# CS1Q Computer Systems
# Lecture 11

(Part two of the notes)

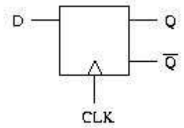
Prof. Chris Johnson

S141 Lilybank Gardens, Department of Computing Science, University of Glasgow, Scotland.
johnson@dcs.gla.ac.uk, http://www.dcs.gla.ac.uk/~johnson

Notes prepared by Dr Simon Gay

## The D FlipFlop

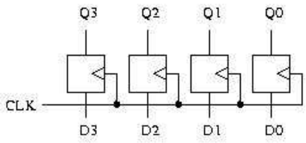
A 1-bit register is called a *D flipflop*. When considering the D flipflop as an individual component, it is common to make both the output *Q* and its inverse $\overline{Q}$ available. Here it is (without the Reset input, which we do not need for the next few examples).






## The D FlipFlop

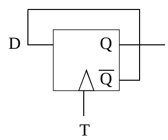
A register can be viewed as a collection of D flipflops. Here is a 4 bit register (without the Reset input). Note that all 4 flipflops are connected to the same clock input.






## The T FlipFlop

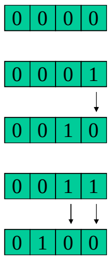
If the $\overline{Q}$ output of a D flipflop is connected to the D input, the resulting circuit changes state (from 0 to 1, or from 1 to 0) at every clock tick. This is a T (for *trigger*) flipflop.




This is an example of the general structure of a sequential circuit. There is one bit of memory, and at each clock cycle, the memory is updated by storing the negated output from the previous cycle.

# A Binary Counter

When counting in binary, each bit changes its value when the bit to its right changes from 1 to 0 (i.e. at a negative edge).



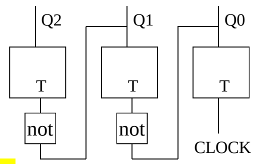

a negative edge causes the bit on the left to change



# A Binary Counter

T flipflops can be used to build a circuit which counts in binary, advancing by 1 on each clock cycle. Assuming that the flipflops are positive edge triggered, we need to convert a negative edge on an output into a positive edge on the next clock input.

a 3 bit counter




the same idea works for any number of bits



# A Binary Counter

This design is sometimes called a *ripple counter* because of the way that the change in output propagates from bit to bit.

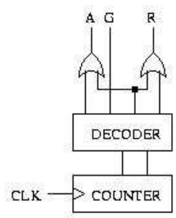
When the outputs are 111, the next clock cycle changes the state to 000.

The design can be extended to any number of bits.



# A Counter Application

A 2 bit counter can be used to complete the design of a traffic light controller. The counter generates the binary numbers from 0 to 3 in sequence, and the circuit from Lecture 9 converts these numbers into the correct outputs for the Red, Amber and Green lights.




Exercise: Why is this not quite a perfect traffic light controller?

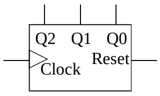
# The Prime Number Machine Again

Recall our design of the circuit which outputs the sequence 2, 3, 5, 7, 11, 13 as 4 bit binary numbers.

The design works but has a couple of undesirable features:

- a 4 bit register is used, but there are only 6 states and therefore only 3 bits of storage should be necessary
- it seems to be a fluke that the "Reset problem" can be solved so easily; this technique might be hard to generalise

The idea for an alternative design is to assume that we have a 3 bit counter as a standard component. It has a clock input and an asynchronous reset input.






# PNM Second Design

Assuming that we make use of states 0 to 5 (i.e. 000 to 101) of the counter, we need a circuit to calculate the outputs P3,P2,P1,P0 according to this truth table:

| Q2 | Q1 | Q0 | P3 | P2 | P1 | P0 |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | X | X | X | X |
| 1 | 1 | 1 | X | X | X | X |

Easy to design the circuit for P3,P2,P1,P0 (exercise).



# PNM Second Design

States 110 and 111 are not needed. We would like the counter to advance from state 101 to state 000.

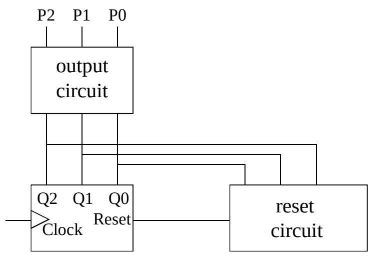
This can be achieved by connecting the Reset input to a small circuit which will activate Reset when the state reaches 110.

