically both Android and IOS are based on Unix kernels, Android on a linux kernel and IOS based on a proprietary Apple kernel). There is a bit more to it than that: apps have to conform to a particular design structure, which is quite complex. Nevertheless, you can write your own apps to run on phones, just as you can write your own programs. Android apps are (mainly) written in Java. IOS apps are written in C++ or Apple's new language, Swift. Both Android and IOS have development environments that will help you generate vanilla apps that you can then extend.

Introductions to app writing for Android are at: `https://developer.android.com/training/basics/firstapp/index.html` and for IOS at: `https://developer.apple.com/library/content/referencelibrary/GettingStarted/DevelopiOSAppsSwift/` We will look at the actual smartphone architectures later.

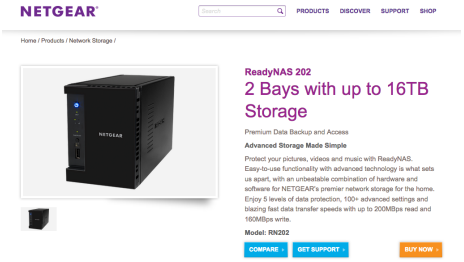
1.3.4 Personal computers disguised as consumer devices

Your house may also contain a number of devices that are basically fully functional computers boxed as something with a special purpose. Many of these run a variety of different apps and/or have software (firmware) that can be updated, and they connect to the internet.

- **(Wireless) router:** a router is a computer whose primary purpose is to direct network traffic. Routers on the internet backbone are powerful computers in their own right. But the router in your own home is much more basic. A common way to construct domestic routers is to build a small basic computer and run a version of linux on it. The advantages of this are that it makes it much easier to re-use existing publicly available code, increasing reliability and security and decreasing cost. Moreover it decouples the control software from the hardware, making it easier to change either independently.

- **Games console:** a games console is basically a desktop PC running slightly different software. Modern games consoles run different apps and connect to the internet for online gaming. Internally they are like PC's but with some different trade-offs made in design, and using the power of having massive production runs to drive construction costs down.

- **Network storage:** lots of homes have network addressed storage that you can use for backups of laptops and PC's as well as centralised storage of music, photos and video. These also run different apps and have firmware that can be upgraded. They are basically a small computer with disks attached.



1.3.5 Compute-heavy consumer devices

We also have many devices that carry out significant amounts of computation, and which themselves may be internet-connected. They may contain microprocessors or high-end microcontroller devices (the difference is disappearing as more computation and control is being placed on individual chips).

1. **Television:** Digital television signals are basically mp4 encodings. Televisions take those encodings and display them on the screen. That takes significant computing power. Many of televisions are now "smart", which

means internet enabled, and they run apps for things like iPlayer and YouTube.

2. **Camera:** Most cameras, whether still or video, do a lot of processing (face recognition, picture encoding and decoding, camera control, ...). They can also be internet enabled.

3. **Printer:** Printers are also networked, and may well do a lot of processing (the simplest ink-jets, though, get the computer you are printing from to do all the work).

4. **Cars:** these use extensive computer-based control covering things like engine-control and monitoring, satnavs,....

In October 2016 there were a number of large distributed denial of service attacks using the massed computing power of internet-enabled devices. The most serious knocked out a number of companies including Twitter. See the BBC report at `http://www.bbc.co.uk/news/technology-37738823` for more detail.

1.3.6 Low-end microcontroller devices

There are lots of low-end devices coming on the market. These range from smart devices for the home, often network enabled, though regular household appliances to children's toys. It is easy to find a microcontroller for just a few pence (search `mouser.co.uk` for microcontrollers and sort by price to find one starting at 22p - if you buy 100). As a result, in most cases if something has a digital control interface, it is likely to incorporate a microcontroller. Examples include: microwave ovens, thermostats, burglar alarms, ovens, digital radios,

1.3.7 Computer checklist

The more of these you can tick off the more likely something is to have a "computer" inside.

- ☐ Runs apps or programs, particularly if you can install these yourself. (Computers, laptops, phones, tablets)
- ☐ Has software or firmware that can be updated. (Routers, network storage).
- ☐ Carries out tasks you know to be computationally complex (Televisions, cameras)
- ☐ Carries out complex configurable control task. (Household items, toys)

Chapter 2

Basic device-level architecture

Aims

The aim of this part of the course is to look inside the box for the types of computers in the last chapter, show what they have in common and how they differ. It is also to introduce the von Neumann architecture, which serves as a basic reference architecture, and to see how computers still conform to it, and, of course, some of the ways they have moved on.

Learning Objectives

By the end of this chapter you should:

☐ understand that computers contain a cpu, short and long-term memory, io devices, and internal comms,

☐ be able to explain the basic roles of the components above

☐ have seen how these components are built into modern computers, including servers, laptops, tablets and mobile phones

☐ be able to describe the standard von Neumann architecture

☐ be able to explain ways in which modern architectures conform to von Neumann, perhaps with some elaboration, or differ from it.

2.1 Basic Components

In this section we look at the basic components and architecture of these types of computer.

We will see that all computers have five basic sorts of components:

- one or more **central processing units** (**cpu's**) to control the machine and actually carry out computation
- memory to store data and programs, almost always split into
  - short-term memory to store working data and programs which is cleared when the machine is turned off (e.g. RAM).
  - long-term memory that stores data and programs permanently (e.g. disk).
- devices that enable the computer to interact with its users and with the outside world (**Input-Output** or **IO**) (e.g. screen, wifi controller).
- a communication mechanism that enables these devices to talk to each other (often provided by the system board).

These components have been present since very earliest modern computers, when the basic reference architecture was set out by John von Neumann in a paper he wrote for the US government: "First Draft of a Report on the EDVAC" (1945).

2.2 Background: a name you should know - John von Neumann

John von Neumann was a Hungarian mathematician working in the US before and after the second World War. Amongst other things he made fundamental contributions to:

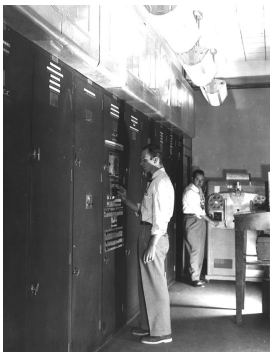
economics: he invented game theory, a basic tool of modern economics in which a system is described as a kind of board game between different players

computer programming: where he did a great deal of work in numeric programming, and in particular invented Monte Carlo methods, one of the basic pillars in machine learning.

During the Second World War he was employed by the US government as a kind of mathematical hitman, and at one point was asked to explain the new electronic computers that were being produced. His report was the result (there is no second draft, and he didn't actually formally publish it).

2.3 The von Neumann Architecture

A diagram of the von Neumann architecture is given in the figure. It includes:



First Draft of a Report
on the EDVAC

by

John von Neumann

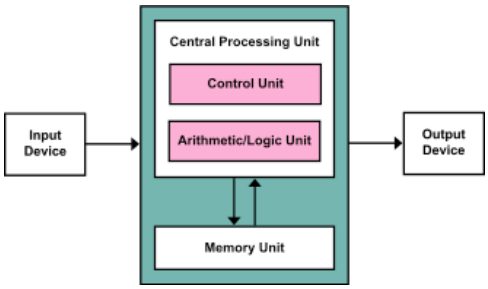
Contract No. W–670–ORD–4926

Between the

United States Army Ordnance Department

and the

University of Pennsylvania



2.0 MAIN SUBDIVISIONS OF THE SYSTEM 1

Figure 2.1: von Neumann, EDVAC and the von Neumann architecture

- a single cpu divided into a control unit and an arithmetic/logic unit (the division is not regarded as an essential feature)
- memory (not divided into short and long-term as disks and other long-term storage had not been invented at the time)
- input and output
- a basic comms structure that includes a single two-way channel between memory and cpu (an essential feature).

2.4 PC architecture

In a standard PC the basic architecture can be seen from the **system board** (or **motherboard**. This board carries the critical components of the PC and contains the basic comms structure linking them.

Figure 2.2 contains a diagram of a motherboard made by the company Gigabyte. This was used in PC's in the ITL in about 2006, and shows a fairly typical structure for tower-based PC's. Compare this to the von Neumann architecture:

1. the cpu (this board uses an AMD processor, not an Intel one).
2. the RAM memory, directly connected to the cpu as in von Neumann.
3. the hard disk(s) would be Serial ATA devices, as would be any optical disks (CD's and DVD's).
4. there are various classes of IO devices including USB and Firewire (IEEE 1394)
5. the communication channels all go via a central comms chip except for the connection to the RAM. There is a single connection from the comms chip to the cpu (the "Hyper Transport Bus").

There are obvious links with the von Neumann architecture (single cpu, single channels from memory to cpu, classification of subdevices into cpu, memory and IO, ...). There are also differences or at least refinements: more than one kind of memory, disks become generic SATA devices, IO devices become generic USB devices, etc.

Here is a physical picture of a similar computer. Note that there are lots of separate components connected by wires. These can be changed by users to either enhance the computer or fix faults. Note also the large fan in the centre that you can just see is covering an equally large piece of finned aluminium. This is the heat sink and fan over the cpu. The cpu typically might use over 100W. This is to get rid of that heat, which would otherwise fry it.

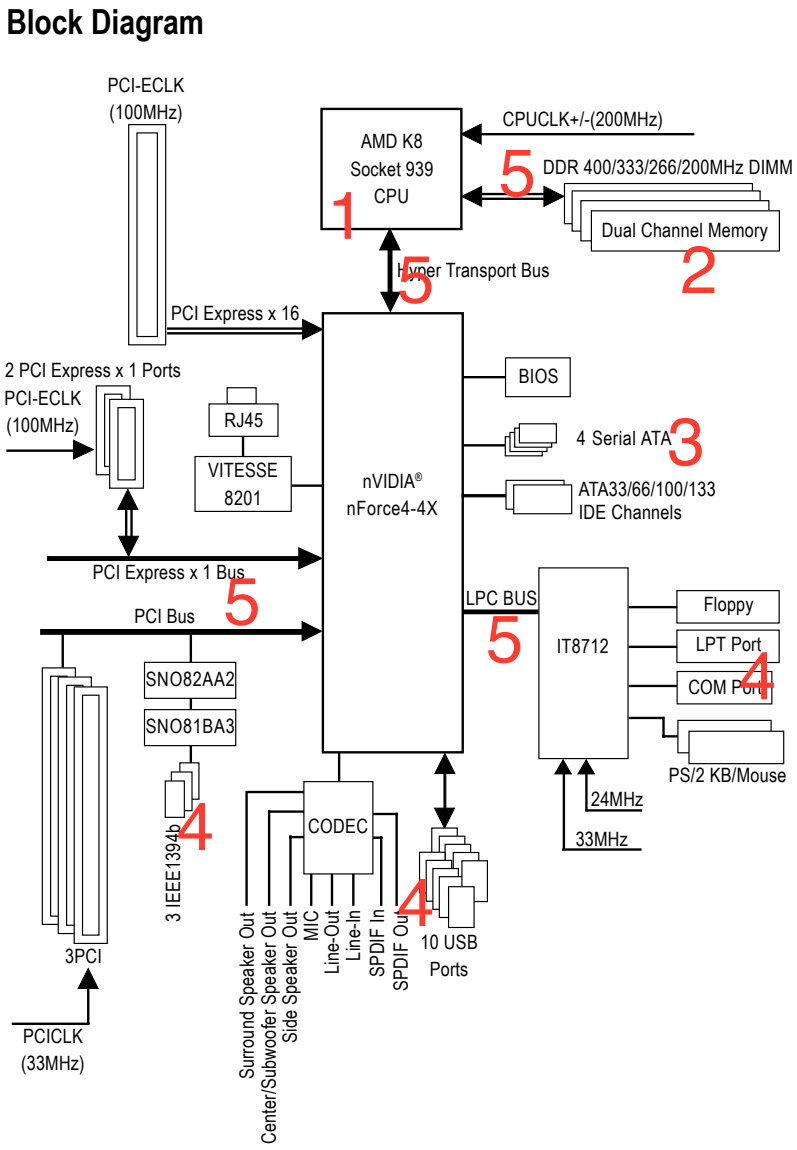
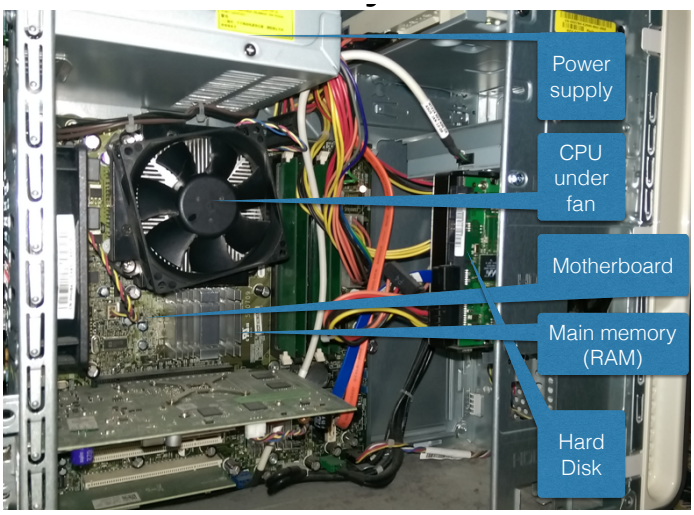


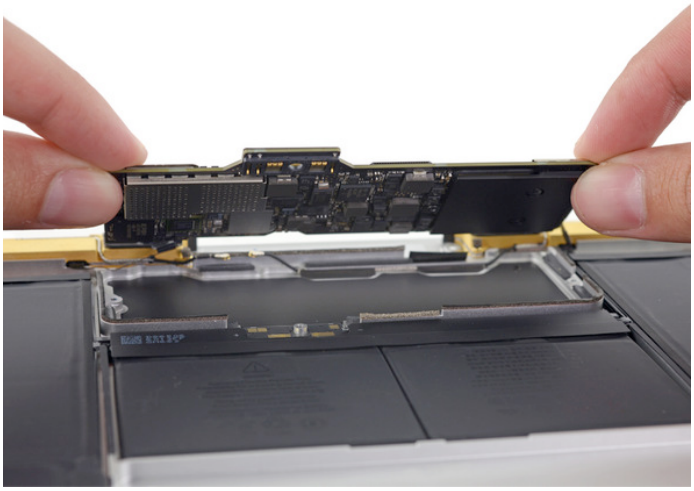
Figure 2.2: A Gigabyte PC motherboard

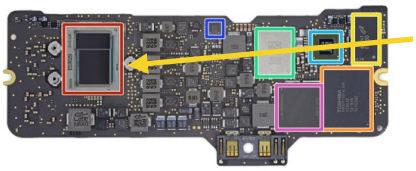


A laptop tends to have fewer user-replaceable devices, with many of its components being soldered onto the motherboard. However the logical architecture is similar to the conventional PC's. Modern all- in-one PC's are much like laptops, as can be seen in this photograph:

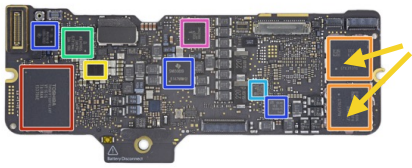
2.5 Laptop architecture

Modern laptops have gone even further. Here is a series of pictures of a Macbook Air from 2015. The first shows the motherboard in the context of the computer itself. The remaining pictures show what is on the motherboard. This is now a consumer commodity device, mass-produced with little or nothing that can be done in the way of customisation or replacement of parts.

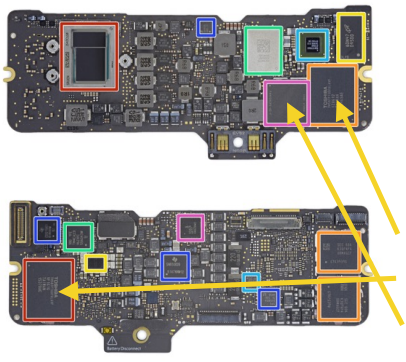




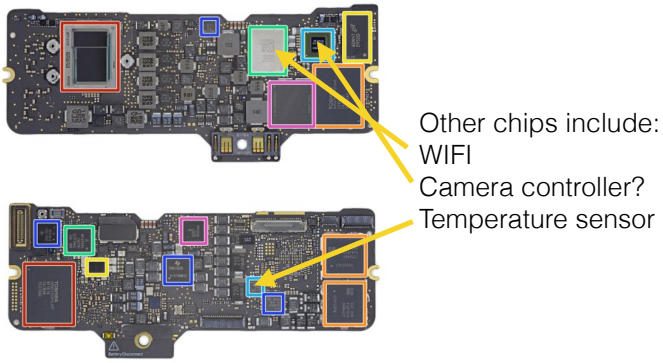
CPU: Intel SR2EN
Intel Core m3-6Y30
Processor



Main Memory: 2 x
Samsung K3QF4F4
4 GB LPDDR3 RAM
(total 8 GB)
just behind the cpu



Long-term memory:
solid state disk
consisting
of 2 x Toshiba
TH58TFT0DFKLAVF
128 GB MLC NAND
Flash mounted
front
and back
with
controller (beneath).

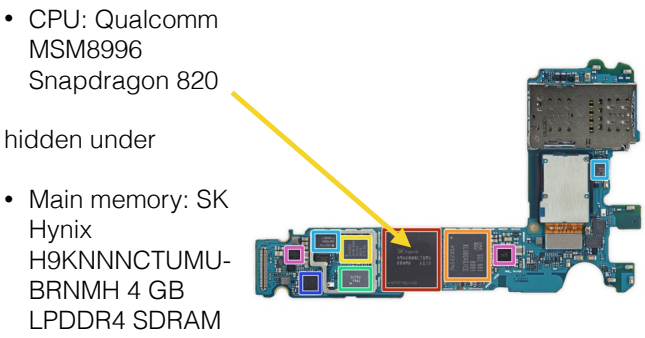


This shows that the Macbook has the components we expect, even if we cannot see the communication channels between them.

If the insides of a Macbook Air look vaguely as if they might be inside an iPad or a mobile phone, this is not a coincidence. The designs have been converging and the fact that high-end laptops like the Macbooks have a very long battery life is down to the use of low-power components like those used in mobile phones, along with the replacement of power-hungry screens and mechanical disks with led screens and solid-state disks. Note that the Macbook has no fan. It can get away with this because the Intel cpu used draws only 5W.

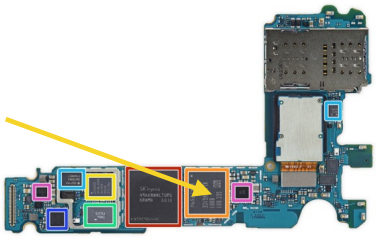
2.6 Mobile phones and tablets

Shown here is the motherboard of a Samsung Galaxy S7. It looks very like the motherboard for the Macbook Air. Chips are soldered on to both sides. The components are broadly similar. The influence of the von Neumann architecture is still apparent.

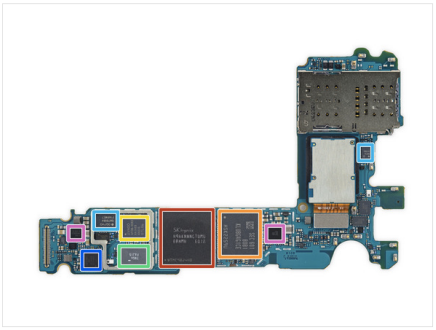


Smartphone construction (Samsung Galaxy S7)

- Long-term memory: Samsung KLUBG4G1CE 32 GB MLC Universal Flash Storage 2.0

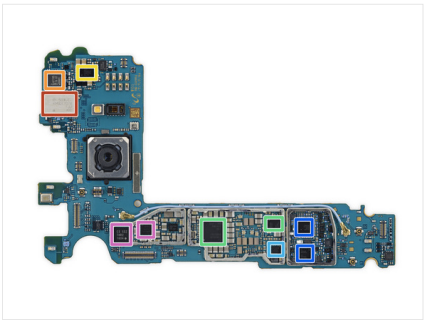


Details from teardown on iFixit



Step 7 Edit 5

- With that, it's time to digitally convey some chip ID. On the front side of the motherboard, we note:
  - SK Hynix H9KNNNCTUMU-BRNMH 4 GB LPDDR4 SDRAM layered over the Qualcomm MSM8996 Snapdragon 820
  - Samsung KLUBG4G1CE 32 GB MLC Universal Flash Storage 2.0
  - Avago AFEM-9040 Multiband Multimode Module
  - Murata FAJ15 Front End Module
  - Qorvo QM78064 high band RF Fusion Module and QM63001A diversity receive module
  - Qualcomm WCD9335 Audio Codec
  - Maxim MAX77854 PMIC and MAX98506BEWV audio amplifier



Step 8 Edit

- With so many similarities to the standard S7's chipset, it almost feels like we're repeating the computer:
  - Murata KM5D17074 Wi-Fi module
  - NXP 67T05 NFC Controller
  - IDT P9221 Wireless Power Reciever (likely an iteration of IDT P9220)
  - Qualcomm PM8996 and PM8004 PMICs
  - Qualcomm QFE3100 Envelope Tracker
  - Qualcomm WTR4905 and WTR3925 RF Transceivers
  - Samsung C3 image processor and Samsung S2MPB02 PMIC

2.7 Rack-mounted computers and servers

Rack-mounted computers are descended from desktop PC's in a somewhat different way. The basic design imperative is to put as much computing power into a box as you can manage for a given price. The way to do that is to put some standard architecture cpu's and as much standard memory as you can get in, with them sharing components such as power supplies, and communication infrastructure.

To illustrate this see the HP Proliant BL680c, below. This has up to four cpu's, and 2.0TB of RAM (that is about 500 times the amount on a basic laptop, or 1000 times the amount on a standard mobile phone).

The first picture gives a block diagram of the basic components and the way in which they are connected.

Differences from von Neumann architecture:

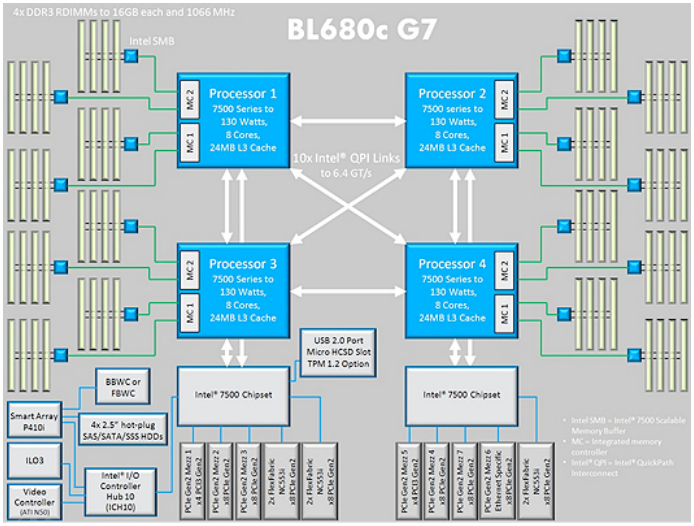
- four cpu's all interconnected (instead of one single one)
- memory split into four bits each connected to single cpu (instead of single memory connected to central processing component)

Similarities with von Neumann architecture:

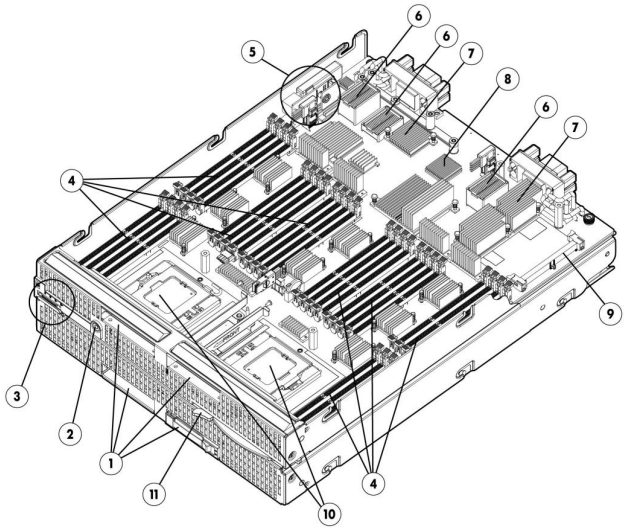
- same general classification of components
- each individual cpu plus its allocated memory is a von Neumann machine

The second picture gives a diagram showing the physical layout. Notice

- there is no power supply because this is intended to go into a chassis that supplies power
- ditto networking
- ditto fan for cooling
- note that the components are aligned front to back. This is so that cooling air can pass over them more easily (see chapter 7 on power). This is not shown in the block diagram.
- notice the rails on the side so that the whole server can be removed from its chassis.



HP ProLiant BL680c Gen7 Server Blade



HP ProLiant BL680c Gen7 Server Blade
Front view image showing side "A"

1. Four hot-plug SAS/SATA/SSD drive bays
2. Power on/standby button
3. UID, health, and network adapter LEDs
4. 64 RDIMM slots supporting up to 2.0TB of DDR3-1333MHz memory (operating up to 1066 MHz) (32 RDIMMs per side)
5. One USB 2.0 port, one MicroSD high capacity (SDHC) port, and one TPM 1.2 connector
6. Seven PCIe Gen2 I/O expansion mezzanine slots (3 on one side, 4 on the other)
7. Six NC553i 10Gb FlexFabric adapter ports (4 one on side, 2 on the other)
8. iLO 3 Management adapter port
9. HP P410i Smart Array flash cache connector
10. Two, three or four Intel® Xeon® 4800 family processors (below the hard drives) (two per side)
11. Server release lever

2.8 Data Centres

This extends the picture beyond what would normally be regarded as an individual machine. But an entire data centre can be regarded as a computational device (see for example the case made by two Google engineers in Barroso, Luiz Andr, Jimmy Clidaras, and Urs Hlzle. "The datacenter as a computer: An introduction to the design of warehouse-scale machines." Synthesis lectures on computer architecture 8.3 (2013): 1-154.

Modern cpu's contain a number of cores, microprocessors in their own right. We have seen that rack-mounted computers can contain several communicating cpu's. In a data centre the computers in a rack can often be linked, and a number of racks are often linked. So the same structure of communicating computational devices is replicated at a number of levels. This kind of structure is called **fractal**.

Google gives a good introduction to their data centres in `https://www.google.com/about/datacenters/`, including detail of their construction.