If Reset is active high, then we can use $R = Q_2 Q_1 \overline{Q_0}$

If Reset is active low, then we need $R = \overline{Q_2} + \overline{Q_1} + Q_0$



# PNM Second Design




Exercise: complete the circuit.

# Implementing Memory/Registers

With current integrated circuit technology, memory can be implemented by taking advantage of *capacitance* - the tendency for electric charge to persist in part of a circuit. The electric current representing a 1 input to a register causes a tiny electric charge to build up in the circuit, and this charge will remain for a short time.

If the contents of the register are only needed for the next clock cycle, and the clock speed is sufficiently high, then nothing more needs to be done. Registers (known as *latches*) which are used for temporary storage, have this property.

In the case of a CPU's general registers, and the RAM of a computer, the contents must be *refreshed* in order to preserve them for longer periods.



# Implementing Memory/Registers

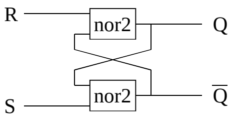
It is also interesting that memory can be built from the purely logical properties of the basic gates that we are familiar with.

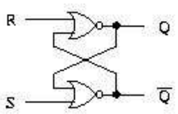
We will now look at the step-by-step development of the D flipflop from a circuit called the RS flipflop, which just consists of two NOR gates.



# The RS FlipFlop

Using two NOR gates we can build a circuit which is able to store one bit of information: either 0 or 1.




Note the essential use of feedback (connections from outputs to inputs), which takes us beyond combinational circuits.



# The RS FlipFlop: Summary

The RS flipflop has two stable states:

- $Q = 0$, $\overline{Q} = 1$
- $Q = 1$, $\overline{Q} = 0$

If R (reset) becomes 1 then Q becomes 0 and $\overline{Q}$ becomes 1, and this state is maintained when R returns to 0.

If S (set) becomes 1 then Q becomes 1 and $\overline{Q}$ becomes 0, and this state is maintained when S returns to 0.

If R and S become 1 simultaneously then the behaviour of the circuit is not defined (in practice, one of R and S would remain at 1 for slightly longer than the other, and the last one to change would determine the subsequent state of the flipflop).

## The RS FlipFlop

The RS flipflop is a basic unit of memory: it can store one bit, and the stored bit can be changed by means of the R and S inputs.

Some refinements are needed before the RS flipflop becomes a useful component for digital circuits.

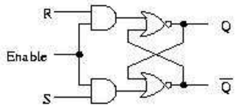
The first issue is that it is an asynchronous component: the R and S inputs can be used at any time. As we are interested in synchronous circuits, we need to introduce a clock, in such a way that setting R or S causes the state to change *at the next clock cycle*, rather than immediately.

The first step is to introduce an *enable* input.

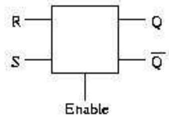


## The Gated RS FlipFlop

A *gated* or *enabled* RS flipflop only responds to its inputs when the Enable input is 1. This is done by passing the inputs through AND gates.




Diagrammatically:

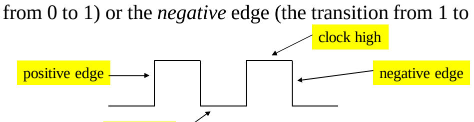





## The Clocked RS FlipFlop

If the clock is connected to the Enable input of a gated RS flipflop, the result is a flipflop which responds to its inputs when the clock is high (has value 1) and ignores its inputs when the clock is low (has value 0). A sequential circuit built in this way can only change state while the clock is high.

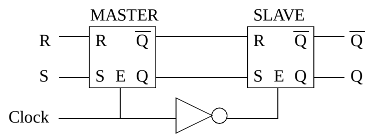
This is a good start, but there are a number of engineering reasons why it is preferable for state changes to occur at a clock *edge*. A circuit can be designed to use either the *positive* edge (the transition from 0 to 1) or the *negative* edge (the transition from 1 to 0).






## The Master-Slave Circuit

Two gated RS flipflops connected together:



## The Master-Slave Circuit: Summary

When the clock is 1, the master flipflop is enabled and responds to its inputs. It can therefore be set or reset by the S and R inputs. The slave flipflop is not enabled, because it receives the inverted clock, which is 0. Therefore the outputs of the circuit, corresponding to the value stored in the slave flipflop and are not affected by the S and R inputs.

When the clock changes to 0, the master flipflop is no longer enabled, so its stored value is fixed according to whether the S or R input was the last one to have value 1. The slave flipflop is now enabled, so the outputs of the master flipflop set or reset the value stored in the slave.

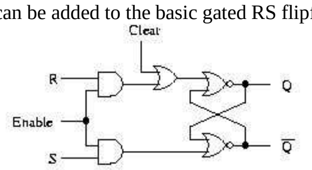
Therefore as the clock changes from 1 to 0, the value represented by the inputs to the master flipflop is transferred to the slave flipflop. Once the slave has received this value, its outputs are fixed until the next time the clock changes from 1 to 0. This circuit is a negative edge-triggered RS flipflop.



## RS FlipFlop with Clear

It is useful to be able to reset a flipflop to 0 at any time, even when it is not enabled. (For example, because an electronic flipflop enters a random state when power is first applied, and needs to be initialised.)

A Clear input can be added to the basic gated RS flipflop circuit like this:

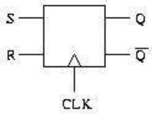



A master-slave circuit built from two flipflops with Clear, by connecting the Clear inputs together, gives a clocked flipflop with an asynchronous Clear input.



## Positive or Negative?

We will use the following diagram for a clocked RS flipflop.




When studying synchronous circuits, we don't care whether components are triggered by the positive or negative clock edge; actually, we can even forget about whether they are edge-triggered or level-triggered. All we need to know is that the clock "ticks" regularly, and a state change is possible at each tick.



## CS1Q Computer Systems Lecture 12

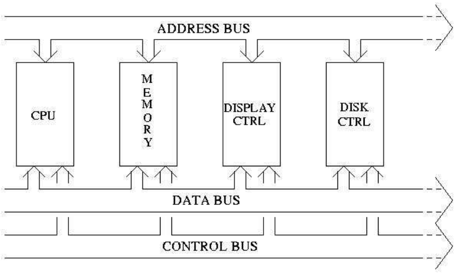
# Where we are

Global computing: the Internet
Networks and distributed computing
Application on a single computer
Operating System
Architecture
Digital Logic
Electronics
Physics

putting together the pieces

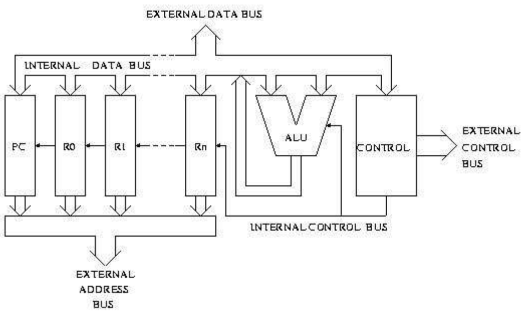


# Structure of a Computer






# Structure of a CPU






# Implementation of the CPU

We will look at how the components of the CPU can be implemented using the techniques of digital logic which we have been studying.

We will look at the details of how to implement a subset of the ITM instructions: the *register-register* instructions. We will then discuss the issues involved in extending this implementation to the full instruction set.

Real CPUs use essentially the same ideas, but developed to a much more sophisticated level.

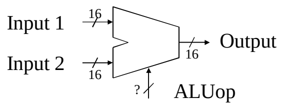
# The ALU

We have already seen (Lecture 9 Slide 12) how to construct an ALU which offers a choice between several operations.

The ALU for the IT Machine must offer the following operations: addition, subtraction, negation, multiplication, comparison (3 kinds).

Exercise: how many bits are needed for the control input, ALUop?.

Its data inputs and output must be 16 bits wide.






# The Registers

We have seen (Lecture 11 Slide 3) that a register can be built from D flipflops. The IT Machine has 16 registers and each register is 16 bits wide. The registers are organised into a structure called the *register file*.