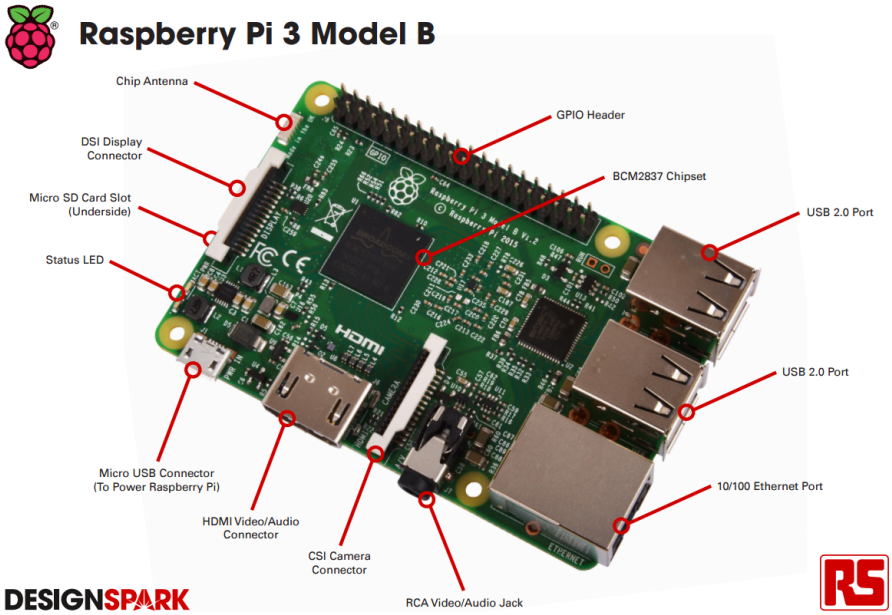
There's an introduction to Microsoft's data centres at `https://www.microsoft.com/en-gb/cloud-platform/global-datacenters`. There's also an introduction to them, including novel forms of construction in an article on ZDet: `http://www.zdnet.com/article/photos-a-tour-inside-one-of-microsofts-cloud-data-centers/`.

2.9 The Raspberry Pi

The Raspberry Pi illustrates the modern architecture for a small machine. It uses a **System on a Chip (SOC)** architecture, where the main chip carries most of the system as well as just the cpu.

Notice that there is a single chip (a Broadcom BCM2835 in the first version, and a Broadcom BCM2837 on the Raspberry Pi 3 version B). This has the cpu and the RAM memory and the various IO devices on it. The CPU uses an ARM architecture, a 1.2GHz 64-bit quad-core ARMv8 CPU.

This illustrates a trend in the production of large-volume consumer systems, where designs for components are bought in and then packaged together on a chip intended for a specific device.


Raspberry Pi 3 Model B
Chip Antenna
GPIO Header
DSI Display Connector
BCM2837 Chipset
Micro SD Card Slot (Underside)
USB 2.0 Port
Status LED
USB 2.0 Port
Micro USB Connector (To Power Raspberry Pi)
10/100 Ethernet Port
HDMI Video/Audio Connector
CSI Camera Connector
RCA Video/Audio Jack
DESIGNSPARK

Chapter 3

Integrated circuits

Aims

The aim of this chapter is to cover the basics of how computer chips function and are built. The story we tell uses a stack of abstractions at different levels, running from the basic physics of semiconductors up through how transistors work and their use as switches to building logic gates and then components such as memory cells and basic adders.

Component
Gate
Transistor
Semiconductors
Quantum physics

Each level of this is built in some way on the next level down. We aim to give a broad overview rather than great detail.

Putting this into the broader context, we will see later that everything in a computer is represented in binary, 1's and 0's. Inside a chip these correspond to high and low voltages, but conceptually they also correspond to the **true** and **false** of **boolean logic**. The computer needs to do calculations with the boolean values. These calculations can be represented as logical operations made up from the standard boolean connectives (**and**, **or**, **not**) and are implemented in the computer through a switching network made using **transistors** as switches.

The aim of this chapter is to show how transistors are made from semiconductors (and why they work), how they can be put together to compute logical operations (gates), and how those gates can be put together to do other computations, such as making a simple one-bit adder. We will also show how to make a simple one-bit memory.

The designs we show are not necessarily the ones used in real integrated circuits. Particularly when we come to the adder and the memory, actual designs are optimised. But they do show that the concept can be made to work.

Learning Objectives

By the end of this chapter you should understand:

- ☐ 1. how transistors are constructed from semiconductors and how they function as switches
- ☐ 2. how to use transistors to build basic logic circuits called gates
- ☐ 3. how to put units of logic together to construct other circuits such as basic adders
- ☐ 4. how circuits with feedback can be used to construct simple memory cells.

3.1 Semiconductors

This section is not examinable.

The physical reason an electric current flows in a conductor is that (negatively charged) electrons are moving in the opposite direction. Metals, such as copper and iron, are conductors.

But the quantum physics underlying this is more complicated, and some substances (like silicon) have some of the properties of metals that would make them conductors, but not all (there are issues with the energy levels of electron bands).

These substances will conduct electricity, but not readily. They are **semiconductors**, neither good conductors nor good resistors, but unlike metals their conductivity increases if they are heated.

The best known of these is silicon. Silicon is an element with atomic number 14, and valency 4. It forms crystals in which each atom is surrounding by four others at the corners of a tetrahedron (a diamond cubic lattice, the same structure as carbon when forming diamonds).

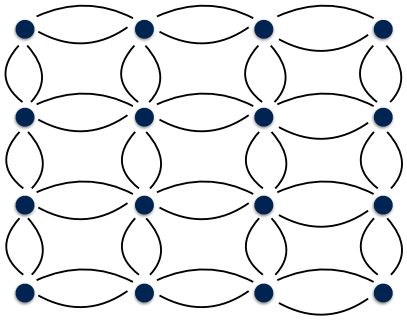
Silicon is very common, it is a basic component of rocks, including sand. Glass is made from Silicon Dioxide.

For use in chips, impurities are added to the silicon. These are called **dopants**, and they fall into two classes: electron **acceptors** or **donors**.

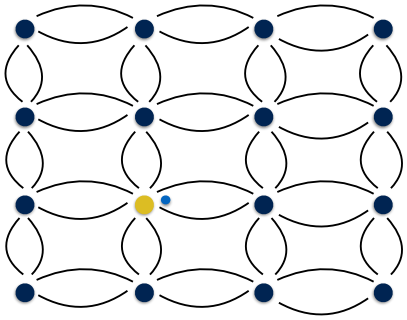
Donors produce **n-type** semiconductors. A commonly used donor is **arsenic**. **Acceptors** produce **p-type** semiconductors. A commonly used acceptor is **boron**.

Silicon crystal structure

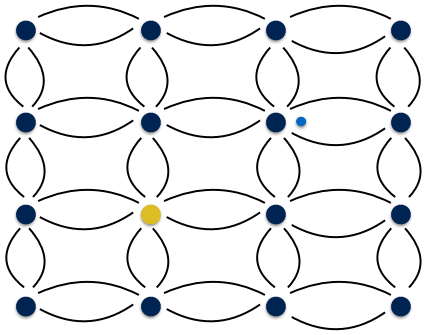
Here is a flattened out picture of the silicon crystal structure (the connectiivity is wrong, but that is not important for our current purposes). The arcs represent standard covalent bonds. In other words they are a pair of electrons shared between the outer shells of the atoms at the two ends.



Doping replaces one of the silicon atoms in the structure with an atom of the dopant (here arsenic). So this gives you a substance with the same crystal structure, but the wrong atoms at some random points. Now arsenic has an extra electron in its outer shell compared with silicon, and so what we end up with is the same basic structure as a pure silicon crystal, but with some extra electrons at various points.

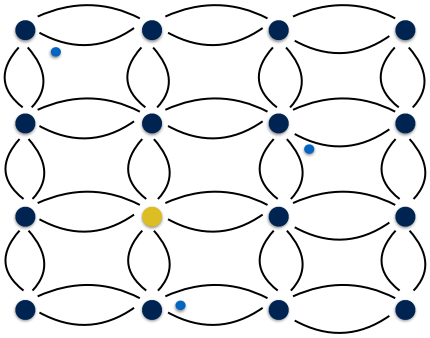


The critical thing is that these additional electrons are not tied to the arsenic atoms, but can move about the crystal structure. This allows the silicon structure to conduct electricity. Remember that metals conduct electricity because electrons are free to move through the metal, so transporting electric charge and thus making an electric current.



Understanding electrical conductivity properly, and understanding the conductivity in semiconductors properly takes some quite serious quantum physics. But the upshot is that the mobile electrons in an n-semiconductor behave as an ideal gas. This is not simply a loose analogy, there is a precise technical meaning to this, in terms of the statistical behaviour of the electrons in terms of their position and motion. As a result we can regard an n-semiconductor as electrically neutral, but holding a negatively charged electron gas at positive pressure.

The story for a p-semiconductor is similar, except that the dopant has fewer electrons in the outer shell than sillcon. This means that there is an electron missing where the crystal structure expects there to be one. This is called an electron **hole**. An electron from a neighbouring atom can be borrowed to fill that hole. If we think in terms of holes, not electrons, that means the hole moves from its original atom to a neighbouring one. As a result, the holes can move about the crystal structure in the same way as electrons, and, like the electrons in an n-semiconductor, they behave as an ideal gas. We can therefore think of the p-semiconductor as filled with a positively charged gas made out of electron holes. Alternatively we can think of it as filled with an electron gas at negative pressure.



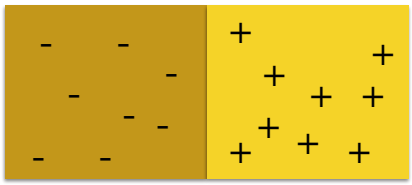
There is a symmetry between n-semiconductors and p-semiconductors.

n-semiconductor	**p-semiconductor**
standard diamond cubic lattice	standard diamond cubic lattice
dopant gives additional electrons	dopant gives additional holes (absences of electrons)
overall electrically neutral	overall electrically neutral
conducts electricity through movement of additional electrons round crystal	conducts electricity through movement of holes round crystal
electrons behave as ideal gas	holes behave as ideal gas
filled with electron gas at positive pressure	filled with electron gas at negative pressure

Extra Reading: Start with Wikipedia on semiconductors.

Diodes

Interesting things happen when you put p-semiconductors and n-semiconductors together. The simplest example of this is when you just put them next to each other. This is called a **diode**.



If we try to pass a current through this from left to right, then this would be given by electrons travelling in the opposite direction. We connect the right hand side to a negative potential (electron source) and the left hand side to a positive one (electron sink).

Now, the p-semiconductor wants to be filled with an electron gas at negative pressure. If electrons come in they increase that pressure. But the electrons cannot flow from right to left across the join between the two semiconductors because the gas on the other side is at a higher pressure, and flows take place from high pressure to low, not vice versa. The result is that the possible current is blocked at the join between the two semiconductors, there is no overall electron flow, and so no current.

The situation is different if we try to pass a current from right to left. In this case the electron source is next to the n-semiconductor and it increases the pressure of the electron gas in it. That electron gas flows easily into the adjacent low-pressure (in fact negative pressure) region in the p-semiconductor, and then across it to the electron sink. The result is that a current flows easily.

So the diode allows a current to flow easily in one direction, (from p to n), but not the other.

FACT TO REMEMBER: current can flow from a p-type semiconductor to an n-type semiconductor. Current can NOT flow from n-type to p-type.

Of course a diode can be forced to allow a current to flow in the "wrong" direction. If we put enough potential on it, then this forces an electron gas into the p-semiconductor at high enough pressure to overcome the differential. But chips are designed so that this does not happen.

3.2 Transistors

This section, and the rest of the chapter, is examinable.

The standard technology to make integrated circuits is **CMOS** (**Complementary Metal-Oxide Semiconductor**). This uses a particular sort of transistor, called a **MOSFET** (**Metal-Oxide Semiconductor Field Effect Transistor**). These are used as switches in the chip. We begin by describing what they are and how they work.

MOSFET's have four connections:

gate: which acts for us as the control on the switch

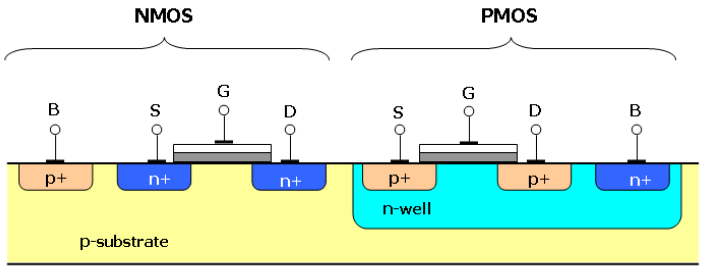
source: a connection for the wire into (or out of) the transistor

drain: a connection for the wire out of (or into) the transistor

body: which we shall ignore

The **gate** controls whether there is a connection between the **source** and the **drain**.

There are two sorts of MOSFETS, **nMOS** and **pMOS** (or **NMOS** and **PMOS**, or **n-channel** and **p-channel**).

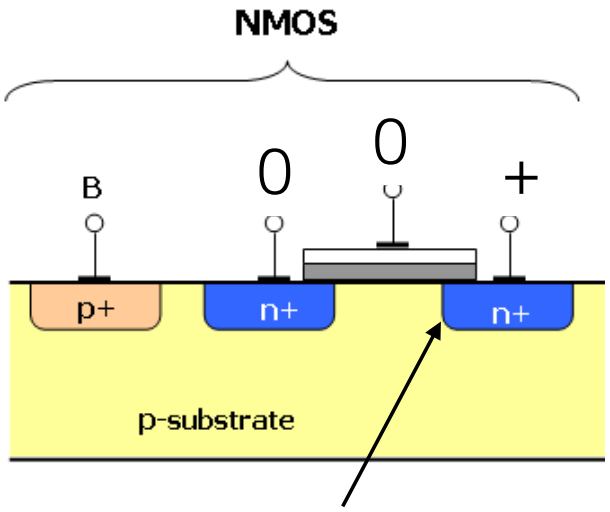


We begin by looking at the nMOS transistor.

The basic transistor substrate is made out of p-type semiconductor. The gate sits on an insulating layer, so that no current can flow from the gate into the transistor. The source and drain both sit on pieces of n-type semiconductor which themselves are connected to the p-type semiconductor substrate.

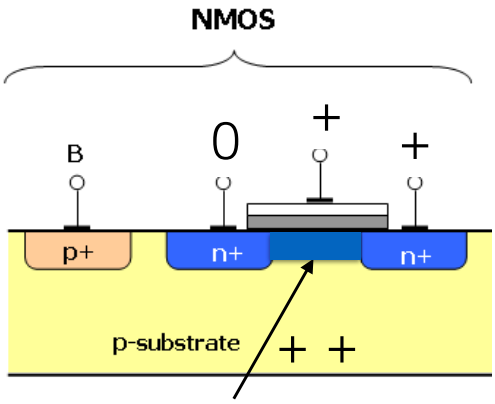
If no potential is applied at the gate, then any current from the source to the drain (or from the drain to the source) would have to flow from an n-type

semiconductor into the p-type semiconductor, through that and into another n-type semiconductor. But we have seen that current can not flow from an n-type semiconductor into a p-type, and so there is no electrical connection between source and drain.



However, remember that the p-type semiconductor is filled with a "gas" made out of positively charged electron holes. If we apply a potential at the gate, that is positive compared with the potential of the substrate, then that gas is pushed away from the gate, leaving a strip in which there are free electrons, but not holes. This strip is marked blue in the diagram below. It changes the p-type substrate so that there is a strip of what is effectively n-type semiconductor running between the source and the drain, and hence a continuous line of conducting n-type semiconductor. This allows current to flow between the source and the drain (in either direction).

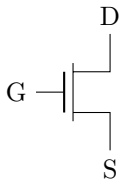
Notice that if we apply a negative potential at the gate of an nMOS transistor, then, holes are attracted to the gate, making the area a stronger p-type semiconductor, and thus ensuring that, as with the neutral gate, there is no connection.



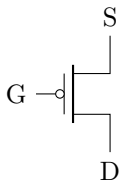
The pMOS transistors are similar, with n- and p-type semiconductors exchanged. The diagram above shows a pMOS transistor built using a large n-type well in a p-type substrate. Source and drain are connected to small p-type wells. So far as positive and negative potentials are concerned, the pMOS transistor is a mirror image of the nMOS. As a result it allows current when there is a negative potential at the gate compared with the potential of the substrate, and does not when there is a neutral or positive.

In implementations, the body of an nMOS is connected to the most negative voltage available (often ground, 0), while the body of the pMOS is connected to the highest voltage available. This means that for the pMOS 0 switches the transistor on, forming a connection, while a high voltage turns it off.

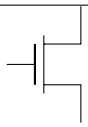
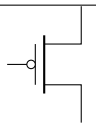
MOSFET's on circuit diagrams In circuit diagrams, an nMOS transistor is represented by the symbol



And a pMOS transistor is represented by



Summary We will use **0** for **low potential (ground)** and **1** for **high potential**. And we will say the transistor is **on** if htere is a connection between source and drain, and **off** if there is not.

	nMos	**pMOS**
AKA	n-channel	p-channel
Symbol		
On	Gate=1	Gate=0
Off	Gate=0	Gate=1
Substrate	p	n
Channel	n	p
Substrate potential	0	1

Additional reading: Find out more about how Intel actually makes these transistors on chips on their website: `http://www.intel.com/content/www/us/en/silicon-innovations/standards-22nm-explained-video.html` and `http://www.intel.com/content/www/us/en/silicon-innovations/standards-14nm-explained-video.html`.

3.3 Gates

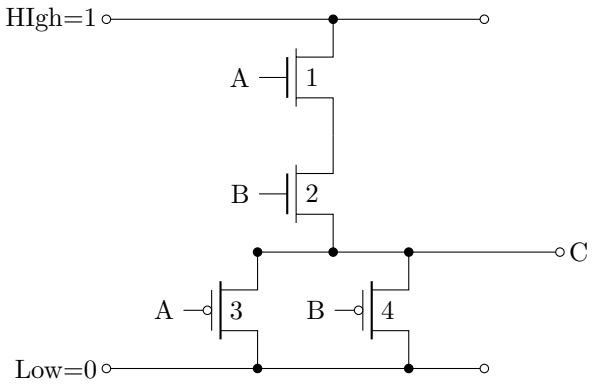
Computers work in binary, 0's and 1's, and in the context of integrated circuits, these are represented by different voltage levels. We are going to take **0** as being a low voltage (ground) and **1** as being high (perhaps 5v). One of the things a cpu does at its heart is to manipulate and do calculations with these 0's and 1's, taking some number of bits as inputs and producing a number of bits as output.

The simplest basic calculations are carried out by standard pieces of circuitry called **gates**. The standard picture is for gates to have one or two inputs and a single output. What they do in essence is calculate a formula in boolean logic.

Example: (And gate) The function of an **and gate** is to compute the boolean operation **and** as given by the following truth table (**0** is **false**, and **1** is **true**):

Inputs		Output
A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

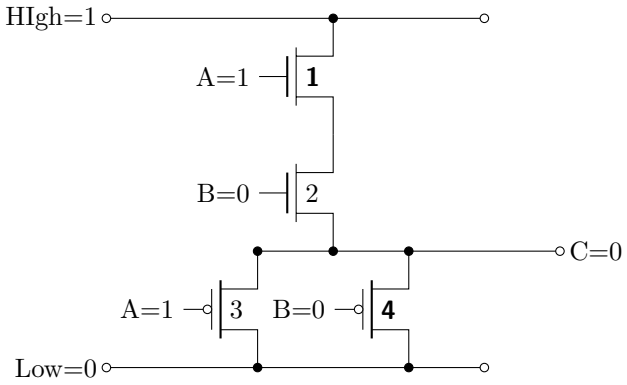
This is computed by the following circuit:



In this circuit, inputs are at A and B, and the output is at C.

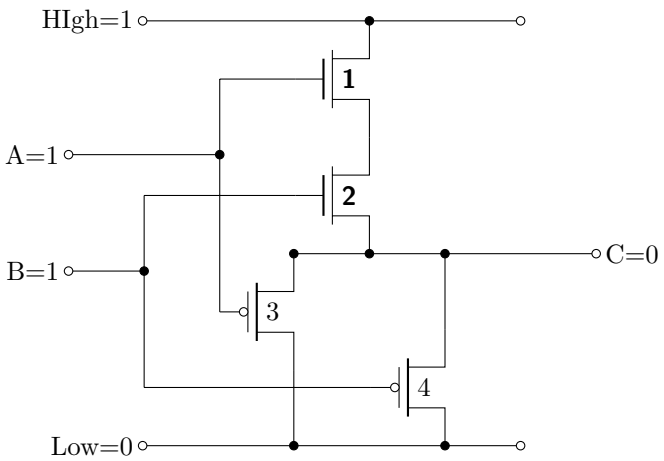
Part of the circuit design is that there is a high potential rail (at the top) and a low potential rail (at the bottom). High potential is never directly connected to low potential, so there is no significant current flow. But the output at C is always connected to one (and only one) of the potential rails at the top and bottom. So C takes the value of the rail that it is connected to.

For example, if A=1 and B=0, then the pMOS transistor at 4 is on, so C is connected to the low rail and the output is 0. The transistors that are on are shown in bold (nMOS are on when their gate is 1, pMOS are on when their gate is 0). Since 2 is off, there is no connection between the high rail and C.



In order to make the diagram simpler, we've left out the links between inputs A at 1 and 3, and inputs B at 2 and 4. In the next diagram we've put them in.

There is a connection between High and C if and only if both of the transistors 1 and 2 are on. This happens exactly when both A and B are 1.



We can summarise the working of the gate in the following table:

A	B	1	2	3	4	C
0	0	off	off	on	on	0
0	1	off	on	on	off	0
1	0	on	off	off	on	0
1	1	on	on	off	off	1

Notice that this table includes the truth table for **and** as the input-output behaviour, and that is why the gate is said to be computing it.

There is a lot of structure to this design:

- top rail as source of High=1
- bottom rail as source of Low=0
- inputs A and B
- output C
- inputs A and B used to control a switching network that connects C to exactly one of HIgh and Low.

In fact there is more structure. The switching network splits into a top half and a bottom half:

- top half connects C to High=1 exactly when (**A and B**) is **true**
- bottom half connects C to Low=0 exactly when (**A and B**) is **false**.

- top and bottom half are **duals** of each other.

We'll explain this last part. The top half contains transistors used as switches in **series**. So there is a circuit only when both switches are closed (both transistors are on). This tests that both of the gates have the right value and so corresponds to an "and" of the gate values.

The bottom half contains transistors used as switches in **parallel**. So there is a circuit when either switch is closed (at least one transistor is on). This tests that at least one of the gates has the right value and so corresponds to an "or" of the gate values.

Moreover, there is an exact correspondence between the transistors in the top half and the transistors in the bottom half, in which both transistors are controlled by the same input and one is an nMOS and the other a pMOS.

FACT: Any boolean function can be implemented by such a gate.

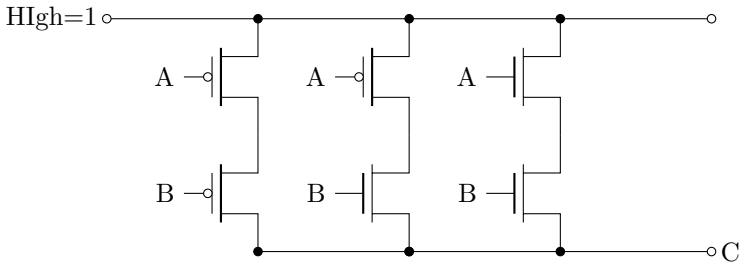
Justification: Start by writing out the input-output behaviour of the function as a table. Let's take for example:

Inputs		Output
A	B	C
0	0	1
0	1	1
1	0	0
1	1	1

(This is (**A implies B**), but we will not use that.)

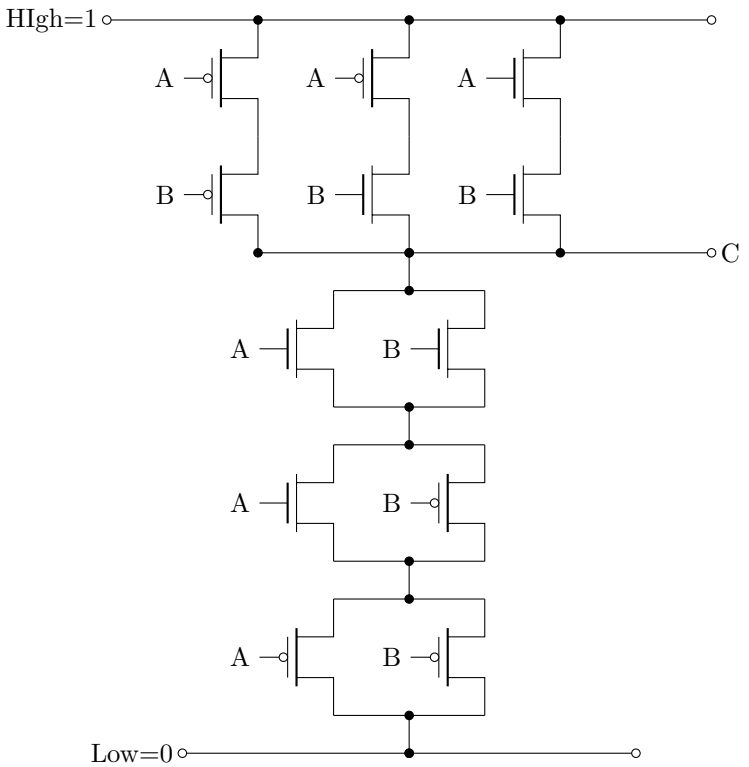
We can now construct the top half of a gate by taking all of the rows that produce 1. Here we have three such rows. For each of these rows we construct a chain with a transistor for each input, where the transistor is an nMOS if the input is 1, and a pMOS if the input is 0. We put these in parallel as the top half of the gate.

In our example we get chains corresponding to rows 1,2, and 4 of the table:



We can now construct a bottom half by dualising the top. We change the type of the transistors, and where we have things in parallel we put them in series, and vice versa.

This gives us:



In fact, what we have in this example is unnecessarily complicated (and in general this algorithm for producing gates will give you circuits that have more transistors than necessary). It would be simpler to take the single line that produces 0, use this to generate the bottom half of the gate, and then dualise that to get the top half.

Self-test

☐ 1. What is the simpler gate that results from doing this?

☐ 2. Follow the algorithm above to produce an **inverter** (a logic gate for the boolean function **not**). (Start with the lower half).

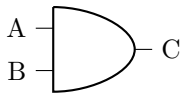
Input	Output
A	C
0	1
1	0

☐ 3. Follow the algorithm above to produce a **nand gate** (a logic gate for the boolean function **nand** = **not and**).