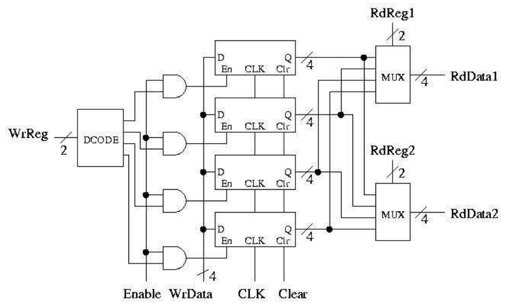
In order to execute an instruction such as ADD in a single clock cycle, it must be possible to read values from two registers and write a new value into a third register, simultaneously.

It is convenient to provide an Enable input, and in reality, a Clear input for zeroing all the registers would be provided.

The next slide shows 4 registers, each 4 bits wide. For more registers the multiplexers and decoder would be bigger, and the WrReg, RdReg1, RdReg2 inputs would be wider. For the ITM, register 0 is wired to 0.



# A Small Register File

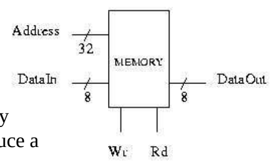





# Memory

Here is a memory component. If the Rd input is 1 then the contents of the location specified by the Address input appears on the DataOut output. If the Wr input is 1 then the location specified by the Address input is updated; its new contents are taken from the DataIn input.

This is a $2^{32}$ by 8 bit RAM: $2^{32}$ locations storing 8 bits each. In reality there is not (yet!) a single chip containing this much memory, but a number of memory chips could be combined to produce a circuit which would be used in the same way.



# Memory

Real RAM chips come in different varieties with different characteristics.

- The number of locations varies.
- The number of bits per location varies.
- There might be no Rd input.
- The Wr input might be edge triggered or level triggered.
- Large RAMs sometimes have only half as many address inputs as the number of bits needed to address all their locations. In this case, the memory is thought of as a 2 dimensional array of locations, and the *row address* and *column address* must be specified separately in two clock cycles.



# RAM and ROM

RAM stands for Random Access Memory. The name is no longer very informative, but RAM is memory which can be read from and written to. Quoted memory sizes of computers refer to RAM (e.g. 512 MB) The most widely used RAM technology loses its contents when power is turned off.

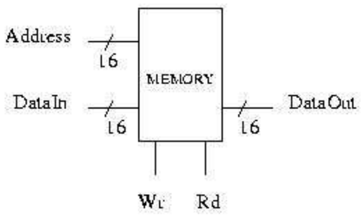
ROM stands for Read Only Memory. The contents are fixed at the time of manufacture and cannot be changed. Every computer has some ROM (a relatively small amount) which contains the instructions which will be executed when the power is turned on.

Other types of memory with different properties: PROM, EPROM, flash memory, ...



# Memory for the IT Machine

For simplicity we will assume that the memory of the IT Machine consists of 65536 locations (addressable by 16 bits), each storing a 16 bit word.






# Physical Implementation of Memory

An obvious way to implement RAM is with D flipflops. This turns out to be too bulky and expensive for large amounts of memory, because each flipflop consists of several components. However, relatively small (several kilobytes) amounts of very fast *cache memory* are made in this way, and called *static RAM*.

Large amounts (megabytes) of memory are made from *dynamic RAM*. Each bit is implemented by a tiny *capacitor* which is capable of storing a small amount of electric charge. Charge = 1, no charge = 0.

This charge leaks away in a few milliseconds and must be *refreshed*. This increases the complexity of memory systems.

# Instructions and Clock Cycles

We have described the execution of machine language programs as a sequence of steps, each step executing one instruction. We have looked at sequential circuits (of which a CPU is an example) driven by a clock. It would be natural to assume that each instruction is executed in one clock cycle, and therefore that the steps of execution are the same as the clock cycles.

The ITM can't be implemented in this way, basically because only one memory location can be accessed during each clock cycle.

There are two issues: some instructions are more than one word long, and some instructions involve further memory accesses when they are executed.



# Type 1 Instructions

The *register-register* instructions:
ADD, MUL, SUB, NEG, CMPEQ, CMPLT, CMPGT

The instruction is 1 word long, and just requires the ALU to take the contents of two registers, perform an operation, and put the result into a third register.

This can all be done in 1 clock cycle.

| | | | | |
|---|---|---|---|---|
| ADD Ru, Rv, Rw | 3 | u | v | w |



# Type 2 Instructions

LDVAL

The instruction is 2 words long, so it takes 2 clock cycles just to fetch the whole of the instruction from memory. During the second cycle, a value is transferred from memory into a register.

| | | | | | |
|---|---|---|---|---|---|
| LDVAL Ru, $number | 2 | u | 0 | 0 | number |



# Type 3 Instructions

LOAD, STORE

The instruction is 2 words long, so it takes 2 clock cycles just to fetch the whole of the instruction from memory. During the second cycle the value of *label* becomes known and the address can be calculated. It then takes a third cycle, and a third memory access, to do the load or store.

| | | | | | |
|---|---|---|---|---|---|
| LOAD Ru, label[Rv] | 1 | u | v | 0 | #label |
| STORE Ru, label[Rv] | 7 | u | v | 0 | #label |

# Type 4 Instructions

JUMP, JUMPT, JUMPF

The instruction is 2 words long, so it takes 2 clock cycles just to fetch the whole of the instruction from memory. During the second cycle the value of *label* becomes known and the address can be calculated. This address is put into the PC to make the jump happen.

| | | | | | | |
|---|---|---|---|---|---|---|
| JUMPT Ru, label[Rv] | b | u | v | 0 | #label |
| JUMPF Ru, label[Rv] | c | u | v | 0 | #label |
| JUMP label[Ru] | d | u | 0 | 0 | #label |



# Implementing Type 1 Instructions

There are two stages in designing the implementation of the CPU.

First we look at the *datapath* - the routes which data takes through the CPU. (Each instruction type has its own datapath; in fact, a different datapath for each clock cycle during its execution.)

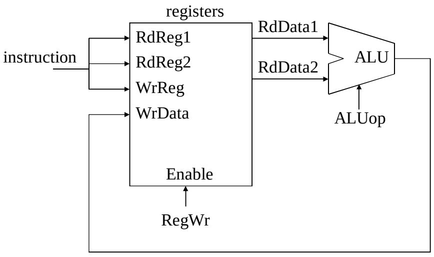
Then we look at the *control unit*, which analyses an instruction and generates control signals which activate the correct datapath for each cycle during the execution of that instruction.



# Datapath for Type 1 Instructions

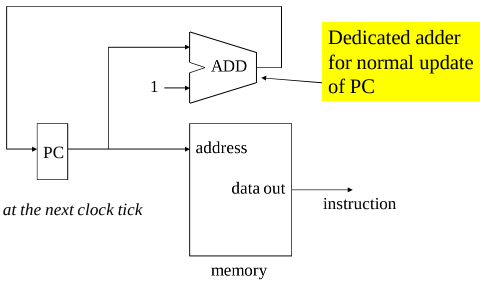
Register-register instructions must use the register file and the ALU.

The register file is updated *at the next clock tick.*






# Datapath for Instruction Fetch




PC is updated *at the next clock tick*

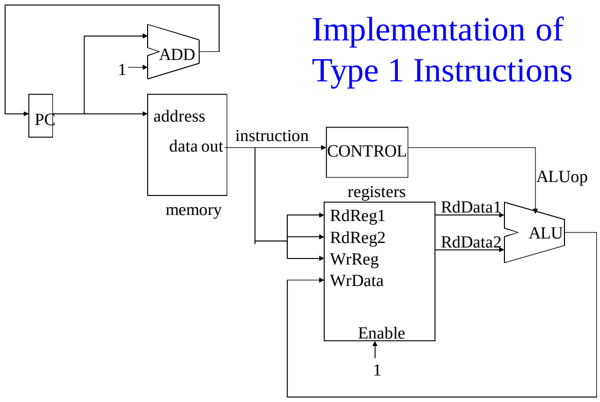
# Control Unit

If we are only implementing type 1 instructions, then the control unit is very simple. RdReg1, RdReg2 and WrReg come directly from the 2nd, 3rd and 4th hex digits of the instruction: each digit is a 4 bit value which identifies a register. RegWr is always 1 because every instruction updates a register. ALUop (a 3 bit value) is calculated from the first hex digit of the instruction by some straightforward combinational logic. (e.g. for ADD, the first hex digit is 0011 but ALUop might need to be 000, depending on the details of the ALU)

Exercise: Pick an arbitrary numbering of the ALU operations. Look up the first hex digit of each type 1 instruction. Design the logic needed to convert from the hex digit (as a 4 bit binary number) to ALUop.