Input		Output
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

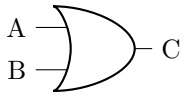
When it comes to designing larger scale circuits, these gates are treated as basic components, and there are symbols for them. We will make particular use of gates for **and**, **or**, **nand**, and **exclusive or**.

and



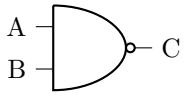
C = A **and** B

or



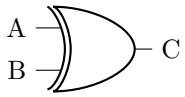
C = A **or** B

nand



C = A **nand** B

exclusive or

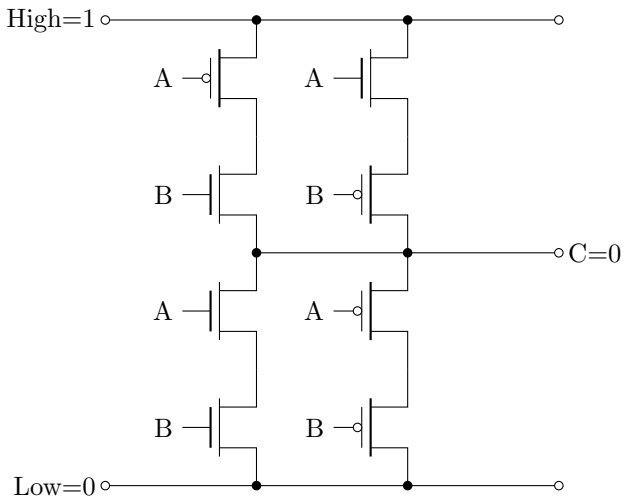


C = A **exclusive or** B

Both **and** and **or** are standard logical operations (or connectives). **Nand** is not, but it is simply the result of applying **not** to A **and** B: A **nand** B = **not** (A **and** B). **Exclusive or** is also unfamiliar, and less simple to write in terms of standard connectives. A **exclusive or** B is true when exactly one of A and B is true, but not both. **Exclusive or** is sometimes abbreviated to **xor**. The truth tables corresponding to these logical operations are:

A	B	A **and** B	A **or** B	A **nand** B	A **exclusive or** B
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

The gate that we have not seen a circuit diagram for is **exclusive or**. This diagram does not make use of the algorithm given previously:



Self-test

☐ 1. Verify that the diagram above is indeed an **exclusive or** gate.

3.4 Components: adder

In this section we will find out how to build a simple adder. First we will build a very simple one-bit adder, and then we will show how to put one-bit adders together to make an n-bit adder. This is a demonstration in principle. Actual adders are not necessarily made like this, but we will see how the technology can be used to add unsigned boolean numbers.

If you are not sure about addition in binary arithmetic, skip this section and come back to it after we have covered it in chapter 10.

Here is the simple boolean single digit addition table:

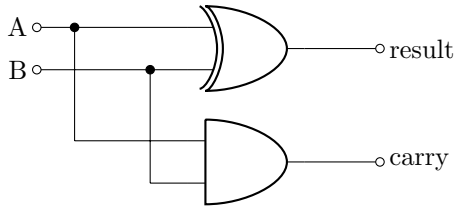
+	0	1
0	00	01
1	01	10

We have given all the results to two digits. You can think of the lower order digit as the result and the higher order as a carry.

The next stage is to look at these as boolean functions:

Input		Output	Logic	
A	B	A+B	A **and** B	A **exclusive or** B
0	0	00	0	0
0	1	01	0	1
1	0	01	0	1
1	1	10	1	0

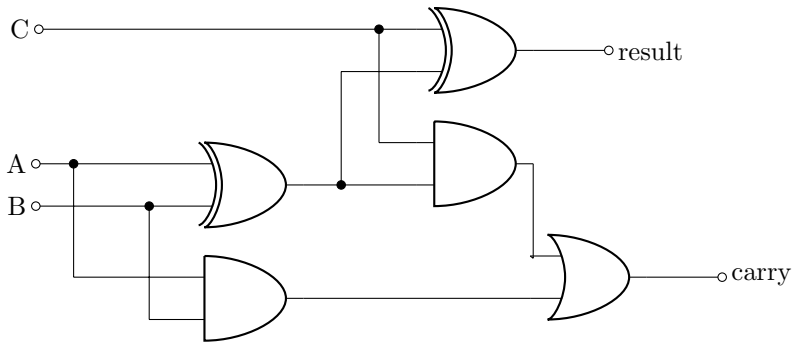
This means that the following circuit can be used to implement this addition operation:



This circuit is called **half adder**. That is because it really only does half the job it needs to. A **full adder** needs to account for a carry coming in as well as its two arguments. It can be implemented as basically two half adders, the first to add the two arguments, and the second to add in the carry. Here is the truth table for the logic you need (using **xor** for **exclusive or**):

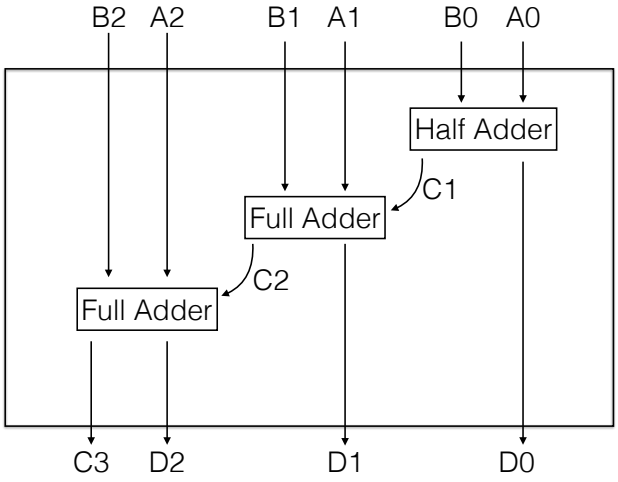
Inputs		Carry	Output	Logic				
A	B	C	A+B	A **xor** B	(A **xor** B) **xor** C	A **and** B	(A **xor** B) **and** C	(A **and** B) **or** ((A **xor** B) **and** C)
0	0	0	00	0	0	0	0	0
0	0	1	01	0	1	0	0	0
0	1	0	01	1	1	0	0	0
0	1	1	10	1	0	0	1	1
1	0	0	01	1	1	0	0	0
1	0	1	10	1	0	0	1	1
1	1	0	10	0	0	1	0	1
1	1	1	11	0	1	1	0	1

Comparing the first and second digits for the result with the logical expressions, we can see that the first (lowest order) digit of the result is (A **xor** B) **xor C**, and the second digit (the final carry) can be calculated as (A **and** B) **or** ((A **xor** B) **and** C). That means it can be calculated using the following circuitry:



Once we have a full adder, we can chain them together to make an n-bit adder. Here is a 3-bit adder built from a half adder and two full adders. The inputs are A's and B's, and outputs D's. The C's are carries. So if we are adding 011 to 110, then the inputs are

A2	A1	A0	B2	B1	B0
0	1	1	1	1	0



Self-test

☐ 1. Verify that you understand how this 3-bit adder works by adding 011 and 110 using binary long addition, and then checking how the adder given below does it. You should see a digit by digit correspondence between the paper and pencil method and the working of this adder.

☐ 2. Draw the diagram of an 8-bit adder.

At this point we have seen how to construct a general n-bit adder, at least in principle. It is not very efficient as the calculation of each bit has to wait on the previous ones. Actual hardware adders do a better job of controlling this wait and doing things in parallel, at least in the average case.

We are not going to cover how to multiply unsigned binary numbers, though you should now be in a position to see that it is at least possible. We are also not going to cover arithmetic operations on binary floating point, but again you should be able to see that they are possible, at least once we have covered the relevant material on floating point representation.

3.5 Flipflops

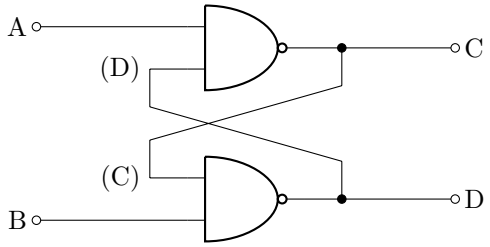
So far we have looked at how digital circuits can be used to do calculations. The circuits we have looked at take inputs and give outputs. Given the same set of inputs they always give the same outputs. The terminology for this is that they are **combinational systems**.

Definition [combinational circuit]: A logic circuit whose outputs at a specified time are a function only of the inputs at that time. (source: encyclopedia.com)

The next circuit we look at does not have this behaviour. Its outputs depend on what happened to the circuit previously. They depend on what **state** the circuit was in when its inputs were supplied. It is a **sequential circuit**.

Definition [sequential circuit]: A circuit whose outputs at a specified time depend not only on the inputs at that time, but on the sequence and ordering of previous inputs.

A basic flipflop In the literature this is also referred to as a **latch**. There are many type of flipflops (latches), but this is the simplest. It is made out of two **nand** gates. It has two inputs (A and B) and two outputs (C and D). Each input is fed into a nand gate, as one of its inputs, and the output of that nand gate is fed back to the other as the nand gate's second input.



This design is symmetric about a horizontal line through the middle (exchange A with B and C with D, and we get the same circuit).

We now find out what happens to this circuit when we give it various inputs.

First, nand gates have the property that if either input is 0, then the output is 1 independent of the other input.

Let's suppose that at least one of A and B is 0. We'll use the example A=0, but the argument when B=0 is similar. If A=0 then the output of the top nand gate is 1: C=1. The inputs to the bottom nand gate are therefore C=1 and B. If B=0, then the output is D=1, and if B=1, the output is D=0.

This means that if either A=0 or B=0 then the circuit behaves in a combinational way, with its outputs depending only on its inputs.

The remaining possibility is that A=1 and B=1. In this case we don't immediately know what value either of C or D takes. Let's try D=1. In this case the output of the top nand gate is C=0. The bottom nand gate has inputs

C=0 and B=1, so its output D=1, which is consistent with our assumption. So this is a possible stable configuration of the circuit.

Now let's try D=0. In this case the output of the top nand gate is C=1. The bottom nand gate has inputs C=1 and B=1, so its output D=0, which again is consistent with our assumption. So this is also possible stable configuration of the circuit.

Since we have covered all the possibilities for D, these are the only two stable configurations of the flipflop when A=1 and B=1.

This is summarised in the table below listing all of the possible configurations of the flipflop.

A	B	C	D
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	1
1	1	1	0

The result is that when A=0 or B=0, the circuit behaves combinationally. But when A=1 and B=1, then the circuit behaves sequentially. Its state does not depend simply on the current inputs. In fact it depends on the previous state of the circuit as follows:

A	B	C	D	next state on A=1, B=1
0	0	1	1	uncertain: flipflop can go to either C=0,D=1 or C=1,D=0
0	1	1	0	flipflop stays in C=1, D=0
1	0	0	1	flipflop stays in C=0, D=1
1	1	0	1	flipflop stays in C=0, D=1
1	1	1	0	flipflop stays in C=1, D=0

If we give the flipflop the input (A=0, B=0), then its behaviour will be unpredictable. But if we avoid that, and only give it inputs (A=0, B=1), (A=1, B=0), and (A=1, B=1), then its behaviour will be perfectly predictable. However, its behaviour for (A=1,B=1) will depend on the previous state. This means that the flipflop has a very simple memory.

3.6 Components: memory

In this section we show how the flipflop we studied in the previous section can be used to implement a simple 1-bit memory cell. This is a proof of concept. These kinds of techniques are used for some of the faster memories, but actual RAM uses a different approach which we will describe later.

We start by describing what a 1-bit memory cell does. It provides four functions:

store 0: stores a **0** in the cell, erasing what was there before

store 1: stores a **1** in the cell, erasing what was there before

hold: holds the value currently stored

read: gives the value currently held in the cell

At each time-point the cell is given one of these instructions. For it to operate as a memory cell, **read** should always produce the value placed into the cell at the most recent **store**.

The flipflop can provide these four functions if it is operated according to the correct protocol.

In this protocol, the flipflop is always given inputs (A=1,B=1) except when a store instruction is issued. The **store 0** instruction is issued by giving the flipflop inputs (A=0,B=1). The **store 1** instruction is issued by giving the flipflop inputs (A=1,B=0). The **hold** instruction is issued by giving the flipflop inputs (A=1,B=1). **read** is implemented by giving the flipflop inputs (A=1,B=1) and taking the value from D, which is always set at the current value stored in the flipflop. The flipflop is never given inputs (A=0,B=0).

Example: Consider the sequence of states the flipflop goes through corresponding to the input sequence: **store 0; store 1; read; hold; store 0; hold; read;**

instruction	A	B	C	D
store 0	0	1	1	0
store 1	1	0	0	1
read	1	1	0	1
hold	1	1	0	1
store 0	0	1	1	0
hold	1	1	1	0
read	1	1	1	0

Notice that the value at D is always that of the last store, and in particular, at the two reads it is correctly first 1 and then 0.

Self-test

☐ 1. Check the flipflop behaves correctly as a memory cell for a different sequence of input commands, e.g. **store 1; read; store 0; hold; read; store 1; store 1; hold; hold; read;**

When we studied the adder, we first built a simple 1-bit adder, and then showed that by replicating it, and connecting the copies together in the right way, we could construct an n-bit adder for any n we liked. A similar thing is true here. In order to construct a large memory, we can build it from replicas of this single cell, connected together in the appropriate way. The details are

complicated, but it should be obvious that it can be done by using the appropriate switching network to send the right signals to the right cells. This is how some kinds of memory are built, but not the RAM that is used to build main memory.

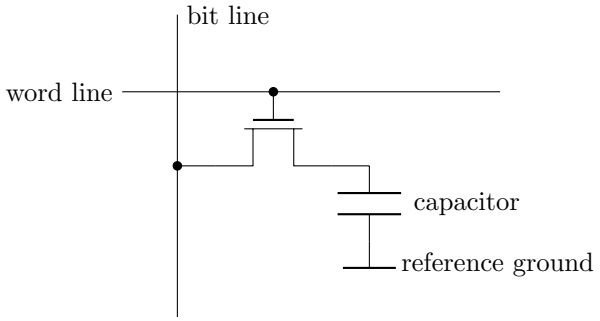
3.7 Memory: RAM

Actual random access memory is not simply built from transistors. It uses **capacitors** to store the data.

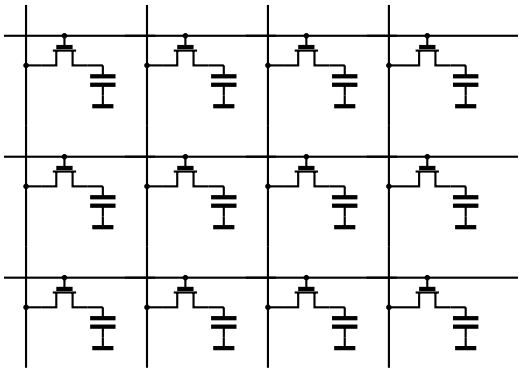
Definition [capacitor]: A **capacitor** is a device that stores an electric charge.

A single RAM device is built as a two-dimensional array of cells, each of which stores one bit. The axis in one direction gives addresses. The axis in the other gives an array of bits corresponding to a single memory read. Even though memory may conceptually be organised as an array of 32-bit "words", a single hardware read can give more than this.

The diagram below shows a single cell. When there is a high voltage on the word line, the transistor is switched on, allowing the potential in the bit line to be stored in the capacitor, or, conversely allowing the potential stored in the capacitor to be read at the bit line.



These cells are arranged in an array:



The basic idea is that when a single word line is high, then the values stored in the RAM can be accessed on the bit lines. But there are a number of difficulties in making this practical, such as:

- In order to get as much RAM onto a chip, everything is as small as possible. In particular the capacitors (which are built using the same kinds of semiconductor materials as the transistors) are small and do not store much charge.
- Reads reduce the charge stored in the capacitor, so the contents need to be put back.
- The capacitance of the bit lines is as much or more than the capacitance of the capacitors, so the potential that can be read is low and has to be amplified.
- Charge leaks from the capacitors, and so the data stored has to be periodically refreshed.

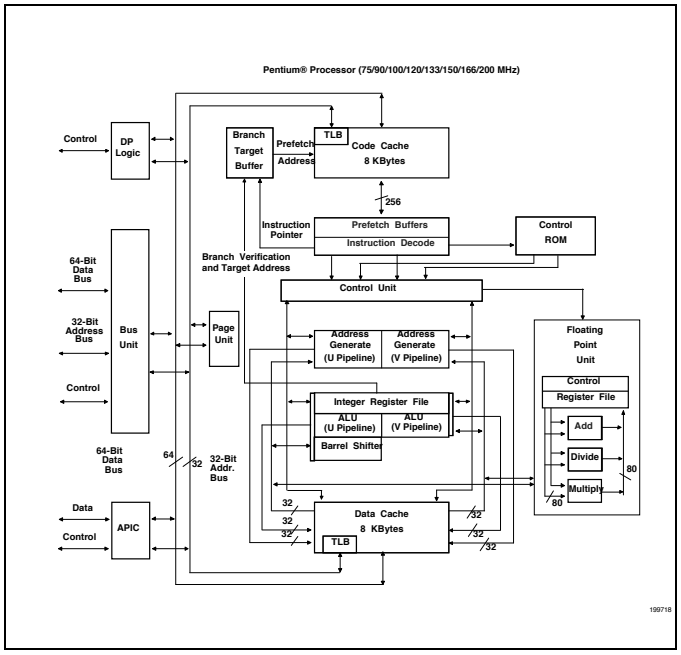
How these problems are solved is beyond the scope of this course. For more information see Bruce Jacob's lectures: `http://www.ece.umd.edu/class/enee759h.S2003/lectures/Lecture2.pdf` or the book by Keeth and Baker: "DRAM Circuit Design: A Tutorial" (`https://www.researchgate.net/publication/293483172_DRAM_Circuit_Design_A_Tutorial`).

3.8 CPU overall design and physical layout

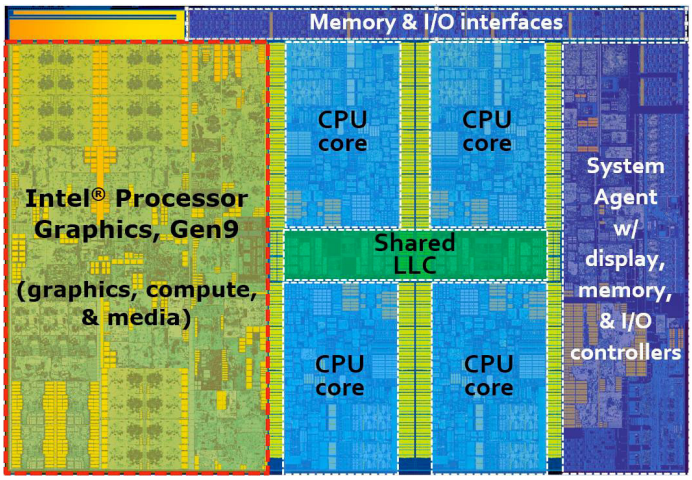
The point we have reached is that we can see roughly how basic computational functionaility can be implemented as circuits that can then be used as components to put on chips.

Complex integrated circuits, such as cpu's have a lot of these components, and are designed at various levels of abstraction. Lower level details, such as the physical layout are often handled by computer programs. One result of this is that layouts have become more complex, with the physical grouping of components less closely linked to the logical structure. Reasons for doing this include improving communications and spreading the areas of activity in order to reduce heat generation. However, major components can still be seen on pictures of the dies used to produce chips.

The following block diagram shows a single core cpu from the 1990's (an Intel Pentium). It is easy to see the different functional parts (for example control, cache, floating point unit, ...).



Here is the actual physical layout for a more recent chip, a four-core Intel Skylake processor, which, along with the four cores incorporates a GPU:



Considerable detail about how this chip is structured can be found at: `https://en.wikichip.org/wiki/intel/microarchitectures/skylake`.

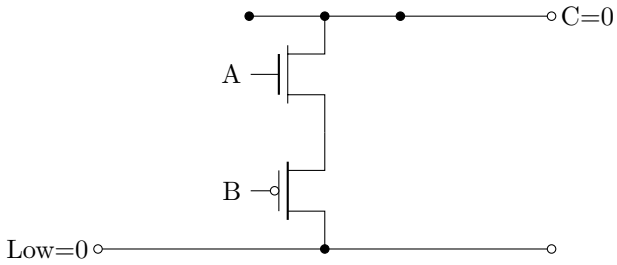
The most famous chip design companies are currently ARM and Intel. Intel has a classic business strategy that involves the company carrying out all levels from chip design through to fabrication. ARM, on the other hand, only sells designs and specifications, and it does this at different levels from a functional spec-

ification through to basic designs. It is useful to learn a little about this different approach (as it exposes some aspects of the production that are hidden inside the company by Intel's overall approach). See `http://www.anandtech.com/show/7112/the-arm-diaries-part-1-how-arms-business-model-works` for an accessible introduction.

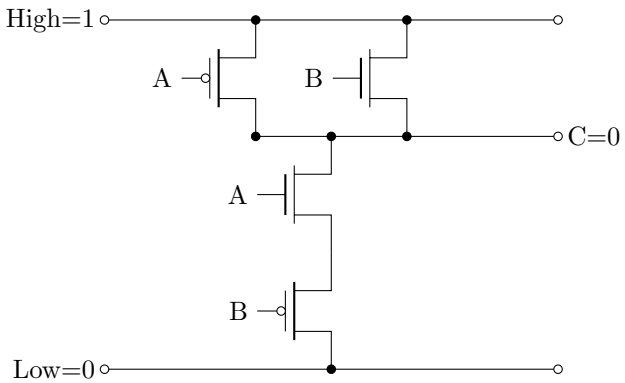
3.9 Self-test solutions

☐ 1. What is the simpler gate that results from doing this?

Solution: We look at the single line in the table that produces 0. This has inputs A=1 and B=0. That gives us a single chain for the bottom half of the gate with A controling an nMOS and B controlling a pMOS.



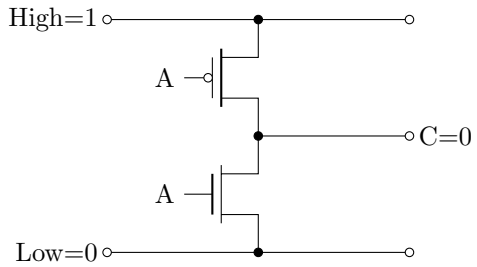
The two transistors are in series, so to dualise, we interchange pMOS and nMOS and put them in parallel.



☐ 2. Follow the algorithm above to produce an **inverter** (a logic gate for the boolean function **not**). (Start with the lower half).

Input	Output
A	C
0	1
1	0

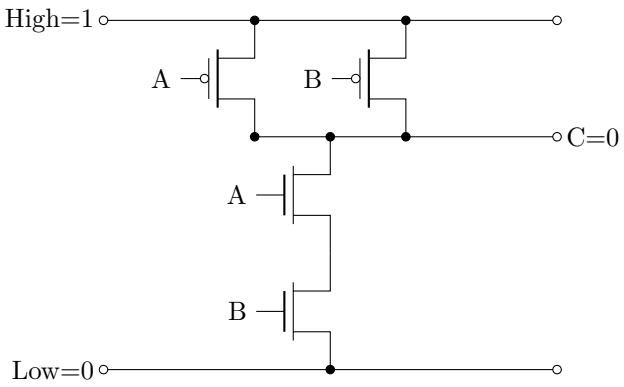
Solution: The top half is a simple pMOS and the bottom is an nMOS.



☐ 3. Follow the algorithm above to produce a **nand gate** (a logic gate for the boolean function **nand** = **not and**).

Input		Output
A	B	C
0	0	1
0	1	1
1	0	1
1	1	0

Solution: Start with the single line 4 to give the bottom half, constructed from two nMOS transistors in series.



Chapter 4

Communications inside the computer

Aims

The aim of this section is to introduce the way data is communicated around the inside of a computer. We will introduce the concept of a **bus** and discuss how designs have changed. We will discuss briefly where communication is happening and introduce basic concepts such as **synchrony** and **asynchrony**, **latency** and **bandwidth**.

Learning Objectives

By the end of this section you should:

- ☐ 1. be familiar with the concept of a **bus**
- ☐ 2. understand how a simple bus can be built from parallel wires
- ☐ 3. understand the difference between **synchronous** and **asynchronous** communication
- ☐ 4. understand the concepts of **latency** and **bandwidth**, and how they impact the efficiency of communication.

4.1 Buses

This section introduces the concept of a **bus** and explains how a simple bus can be constructed.

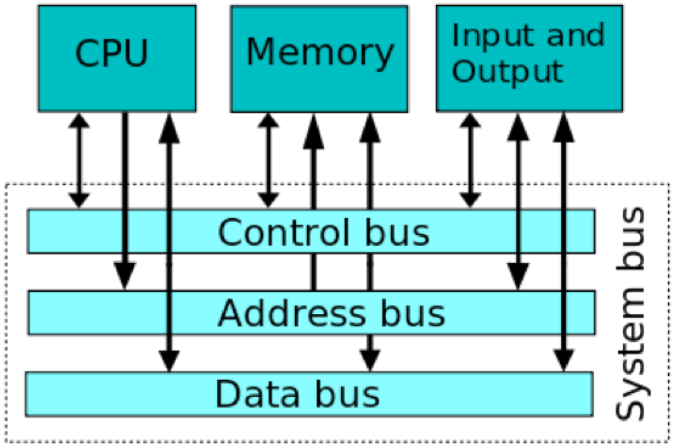
Definition [bus]:

1. a **bus** is a communication system that transfers information inside a computer
2. a **bus** is a set of physical connections that can be shared by a number of components
3. a **bus** is a collection of data lines that is treated as a single logical signal.

We find buses internally on the cpu where they are used to ferry data and program instructions around the cpu (for example from a core to the on-board cache), and also inside the computer, connecting the cpu either directly or indirectly to RAM, connecting the other memory devices (PCI and SATA), and connecting in various IO devices (USB).

The speed and efficiency of the buses has great impact on the speed and efficiency of a computer as a whole. There is no use in having a fast efficient cpu, or lots of RAM, if the cpu cannot get the data it needs rapidly enough. A slow bus can be the bottleneck that slows a computer down. Sun Microsystems (the company responsible for Java) was able to establish itself in part because the computers it built had fast buses and so ran faster than those made by its competitors that used the same cpu's and similar memory.