# Implementation of Type 1 Instructions





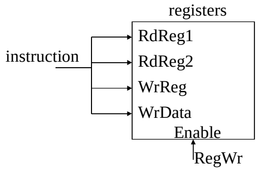

# Simplified Type 2 Instructions

To illustrate how different types of instructions can be implemented, consider a simplified form of the LDVAL instruction:

LDVAL Ru, $number

| 2 | u | number |
|---|---|---|

Here *number* is restricted to 8 bits, so that the instruction fits into a single word and can be executed in a single clock cycle.






# Type 1 & Type 2 Instructions

WrData now comes from *either* the ALU *or* the rightmost 8 bits of the instruction. (Actually the 8 bit number in the instruction must be *sign extended* to form a 16 bit value for WrData.)

A 2-1 multiplexer is used, controlled by a new control signal: RegSrc. The control unit must calculate the appropriate value of RegSrc, depending on the instruction type.

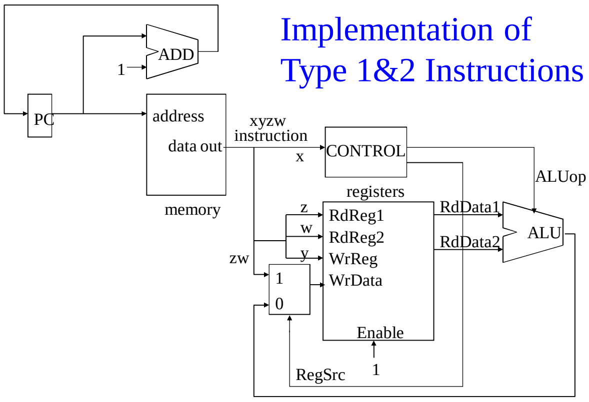
instruction

| x | y | z | w |
|---|---|---|---|

if x = 2 then RegSrc=1 else RegSrc=0

Exercise: what is the definition of RegSrc, assuming that the 4 bits of x are $x_3x_2x_1x_0$ ?

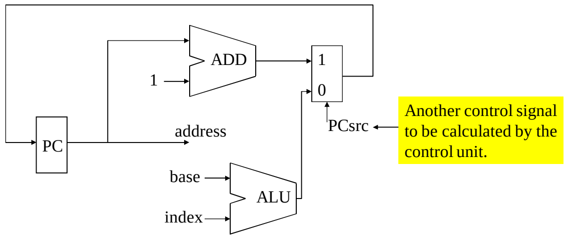
# Implementation of Type 1&2 Instructions





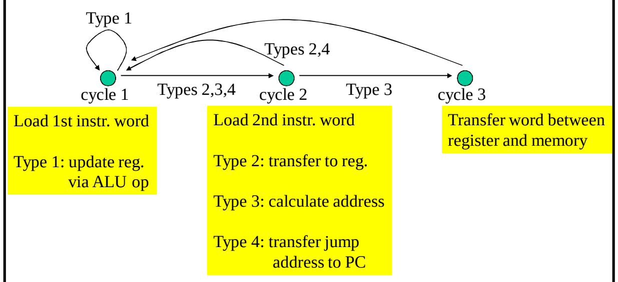

# Datapath Modifications: JUMP

Implementing JUMP instructions requires a modification to the datapath for instruction fetch. The next value of PC can come either from the ADD unit or from an address calculation.






# Multi-Cycle Instructions






# Multi-Cycle Instructions

In each state, appropriate control signals are generated, depending also on the instruction type.

The next state depends on the instruction type.

Communication between states: e.g. for a load/store, an address is calculated during cycle 2 and then used during cycle 3. Extra registers ("latches") must be introduced into the datapath to preserve values (e.g. addresses) from one cycle to the next.

# Microcode

In some CPU designs (particularly CISC) a lookup table is used to work out the state transitions and control signals for each instruction.

In effect, a machine language instruction is translated into a sequence of even lower-level instructions: *microcode*.

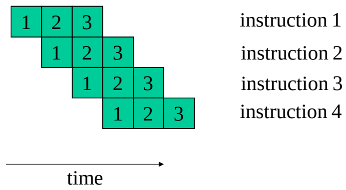
The microcode interpreter can be viewed as a RISC CPU embedded within the CPU.

Some early computers made the microinstruction level accessible to the programmer, allowing the implementation of machine language instructions to be modified.



# Pipelining

*Pipelining* is the idea of overlapping the execution steps of successive instructions, by taking advantage of the fact that different parts of the CPU are active at different steps of the execution of an individual instruction.






# Pipelining

Works best if all instructions are the same length and take the same amount of time to execute.

Complications are caused by jump instructions because there's a danger of executing steps from the wrong instruction.

The basic idea is the same as a production line.

Pipelining originated as a key feature of RISC designs.



# CPU Structure/Computer Structure

We can make a close analogy between the structure of a CPU and the structure of a computer.

| | |
|---|---|
| registers | memory |
| ALU | peripheral device |
| control unit | CPU |
| buses | buses |

The control unit is like a CPU within the CPU. A microprogrammed CPU has another layer within it: the microcode interpreter.

An infinite regress does not occur because the control unit is not a general-purpose programmable computer: it executes a *fixed* program, the fetch-decode-execute cycle.

# CS1Q Computer Systems
# Lecture 13



## Where we are

Global computing: the Internet
Networks and distributed computing
Application on a single computer
Operating System
Architecture
Digital Logic
Electronics
Physics

A general impression of the lowest levels of hardware



Computer Systems

## The Lowest Level of Hardware: Transistors and Chips

John O'Donnell
Computing Science Department
University of Glasgow


## And Now for Something Completely Different!

*How do logic gates work anyway?*

A brief introduction…

to VLSI electronics

A logic gate is actually a circuit comprising primitive components (MOSFET pass transistors) and we can understand (roughly) how these work just using a little elementary physics

## Physics Background

***Atoms***

Electrons have negative charge

Nucleus contains protons with positive charge

- Like charges repel
- Unlike charges attract



## Chemistry Background

- Electron Shells
  - The electrons around a nucleus are grouped into shells. Each shell has an ideal number of electrons that will fit into it, and the atom "likes" to have the outer shell filled
- Covalent Chemical Bonds
  - Two atoms with partially filled outer shells can "share" some electrons; they bond together



## Insulators and Conductors

- If an atom has just enough electrons to fill the outer shell exactly, then electricity (moving electrons) can't go through it. It's an insulator.
- If an atom has only a few electrons in a big shell far from the nucleus, it's easy to get those electrons moving. It's a conductor (silver, copper, gold, …)



## Semiconductors

- Silicon is a semiconductor, halfway between an insulator and a conductor.
- Its outer shell wants to have 8 electrons, but the Silicon atom provides only 4.
- The atom forms covalent bonds with its neighbors, ending with filled shells.
- This is a silicon crystal, which is a good insulator

# Doped Silicon

We could take a silicon crystal, but replace a few of the silicon atoms with Boron or Phosphorous.

- This would give us either one more or one fewer electrons in the outer shell
- And that means that an electrical field can more easily get some of those electrons moving, since the outer shell is not completely stable

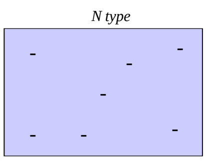


# N and P Type Silicon

- Pure silicon is an insulator, but it can be doped, turning it into:
  - N type silicon, a semiconductor with negative charge carriers (called free electrons)
  - P type silicon, a semiconductor with positive charge carriers (called holes). *These are spots where an electron would like to be (would lower the energy) but isn't.*

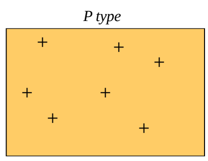


# Negative Charge Carriers



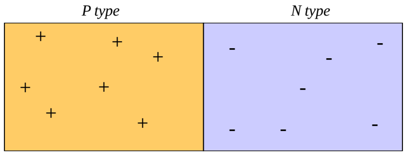



# Positive Charge Carriers




*In reality, the only movable charge carriers are negative electrons. However, they behave as if there are holes that can move, and the holes act as if they have a positive charge.*