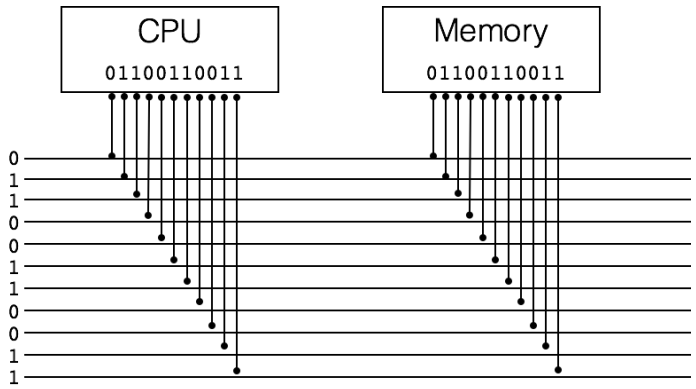
Early buses were simple and constructed using the parallel wire approach we are about to explain. More modern buses get much closer to the kind of layered comms network that we see between computers and will study later under networks.



The diagram shows a basic bus. The fact that cpu, memory and IO are all connected to the same bus is not necessarily realistic. Note:

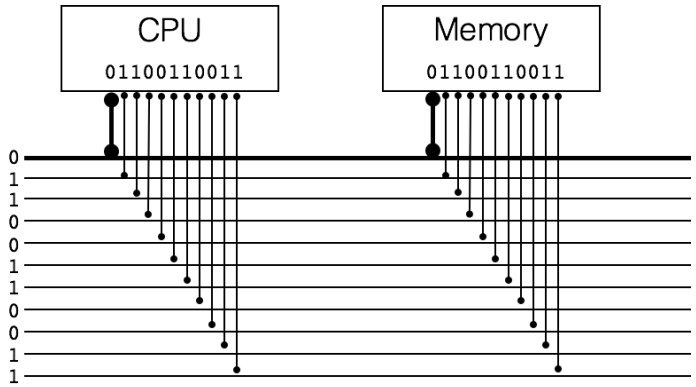
- it is split into three parallel parts: control, address and data
- the attached components each connect to all of the parts

Early buses were built from a set of parallel wires. Each component was connected to each wire.

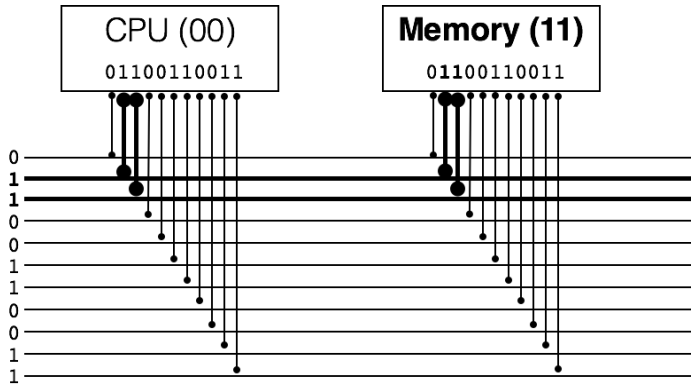


In this diagram:

- there are 11 wires
- each wire is currently carrying either a high (1) or a low (0) potential
- because the two components are connected to the bus, they have connection pins that are also at these potentials
- those connection pins connect to circuitry that allows the component to send and receive messages on the bus
- in this illustration the top wire is used for control, the next two for address, and the bottom eight for data

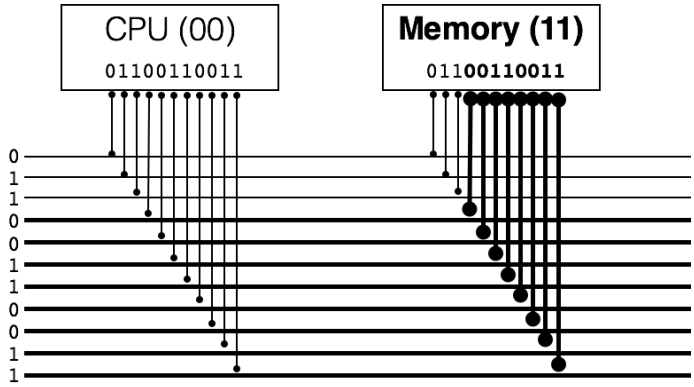


Timing and synchronisation is handled by the control wires (in this illustration 0 means that the bus is carrying a message).



Each component has an address in binary. In this example the cpu has address 00 and the memory has address 11. Since the bus wires are carrying 11, the message is for the memory. The cpu can see the message (even if it did not send it), but it is expected to ignore it because it is addressed to another component.

In early systems, this addressing configuration had to be set up when the computer was being built (or at least a very low-level system table had to be edited to tell the bus what was where. Modern buses can often cope with components being added dynamically.



Finally, the data wires carry the content of the message in binary. In this case it is 00110011.

Additional reading: Comer chapter 14 covers this kind of bus in much more detail. See 14.1-14.18 and 14.28-14.32.

4.2 Beyond the simple bus

The example we have just seen had two address lines. This means that the only possible addresses are 00, 01, 10, and 11. So we can connect at most four

devices to that bus. Similarly, it has only eight data lines. This fits with very early computer architectures, but modern ones use 32 and 64 bit designs. We can see that we might need a lot of parallel wires. This is clumsy and, if we are using actual metal wires, there are increasing issues about possible breaks in thin wires, and problems at connections.

Moreover, we have communication mechanisms functioning at various levels. If we turn to the motherboard diagrams of chapter 1 we see various levels on the motherboards. Looking at Figure 2.2, we can see a special purpose bus for main memory, lots of PCI Express for cards and other devices, various ATA for disks, and lower down, the USB and some legacy PCI.

We have at least the following:

- internal comms on the cpu, between different subcomponents of the cpu, implemented in silicon
- a very high-traffic fast link on the motherboard between cpu and main memory
- further high-traffic links connecting the motherboard to devices that are internal parts of the computer and will not change, e.g. a graphics card, the hard disk, or optical disks (dvd) (SATA and PCIexpress)
- lower-traffic links to IO devices such as the keyboard and mouse (usb)
- external networking linking this machine to other computers (ethernet and wifi)

Being cruder we have:

- the cpu is a cluster of communicating subcomponents
- the main box of the computer is a cluster of communicating devices
- the computer as a whole (main box plus peripherals) is a cluster of communicating devices
- the computer is part of a network which consists of communicating computers

Communications at each of these levels use different technologies, but some of the design issues are in common, and some of the techniques used to solve these are shared.

In particular, some of the design methods for network-level communication are also now used in the design for the modern buses used inside computers for internal communication. The details are different.

In particular, modern buses:

use fewer wires for the physical connection: For example SATA, the bus mechanism used to connect hard disks (and internal optical disks) to the motherboard, uses four wires. A comparison is wired ethernet, which uses two (twisted pair).

send data as a sequence of bits, not in parallel: This is a simple consequence of there being fewer physical wires.

use a layered packet-based approach, so that messages are sent in structured packets: this is true for both SATA and PCIexpress, the slightly older protocol used to connect network cards.

4.3 Synchronous and asynchronous communication channels

Definition [Synchronous]: Communication is said to be **synchronous** when the message is received at (essentially) the same time it is sent. This is sometimes refined to mean that the message channel has a fixed externally set schedule (for example set by an external clock).

Definition [Asynchronous]: Communication is said to be **asynchronous** when the message may not be received at the time it is sent. This is sometimes relaxed to mean that the message channel organises its own schedule by sending suitable signals.

Everyday examples:

Synchronous: phone calls, face to face communication

Asynchronous: texts, emails, letters

The simple bus described above is synchronous. Synchronous communications are often easier to design and build, but they depend on the hardware attached being able to cope. Asynchronous communications are often harder because you need to have something like a **buffer** in order to store messages that are waiting for the next stage of processing (messages that the source has sent, but that have yet to be transmitted on the channel, and messages that have arrived at their destination, but have not yet been picked up).

4.4 Bandwidth and Latency

Bandwidth and latency are also important general communication concepts.

Definition [bandwidth]: **Bandwidth** measures the capacity of a channel in terms of the amount of data it can transport in a given period of time. It is measured in bits per second, or a similar variant (e.g. kilobits per second).

Bandwidth gives the maximum rate of data transfer.

Warning: (bits and bytes) Bandwidth is traditionally measured in **bits** (kilobits per second). Storage (RAM or disk) is generally measured in **bytes**. 1 byte = 8 bits. So if you have a 10MB (megabyte) file, and send it over a 1 Mbps (megabit per second) channel, then it will take 80 seconds, not 10.

Definition [latency]: Latency measures the responsiveness of the channel. It is the time taken before you start to receive a response, and it is measured in seconds. It can be measured either one-way (the time a message takes to traverse the network from source to destination), or two-way (the time a request takes to get a response).

Everyday examples:

1. You want to download a complex webpage (e.g. Facebook). This can involve fetching 100 files from remote servers. None of them is huge, so the issue is the extra small amount of time you have to wait for each connection, in other words, latency.

2. You and all your family want to watch different programmes from Netflix. You all want a large amount of data from the same place. The issue is whether the network can deliver this, in other words, bandwidth.

Latency is a major issue in the internal communications of a computer. Consider the following timings:

CPU clock cycle: 0.5ns

CPU single instruction execution: 1-5ns (approx)

RAM memory access and transfer rate: 100ns and 6GB/s

HDD access and transfer rate: 15ms and 150MB/s

SSD access and transfer rate: 0.03ms and 500MB/s

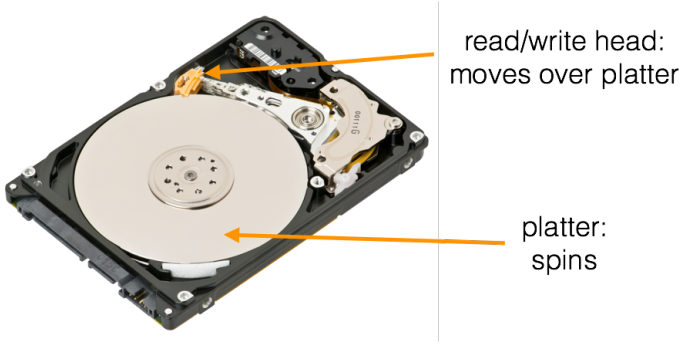
Obviously these depend on the specific devices, but these are current orders of magnitude as of 2016.

You can see that form RAM access, the 100ns latency compared with cpu clock speed is what will slow up a computation that requires a great deal of memory access.

But the really interesting case is SSD vs HDD.

SSD v HDD The conventional HDD (hard disk drive) has a magnetic platter that spins. Information is kept on tracks and read by a head that moves between tracks. So, unless you are lucky, when you ask to read or write data, the disk has to move the head to the correct track, and then wait for the platter to spin to the point that the data wanted is under the head and can be read. Moving the head takes about 100ms, and the disk spins at either 5400 or 7200rpm. This means it takes about 4ms for a 7200rpm disk to make half a revolution, the

expected time you would have to wait. It follows that the latency for an HDD is both large and uncertain. By contrast the latency for an SSD (solid state disk) is much lower and fixed. The major impact of this is if you are accessing numerous small files. But that is what modern computers do when they are running programs. So the SSD makes the computer much faster. By contrast the difference in transfer rates is relatively insignificant.



Chapter 5

Data storage

Aims

The purpose of this chapter is to introduce the various levels of storage in a computer, introducing the concept of the memory hierarchy. We will also discuss how different levels of the hierarchy can be used to mitigate issues with other levels, concentrating on the use of caches (small low-latency memories used to reduce the overall latency of larger high-latency memories), and virtual memory (a small fast memory and large slow memory used together to simulate a large fast memory).

Learning Objectives

By the end of this chapter you should:

- ☐ 1. understand the concept of the memory hierarchy
- ☐ 2. be able to list is levels and
- ☐ 3. explain its ordering.
- ☐ 4.

5.1 Introduction

Computers have various levels of memory. There are typically at least three:

CPU registers: the cpu has registers storing information that it can actually process

main memory: used for storing the data required by current processes

disk memory: used for storing long-term data that has to survive the computer being turned off

This is a simplified picture. There are other small areas of memory, and we will see that some levels are used to simulate others. The account above ignores other levels such as network and cloud-based storage.

In addition, memory shows very strongly two of the characteristic features of computer design: the use of virtualisation, and the importance of different levels of abstraction. It is very difficult to say what the hardware actually does because what used to be a relatively simple piece of hardware has now often changed into a more complex mix of hardware and software control and interface that behaves in a similar but more efficient way. Moreover, the the account we give of the structure of memory and the way that it is accessed depends on what system level we are using.

- hardware (actual implementation)
- hardware (as it appears to the cpu)
- operating system
- programming language

We will see all of these when we talk about main memory, but first a brief account of the cpu.

5.2 CPU registers

The standard interface to a memory device has operations to read data from a specific location and to store given data in a specific location. There are no operations that change data in situ, memory hardware does not support such operations, and including them would significantly complicate its design. As a result, all of the work in changing data has to be done by processing units, most importantly the cpu.

The cpu has a small number of storage locations called **registers**. These each have a fixed size.

Examples:

MIPS: 32 general purpose registers, originally 32-bit, now 64-bit

ARMv8: 31 general purpose registers, 64 bit

x86: A complex structure of registers of different sizes and purposes

Keeping things simple, the contents of the registers determine the state of the cpu. The registers change as the cpu executes instructions. MIPS has two sorts of instructions. One kind of instruction loads or stores information from main memory into registers. The other kind performs operations on data in registers

and stores the result in another register. x86, by contrast, has instructions that refer to locations in main memory and the data for these, therefore, has to be pipelined into or out of the cpu.

5.3 Main memory: the simple picture

Main memory is the primary location where the temporary data is stored. It consists of a sequence of memory locations, indexed by addresses and all the same size, usually one byte (= 8 bits). The addresses are bit sequences, all the same length, and we can think of them as numbers running from 0 to (MAXMEM-1), where MAXMEM is the number of addresses available.

0	
1	
...	...
MAXMEM - 1	

The length of the address is machine dependent, but in modern computers will be either 32 or 64 bits. If 32 bits is used then only about 4 billion addresses are available, and if each location contains 1 byte (= 8 bits), then the maximum size of memory that can be addressed is 4GB. This is smaller than on many modern machines.

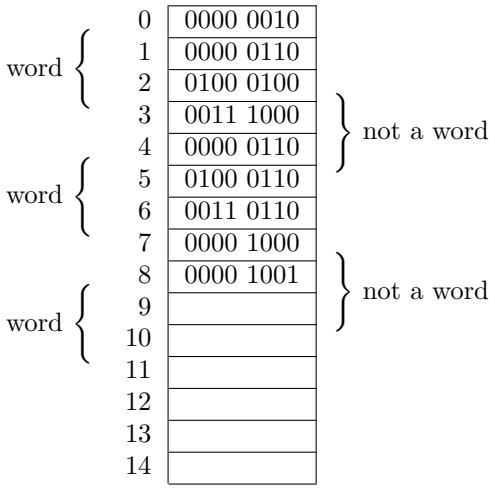
The memory supports two basic operations: read and write. A read takes a single argument, a memory address, and returns the value stored at that address as binary data. A write takes two arguments, an address, and a piece of binary data (of the right size). It does not return a useful value, it simply stores the data at the address.

A read will always return data of some fixed size. The amount of binary data that a memory naturally returns is called a **word**. It is also usually 32 or 64 bits. Again this is machine-dependent.

When you consider how languages such as Java behave, then the account refers to main memory. Program variables and method and procedure parameters refer to storage contained in a structure called the stack. Objects refer to storage contained in a structure called the heap. Both of these are part of the main memory assigned to running the program.

5.4 Main memory: complications

Bytes versus words A byte is 8 bits, and a word is 32. So there are four bytes in a word. Treating memory as an array of bytes, words consist of four consecutive bytes starting at an address divisible by 4. In binary this means that the last two digits of the address are 0.



The MIPS assembly language studied later has a number of instructions for reading and writing from memory. Since the language is written from the perspective of the cpu and its registers, these are called **load** (= read from memory into register) and **store** (= write from register into memory). These instructions include **lw** and **sw** (load and store words) and **lb** and **sb** (load and store bytes).

Most modern architectures use this plan of addresses that refer to bytes (**byte addressing**), but there is an alternative of using addressing that refers to words (**word addressing**). However the structure of memory is such that addresses pass through several layers of translation and implementation, in both hardware and software, on the way from the cpu to the actual memory chips. This means that there isn't a concrete hardware reason to use one or the other.

Big Endian and Little Endian In a byte addressed memory, a 32-bit word occupies four sequential bytes, but there is a choice of which order the bytes of the word come in. Take for example the 32-bit word below:

0000 1111	0000 1100	0000 0011	0000 0000

Suppose we want to store this in memory at address 100. We can either choose to store the first byte at the lowest address:

100	0000 1111
101	0000 1100
102	0000 0011
103	0000 0000

This approach is called **Big Endian**. It seems the obvious approach. It looks as if the bits are laid out in the same order in memory as they are in the word, given the way that we write the word.

The alternative is to store the word with the lowest order bits in the first byte:

100	0000 0000
101	0000 0011
102	0000 1100
103	0000 1111

This approach is called **Little Endian**. It means that the natural order of indices for the bits agrees with the natural ordering of the bits in the word when we view its contents as a number. Lower order bits get lower addresses. Most computers including the x86 architectures at the base of both Macs and PCs use Little Endian.

Ultimately the issue is that we write text from left to right, but numbers from right to left. So if we want to put characters into an array, then the natural indexing starts at the left:

r	h	o	m	b	o	i	d
0	1	2	3	4	5	6	7

While if we want to put the digits of a number into an array, then the natural indexing starts at the right, since it corresponds with the power of 10 that the digit represents:

1	9	8	4
3	2	1	0

$$1984 = 1 * 10^3 + 9 * 10^2 + 8 * 10^1 + 4 * 10^0$$

Little Endian takes a consistently numeric view of the data. Big Endian mixes viewing it as text and number. As a result, despite appearances, the Little Endian approach is more closely linked to what happens on silicon.

In hardware x86 architectures are little endian, while ARM architectures allow either, but favour little endian. Few types of machine now use big endian. However big endian is not dead, network protocols use big endian representations.

Self Test Exercises:

☐ 1. Which of the following byte addresses written in decimal are valid word addresses:

- ☐ *a*) 1024
- ☐ *b*) 2034

☐ 2. Which of the following byte addresses written in binary are valid word addresses:

- ☐ *a*) 0000 0000 0000 0000 0000 0000 1100 0101
- ☐ *b*) 0000 0000 0000 0000 0000 1000 0101 1100

☐ 3. Given the following memory, what is the word at 12 given that the representation is

- ☐ *a*) little endian
- ☐ *b*) big endian

10	0100 0000
11	0000 0011
12	0010 1100
13	1001 1001
14	0000 1100
15	0101 1111
16	0110 1101
17	0000 1111

Base-offset addressing Writing code that puts data into specific addresses in memory is almost always a bad idea. Most computers run more than one program simultaneously, and you do not want two programs to be using the same memory address. In fact you want each program to have its own separate area of memory, assigned when the program is run. This means you want the program to be capable of running either in a block of memory towards the start of memory, or in a similar block towards the middle or end. In order to assist this, assembly languages do not just use fixed addresses, they also use a method called base-offset (or base or displacement addressing).

In MIPS base-offset addressing, the instruction has three arguments: a register for the data value, a register containing the base address and an offset (given as a small binary number). The offset is added to the value of the base and the result is used as the memory address for the operation.

This means, for example, that if a register contains the base address of a block in memory containing program data, then that block can be moved about (so long as the base address register is kept in sync).

Base-register addressing makes it much easier and more efficient to implement various sorts of code:

> **arrays:** the base is the start of the array and the offset corresponds to the index times the entry size
>
> **objects:** each object has an area of storage, with the base pointing to the start of that area and each variable at a fixed offset
>
> **procedure and method calls:** use an object-like frame with the base being the start of the frame and parameters and variables located at specific offsets from the start of the frame

Further reading: Patterson and Hennessy 2.3 (p69) and 2.10 (p118)

The following lengthy quote from "Understanding the Linux Kernel" [**?**] makes some of the different layers clear:

Programmers casually refer to a memory address as the way to access the contents of a memory cell. But when dealing with 80 x 86 microprocessors, we have to distinguish three kinds of addresses:

Logical address *Included in the machine language instructions to specify the address of an operand or of an instruction. This type of address embodies the well-known 80 x 86 segmented architecture that forces MS-DOS and Windows programmers to divide their programs into segments . Each logical address consists of a segment and an offset (or displacement) that denotes the distance from the start of the segment to the actual address.*

Linear address (also known as virtual address) *A single 32-bit unsigned integer that can be used to address up to 4 GB that is, up to 4,294,967,296 memory cells. Linear addresses are usually represented in hexadecimal notation; their values range from* `0x00000000` *to* `0xffffffff`.

Physical address *Used to address memory cells in memory chips. They correspond to the electrical signals sent along the address pins of the microprocessor to the memory bus. Physical addresses are represented as 32-bit or 36-bit unsigned integers. The Memory Management Unit (MMU) transforms a logical address into a linear address by means of a hardware circuit called a segmentation unit ; subsequently, a second hardware circuit called a paging unit transforms the linear address into a physical address (see Figure 2-1).*

Figure 2-1. Logical address translation

5.5 Disk memory

Long-term memory traditionally takes the form of a **hard disk drive (HDD)**. Unlike main memory, the cpu does not see this directly. The memory here is significantly slower than main memory, but much larger, and its contents do not disappear when the computer is switched off.

A traditional hard disk drive has a number of platters. These spin at about 7200rpm (120 revolutions a second), slower in low performance, and faster in high performance drives. These contain magnetic material which is read by a head that moves to and fro over the surface of the platter.

Before use, the disk has to be set up by being formatted. This creates a system of concentric rings on the disk called **tracks**. The tracks are further divided into slices called **segments**. The segments are then grouped together into **blocks**. Blocks are the basic read/write entities for disks. They are ordered, and so disk memory, like main memory can be regarded as an array of storage.

File systems are extra structure that lives on top (at the operating system level, rather than hardware).

5.6 Memory hierarchy

We have seen that information gets stored by different devices (or components). These devices can be ordered by response time (how long does it take for the device to respond once it received a memory read or write request). The result is a ranked ordering of memory devices that form part of the computer. This is called the **memory hierarchy**.

Definition [memory hierarchy]: The **memory hierarchy** is the list of memory components and storage devices that form part of a computer's architecture, ranked by response time from the point of view of the component, with cpu registers at the top, passing through main memory (RAM) and long-term storage (HDD, SSD, flash memory) and potentially including network or cloud storage at the bottom.

Important note This concept presupposes an understanding of which physical devices are part of the computer and which are not. This understanding is challenged by temporary storage devices attached to the machine (eg usb disks) and by network and cloud storage. It is clear that if we ask: "which components are part of the machine", then the physical description layer will suggest different answers from more abstract conceptual layers (it won't include anything that is not a permanent physical part of the machine, however servers may not physically include disk storage, but are inoperable without it).

Other levels The memory hierarchy may also include other levels between those mentioned above, notable cache.

Other ranking mechanisms The response time ranking also correlates well with other possible ranking mechanisms. These include:

- the amount of storage available
- the cost of the storage per Gb
- the route that data follows through the machine

Essential Reading: Patterson and Hennessy: 5.1 and 5.2 p374–383

5.7 Cache

The abstract concept **Caches** are an example of the way a technique used in everyday life by ordinary people is also used in computers.

A **cache** is a relatively accessible store of some resource that you believe you are likely to need in the near future. The word comes from the French, cacher: to hide. Trappers in the wilderness used to hide caches of supplies, so that they did not have to carry as much on them.

When you are studying in a library, you take books of the shelf and put them on your desk. Over time, you may put some back, and take more out. But you keep a working set, and don't keep putting the same book back and taking it out again. In this case the desk is acting as a cache, in which you are storing the books from the library that you think you are likely to need in the rest of your session.

The reason you do this is because it takes you longer to get a book off the shelves than to pick it up off your desk. In other words, the latency of access for a book on the shelf is longer than that for a book on the desk. Keeping a working set improves the overall latency of access.

Caches are used to improve the latency of access throughout computer science. For example they are used in web services, where caches of recently accessed web pages are kept.

They are also used in the memory hierarchy.

Caches in the memory hierarchy In the memory hierarchy a **cache** is a small amount of relatively fast memory that sits between the client (the cpu) and the server (the actual memory). The job of the cache is to store a small set of data from the slower memory device.

A basic cache consists of a store and an index (or table) of the memory items currently in the store. Simple memory accesses are replaced by more complicated processes. The basic picture for a read is as follows:

> **Lookup:** the memory address is looked up in the cache index (or table). If the item is already there then this is a **cache hit**. If it is not then it is a **cache miss**.
>
> **Cache hit:** the original memory address is replaced by the address of the item in cache, the item in cache is used instead of retrieving the original from memory
>
> **Cache miss:** the item is retrieved from memory, placed in cache, indexed, and then sent on to the cpu.

Notice that both possibilities result in a more complicated operation than a basic memory access. The cache miss includes a basic memory access, so it is **slower** than the basic memory access. The cache hit includes an additional index lookup and a cache access. So it will be faster only if the lookup and the cache access are both much faster than the basic memory access.

In this context it is good to have a clear idea of the relative access times of various technologies (source Patterson & Hennessy):

Technology	Access time	$ per GB 2012	Use
SRAM	0.5-2.5ns	$500 - $1000	registers, cache
DRAM	50-70 ns	$10 - $20	main memory
Flash	5,000-50,000 ns	$0.75 - $1.00	SSD
Magnetic disk	50,000,000 - 20,000,000 ns	$0.05 - $0.10	HDD

The cache as a whole will be faster than using basic access if the cache memory being used is fast enough to allow the additional overheads of a cache hit and there are few enough cache misses (the hit rate is good). DRAM access time is of the order of 50-100 times slower than SRAM, and moreover the access time is about 100 clock cycles. The difference between DRAM and the two disk technologies is wider, and therefore caching disk accesses makes even more sense.

There are further complications:

- After a while the cache becomes full and you have to start working out which data to replace.
- If you do a write then you have to remember to write the changed item in cache back to the main memory before you replace it in cache with something else. But you do not have to write it back every time it changes, and you do not have to write back something that has not changed.
- Some memory devices (notably hard disks, but also some RAM) are set up so that they naturally return more data than a standard memory request asks for. The extra data also gets put in cache and it means that if you ask for a piece of data, then you automatically also get some of its surrounding data put into cache. This reduces the number of cache misses.

The cache enables a large slow memory plus a small fast memory to simulate a large fast memory.

It works **well** when the data requested is usually in the cache. This happens when the data satisfies a **locality** property: at any given time, the data used in a short interval about the time can mainly be found close to the data in use at the time.

Self Test Exercises:

☐ 1. Which of the following data access patterns have a good locality property, and hence work well with caches?

- ☐ *a*) a small number of individual variables
- ☐ *b*) an array accessed at random
- ☐ *c*) an array accessed in increasing sequence
- ☐ *d*) an array access in decreasing sequence
- ☐ *e*) a matrix stored in column order and accessed in column order
- ☐ *f*) a matrix stored in column order and accessed in row order.

☐ 2. Which of the following code patterns have a good locality property and hence work well with caches?

- ☐ *a*) straightforward code with few or no while loops or if statements

- ☐ *b*) code where the while loops have short bodies
- ☐ *c*) code where there are while loops with very long bodies
- ☐ *d*) code with lots of procedure or method calls to external libraries

Essential Reading: Both Patterson and Hennessy and Comer have extensive material on caches:
Patterson and Hennessy: 5.3 Basics of Caches p383–398
Comer ch 12: 185–204

5.8 Virtual Memory

The idea behind virtual memory is very similar to the idea behind caches. But caches are used to make a slow memory faster. Virtual memory uses similar techniques to make a small memory larger. The basic idea behind traditional **virtual memory** is to put temporary storage onto the disk, and then use RAM as a cache for that temporary storage.

In Comer, virtual memory is presented as a general abstraction layer between a system view of memory and hardware.

Essential reading: Comer 11.0–11.12 (p 163–171)
Patterson and Hennessy 5.7 (p427–430 as essential and rest of section additional).

5.8.1 History

Virtual memory was first used at a time when most computers were mainframes and developers wanted to get them to run multitasking. The standard main memory was extremely expensive hand-woven magnetic core, and therefore small. It could not contain many processes. The obvious get-out was to park some of the processes on the hard disks. Virtual memory not only does that, but it also allows you to park parts of the current process that are not active.

Patterson and Hennessy explain how virtual memory can be regarded as a form of cache. The main memory acts as a cache for system memory held on disk.

5.9 Self-test solutions

Self Test Exercises:

☐ 1. Which of the following data access patterns have a good locality property, and hence work well with caches?

- ☐ *a*) a small number of individual variables: these can all be kept in a small area of storage, so there is good locality
- ☐ *b*) an array accessed at random: if the array is large, and say only half will fit in cache, then a random access will cause a cache miss about 50% of the time. So there is poor locality.
- ☐ *c*) an array accessed in increasing sequence: if the cache algorithm fetches data in blocks, so that it also loads data close to the data being requested, then each cache miss will load a sequence of elements of the array, and generate further cache hits. So this exhibits good locality.
- ☐ *d*) an array access in decreasing sequence: it does not matter whether the access is in increasing or decreasing sequence.
- ☐ *e*) a matrix stored in column order and accessed in column order: matrices are typically stored as shape information, and then an array of the actual elements. Column order means that the first n elements of the array are the first column or the matrix, the next n are the second column and so on. Accessing in column order means that this array is accessed sequentially. So this has good locality.
- ☐ *f*) a matrix stored in column order and accessed in row order: in this case the accesses jump about the array, so have poor locality if the array is large.

☐ 2. Which of the following code patterns have a good locality property and hence work well with caches?

- ☐ *a*) straightforward code with few or no while loops or if statements: at the machine code level accesses are largely sequential, so there is good locality.
- ☐ *b*) code where the while loops have short bodies: in this case the entire body of the loop can be kept in cache, and so there is good sequentiality.
- ☐ *c*) code where there are while loops with very long bodies and lots of if statements: in this case the cpu jumps about the code, and so there is comparatively poor locality.
- ☐ *d*) code with lots of procedure or method calls to external libraries: in this case the cpu jumps from the current program to external libraries, so there is poor locality.

Chapter 6

Input-Output

Aims

A short introduction to how computers handle Input and Output.

Learning Objectives

☐ 1. understand the basics of how computers carry out input and output, including the concept of interrupt

6.1 Introduction

Modern computers have a fractal structure: the same kinds of features occur at different levels of abstraction. For example, the cpu contains subsystems that communicate with buses, and its own memory in the form of onboard cache. Standard computers are connected to a local network, which might feature its own storage and other compute servers. This is particularly the case in modern data centres, where the computers are in racks and groups of racks are tightly coupled.

In this context it is not always easy to identify the boundaries of the machine, and therefore to specify what counts as input and output for it. In the modern context it may not be sensible to distinguish between IO and other forms of communication with devices, which could be internal or external.

However there are some aspects of IO that it is beneficial to discuss.

6.2 Interrupts

There is a fundamental difference between early computers and modern ones. Early computers only did one thing at a time. If you got the computer to print a document, then that was all it would do. It would simply sit there unresponsive

until the document had finished printing. You could not edit another document or do any other work. We however, are used to being able to play video while editing and compiling code. Computers appear to do all of these simultaneously. However, appearances are deceptive. The main way that computers do things simultaneously is by switching between different tasks very quickly. In order to do that effectively they need a mechanism that can be used to stop them doing whatever they are doing at the time, and pay attention to something else. That mechanism is the **interrupt**.

Basic assembly language execution model:

- execute a sequence of simple instructions
- keep track of the next instruction, and then execute it.

In this model, once a computer is executing a program it will simply continue to execute it until the program terminates. There is no way to interrupt the program.
Scenario 1:

- Computer is running a program Lets say its playing Beethovens 9th on iTunes.
- You want to do something else (lets say start typing answers to your coursework).
- How do you get the computer to stop running its program and do something else, given the account above?

The answer is you can't stop the program without there being something else.
Scenario 2:

- Your computer calls for some information from a network mounted disk.
- Then it goes off to do something else.
- And then the information comes back (on the bus).
- How does the computer know that it has to stop doing things and deal with the information on the bus before it gets replaced by more information coming in?

The answer in both these cases is an **interrupt**.

An **interrupt** is a hardware instruction that stops the cpu doing whatever it is doing at the moment and passes control to a specified **handler**. Think of it as a wire going into the cpu.

Handling interrupts

When there is an interrupt, e.g. there is something on the bus for the cpu, the cpu saves the state of whatever program happens to be running at the time, and then runs a small"deal with incoming data" program'.

A running program (or process) has essentially three parts:

- the code
- its live data (held in main memory)
- its immediate working data (cpu registers)

The code does not change. The live data in main memory can stay there safely. All that needs to be saved on interrupt is the immediate working data (the contents of the cpu registers). This can be done very quickly using code that is the same for every process. Similarly, when the interrupt has been handled, the registers can be restored and the program allowed to resume running. From the program's perspective it will be as if nothing had happened.

6.3 Direct memory access

See Comer p 250-251

6.4 Reading

Comer Chapter 13: Input/Output Concepts and Terminology p 207-214
Chapter 15: Programmed and Interrupt-Driven I/O p 237-254

6.5 To be completed

Chapter 7

Power

Aims

Power consumption is often ignored in Computer Architecture courses, but particularly over recent years it has had a major impact on the design of devices and on the ways they are used. This is a short chapter whose aim is to give some insight into why power considerations are important. We approach this from two directions.

The purpose of sections 7.2 and 7.5 is to give a feeling for how much power is used directly first in server computers and data centres, and then in mobile devices. Sections 7.3 and 7.4 give a picture of the implications in terms of cooling and running costs respectively. Section 7.6 presents the current data about the power usage of different areas of IT expressed in terms of data centres, consumer devices and networks.

Learning Objectives

By the end of this section you should:

☐ 1. understand approximately how much power is used by

- ☐ *a*) a major data centre
- ☐ *b*) a desktop PC
- ☐ *c*) a laptop
- ☐ *d*) a mobile phone

and be able to relate that to the power used, say, in a house or in a refrigerator.

☐ 2. understand why power consumption is a significant issue at both ends of the computing scale, and in particular in:

- ☐ *a*) a major data centre
- ☐ *b*) a mobile phone
- ☐ *c*) a battery-driven device

☐ 3. understand how much of the world's electricity is used to run various kinds of IT.

7.1 Introduction

Power consumption has had a major impact on computer design and architecture. It has also had an important impact, both directly and indirectly, on how computers are used. Power considerations are a major reason for companies using specialist cloud providers. At the other end of the scale, they are also a major issue in the design of mobile devices.

We will see that data centres use massive amounts of power, and this impacts their design as it makes sophisticated cooling systems necessary. Moreover, the annual cost of the power consumed in data centres is high to the point of being a significant fraction of the equipment costs of the data centre.

We will also see that IT constitutes a significant and rising amount of the electric power usage globally, with estimates of up to 10%. Estimates also show that this is split roughly evenly between the three headings of data centre, consumer devices, and network usage.

7.2 Data centres: power consumption

The purpose of this section is to (get you to) make a rough calculation of how much power is used in a major data centre. It would be easy simply to give figures, but actually doing the calculations gives a better sense of scale.

You don't need to know much to do this. The basic information is how much power a single computer uses, and then you need some way of estimating how many computers there are in a data centre.

The HP Proliant BL680c shown in chapter 2 has up to 4 cpu's, each rated at 130W. This makes a total of 520W. A typical rack-mounted server is likely to take between 600W and 1000W.

Self Test Exercises: The purpose of this exercise is for you to do some calculations so that you understand the power requirements of large-scale computing.

☐ 1. Using an average figure of 800W as a guide, and an average of 14 computers in a rack, how much power (in KW) is drawn by a single rack of computers?

☐ 2. A corporate server-room contains 20 racks, what is the power drawn by it?

☐ 3. Estimate (very roughly) the amount of power that might be required by a major computer centre, such as those owned by Google or Facebook.

Assume the computers occupy a building the size of an American football field (approximately 110m by 50m), and that one third of the floor space is occupied by racks, each of which takes up about $0.5m^2$. These figures are very approximate and intended only to give a rough guide. Also estimate how many racks and how many computers the centre contains.

☐ 4. Using your electricity bill, calculate your average home use of electricity in kW (one month is about 2.6 million seconds, and three months about 7.8 million, or equivalently, one month is 720 hours, and three months 2160).

☐ 5. The major fuse into a typical house is rated at $80a$. What is the maximum power that can be drawn by such a house?

☐ 6. How many houses at average power does it take to use the same electricity as:

- ☐ *a*) a single rack of computers?
- ☐ *b*) a corporate 20-rack server room?
- ☐ *c*) a major computer centre?

☐ 7. Using the maximum power drawn, calculate roughly what block of houses needs the same power supply as:

- ☐ *a*) a single rack of computers?
- ☐ *b*) a corporate 20-rack server room?
- ☐ *c*) a major computer centre?

Note that this is inaccurate. Planners would assume that the houses in a block were not all drawing maximum power at the same time, but computers in a server centre are all designed to be operating at full power for as much of the time as possible.

7.3 Cooling

All of the power used in a computer ends up as heat, which needs to be removed first from the device, and then in the case of rack-mounted equipment, from the building.

If you look at the picture of a PC in 2.4, you can see a large fan. Beneath it you can just make out a finned piece of metal forming a heatsink, which sits on top of the cpu that is invisible beneath it. This is because this particular cpu runs at about 90W. The metal absorbs the heat that arises from this and the fan keeps the metal heatsink cool. There is another fan that moves air out of the casing and keeps that cool. Without these the temperature of the cpu would build up rapidly just as in the filament of a light. Tower-style desktop PC's draw about 500W, and this ends up heating the room that they are in.

Laptops draw only a fraction of this power, but nevertheless most need a fan to keep the cpu cool (a typical laptop cpu might run at 15W, but even this is enough to get very hot if it is in an enclosed box).

Laptops have benefited from the power-reduction technology developed for mobile phones.

We have seen that a single rack draws about 11kW of power, while a large data centre could use 40MW. To put this into context, a larger central heating radiator delivers about 2kW, so 11kW would comfortably heat a two-bedroom flat, while 20kW would heat a fairly poorly insulated house. So the power used by a data centre would heat a town of about 30,000 dwellings, and all has to be removed from the building housing the centre. The traditional way to do this is to use lots of fans to remove the heat from the machines, and then lots of air conditioning to get the heat out of the building. As a rough guide, the cost of the cooling for a data centre is of the same order of magnitude as the cost of the computers in it, and uses a significant fraction of the total energy budget for the building.

All of this cooling is done by air, and so the heat has to go into making cool air hotter. Let's consider a single 800W computer and suppose there's a 20K difference between the temperature of the air coming in and the air going out. The specific heat of air at constant pressure is 1.005 kJ/kg.K. So we require $(800/(1005 * 20)) \simeq 0.4kg/s$ of air to carry this heat. The density of air is roughly $1.2kg/m^3$, so this amounts to $0.4/1.2 \simeq 0.33m^3/s$ or equivalently $330l/s$ or $1200m^3/hr$. This is about twice the rate of a medium sized kitchen extractor running at maximum speed. A rack produces maybe 14 times this amount of heat, and a large data centre, as we have seen, is producing enough heat to provide central heating for a small town. This all needs to be removed from the building and, ideally, recycled in some way. You also need to make sure that the air on one side of a rack is quite cool, while the air on the other will be quite hot. If the hot air flows back to the cold side of the rack, then it will make the computers harder to cool. Managing all of this is a significant part of data centre design.

Further reading: Google's information about their data centres shows how far they will go to improve their efficiency: `https://www.google.com/about/datacenters/efficiency/` and in particular the information about "How we do it".

7.4 Costs

The Samsung Galaxy S7 Edge has a 3600mAh battery rated at 3.85V. This means it contains 13.9Wh of energy. So if you run it flat and charge it every day, then over a year you use about 5kWh of energy. At current rates this would cost you about 1.25 (using 25p per kWh).

If you have a 500W desktop, and run it day and night over a year, then this uses $0.5 * 24 * 365 = 4380kWh$. Doing this will cost you about 1100 a year, or more than the cost of the computer.

Moving up, each rack in a machine room will use $0.8 * 14 * 24 * 365 = 98,112kWh$ if used at full capacity throughout a year. This is unlikely, but even 50% would give about $50,000kWh$, and cost 12,500. This is a similar order to the cost of the computers, and in this case we also need to factor in additional costs for running the cooling.

When we come to a full-scale data centre, we have $40MW$, so running the centre for an hour would cost 10,000, plus the cost of cooling, assuming that the data centre owners pay the same for their electricity as domestic consumers (which they don't). This means that the annual electricity bill is of the same order of magnitude as the cost of buying the computers. It also means that the owners have a very strong incentive to maximise the use of their computers, and to minimise the use of energy.

Further reading: Google has announced that it will buy all of its energy ($2.6GW = 2,600MW$) from sustainable sources. See `https://environment.google/projects/announcement-100/` for their account of what they are doing and why.

7.5 Mobile phones and other small devices

We saw that the Samsung Galaxy S7 Edge has a 3600mAh battery rated at 3.85V. This is a large battery for a mobile phone, but it contains only $13.9Wh$ of energy.

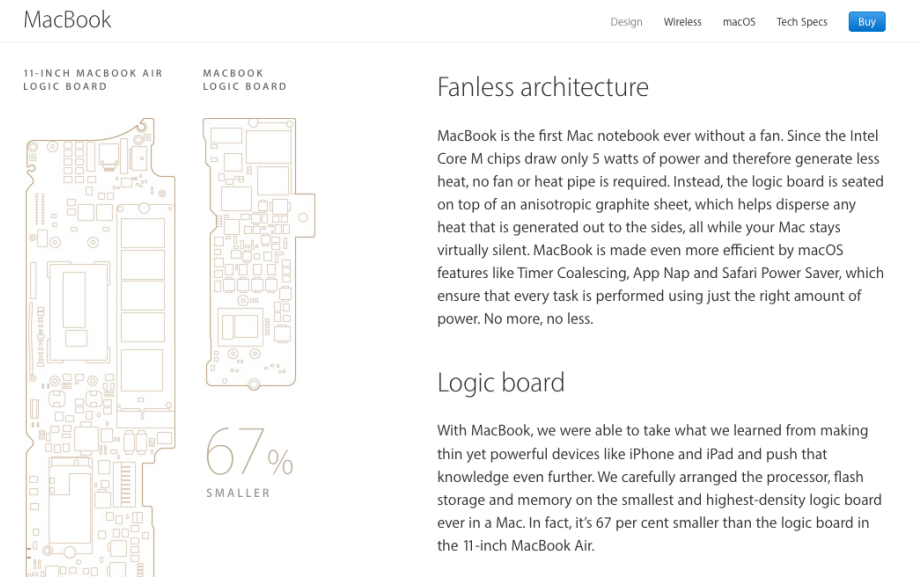
If you want the phone's battery to last through seven hours' use, then the phone can only use $2W$. This contrasts with $500W$ for an old-style desktop, or $60W$ for a laptop. So the mobile phone (or tablet) has to be designed so as to be very stingy with its use of power.

The mobile phone achieves its low-power by:

- using components that are designed to use little power, and in particular can be configured so that their power usage can be controlled
- switching off power-hungry elements (the screen, radios, parts of the cpu) when they are not needed.

in particular the cpu on a mobile phone is different from the cpu's used on a laptop, which are in turn different from the cpu's used for desktops. An Intel i7 desktop cpu uses between 35W and 140W, depending on model. The i7 laptop cpu's go down to 11.5W and the Intel Core M processor used on the Macbook uses only 5W. The Qualcomm SnapDragon uses about 3W. A mobile cpu is designed so that some of the subcomponents (eg some of the cores) can be switched off when they are not needed, so that they do not waste power. These designs are filtering through to laptop design, with the consequence that effective battery life has gone up significantly, and some laptops are being produced that do not need fans.

Apple made this a selling point for their 2016 Macbook, where they are quite explicit about the benefits of the use of technology derived from their tablets and mobile phones:



Further reading: See `http://www.greenbot.com/article/2882307/everything-you-need-to-know-about-the-qualcomm-snap` `html` (or `http://bit.ly/1KM1Mct`) for a discussion of the structure of the Qualcomm Snapdragon cpu's and the kinds of things they do to reduce power consumption.

Power consumption is even more significant for some internet of things devices that depend on batteries as they cannot use mains power. An example would be a battery-powered sensor for a burglar alarm. This has to be active, and to communicate on a wireless network with its controller, but its battery might be a coin cell.

A coin cell (CR2032) of the kind that runs a lot of small devices, has about 200mAh at 3V. This means it contains about 0.6Wh (2160J) of energy. That is about 1/20'th of the power in a mobile phone battery, and it has to power the sensor for months. In order to do this, the sensor uses a low-power microcontroller, switches off its radio as much as it can, and uses particular radio communication protocols adapted for low power devices.

7.6 World-wide power use

We have just seen that data centres use a large amount of power. But even the largest companies, like Google, don't have very many of them. Similarly, our phones, tablets and laptops don't take that much power to keep charged. But there are huge numbers of them. And in between all of these is the communication infrastructure of the internet, including all of the mobile phone masts and other infrastructure. It is far from clear how much energy each of these consumes compared with other areas of human activity.

Published figures differ significantly. But they all agree that each of these various forms of activity consumes a lot of energy.

At the lower end, a paper: "Trends in worldwide ICT energy consumption from 2007 to 2012" estimates that in 2012, ICT amounted to 4.6% of worldwide energy usage. Inside that communication networks used $330TWh$ (that is 330 billion kWh), personal computers (not including mobile phones and tablets) used $307TWh$, and data centres used $268TWh$, of which about half were infrastructure costs associated with power supply and cooling. These are all growing at 10%, 5% and 4%, which is faster than the overall growth rate in electricity consumption of 3%.

A paper from 2013 estimates that the global ICT ecosystem uses $1500TWh$ annually, (which may be an estimate) and notes that this is equivalent to the annual electricity consumption of Japan and Germany combined (this should be accurate, and these are two of the world's largest economies). It also claims that this is 50% more than the energy used in global aviation.

Your mobile devices are a significant culprit in this. Simply charging a device is cheap. But if you watch an hour of video a week, then the network costs of doing this dwarf the energy costs of the actual device. These energy costs vary between 2kWh per GB (at the low end) and 19kWh at the top.

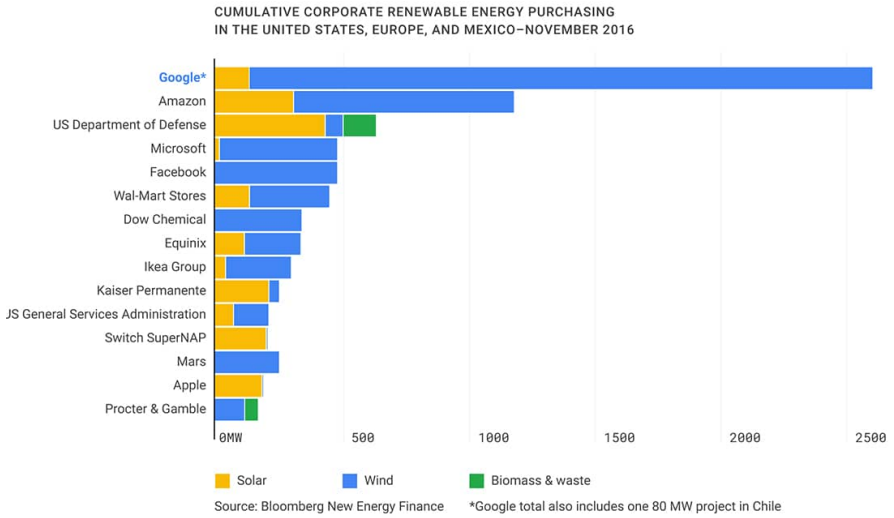
We quote the argument from the end-notes for the paper above, arguing that this energy use is the equivalent of running a new efficient refrigerators:

- a new refrigerator uses $350kWh/yr$ according to EPA Energy Star; plus $5 - -10\%$ to include the embodied energy (manufacturing).
- weekly streaming HD from Netflix is $2.8GB/hr$, and so just under $150GB/yr$. Using the lower end of estimates for network costs this is around $300kWh/yr$ (close to the running cost of a single fridge).
- we also consider the tablet embodied energy, estimated at about $100kWh/yr$.
- Putting these together, the energy cost of a tablet is more than the energy cost of a fridge.

Further reading: The paper used above is: Mills, Mark P. "The cloud begins with coal." Digital Power Group. Online at: `http://www.tech-pundit.com/wp-content/uploads/2013/07/Cloud_Begins_With_Coal.pdf` (2013).

Here is what Google has to say about their use of renewable energy: `http://static.googleusercontent.com/media/cfz.cc/en/us/green/pdfs/renewable-energy.pdf`

And here is their summary of how much renewable energy other bodies are buying (gives some idea of the scale of IT-based companies).



7.7 Self-test solutions

☐ 1. Using an average figure of 800W as a guide, and an average of 14 computers in a rack, how much power (in KW) is drawn by a single rack of computers? **The power consumption is** $800 * 14 = 11200W = 11.2kW \simeq 11kW$**.**

☐ 2. A corporate server-room contains 20 racks, what is the power drawn by it? **The power drawn is** $20 * 11 = 220kW$**.**

☐ 3. Estimate (very roughly) the amount of power that might be required by a major computer centre, such as those owned by Google or Facebook. Assume the computers occupy a building the size of an American football field (approximately 110m by 50m), and that one third of the floor space is occupied by racks, each of which takes up about $0.5m^2$. These figures are very approximate and intended only to give a rough guide. Also estimate how many racks and how many computers the centre contains. **The centre is** $110 * 50 = 5500m^2$**, with** $1/3 * 5500 \simeq 1800m^2$**, occupied by racks, making** $1800/0.5 = 3600$ **racks. This is** $3600 * 14 = 50,400$ **computers. So the power requirement using our assumptions is about** $3600 * 11 = 39600 \simeq 40,000kW = 40MW$**.**

☐ 4. Using your electricity bill, calculate your average home use of electricity in kW (one month is about 2.6 million seconds, and three months about 7.8 million, or equivalently, one month is 720 hours, and three months 2160). **My home use of electricity is about** $300kWh$ **per month. This equates to** $300/720 = 0.42kW$**.**

☐ 5. The major fuse into a typical house is rated at $80a$. What is the maximum

power that can be drawn by such a house? **The power is amps∗volts, or** $80 * 210 = 16800W = 17kW$**.**

☐ 6. How many houses at average power does it take to use the same electricity as:

- ☐ *a*) a single rack of computers? $11/0.4 = 27.5$
- ☐ *b*) a corporate 20-rack server room? $220/0.4 = 550$
- ☐ *c*) a major computer centre? $40,000/0.4 = 100,000$

☐ 7. Using the maximum power drawn, calculate roughly what block of houses needs the same power supply as:

- ☐ *a*) a single rack of computers? $11/17 = 0.65$
- ☐ *b*) a corporate 20-rack server room? $220/17 \simeq 13$**, or a small street.**
- ☐ *c*) a major computer centre? $40,000/17 \simeq 2,353$

Note that the figure for the major computer centre is true but misleading. A better measure might be to look at the output from a mid-range power-station. This is about $600MW$. So the data centre uses $1/15$ of that. An alternative is to return to the average use figure of $100,000$ houses, and deduce that a major data centre uses about the same amount of electricity as a town.

Chapter 8

History of Computer Development

Aims

The aim of this chapter is to give a brief introduction to the history of the development of modern computers. It has two core aims. The first aim is to convey the rapidity of development in the hardware, driven by Moore's Law. The second is to show how that development in hardware has gone hand in hand with development in the capabilities of computer systems, triggering different ways in which we can use computers (and often different companies that support us in our use of those ways).

Learning Objectives

By the end of this chapter you should:

- ☐ 1. have a sense of the overall chronology of the development of computers
- ☐ 2. understand Moore's Law and its impact on computer development
- ☐ 3. understand how the development of computers fits with their increasing availablility and their development as consumer devices
- ☐ 4. understand how the development and availability of hardware has enabled consumers to use computers for an increasing range of purposes

8.1 The earliest computers

Modern computers came out of machines built around the end of the second world war. These were mainly experimental machines built at a range of uni-

versities in the US and the UK (notably the University of Pennsylvania and the University of Manchester).

Early computers used either relays (electro-mechanical devices) as switches, or valves (early electronic). Some of them had strange basic mechanisms, so that recently there have been a couple of announcements that, programmed correctly, they would actually function as general purpose computers. This is expressed by saying that they are **Turing complete**. Their input mechanisms tended to be time-consuming and odd (for example, patching a lot of cables). And occasionally the machines used decimal, rather than binary representations.

These are all largely experimental machines (even if they were used for serious purposes). There was very little idea at the time how many computers might be needed. Many people thought that advanced countries like the UK might have one or two.

1941: Z3, Konrad Zuse, Germany Produced what is now generally regarded as the first modern computer, based on using relays as switches, binary representation and punched film for input. But no mechanism for storing programs and people did not realise it was Turing complete until 1998. The original was destroyed during the war and does not seem to have had much practical use.

1943: Colossus, Tommy Flowers, UK Flowers was a Post Office engineer, brought up in the East End of London. He built Colossus as a special-purpose machine, part of the code-breaking work at Bletchley, to replace the electromechanical "bombes" with something based on valves. Whether or not it was Turing complete is debatable (an individual Colossus is supposed not to be, but the ten machines constructed could have been used as a Turing complete computer).

1943 (operational 1945): ENIAC, Mauchly and Eckert, University of Pennsylvania For a long time this was regarded as the first operational Turing-complete computer. It was built mainly using valves, and was programmed using switches and patch cables. It remained in practical use until 1955 and was upgraded several times. Originally it had no internal memory.

1946: EDVAC, Mauchly and Eckert : a successor to ENIAC and the subject of von Neumann's famous paper. This machine had an internal memory.

1947: Transistor, Bardeen, Brattain and Shockley : used semiconductors to make the first transistor. The team received the Nobel Prize for this in 1956. Getting a Nobel Prize is good. Getting it only nine years after the work was done is incredible.

1948: SSEM, Williams, Kilburn, Toothill, University of Manchester The small-scale experimental machine, nicknamed Baby, was built as a test-bed for the Williams tube, which provided its memory. This was a small machine, but was the first stored program computer. The university turned this design into the larger Manchester Mark 1, which then formed the basis for the Ferranti Mark 1, the world's first commercially available computer (1951).

During the 1950's, more computers were built, and companies like Ferranti (in the UK) and IBM (in the US) started building them to sell them to corporate clients. They began to be used for a much wider range of purposes, but they were still exotic rare beasts.

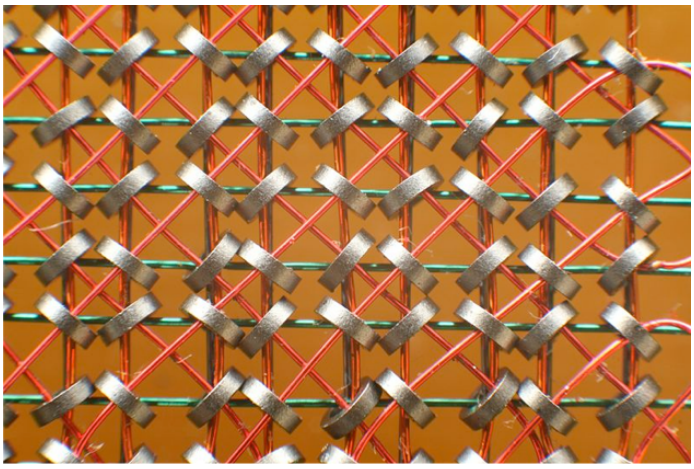
Some sense of how special and rare computers were can be gathered from this film about the installation of Queen Mary's first computer: `https://vimeo.com/77387308`

1960: IBM 1401 This was IBM's first large-run computer, and sales wildly exceeded their expectations. Between 1960 and 1964 they "sold" 10,000 1401's. In fact their business model was to concentrate on larger corporations, rent computers rather than sell them, control everything (including the software) and make lots of money from support/consultancy. Computers became commonplace. In the course of that IBM established itself as the world's leading (but not only) manufacturer.

Summary: The mid to late 1940's saw the development of a series of machines which gradually introduced the features we expect to see in a modern computer. At the time, each of these was a national level resource.

8.2 Semiconductors

1964: IBM360 IBM started marketing a family of mainframe computers (based on transistor technology) to corporate clients. IBM consolidated its position. Main memory was made from hand-woven magnetic core, and long-term storage was (huge) disk packs and magnetic tape.

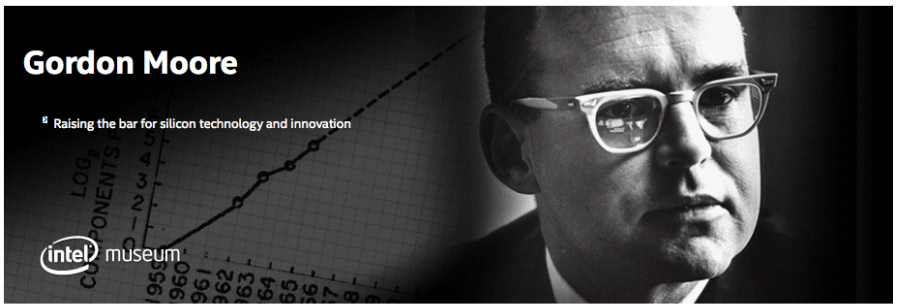


Early 60's: programming languages : Saw the development of high-level programming languages. Fortran and Cobol were both introduced in the late 50's, but Algol and Lisp were early 60's. C came in the early 70's and Java in the 80's.

8.3 Moore's Law

1965: Moore's Law : The number of transistors incorporated in a chip will approximately double every 24 months. (Modern Intel statement)

Moore's Law is extremely important. The development in chip technology that it expresses has largely driven the development of computing, and has certainly driven the availability of computers as capable consumer devices.



Gordon E. Moore Moore was one of the founders of Fairchild semiconductors. He became director of R&D there.

In 1965 Fairchild was a leading US semiconductor manufacturer, distinguished by being able to produce cheap semiconductors through manufacturing innovation. They were starting to produce integrated circuits. He was asked by the IEEE to write a magazine article on where the semiconductor industry was going over the next ten years.

In 1968 he left Fairchild to help found Intel, where he remained as one of the senior figures for the rest of his career.

Moore talked about how many components (transistors) you could sensibly get on a chip. If you only have a few, then the fixed costs of connections and production dominate, and the cost per transistor goes up. But all manufacturing processes are imperfect, and if you try and put as many transistors on a chip as you possibly can, then you end up finding that a lot of your chips have faulty transistors somewhere and have to be rejected. The price goes up both because you are pushing processes to the limit and because you have high failure rates. Somewhere in between these is a sweet spot where your chips can be produced at a minimum cost per transistor. Moore predicted that the industry could improve their processes so that the number of transistors on a "sweet spot chip" would double every year.

Cramming More Components onto Integrated Circuits

GORDON E. MOORE, LIFE FELLOW, IEEE

The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65 000.

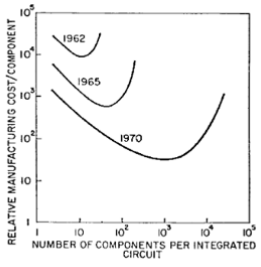
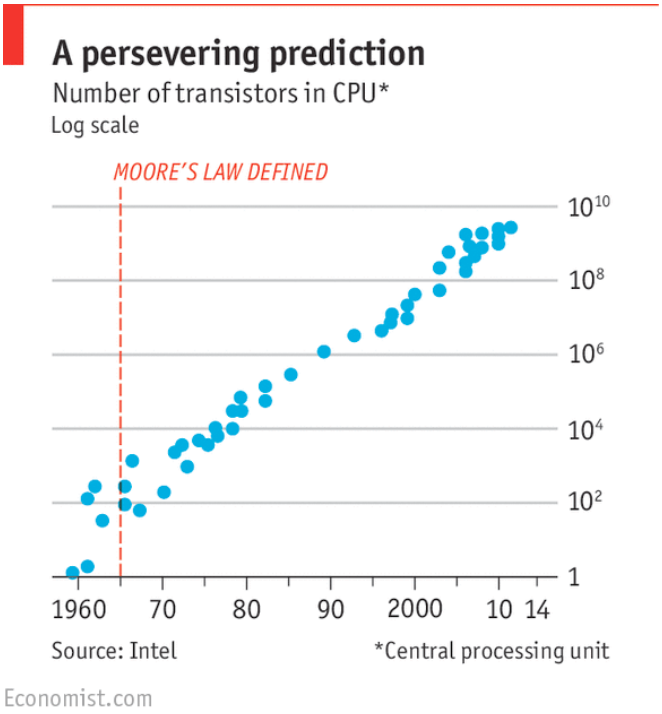


Fig. 1.

This doubling meant an amazingly rapid rise in the number of transistors available to use in a cpu, and was accompanied by exponential shrinking in their size and exponential increases in speed. This in turn meant that microprocessors became much more capable, and could be produced in huge volumes very cheaply. This enabled first the home computer industry, and then the mobile phone industry.

Moore's prediction held for an astonishingly long time, as shown by the number of transistors on cpu's. This graph uses a log scale for the y-axis, so that the straight line corresponds to the exponential growth.



Where we are now

- The exponential increase in clock speeds has stopped. Chips run at or around 2GHz.
- Components continue to shrink, but at a slower rate. We are now down to a feature size of 14nm, which equates to about 70 Silicon atoms. So the end of the road is looming. Nevertheless Intel and the rest of the industry do have a **semiconductor roadmap**. Under this the aim is to continue shrinking to about 10nm.
- The number of transistors on chips continues to grow, though more slowly than Moore predicted (he actually said it would double every year, but Intel now has it as every 24 months). The semiconductor roadmap mentioned above points to different mechanisms for increasing chip densities (such as increasing the number of layers, and making chips more 3-dimensional).

8.4 Microprocessor-based computers

Intel started producing microprocessors in 1971, with an 8-bit processor in 1972 and the 8080 in 1974. This meant that almost anyone could make a computer, and they did. In 1973 Xerox (PARC) produced the first personal workstation, the Alto, changing the computer from a large mainframe living in a special room, watched over by a team of people, to something that was the property of an individual and lived in their office. At about the same time they invented the ethernet to connect these computers together. And rapidly thereafter Xerox also invented the graphical user interface, with windows, a point and click interaction model (1975).

One year later, in 1976, Steve Jobs and Steve Wozniak used cheap microprocessors to produce the Apple I as a hobbyist computer.

Microprocessors were originally used in low-end specialist control applications and hobbyist kits, but as they became more powerful they began to be used in personal workstations whose machines were roughly as powerful as the smaller mainframes on the market. These included lispmachines (from MIT), an obscure computer called the Whitechapel (from Queen Mary) and most famously the workstation range from Sun (founded 1982). Suns were individual workstations running Unix and the X-window system developed at MIT that is still the basis of the windows interfaces on linux.

8.5 Personal Computers

Various companies (Apple, Commodore, Tandy) were producing workstations that could be bought and operated by individual consumers, but in 1981 IBM decided to bite the bullet, and produce a product that would go completely against their business model. They built a small personal computer from largely standard components that they would sell (not rent) to individual customers, and they would release details of the architecture so that others could make copies.

> On August 12, 1981, at a press conference at the Waldorf Astoria ballroom in New York City, Estridge announced the IBM Personal Computer with a price tag of $1,565. Two decades earlier, an IBM computer often cost as much as $9 million and required an air-conditioned quarter-acre of space and a staff of 60 people to keep it fully loaded with instructions. The new IBM PC could not only process information faster than those earlier machines but it could hook up to the home TV set, play games, process text and harbor more words than a fat cookbook.
>
> The $1,565 price bought a system unit, a keyboard and a color/graphics capability. Options included a display, a printer, two diskette drives, extra memory, communications, game adapter and application packages — including one for text processing. The development team referred to their creation as a mini-compact, at a mini-price, with IBM engineering under the hood.

It used an Intel 8088 CPU, and the MS-DOS operating system provided by Microsoft, and it swept the world. It was so successful that people bought PC's and stopped buying IBM mainframes. But people bought clones and IBM

no longer had control of what they were doing. The success of the PC nearly bankrupted the company.

The first PC's were very primitive. While mainframes and Unix-based workstations could **multitask** (i.e. do several jobs at once), early PC's could not. You could not edit a document and print at the same time.

During the 1980's and through into the 90's PC's got more and more powerful cpu's, and more and more memory. They gradually added capabilities, many of which had been possessed by the Unix-based workstations for some time. This can be seen if you look at successive generations of the Microsoft operating system and see what new capabilities it was making available to consumers:

1985: Windows 1.0 Graphical User Interface for MS-DOS. Windows, but they could not overlap. Some multitasking.

1987: Windows 2.0 Better interface. Introduction of strong graphical applications Word and Excel.

1990: Windows 3.0 Introduced virtual memory (mainframe feature) allowing a much larger range of applications to run simultaneously. Required hard drive, not just floppy disks. First multimedia support (play CD's).

1992: Windows 3.1 Scalable fonts, better disk operation, improved multimedia support, serious security flaws.

1994: Windows NT A completely new 32-bit operating system. Better hardware model and better engineering under the hood. More for workstations than consumer PC's.

1995: Windows 95 Brought advances of NT to consumer PC's. Introduction of plug and play for peripherals, some networking.

1998: Windows 98 Integrated networking, bundled Internet Explorer.

8.6 Laptops

People began to attempt to build battery powered computers very soon after the establishment of the microprocessor as a viable basis for a computer. But the first real commercial laptops arrived in about 1990. At that stage the screens and the battery life were awful. Since then improvements have gone hand in hand with the development of mobile phones and other mobile devices, a development that has been most obvious in recent years.

In order to build a successtul laptop you need to cut down the power being used by the cpu (lowering the voltage it runs at). You will want to pack as much as you can into a small space, so for a long time l aptops have used smaller 2.5in disks, rather than 4.5 inch ones as in PC's. And you need to make sure you can get the heat generated out of the device.

In order to get better battery life you need to improve batter technology, but you also need to cut down on the power used by other devices, particularly the more power hungry.

In a laptop built between 1995 and 2005 the power would go into:

cpu: laptops have always used lower power cpu's than desktops, but in recent years manufacturers have worked very hard on this

screen: some power went into running the screen, but more went into the backlighting. A major step change was the change from using small incandescent bulbs to led's.

disk: a lot of power went into mechanical disks, notably starting and stopping the rotation. Modern solid state disks are much more efficient

floppy and optical disks: early laptops had these but they have been replaced by more efficient forms of storage.

In addition, battery technology has developed so that batteries can hold more power, and shrinkage in other components has meant that they could be physically larger.

Putting this together has meant that laptop battery life has risen from about an hour to ten hours.

What we are seeing now is experimentation with hardware forms. In addition to developments of the old-style laptop, we have low-end laptops with what amounts to mobile phone technology, and high-end tablets with detachable keyboards.

8.7 Mobile Phones

Moore's law has also driven the development of mobile phones. Over the nearly thirty years since the first mobile digital handsets phones have become more and more capable, and this has been reflected in changes in the kinds of things we get them to do. Originally, they were just telephones. Then they incorporated cameras, completely destroying the market in basic amateur cameras in the process and forcing camera companies to change their business strategies. Now they are the basic way many people access the internet, not simply a phone, but a personal portal to the digital world (not to mention portable tv and games console).

There are two parts to having a mobile phone: the digital network infrastructure and the handset itself.

The earliest mobile phones were analogue (1G). The first digital network (2G) was launched in Finland in 1991, and many of the most popular phones that used this technology were made by the Finnish company, Nokia. The picture shows a Nokia 9110 from 1998.

By 2002, phones had developed to include basic games and networking capabilities. The first camera phones were introduced. The 2G network had been patched to allow some use of data, but 3G networks were introduced that were basically digital data networks delivered by wireless. In April 2000, the UK government made £22 bn from the auction of licences to run 3G networks. That's about £300 per person in the UK.

On the left is Nokia's iconic 3310 from 2000, an indestructible basic phone owned by everybody. And on the right is the 7650 from 2002, their first phone with a camera and their first phone with a colour screen. The camera had a magnificent 0.3 megapixels.

Phones became smarter, and a Canadian company, RIM, produced a phone called a Blackberry that combined the phone with a personal organiser, email client, and text messaging platform. It became a standard business device.

Blackberry 7210 from 2003

In 2007, Apple introduced the iPhone, the first touchscreen smartphone, and the inspiration for the modern smartphone. It ran a new operating system called iOS, that was radically different from MacOs, in particular in its security model. And it introduced a new ecosystem of apps. This gave consumers a handheld computer, but as a consumer device, not as something you administer as a computer system. Access to internals was locked down, both physically and metaphorically (the filesystem). The hardware was not always world-beating: the camera was not great and, bizarrely for a flagship phone designed to access the internet, it did not get 3G until the second generation.

A device as successful as the iPhone, particularly one built from readily available components, always sparks competing clones. But Apple's operating system was not available to these competitors, and so this produced the commercial drive behind Android, which is now the world's most popular phone operating system. Android was developed by Google. It is based on the Linux kernel, which means that Linux transitioned from first being a port of Unix to IBM PC's, then the operating system kernel of choice for servers running open source software, and finally to the kernel of choice for mobile phones and other consumer devices.

Since then phones have continued to follow much the same model, getting larger and more powerful, and with better internet connections. 4G, introduced by EE in the UK in 2012, is intended to increase the network capacity available to mobile phones, in order to allow internet access comparable to broadband.

8.8 Tablets

In 2010, only three years after the introduction of the iPhone, the computational model was stable enough for Apple to apply it to a whole new category of device. It introduced the iPad, the first popular tablet. A tablet is basically a mobile phone, with a larger screen and battery, but without some of the comms hardware.

8.9 The Future?

The chief function of a section on the future is usually to get it laughably wrong.

What we are seeing now is that computers are splitting into radically different forms of device that share certain forms of technology.

Most computers are consumer devices that are owned and operated by ordinary people who do not want to operate a computer and are not conscious that they are doing so. They want to install software as a kind of add-on, and do not want to spend lots of time understanding how these programs fit together and might communicate. They just want to press a button and send their photo to Instagram. They use mass-produced devices (phones, tablets, low-end laptops),

and have no more desire to change or fix them than they have to do their own car maintenance.

In addition there is an increasing population of devices that most people would not remotely think of as computational, often connected to the Internet of Things.

Quite a lot of development will be devoted to keeping these people safe and secure.

This tendency will increase, and it will be supported by manufacturers using increased chip sizes to put more of the components of these devices onto single chips. In particular where phones now have separate cpu's and RAM chips, we expect those to be merged.

At the same time, we have back-end services provided by vast data centres. These are now providing off the shelf raw computing power.

Btu we are also starting to see the introduction of genuinely new technologies. For example, in 2015 Intel and Micron announced a radical new technology capable of replacing both RAM and Flash memory, 3D XPoint. This depends on storing bits by making physical changes in resistance. It does not use transistors. This first came to market in Spring 201:
. `https://www.theregister.co.uk/2015/07/28/intel_micron_3d_xpoint/`
and
`https://arstechnica.co.uk/information-technology/2017/03/intel-optane-3d-xpoint-details-pri`

8.10 Timelines

Table 8.1: Timeline of computing.

Year	Operating Systems	Networking	Multimedia	Companies	Hardware	Other
Before 1960				Nokia 1865 IBM 1911		
1961						
1962						
1963						
1964					IBM360	
1965						
1966						
1967						
1968				Intel		
1969		Arpanet (DARPA)			RAM (Intel)	
1970						
1971					Intel 4004 microprocessor	
1972	Unix Bell Labs				Intel 8008 microprocessor	
1973		Ethernet (Xerox PARC)				
1974						
1975	MS-DOS			Microsoft		
1976				Apple		
1977	Berkeley Unix					
1978						
1979						
1980						

Year	Operating Systems	Networking	Multimedia	Companies	Hardware	Other
1981	MS-DOS shipped with IBM PCs	SMTP (email): UCLA			IBM PC	
1982			CD introduced	EA Games		
1983		Internet (approx) DARPA				
1984						
1985	Windows 1.0: windows some multitasking					
1986						
1987	Windows 2.0					
1988						
1989		World-Wide Web: CERN				
1990	Windows 3.0 - some network capability			ARM		
1991	Solaris Linux					
1992	Windows 3.1					
1993			MP3			
1994	Windows NT - first 32 bit, first high-end business Windows OS			Amazon Netscape		
1995	Windows 95 real integrated networking	SSL/TLS Security	DVD			

Year	Operating Systems	Networking	Multimedia	Companies	Hardware	Other
1996						
1997						
1998	Windows 98	Digital broadcasting UK		Google		
1999						
2000						
2001	Windows XP break from MS-DOS					
2002			Blu-Ray Apple ipod			
2003			Apple iTunes			
2004	Ubuntu			Facebook		
2005						
2006	Windows Vista			Twitter		
2007	iOS			Dropbox	iPhone	
2008	Android					
2009	Windows 7					
2010					iPad	
2011						
2012	Windows 8 - integrated tablets phones					
2013						
2014						
2015						
2016						
2017						

Part II

Data Representation

Chapter 9

Introduction

9.1 Aims

These are the overall aims of the course, with the aims dealt with by this part picked out in bold.

The aim of this course is to provide students with a basic understanding of how a computer works, how programs are executed by the CPU at the machine level, and how computer networks function. They will gain this firstly by studying the major components of a computer, the interaction between them, and how computers communicate over networks. **Secondly, they will learn how data is represented to be processed by computer.** Thirdly, students will learn some assembly language and understand how high- level programming concepts are related to their machine language implementation. Finally, they will learn how computer networks function, and will understand how low-level network traffic implements communication between computers.

In more detail: The aim of this part of the course is to familiarise you with the way computers represent the data that they are working with. We will cover how numbers and characters are represented in some detail, aiming for you to understand exactly what bit sequences are used. We will then move on briefly to different forms of media, aiming to convince you that the kinds of representation used before can be extended to allow the representation of sound and video. In this context we will also discuss the compression techniques used and their importance.

9.2 Summary

This is the overall summary of the course, with the items dealt with by this part picked out in bold.

The course presents the concepts needed to understand typical computers at the level of their 'machine-code' instruction set, and to understand the basic concepts of computer networks.

The material covered includes

1. the major components of a computer, including CPU, memory, I/O and buses and the role of bandwidth, latency and power dissipation in determining the relationship between them.

2. **the use of bits, bytes and data formats to represent numbers, text and programs**

3. CPU structure and function: the conventional (von Neumann) computer architecture

4. data types, addressing modes and instruction sets

5. machine-level program structure and its correspondence to higher-level programs

6. the role of wired and wireless networks in modern computer systems

7. a basic understanding of typical network technologies, e.g. ethernet, wifi

8. the role of protocols such as ethernet in the implementation and use of network technology

In more detail:

We will cover:

- **Unsigned integers**

- **Signed integers**

- **Floating point numbers**

- **Character sets and text**

- **Media: sound, pictures and video**

Chapter 10

Unsigned integers

Aims

To cover the the basic use of bits to represent data, concentrating on the representation of unsigned whole numbers.

Learning Objectives

By the end of this chapter you should:

- ☐ 1. understand that a bit is a single binary digit, and understand the difference between bits and bytes;
- ☐ 2. understand the correspondence between bit sequences and unsigned integers and in particular:
  - ☐ *a*) be able to translate numbers from decimal to binary and vice versa
  - ☐ *b*) be able to carry out long addition and long multiplication in binary
- ☐ 3. understand the use of hexadecimal to represent bit sequences, and in particular:
  - ☐ *a*) be able to translate between hexadecimal and binary

10.1 Introduction

Computers use binary representations, and at the hardware level these representations are almost always fixed width (i.e. they have a fixed number of binary digits, usually 8, 32 or 64). There are two ways of looking at these representations:

1. as a sequence of binary digits

2. as the binary representation of a number

Example:

1. An IP address is given as a sequence of four numbers: 192.168.0.1 Each number is between 0 and 255, and can be represented as an 8-bit binary sequence. The IP address itself therefore has $4 * 8 = 32$ bits, and can be thought of as a sequence of binary digits in which each group of 8 represents some level of the address.

2. A memory address is given by a fixed length bit sequence, but we think of the memory as being one long sequence of memory words. So it makes sense to think of the address as a number.

This chapter is primarily about how sequences of binary digits represent numbers, how we can convert between binary and decimal and other forms of representation, and how we can adapt familiar algorithms to carry out arithmetic on these binary numbers.

10.2 Types of Numbers

You first meet numbers at primary school, and there you are introduced to different types of numbers in a particular order. First you meet small positive whole numbers. You learn about addition, subtraction and multiplication, and you start to learn addition and multiplication tables. Then you learn about larger numbers and positional notation for numbers bigger than ten. You learn long addition and multiplication. At about the same time you find out about fractions (which we won't study) and negative numbers, and you learn how to add, subtract, multiply and divide. Then you learn about decimals. You learn (again) how to add, subtract, multiply and divide. And you learn that some numbers ($\pi = 3.1415\ldots$) have infinite decimal expansions, and that this is true even of many simple fractions ($1/3 = 0.3333\ldots$). Finally (by which time you'll be at secondary school), you learn that some numbers are simply too large be easily written down as simple decimals, and that you need scientific notation for them (the speed of light is $2.9978 * 10^8 ms^{-1}$ and the mass of an electron is $\ldots$). And you may learn about the square root of -1 and complex numbers.

The numbers used by computers follow a similar pattern, but don't have all the same stages. You may meet some of the same stages in maths courses.

Numbers	Maths name	CS name	Examples
Positive whole numbers	Natural numbers	Unsigned integers	0,1,2,3,...
Positive or negative whole numbers	Integers	Integers	-3,-2,-1,0,1,2,3,...
Fractions	Rationals	no equivalent	$2\frac{3}{4}$
Decimals (scientific notation)	Reals	Floating point	$2.9979 * 10^8$
Complex numbers	Complex numbers	no equivalent	$3 + 4i$

Computers typically do not have all of these supported by hardware and programming languages do not have them as standard types. The ones that are missing are fractions and complex numbers.

Class	Examples	Computer Representation	Java types
unsigned	$0, 1, 2, 3, \ldots$	unsigned binary	
signed	$\ldots -3, -2, -1,$ $0, 1, 2, 3 \ldots$	2's complement	byte, short, int, long
floating point	$-2,80,\ 1.00,\ 3.14$	IEEE floating point	float, double

10.3 Number bases

We count in tens. There is an obvious reason for this: we have ten digits (digit is latin for finger) on our hands (including thumbs). That is a biological, not a mathematical, explanation.

Thought experiment Suppose we had a different number of fingers and thumbs (say eight, which would be one less on each hand, or twelve, which

would be one more). Would we then count in 8's or 12's (answer: yes), and would this make a real difference to us (answer: no).

Mathematicians have studied this idea under the name of **number bases**.

When we write numbers down we are trained to use positional notation: the position that a digit is in controls how much it is actually worth. A digit one place to the left is worth ten times what it was in its original position. This idea is incredibly important. It is critical to our being able to do calculations with large numbers. And it explains why the invention of 0 is so important. Without 0 we would not be able to see all the positions.

1984

1	9	8	4
10^3	10^2	10^1	10^0
1000	100	10	1
1000	900	80	4

In the number 1984 the digit 8 is second from the right, so it is multiplied by 10, and is worth 80. The digit 9 is third from the right, so it is multiplied by 100 and worth 900.

$$1984 = 1000 + 900 + 80 + 4 = 1 * 10^3 + 9 * 10^2 + 8 * 10^1 + 4 * 10^0$$

We can use the same kind of representation mechanism with any number we like, not just 10. This number is called the **base** of the representation.

In Computer Science, the most important base is 2 (**binary**). But we also use base 16 (**hexadecimal** or **hex**).

10.3.1 Converting from binary to decimal

in binary, the powers of 2 play the same role as the powers of 10 in decimal, and the only digits are 0 and 1. This means that a 1 in the fourth position from the right in a binary number is worth $2^3 = 8$ (not $10^3 = 1000$).

position	...	7	6	5	4	3	2	1	0
worth	...	128	64	32	16	8	4	2	1

Conversion method: binary to decimal Example: to convert 1110 0101 to decimal

Write down the digits in a table:

digits	1	1	1	0	0	1	0	1

Insert the powers of 2:

digits	1	1	1	0	0	1	0	1
powers	128	64	32	16	8	4	2	1

Multiply the columns:

digits	1	1	1	0	0	1	0	1
powers	128	64	32	16	8	4	2	1
worth	128	64	32	0	0	4	0	1

Add the results:

digits	1	1	1	0	0	1	0	1
powers	128	64	32	16	8	4	2	1
worth	128	64	32	0	0	4	0	1
sums	229	101	37	5	5	5	1	1

So 1110 0101_2 (the subscript indicates binary), converted to base 10 is 229 (229_{10}).

10.3.2 Converting from decimal to binary

The process can almost be reversed. Taking the same example -
Write out powers of 2 till you reach the largest possible power of 2 that is contained in the number:

sums		229							
powers	256	128	64	32	16	8	4	2	1

Then subtract that power and find the largest power that is contained in the result:

sums		229	101						
powers	256	128	64	32	16	8	4	2	1
sub		101							

And keep on repeating until you get 0:

sums		229	101	37			5		1
powers	256	128	64	32	16	8	4	2	1
diff		101	37	5			1		0

Then put 1 where you have used that power of 2 and 0 otherwise:

sums		229	101	37			5		1
powers	256	128	64	32	16	8	4	2	1
diff		101	37	5			1		0
binary		1	1	1	0	0	1	0	1

The bottom line gives the binary expansion: 1110 0101_2.

Of course you can do this with larger numbers. For example: 1348_{10}

sums		1348		324		68			4			
powers	2048	1024	512	256	128	64	32	16	8	4	2	1
diff		324		68		4				0		
binary		1	0	1	0	1	0	0	0	1	0	0

So $1348_{10} = 101\ 0100\ 0100_2$.

10.4 Other bases

Let b be any positive number greater than 1, then any positive number n can be written uniquely as:

$$n = a_m b^m + a_{m-1} b^{m-1} + \ldots + a_2 b^2 + a_1 b^1 + a_0 b^0$$

where each a_i is between 0 and $m-1$ ($0 \leq a_i \leq m-1$) and $a_m \neq 0$. b is called the **base** of the representation.

Example: In this example the subscript is used to indicate that the expression is written in base 8.

$$100 = 64 + 32 + 4 = 1 * 8^2 + 4 * 8^1 + 4 * 8^0 = 144_8$$

10.4.1 Converting from base 10 to base b

Method 1: Successive division

Construct the base b representation from the right. To calculate the base b representation of the number n:

```
repeat
   divide n by b and let q be the quotient and r the remainder
   write r as current digit
   move left
   replace n by q

   until n = 0
```

Examples:

1. **Convert** 100_{10} **to base** 8**:**
   $n = 100$, $100/8 = 12$ remainder 4, write 4, move left, $n = 12$: current output 4
   $n = 12$, $12/8 = 1$ remainder 4, write 4, move left, $n = 1$: current output 44
   $n = 1$, $1/8 = 0$ remainder 1, write 1, move left, $n = 0$: current output 144
   $n = 0$, stop, output 144. So $100_{10} = 144_8$, as above.

2. **Convert** 100_{10} **to base** 3**:**
   $n = 100$, $100/3 = 33$ remainder 1, write 1, move left, $n = 33$: current output 1
   $n = 33$, $33/3 = 11$ remainder 0, write 0, move left, $n = 11$: current output 01
   $n = 11$, $11/3 = 3$ remainder 2, write 2, move left, $n = 3$: current output 201
   $n = 3$, $3/3 = 1$ remainder 0, write 0, move left, $n = 1$: current output

0201
$n = 1$, $1/3 = 0$ remainder 1, write 1, move left, $n = 0$: current output 10201
$n = 0$, stop, output 10201. So $100_{10} = 1201_3$.
We can check this: $10201_3 = 1 * 3^4 + 2 * 3^2 + 1 * 3^0 = 1 * 81 + 2 * 9 + 1 * 1 = 81 + 18 + 1 = 100$.

This is not a very efficient way of writing this down, and you might want to use:

0	1	3	11	33	100
	1	0	2	0	1

Answer: $100_{10} = 10201_3$.

Examples:

1. **Convert** 123_{10} **to base** 5**:**

0	4	24	123
	4	4	3

Answer: $123_{10} = 443_5$.

2. **Convert** 246_{10} **to base** 7**:**

0	5	35	246
	5	0	1

Answer: $246_{10} = 501_7$.

10.4.2 Converting from base b to base 10

You could, in principle, use the same algorithm as before. But that would involved dividing a number written in base b by 10_{10}, and this is not something we are practised at. So another way is to invert the algorithm above.

Method 2: Successive multiplication

Decompose the base b representation from the left. To calculate the base 10 representation of a number given in base b:
`proceed from left to right: take the first digit of the number, multiply by` b `and add the second digit, multiply again by` b `and add the third, proceed until the end of the number.`

Examples:

1. **Convert** 144_8 **in base** 8 **to base** 10**:** Start with 1,
multiply by 8 and add 4 to get 12,
multiply by 8 and add 4 to get $96 + 4 = 100$

2. **Convert** 10201_3 **in base** 3 **to base** 10**:** Start with 1,
multiply by 3 and add 0 to get 3,
multiply by 3 and add 2 to get 11
multiply by 3 and add 0 to get 33
multiply by 3 and add 1 to get $99 + 1 = 100$

3. **Convert** 443_5 **in base** 5 **to base** 10**:** Start with 4,
   multiply by 5 and add 4 to get 24,
   multiply by 5 and add 3 to get $120 + 3 = 123$

4. **Convert** 501_7 **in base** 7 **to base** 10**:** Start with 5,
   multiply by 7 and add 0 to get 35,
   multiply by 7 and add 1 to get $245 + 1 = 246$

You could also use the arrays we used previously to write these calculations down:

0	4	24	123
	4	4	3

0	5	35	246
	5	0	1

10.5 Hexadecimal (Hex)

From the Computer Science point of view you need to know about two number systems:

binary: base 2

hexadecimal (or **hex** for short): which is base 16.

It is also useful to at least know about a third: **octal**, which is base 8.
Each binary digit is called a **bit**.

10.5.1 Hexadecimal

Hex digits need to run from 0 to 15, so we use a. . . f for 10 . . . 15:

hex	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Since $16 = 2^4$, each hex digit corresponds exactly to four bits. This means that hex is a compact way of representing bit patterns that is both close to the actual pattern and that people can read.

Example: For example, if you look at a Mac address or a WIFI base station ID, you will see something like:

00:24:a8:b7:e2:60

This is a way of writing down a bit pattern using hex. Each hex digit corresponds to four bits as follows:

hex	0	1	2	3	4	5	6	7
decimal	0	1	2	3	4	5	6	7
binary	0000	0001	0010	0011	0100	0101	0110	0111

hex	8	9	a	b	c	d	e	f
decimal	8	9	10	11	12	13	14	15
binary	1000	1001	1010	1011	1100	1101	1110	1111

The hex expression gives a bit pattern in which each hex digit is replaced by the corresponding four bits:

$$\begin{array}{ccccccc} 00 & : & 24 & : & a8 & : \\ 0000\ 0000 & & 0010\ 0100 & & 1010\ 1000 & \\ b7 & : & e2 & : & 60 & \\ 1011\ 0111 & & 1110\ 0010 & & 0110\ 0000 & \end{array}$$

Similarly, colours are sometimes represented by hex string. In Word, choose the text colour "dark purple", and then open "More Colors" to get a dialogue box. Choose RGB sliders and you will see that colours have a hex representation. Dark purple is 660066. This represents a 24-bit pattern, with 8 bits for each of the red green and blue components:

$$\begin{array}{cccccc} 6 & 6 & 0 & 0 & 6 & 6 \\ 0110 & 0110 & 0000 & 0000 & 0110 & 0110 \end{array}$$

Hexadecimal key learning points

- the 15 hexadecimal digits are $0 \ldots 9$ and $a \ldots f$
- because each hex digit represents four bits, hexadecimal is mainly used as a compact human-readable way of writing down binary
- there is a very simple digit by digit conversion algorithm for hex to binary conversion as above
- it can easily be reversed to give a binary to hex conversion algorithm

Self Test Exercises:

☐ 1. Convert the following from hex to binary:

☐ *a*) a9b

☐ *b*) c08

- ☐ *c*) 174
- ☐ *d*) 1100 1101

☐ 2. Convert from binary to hex:

- ☐ *a*) 1110 1001
- ☐ *b*) 0101 1101 0000 1100

Binary Binary is often used as a number representation. Where our standard decimal representation uses powers of ten, binary uses powers of two. You should know and recognise the first of these:

n	0	1	2	3	4	5	6	7	8	9	10
2^n	1	2	4	8	16	32	64	128	256	512	1024

Notice that $2^{10} = 1024$ is very close to 1000 (it is 2.4% out, which is a good approximation).

n	10	20	30	40
2^n	1024	1048576	1073741824	1099511627776

Similarly 2^{20} is close to a million, and 2^{30} is close to a billion, though the errors are increasing (nearly 10% for 2^{40}). Nevertheless this means that decimal prefixes (kilo, mega, giga, and tera) are also used for binary multiples, though incorrectly. There are strict binary analogues: kibi, mebi, gibi, tebi, see `https://en.wikipedia.org/wiki/Binary_prefix` for details.

In decimal, using 3 digits we can write down the numbers from 0 to $999 = 1000 - 1$. Using four it is 0 to $9999 = 10000 - 1$, and using n it is 0 to $99\ldots9 = 10^n - 1$. Similarly in binary, using 3 bits we can write down the numbers 0 to $7 = 8 - 1$, using four bits the numbers 0 to $15 = 16 - 1$, using eight bits the numbers 0 to $255 = 256 - 1$, and using n bits we can write down numbers 0 to $2^n - 1$. This means that with n bits we can produce 2^n different bit-strings to act as labels, and so label 2^n different objects.

Examples:

1. There are 26 characters in the standard English alphabet, each of which can be either upper or lower case (52). There are also 10 digits $0\ldots9$, and perhaps 20 other common characters used in text and for punctuation `".,;:$ ()[]&..."`. This makes a total of something like $52+10+20 = 82$ separate characters. 7 bits would give us a total of 128 bits strings, so these characters can be encoded using 7 bits.

2. The standard IPv4 address is (effectively) a 32-bit bit sequence. This means that there are $2^{32} = 4,294,967,296$ possible IP addresses. Since there are about 8 billion people in the world, only one person in every two can have their own individual IPv4 address.

Logarithms

Logarithms are the inverse operation to exponentiation:

$$m = b^n \iff n = \log_b m$$

in particular:

$$m = 2^n \iff n = \log_2 m$$

For example

$$\begin{array}{rcr} 128 = 2^7 & \text{so} & 7 = log_2 128 \\ 4,294,967,296 = 2^{32} & \text{so} & 32 = log_2(4,294,967,296) \end{array}$$

If you have m objects, then $\log_2 m$ is the number of bits you need to give each of them an individual representation.

10.6 Long addition

Back in primary school you learned how to do long addition.

First you learned your basic addition tables: how to add small numbers together (where here small basically means less than 10, so single digits). Long addition lets you add larger numbers together. Viewed from an algorithmic perspective it is a method for extending single digit addition to the addition of larger numbers represented using place notation. So it works for numbers written in any base. In particular it works for binary.
Recap: $482 + 364 = 846$:

4	8	2		
3	6	4	+	
8	4	6		
1			carries	

The same method works for binary:

1	0	1	0	1	0	1	0	
		1	1	0	0	1	1	+
1	1	0	1	1	1	0	1	
	1				1			carries

A key difference is that the addition table you need is much smaller (two digits given so that carries are explicit):

+	0	1
0	00	01
1	01	10

as opposed to:

+	0	1	2	3	4	5	6	7	8	9
0	00	01	02	03	04	05	06	07	08	09
1	01	02	03	04	05	06	07	08	09	10
2	02	03	04	05	06	07	08	09	10	11
3	03	04	05	06	07	08	09	10	11	12
4	04	05	06	07	08	09	10	11	12	13
5	05	06	07	08	09	10	11	12	13	14
6	06	07	08	09	10	11	12	13	14	15
7	07	08	09	10	11	12	13	14	15	16
8	08	09	10	11	12	13	14	15	16	17
9	09	10	11	12	13	14	15	16	17	18

This makes the implementation on the computer simpler.

10.7 Long multiplication

The same holds true for long multiplication. The same algorithm works, with a simplification in binary.

Once again, you should have learned the method for decimal in primary school:

		2	5	6	
			2	1	*
		2	5	6	
	5	1	2	0	
	5	3	7	6	

The method for binary is similar:

				1	0	1	0	1	0	1	0	
								1	1	0	1	*
				1	0	1	0	1	0	1	0	
		1	0	1	0	1	0	1	0	0	0	
	1	0	1	0	1	0	1	0	0	0	0	
1	0	0	0	1	0	1	0	0	0	1	0	
1	1	1	1	1	1	1	1					carries

Notice that we only ever multiply by:

0: we can simply ignore this

1: from the point of view of the bit pattern, this is just a shift.

So long multiplication in binary can be carried out using the operations of shift left and addition. We do not need to learn a multiplication table.

The example above was fairly simple because we never had a number larger than 1 to carry. But, that can often happen:

12	11	10	9	8	7	6	5	4	3	2	1	columns
				1	1	1	0	1	0	1	1	
								1	1	1	1	*
				1	1	1	0	1	0	1	1	
			1	1	1	0	1	0	1	1	0	
		1	1	1	0	1	0	1	1	0	0	
	1	1	1	0	1	0	1	1	0	0	0	
1	1	0	1	1	1	0	0	0	1	0	1	
		1		1		1	0		1			carries
		1	0			1		1				carries
1	0			1	0							carries

In this example,
The first column contains a single 1, so the sum is 1 carry 0.
The second column contains two 1's and no carries, so the result is 10, written as 0 carry 1. That 1 is in the first row of carries.
The third column contains two 1's and a carry 1, so the result is 11, written as 1 carry 1. That carry 1 is in the fourth column in the second row of carries.
The fourth column contains three 1's and a carry 1, so the result is 100, written as 0 carry 10. That carry is written in the first row of carries, with 0 in the fifth column and 1 in the sixth.
The fifth column contains two 1's and a carry 0, so the result is 10, written as 0 carry 1, with the carry 1 in the sixth column of the second row of carries.
The sixth column contains two 1's and two carry 1's, so the result is 100, written as 0 carry 10, with the carry now written in the seventh and eighth columns of a third row (so that we can distinguish where we are getting carries from). 1 in the sixth column of the second row of carries.
The seventh column contains three 1's and a carry 0, so the result is 11, written as 1 carry 1, with the carry 1 in the eighth column of the first row of carries.
The eighth column contains three 1's and two carry 1's, so the result is 101, written as 1 carry 10, with the carry 10 in the second row of carries.
The ninth column contains three 1's and a carry 0, so the result is 11, written as 1 carry 1, with the carry 1 in the first row of carries.
The tenth column contains two 1's and two carry 1's, so the result is 100, written as 0 carry 10, with the carry 10 in the third row of carries.
The eleventh column contains a single 1 and a carry 0, so the result is 1.
Finally the twelfth column contains no 1's and a single carry 1, so the result is 1.

10.8 Self-test solutions

Self Test Exercises:

☐ 1. Convert the following from hex to binary:

- ☐ *a*) a9b: 1010 1001 1011: straightforward using table
- ☐ *b*) c08: 1100 0000 1000: note the zeroes in the middle
- ☐ *c*) 174: 0001 0111 0100: these are bit patterns so leading zeroes are included
- ☐ *d*) 1100 1101: if you had cd, then you were deceived into converting this from binary to hex. The correct solution is: 0001 0001 0000 0000 0001 0001 0000 0001

☐ 2. Convert from binary to hex:

- ☐ *a*) 1110 1001 : e9: this is conveniently split into groups of four bits, so just look them up in the table.
- ☐ *b*) 0101 1101 0000 1100: 5b0a

Chapter 11

Signed integers

Aims

This chapter covers the standard way in which computers represent signed integers: two's complement.

Learning Objectives

By the end of this chapter you should:

- ☐ 1. understand the two's complement system for the representation of signed integers, and in particular
  - ☐ *a*) be able to translate numbers between binary two's complement and signed decimal and vice versa
  - ☐ *b*) understand why two's complement is preferred to sign and magnitude representation
  - ☐ *c*) understand how to add and multiply numbers in two's complement
  - ☐ *d*) know how to negate twos complement numbers and how to test for positivity.

11.1 Signed integers

Soon after we start doing arithmetic we meet negative numbers. In ordinary decimal we write things like:

$$-345$$

or perhaps when doing accounts:

$$-£15.89$$

We show that a number is negative by writing a minus sign in front of it (thus overloading the minus sign as both a unary and a binary operator). We show that a number is positive, either by putting nothing, or by putting a plus sign.

What this means is that we use a sign and magnitude method for writing down signed integers:

$$\underbrace{-}_{\text{sign}}\ \underbrace{211}_{\text{magnitude}}$$
$$\underbrace{\ }_{\text{sign}}\ \underbrace{1352}_{\text{magnitude}}$$
$$\underbrace{+}_{\text{sign}}\ \underbrace{784}_{\text{magnitude}}$$

11.2 Sign and magnitude

One way to represent signed numbers in binary is simply to copy this.

Computers use fixed width representations for numbers (i.e. the numbers have a fixed number of bits), usually 32 or 64. We will use 8 for examples.

The sign is either + or -, and so can be represented by one bit:

$$\begin{aligned} + &= 0 \\ - &= 1 \end{aligned}$$

In sign and magnitude the first bit is used to represent the sign, and the remaining 7/31/63 are used to represent the magnitude:

$$\begin{array}{cc} & 1001\ 1011 \\ 1 & 001\ 1011 \\ - & 27 \end{array}$$

This means that we can represent numbers from $-2^{n-1}-1$ to $+2^{n-1}-1$

n	min	max
8	-127	+127
32	-2,147,483,647	+2,147,483,647
64	-9,223,372,036,854,775,807	+9,223,372,036,854,775,807

There are, though, problems with this:

- there are two zero's (+0 and -0, 0000 0000 and 1000 0000)
- testing for zero, equality generally, and the order relation is consequently more complicated
- addition and other operations are also significantly more complex (and hence slower).

Specifically, if we want to compute, say, $27 + 5$:

$$0001\ 1011 + 0000\ 0101$$

then we have to add the magnitudes:

$$001\ 1011 + 000\ 0101 = 010\ 0000$$

But if we want to compute $27 + (-5)$,

$$0001\ 1011 + 1000\ 0101$$

then we have to subtract:

$$001\ 1011 - 000\ 0101 = 000\ 0110$$

And we have to work out what sign the result is.

In other words: if we use sign and magnitude, then the signed addition algorithm requires both unsigned addition and unsigned subtraction.

For these reasons, computers use a different representation.

11.3 Two's complement

In n-bit two's complement, the first bit (only) is negated. So the first bit in 8-bit two's complement is worth either 0 or -2^{n-1}:

n	value
8	-128
32	-2,147,483,648
64	-9,223,372,036,854,775,808

Examples:

1. $0001\ 1011 = 27_{10}$, as before.
2. $1001\ 1011 = (-128 + 27)_{10} = -101_{10}$
3. $1000\ 0000 = (-128+0)_{10} = -128_{10}$ is the minimum number representable in 8-bit two's complement
4. $0111\ 1111 = 127_{10}$ is the maximum number representable in 8-bit two's complement
5. $0000\ 0000 = 0_{10}$ is the only zero
6. $1111\ 1111 = (-128 + 127)_{10} = -1_{10}$

This means that in two's complement we can represent numbers between the following minima and maxima:

n	min	max
8	-128	+127
32	-2,147,483,648	+2,147,483,647
64	-9,223,372,036,854,775,808	+9,223,372,036,854,775,807

11.3.1 Converting from two's complement to decimal

We can use essentially the same algorithm as for unsigned, except that the first column contains -2^{n-1}. So for 8-bit two's complement it contains $-2^7 = -128$.

Examples:

1. `11011001` represents -39

digits	1	1	0	1	1	0	0	1
powers	-128	64	32	16	8	4	2	1
worth	-128	64	0	16	8	0	0	1
sums	-128	-64	-64	-48	-40	-40	-40	-39

2. `01001101` represents $+77$

digits	0	1	0	0	1	1	0	1
powers	-128	64	32	16	8	4	2	1
worth	0	64	0	0	8	4	0	1
sums	0	64	64	64	72	76	76	77

11.3.2 Converting from decimal to two's complement

Similarly, essentially the same algorithm can be used for converting from decimal to two's complement as from decimal to unsigned. You just need -128 in the first column.

Examples:

1. -83 is represented by `10101101`.

sums	-83		45		13	5		1
powers	-128	64	32	16	8	4	2	1
diff	45		13		5	1		0
digits	1	0	1	0	1	1	0	1

2. +103 is represented by `01100111`.

sums		83	39			7	3	1
powers	-128	64	32	16	8	4	2	1
diff		39	7			3	1	0
digits	0	1	1	0	0	1	1	1

11.3.3 Addition in two's complement

Two's complement may seem odd, but there is one key reason why this representation system is used:
if two's complement is used, then the same circuitry can be used to implement signed addition/substraction/multiplication as unsigned.

We are now going to look at this in detail for addition.

Take a signed number, x, write down its representation in two's complement and interpret it as an unsigned number, $\bar{x}$.

If $x \geq 0$, then $\bar{x} = x$.

If $x < 0$, then $\bar{x} = x + 256$ (we have interpreted the first bit as $+128$ instead of -128, a difference of 256.

Suppose we take two signed numbers, x and y, write down their representations in two's complement, and add them as unsigned numbers. Finally we take the rightmost 8 bits of the result. We claim that if $x + y$ is within range to be represented in 8-bit two's complement, then this is indeed the 8-bit two's complement representation of $x + y$.

To see this we have to consider various possibilities:

> Both $0 \leq x \leq 127$ and $0 \leq y \leq 127$: then $\bar{x} = x$ and $\bar{y} = y$, and so $\bar{x} + \bar{y} = x + y$. Moreover, $barx + \bar{y} \leq 254$, and so the binary representation has at most eight bits, and is equal to $x + y$ as an unsigned integer. This is fine so long as the result is less than or equal to 127. The only thing that can go wrong is that the result is greater than 127, and in this case $x + y$ is not within range to be representable in 8-bit 2's complement. This problem is called overflow. We can easily detect this overflow because it must be between 128 and 254, and so its leading bit is 1. Thus the addition of two numbers with leading 0's has given us a leading 1, an easy thing to check for.
>
> $0 \leq x \leq 127$, but $-128 \leq y < 0$: then $\bar{x} = x$ and $\bar{y} = y+256$. This addition is safe because $-128 \leq y \leq x + y < x \leq 127$, and so the result is in range to be represented, and moreover the maths says that $\bar{x} + \bar{y} = x + y + 256$. Now, $0 \leq \bar{x} \leq 127$ and $128 \leq \bar{y} \leq 255$, so $128 \leq \bar{x} + \bar{y} \leq 382$, which means $\bar{x} + \bar{y}$ is between 0 1000 0000 and 1 0111 1110 (using 9 bits). Now, if $\bar{x} + \bar{y}$ is between 0 1000 0000 and 0 1111 1111, then the rightmost

eight bits begin with a !. So their value in two's complement is 256 less than their value as unsigned, hence their value in two's complement is $\bar{x} + \bar{y} - 256 = x + y$, as we want. If the value is between 1 0000 0000 and 1 0111 1110, then the value of the leftmost eight bits as an unsigned integer is 256 less than the value of all nine bits (because of the leading 1). Moreover these eight bits begin with 0, and hence represent a positive number in two's complement, equal to the value as an unsigned. Hence once again this represents $\bar{x} + \bar{y} - 256 = x + y$, as we want.

$-128 \leq x < 0$, but $0 \leq y \leq 127$: identical to the previous case with x and y reversed.

Both $-128 \leq x < 0$ and $-128 \leq y < 0$: This case is similar to the first, but more complex. We have $\bar{x} = x + 256$ and similarly for y. Also $128 \leq \bar{x} \leq 255$, and similarly for y. Hence $\bar{x} + \bar{y} = x + y + 512$, and $256 \leq \bar{x} + \bar{y} \leq 510$. This means $\bar{x} + \bar{y}$ is between 1 0000 0000 and 1 1111 1110. If $\bar{x} + \bar{y}$ is between 1 1000 0000 and 1 1111 1110, then taking the rightmost eight bits reduces it by 256 as an unsigned number, and regarding it as two's complement reduces it by a further 256 as it begins with 1. Hence as a two's complement number that represents $\bar{x} + \bar{y} - 512 = x + y$, as we want. If, on the other hand, $\bar{x} + \bar{y}$ is between 1 0000 0000 and 1 0111 1111, then $\bar{x} + \bar{y}$ is between 256 and 383. Since $x + y = \bar{x} + \bar{y} - 512$ it follows that $x + y$ is between -256 and -129, which is out of range for 8-bit two's complement. This problem is easily detectable as before, because we have two negatives (beginning with 1) apparently giving us a positive (beginning with 0).

This argument is quite complicated. Basically it boils down to this:

- the 8-bit 2's complement value of an 8-bit sequence, and the unsigned value of the same sequence differ by a multiple of 256 (either 0 or 1)
- moreover, the unsigned value determines the two's complement value as the unique number in the appropriate range that differs from it by a multiple of 256, and vice versa
- the unsigned value of a bit sequence and the unsigned value of its rightmost eight bits also differ by a multiple of 256
- these properties are preserved by addition, subtraction and multiplication.

What this means is:

in two's complement signed addition requires only unsigned addition.

Similarly:

in two's complement signed multiplication requires only unsigned multiplication.

In fact, in the MIPS instruction set the real difference between signed and unsigned addition (resp. multiplication) is that signed addition checks for overflow problems and unsigned doesn't.

11.3.4 Negation and Subtraction

Subtraction in two's complement can also be implemented using unsigned subtraction, and the argument that shows this is the case is similar to that for addition.

However, it can also be implemented using negation and addition.

$$x - y = x + (-y)$$

Sign and magnitude has a very simple negation procedure: invert the sign bit. But then the addition is more complex.

Two's complement has a more complicated negation procedure, but it is still simple to implement.

To negate a number in two's complement:

- **invert all of the bits**
- **add 1**

"Invert all of the bits" means that where there is a 1 put 0, and where there is a 0 put 1.

Examples:

1. We saw above that `11011001` represents -39. To negate this:
   - invert all of the bits: `00100110`
   - add 1: `00100111`

   You may want to check that this does indeed represent $+39$.
2. Similarly `01001101` represents $+77$. To negate this:
   - invert all of the bits: `10110010`
   - add 1: `10110011`

   Check result:

digits	1	0	1	1	0	0	1	1
powers	-128	64	32	16	8	4	2	1
worth	-128	0	32	16	0	0	2	1
sums	-128	-128	-96	-80	-80	-80	-78	-77

So `10110011` represents -77 as we want.

This is an odd algorithm because it is not obvious why it works. Here is an explanation.

Start with the fact that if you take a number, invert all the bits, and then add the two together you will get `111111111`. This is because you are always adding a 1 and a 0 and never have any carries.

Example:

```
  10010101
  01101010   +
 ----------
  11111111
 ----------
```

Observe that `11111111` represents $2^n - 1$, which in this case is $2^8 - 1 = 255$. So inverting (or flipping) the bits of an 8-bit unsigned number x is a fast way of computing $255 - x$. If we add 1 to that, then we get $256 - x$, or at least the last eight bits.

Now we see how this fits with two's complement. If our two's complement number is positive, then we have computed $256 - x$, which as we saw in the addition argument is the two's complement representation of $-x$ (because that will be negative). And if the two's complement is negative, then it is represented by $256 + x$, and $256 - (256 + x) = -x$, which is what we want. So the algorithm always works.

11.3.5 Ordering and other tests

One of the key properties of two's complement is that there are simple tests for equality and being greater than or equal to 0:

- given two bit sequences `b1` and `b2`, then `b1` and `b2` represent the same number in two's complement iff they are bitwise identical (this is not the case for sign and magnitude, because of the two 0's).
- a bit sequence `b1` represents a number greater than or equal to 0 if and only if its first bit is 0.
- given two bit sequences `b1` and `b2`, then `b1` $<$ `b2` as two's complement numbers if and only if
  - the first bit of `b1` is 1 and the first bit of `b2` is 0
  - OR, this is not the case and the first subsequent bit at which they differ is 0 in `b1` and 1 in `b2`.

11.4 Real World

In the real world computers usually use either 32 or 64 bits to represent integers. 64 bits may be called **long**.

We can do an experiment that illustrates this:

Here is a simple C program:

```
#include <stdio.h>
int main()
{
 int x=1;
  int i=0;
  for (i=0; i<40; i++) {
     printf("2 to the %i is %i\n",i,x);
     x = 2*x;
     }
}
```

This program sets `x` to be 1, and then runs 40 times through a loop. Each time round the loop it doubles `x` and prints the current value. Mathematically this means it will print the values of $2^{\texttt{x}}$.
We can save this to a file, `test1.c`, then compile it:
`gcc -o test1 test1.c`
Run it:
`./test1`

The output is:

```
2 to the 0 is 1
2 to the 1 is 2
2 to the 2 is 4
2 to the 3 is 8
2 to the 4 is 16
2 to the 5 is 32
.....
2 to the 29 is 536870912
2 to the 30 is 1073741824
2 to the 31 is -2147483648
2 to the 32 is 0
2 to the 33 is 0
2 to the 34 is 0
.....
2 to the 38 is 0
2 to the 39 is 0
```

This is unexpected but it fits with a 32-bit two's complement representation. 1 is represented by `0000...0001`. Each time round the loop the 1 is pushed a single place to the left, so that after 32 times, we have `1000...0000`. In two's complement that is a large negative number. The next time round, the 1 gets pushed off end and we get `0000...0000`, representing 0, and that is where we stay.

We can see directly how many bytes the representation uses through a simple program:

```
#include <stdio.h>
```

```
int main (int argc,char* *argv) {
printf("The size of an integer is %lu\n", sizeof(int));
}
```

This shows that integers have 4 bytes, or 32 bits.

We can also see a representation directly, using a slightly tricky program. The function `atoi` converts a string argument to an integer, and the function `fwrite` writes the binary representation to the standard output (which can't cope with it):

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc,char* *argv) {
int a[1];
int i;
// convert argument string to integer and assign to array
a[0] = atoi(argv[1]);
// write binary to stdout
fwrite(a,sizeof(int),1,stdout);
}
```

Since the standard terminal can't display general binary, we need to patch it with another program that takes the binary and displays the individual bits. The standard UNIX utility to do that is `xxd`, which will usually display things as hex. That is useless to us, so we need tell it to use binary.
`./dumpint 3 | xxd -b`
This breaks down as:
`./dumpint 3` – run the program above on input 3
`|` – take the output of that and give it as input to the next program `xxd -b` – take your input and print out 0's and 1's corresponding to the binary.

We get:

```
$ ./dumpint 3 | xxd -b
0000000: 00000011 00000000 00000000 00000000                    ....
```

What this shows is a 32 bit representation. Note the first byte is `00000011` and the rest are all 0. This means that this is a Little Endian representation. That is the most common form.

Similarly we have:

```
$ ./dumpint 63 | xxd -b
0000000: 00111111 00000000 00000000 00000000                    ?...
$ ./dumpint -63 | xxd -b
0000000: 11000001 11111111 11111111 11111111                    ....
```

Reversing the byte order, this corresponds to 63 being represented as `00000000 00000000 00000000 00111111` and -63 being represented as `11111111 11111111 11111111 11000001`.

This is the standard 32-bit two's complement representation.

Chapter 12

Floating Point

Aims

Computers represent real numbers in a way that is heavily influenced by the way that scientists represent them. The aims of this section are to remind you how that works, remind you also of the perils of working to a fixed level of accuracy, and to introduce you to the standard mechanism for representing real numbers on computers, the IEEE floating point standard.

Learning Objectives

By the end of this chapter you should:

☐ 1. have confirmed previous background knowledge:

- ☐ *a*) understand the structure of the standard decimal scientific notation for real numbers
- ☐ *b*) be able to add, subtract, multiply and divide numbers written in standard scientific notation

☐ 2. understand the floating point representation of real numbers:

- ☐ *a*) understand the concept of rounding error and be able to give examples of how calculations to a fixed number of significant figures can include rounding errors
- ☐ *b*) understand the concepts of BigEndian and LittleEndian byte order in representations
- ☐ *c*) understand the way in which IEEE754 floating point broadly replicates standard scientific notation
- ☐ *d*) be familiar with the detail of the fields used in the 32 and 64 bit IEEE floating point representations

12.1 Introduction

So far we have been working with whole numbers (**integers**). But we know that many calculations need numbers that are not whole. In maths at school you first learn fractions, then other decimals. Then in science you learn about scientific notation. Computers generally don't do fractions. They appear to do decimals. But what they actually have is a form of scientific notation.

fractions	$1\ 3/4$	$273/7$
decimals	6.58	$3.14159\ldots$
scientific	$2.9979 * 10^8$	

Scientists use scientific notation because:

- they often need to work with numbers that are either very large or very small
- they can typically only measure quantities to a certain degree of accuracy, and that accuracy is usually a percentage of the number involved.

For example, Avogadro's constant, the number of molecules in an ideal gas at standard pressure and temperature, is

$$6.02214179 * 10^{23}$$

It is much more convenient to write that than

$$6,022,141,790,000,000,000,000,000$$

Similarly the diameter of a silicon atom is

$$2.22 * 10^{-10}$$

which is easier to read than

$$0.0000000000222$$

Of course we could always write this as

$$222pm$$

but then answering questions that involve things at different scales, for example, how many silicon atoms we can get in a centimetre, is harder.

To prove the point that scientific notation is easier to read, check that I got these conversions right.

Self Test Exercises:

☐ 1. Did I actually get these conversions right?

12.2 Standard Scientific Notation

The basic format is:

$$\textbf{sign significand} * 10^{\textbf{exponent}}$$

The **sign** is either '+' or '-' (or omitted and read as '+'). The **significand** is a standard decimal number with a single non-zero digit to the left of the decimal point (so it is between $1.000\ldots$ and $9.999\ldots$). Finally the exponent is a positive or negative integer.

Examples:

1. The speed of light: $2.997930 * 10^8 ms^{-1}$: the sign is '+', the significand is 2.997930 and the exponent is 8.
2. The mass of a hydrogen atom: $1.67x10^{-24}g$: the sign is '+', the significand is 1.67 and the exponent is -24.
3. $-1.385 * 10^{-32}$: the sign is '+-, the significand is 1.385 and the exponent is -32.
4. $13.85 * 10^7$: this is not in standard form, 13.85 is not in the right form, instead of this we should write $1.385 * 10^8$:.
5. $1.385 * 10^{2.4}$:this is not in standard form, 2.4 is not a whole number, working out the equivalent in standard form is complicated.

12.2.1 Multiplying numbers in scientific form

Multiplying numbers in scientific form is easy:

$$(3.21 * 10^5) * (4.15 * 10^3) = 1.33215 * 10^9$$

Multiply the significands: $3.21 * 4.15 = 13.3215$
Add the exponents: $5 + 3 = 8$
Giving: $13.3215 * 10^8$, which is not in standard form, so we normalise to $1.33215 * 10^9$.

12.2.2 Adding numbers in scientific form

Adding numbers is a little harder:

$$(3.21 * 10^5) + (4.15 * 10^3) = 3.2515 * 10^5$$

First we have to restructure the number with the smaller exponent so that both have the same exponent: $4.15 * 10^3 = 0.0415 * 10^5$
Then we can add the significands: $(3.21 * 10^5) + (0.0415 * 10^5) = 3.2515 * 10^5$
Finally, but not in this case, we might need to renormalise again.

Scientific numbers are usually expressed to a certain number of **significant figures**. For simplicity, we can take this as a the number of digits in the significand. But if we have a number that is supposed to have a certain number

of significant figures, and the significand has more, then we will truncate (and round).

12.2.3 Rounding and other errors

In scientific work it is common to have only a fixed degree of accuracy (better would be an error estimate). This equates to working to a number of significant figures. What that means is that the significand has a fixed limited number of digits (e.g. 3).

If we are working to three significant figures, then we may have to **round** a number we are given that has more significant figures so that it only has three.

Mostly this is easy: $1.356 * 10^4$ (four digits in the exponent) rounds to $1.35 * 10^4$, as does $1.3482 * 10^4$. But what about $1.355 * 10^4$? There are a number of possible strategies:

- round up
- round down
- round towards 0
- round away from 0
- round to even

Round to even means rounding so that the last digit is even: $1.355 * 10^4$ rounds to $1.36 * 10^4$, while $1.345 * 10^4$ rounds to $1.34 * 10^4$, and it may be the best strategy. The others all introduce systematic biases.

If we are working to a fixed number of figures, then we usually do not get exactly accurate results from arithmetic operations, we need to round the exact answer. But sometimes our problems are worse.

Suppose we subtract two numbers that are close together, we may not be able produce an answer to similar accuracy. For example

$$1.36 * 10^0 - 1.35 * 10^0 = 0.01 * 10^0 = 1 * 10^{-2}$$

and we have at most one figure of accuracy. Saying that this is $1.00 * 10^{-2}$ would be indefensible (but computers would do it).

Similarly, basic laws of arithmetic no longer hold. For example, in arithmetic $(x + 1) - 1 = x$. But

$$(1*10^{-3}+1*10^0)-(1*10^0) = (0.001*10^0+1*10^0)-(1*10^0) = (1*10^0)-(1*10^0) = 0$$

because of the rounding in order to get a result for the first sum.

What this means is that algorithms that use floating point numbers need to be carefully structured in order to avoid gradually accumulating errors that can end up seriously impacting the result. Simply using double rather than float will not do.

12.3 IEEE754

It is important that we can run computer programs on different machines and get the same results. That will not happen if the machines do something as basic as represent floating point numbers in different incompatible ways. As a result, the IEEE, on of the leading Engineering Societies sponsors standards for computing equipment. One of them, IEEE754 is about how computers represent floating point numbers.

Its abstract explains: "This standard specifies interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. This standard specifies exception conditions and their default handling. An implementation of a floating-point system conforming to this standard may be realized entirely in software, entirely in hardware, or in any combination of software and hardware. For operations specified in the normative part of this standard, numerical results and exceptions are uniquely determined by the values of the input data, sequence of operations, and destination formats, all under user control."

The standard defines three binary formats, for 32, 64 and 128 bits. This is described in general in section 3.4 of the standard, and the particular parameters are given in Table 3.2. We will concentrate on 64, the size of a **double** in C (floats have 32 bits).

The basic format is:

S	E	T
Sign	Exponent	Significand
1 bit	$w = 11$ bits	$t = p - 1 = 52$ bits

These basic components correspond to standard scientific notation (albeit in binary not decimal). But the computer representation uses some tricks.

- The exponent represents an unsigned integer. Since this is 11 bits, that is $0 \ldots 2047$. So that we can get negative exponents a constant bias of 1023 is subtracted from this. Moreover, the exponent fields 0 and 2047 are treated specially. That means we can exponents of $-1022 \ldots 1023$.
- Since we are using binary, the only non-zero digit is 1. This means the leading digit in any expression in standard scientific form is 1, and we don't have to store it. So the bit sequence $b_1 b_2 \ldots b_{52}$ is used to represent $1.b_1 b_2 \ldots b_{52}$ (using the binary here, not decimal).

Examples:

1. `0100 0000 0001 1000 0000 ...0000` has:

   sign: `0`, representing positive

   exponent: `100 0000 0001` representing $(1024 + 1) - 1023 = 2$

significand: `1000 0000 ...0000` representing $1.1000\ldots$, which in decimal is 1.5.

As a result this represents $(+)1.5 * 2^2 = 1.5 * 4 = 6$.

2. `1011 1111 1111 1110 0000 ...0000` has:

   sign: `1`, representing negative

   exponent: `011 1111 1111` representing $(1023) - 1023 = 0$

   significand: `1110 0000 ...0000` representing $1.1110\ldots$, which in decimal is $15/8 = 1.875$.

   As a result this represents $(-)1.875 * 2^0 = -1.875$.

In addition:

if the exponent E=0 and the significand field T=0: then this represents +0 or -0, depending on S

if the exponent E=0 and the significand field T is non-zero: if T is $b_1 b_2 \ldots b_{52}$ then this represents $(-1)^S * 2^{-1023} * 0.b_1 b_2 \ldots b_{52}$

if the exponent E=2047 and the significand field T=0: then this represents $+\infty$ or $-\infty$

if the exponent E=2047 and the significand field T is non-zero: then this represents something that is not a number (a NaN).

12.3.1 Floats and doubles

I'm using float as a synonym for 32 bit and double as a synonym for 64 bit. We've just looked at doubles. Floats follow a similar structure/pattern.

Property	Float	Double
Exponent bits (E)	8	11
Significand bits (T)	23	52
Bias	127	1023
Max exponent	127	1023
Decimal Sig Figs	7	16

12.4 Real world

When it comes to the real world we have the same big and little endian issues as before with integers. We can use similar C programs to find the sizes of the representations (number of bytes), and we can also use a similar program to print binary to standard output, where we can pipe it through `xxd`. These allow us to check that the representations coming out correspond to the account above.

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc,char* *argv) {
double a[1];
int i;
// convert argument string to double and assign to array
a[0] = atof(argv[1]);
// write binary to stdout
fwrite(a,sizeof(double),1,stdout);
```

```
$ gcc -o dumpdouble dumpdouble.c
$ ./dumpdouble 6 | xxd -b
0000000: 00000000 00000000 00000000 00000000 00000000 00000000  ......
0000006: 00011000 01000000                                      .@
$ ./dumpdouble -1.875 | xxd -b
0000000: 00000000 00000000 00000000 00000000 00000000 00000000  ......
0000006: 11111110 10111111                                      ..
```

Reverse the byte order here to compare these results with the ones above:

```
00000000 00000000 00000000 00000000 00000000 00000000 00011000 01000000
```

becomes

```
01000000 00011000 00000000 00000000 00000000 00000000 00000000 00000000
```

and

```
00000000 00000000 00000000 00000000 00000000 00000000 11111110 10111111
```

becomes

```
10111111 11111110 00000000 00000000 00000000 00000000 00000000 00000000
```

These are as we want.

Chapter 13

Text representation

Aims

The basic aim of this section is to lay out how text is represented on computers. In fact, since text can be regarded as a sequence of characters we concentrate on character representation, covering the traditional ASCII format and the now standard Unicode.

Learning Objectives

By the end of this chapter you should:

☐ 1. Understand the representation of characters and text:

- ☐ *a*) understand that characters are represented in computers by coding them as numbers/bit patterns
- ☐ *b*) understand the original ASCII representation of characters, the number of bits used to represent a character, the way it uses sequences, the distinction between upper and lower case, and the limitations of the character set
- ☐ *c*) have seen instances of 8-bit extensions of ASCII
- ☐ *d*) have been introduced to unicode, and understand the distinction between the code point of a character and its representation as a bit pattern
- ☐ *e*) understand that UTF-8, UTF-16 and UTF-32 are distinct ways of representing unicode characters
- ☐ *f*) understand that UTF-8 is a variable length representation and be able to explain the link between certain example characters and their representations.

13.1 Text

When you create a document on a word processor such as Word, it contains a lot of formatting information, such as the font, the type size, its colour, and the like. If you then save the document in text format (in Word, choose "Save As" from the file menu and then select"Plain text (.txt)" as format), then all of this information disappears and what you are left with is basically the sequence of characters that you typed in. This is what Computer Scientists mean by text. You will have seen in Procedural Programming that computer programs are written as text files, often with a much simpler text editor like Gedit.

These text files are basically sequences of characters. The computer knows when a file finishes because it encounters a special end of file character (Ctrl-D on Unix).

On Unix you can create a text file even without an editor by typing into a terminal window:

```
 $ cat > MyNewTextFile
 abcdefg
 hijklmnop
^D
 $
```

creates a new text file containing two lines of text. The end signal is a Ctrl-D.

In a file like this, each character is represented as a bit pattern, where the bit pattern is given by some encoding system.

13.2 ASCII

The first and archetypal encoding system is ASCII. In its original form it used seven bits to represent each character, but any modern system will use eight in practice. The basic idea is simple:
each character gets a number between 0 and 127, that number is represented as an 8-bit unsigned integer.

There is a very comprehensive article on ASCII on wikipedia:
https://en.wikipedia.org/wiki/ASCII

Here is a table of the values:

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

Things to note:

- The character set includes both upper case and lower case letters.
- Uppercase begins at 65, which is hex 41, and lower case begins at 97, which is hex 61. Moreover the characters are sequential in alphabetical order. So the difference between an upper case character and its corresponding lower case is 32, which is precisely one bit. Moreover the printable letters start close to the start of the upper half of the range.
- Digits also start at a boundary 0 is 48 which is hex 30, and are also sequential.
- The character set is based on codes that might need to be sent to an old-style teleprinter or line- printer (it was originally developed in the early 1960's). It includes codes that are intended to control such a printer and not just produce printable characters. For example it includes a code to sound a bell.
- It also includes both a carriage return and a linefeed. On an early printer, the linefeed would move the paper up, while the return would move the printing head back to the start of the line.
- ASCII stands for American Standard Code for Information Interchange, and it is a US-centric design It uses the US alphabet, and has a dollar sign ($) but not other currency symbols like a pound sign.

13.3 Unicode

One of the key problems with ASCII (apart from its strange range of unprintable characters) is that it is unashamedly US-centric. But we do need to communicate in languages other than English, and ASCII does not allow us to use their alphabets. As a result other character sets were developed, mainly using eight bits. But with eight bits it is only possible to represent 256 characters, and this is not enough:

- for documents that contain multiple languages with incompatible alphabets
- for documents that contain huge numbers of symbols (eg maths)
- for some languages that have a vast range of characters (eg Chinese).

As a result, the Unicode system was developed. This is the standard system used by Java, and on the World-Wide Web. Unicode attempts to encode all of the characters used in all of the languages in the world, plus punctuation marks and other symbols. Its coverage is immense, but still not complete. And it is also not completely consistent. For example the criteria of some form of compatibility with previously used character sets, and the avoidance of having different representations of the same character can come into conflict. There are also issues where characters can be seen as being built from components, and where apparently the same character occurs in different alphabets.

The basic idea is that the character set of a group of languages is encoded as a block. Each character is given a distinct number, and the numbers corresponding to the characters in the block are contiguous. The number corresponding to a given character is called its **code point**. Unicode 10.0 (June 2017) has 1,114,272 code points, of which 276,377 are actually allocated. The standard way to refer to a unicode code point is to use four hex digits preceded by "U+": U+0041 is the code point for "LATIN CAPITAL LETTER A".

Examples:

1. Code points U+0000 to U+007F ($0 \ldots 127$) are allocated to the same characters as in ASCII
2. Code points U+0000 to U+00FF ($0 \ldots 255$) are allocated to the same characters as in a character set called iso-latin-1 (which includes a number of letters with accents in the range $128 \ldots 255$). Code point U+00C2 is "LATIN CAPITAL LETTER A WITH CIRCUMFLEX".
3. Code points U+0400 to U+04FF are Cyrillic (Russian and other languages).
4. Code points U+0600 to U+06FF are Arabic with some additions so that this will cope with some variants, such as Farsi. Other variants require characters from different blocks.
5. Code points U+12000 to U+123FF are ancient Babylonian cuneiform.

13.3.1 Unicode character encodings

In ASCII the code numbers were used to give an immediate bit pattern, but in Unicode there are several ways in which the code points can be translated into the bit sequences that represent the characters. The simplest is UTF-32, which uses 32 bits per character and is a simple direct enccoding of the code point. UTF-16 is based on 16-bit code units. It uses a single 16-bit unit to encode the most common characters and two units to encode less common characters. Java uses UTF-16. UTF-8 is the recommended encoding for the web, and it uses 8-bit code units. The ASCII character set is encoded with a single 8-bit unit. This means that ASCII text files are also Unicode. Less common characters use 2, 3 or 4 code blocks.

UTF-8

The following table (from www.unicode.org) shows how UTF-8 splits up code points from different blocks:

Table 3-7. Well-Formed UTF-8 Byte Sequences

Code Points	First Byte	Second Byte	Third Byte	Fourth Byte
U+0000..U+007F	00..7F			
U+0080..U+07FF	C2..DF	80..BF		
U+0800..U+0FFF	E0	***A0***..BF	80..BF	
U+1000..U+CFFF	E1..EC	80..BF	80..BF	
U+D000..U+D7FF	ED	80..***9F***	80..BF	
U+E000..U+FFFF	EE..EF	80..BF	80..BF	
U+10000..U+3FFFF	F0	***90***..BF	80..BF	80..BF
U+40000..U+FFFFF	F1..F3	80..BF	80..BF	80..BF
U+100000..U+10FFFF	F4	80..***8F***	80..BF	80..BF

Since the encoding has variable length it is critical to be able to recognise when a single character begins and ends.

The continuation blocks are all 80...BF, or in a few cases part of that. These are bytes that start `10....` Starting blocks are either 00...7F (exactly the bytes starting `0...`, or they are contained in C0...F4, bytes starting `11....`

As a result there is the following simple characterization of bytes:

starts `0`	ASCII character	always valid
starts `10`	continuation block	sometimes invalid
starts `11`	start block	sometimes invalid

The following table (also from www.unicode.org) indicates how the actual bit patterns are decided (these details are beyond the scope of the course but included to assist in understanding the general concept).

Table 3-6. UTF-8 Bit Distribution

Scalar Value	First Byte	Second Byte	Third Byte	Fourth Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Examples:

1. U+004D (Latin capital M) has scalar value `00000000 01001101`, is ASCII character 4D, and is represented in UTF-8 by 4D=`01001101`.

2. U+00C2 (LATIN CAPITAL LETTER A WITH CIRCUMFLEX) is from the iso-latin-1 character set, has scalar value `00000000 11000010`, matches row 2 of the table, and so is represented in UTF-8 by `11000011 10000010` (`yyyyy=00011` and `xxxxxx=000010`)

3. U+0627 (ARABIC LETTER ALEF) has scalar value `00000110 00100111`. It also matches row 2 of the table. `yyyyy=11000` and `xxxxxx=100111` and so it is represented by `11011000 10100111`.

4. U+3025 (HANGZHOU NUMERAL FIVE) has scalar value `00110000 00100101` matches row 3 of the table and so is represented by 3 bytes. `zzzz=0011 yyyyyy=000000` and `zzzzzz=100101`. So it is represented by `11100011 10000000 10100101`.

UTF-16 and UTF-32

We will not discuss the UTF-16 and UTF-32 encodings other than to say that in these cases big and little endian issues arise. The UTF encodings work as a sequence of code units. There is no big or little endian issue about the ordering of that sequence. But in the case of UTF-16 each code unit contains 2 bytes, and in the case of UTF-32 each code unit contains 4 bytes. There are big and little endian orderings of the bytes inside each code unit, and hence big and little endian variants of UTF-16 and UTF-32, but not UTF-8.

13.4 Real world

We create a small text file and then dump the contents as binary, and as hex. As expected we see the ASCII character codes.

```
 $ cat > MyNewTextFile
 abcdefg
 hijklmnop
 ^D
 $ xxd -b MyNewTextFile
0000000: 01100001 01100010 01100011 01100100 01100101 01100110  abcdef
```

```
0000006: 01100111 00001010 01101000 01101001 01101010 01101011  g.hijk
000000c: 01101100 01101101 01101110 01101111 01110000 00001010  lmnop.
0000000: 6162 6364 6566 670a 6869 6a6b 6c6d 6e6f  abcdefg.hijklmno
0000010: 700a                                     p.
```