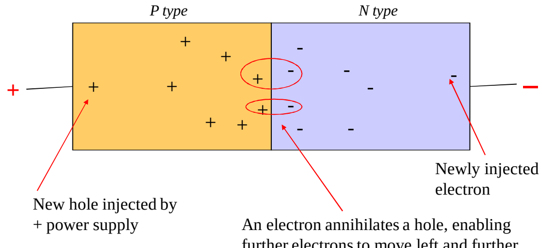
## Junctions




a PN junction is formed when a P-type region is fabricated adjacent to an N-type region



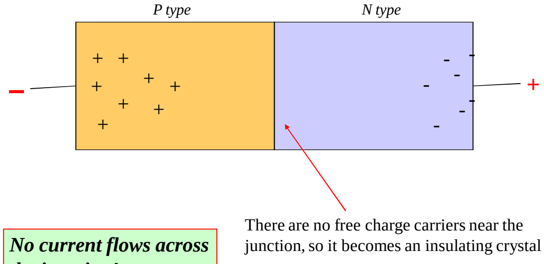
## Forward Biased Junction




***Current flows steadily!***



## Reverse Biased Junction




***No current flows across the junction!***

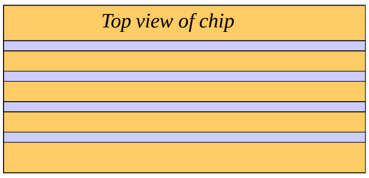


## The Junction Diode

A diode allows current to flow across it in one direction but not the other

A PN junction does exactly that!

# N-Type Wires




*Each of the four N-type wires carries a signal safely; a short circuit is impossible because it would have to cross a reverse biased junction*

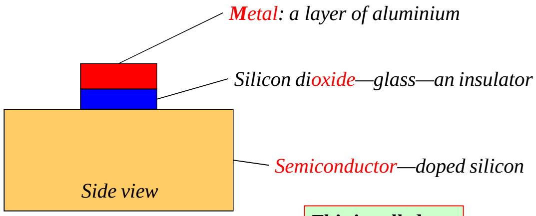


# Integrated Circuits

- Long ago, circuits were built by connecting the components one at a time, soldering in wires one at a time
- Using channels to hold wires suggests a new idea
  – Fabricate many components together on a chip, along with all their wires
  – The wiring is integrated with the components



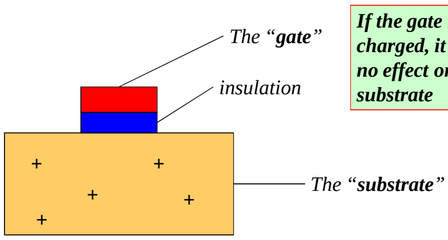
# Metal—Oxide—Semiconductor




***This is called MOS technology***



# Neutral Gate

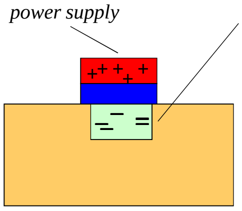



***If the gate is not charged, it has no effect on the substrate***

# Positive Gate, P-type Substrate

*Positive power supply*

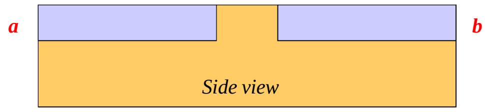
*Free electrons in the substrate are attracted to the region under the gate, where they are stuck because of the insulator*

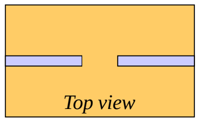


***If the gate is positively charged, it temporarily transforms part of the P-type substrate into N-type***



# Gap in a Wire

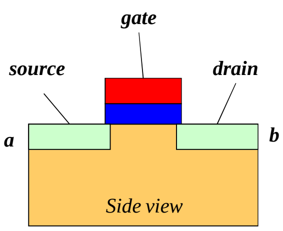



*No current can flow between **a** and **b**, in either direction, since it would have to cross a reverse biased junction*



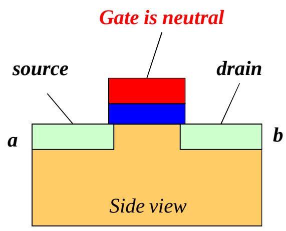
# N-Channel Pass Transistor




***A pass transistor consists of a MOS capacitor built right over a gap in a wire***



# Open N-Channel



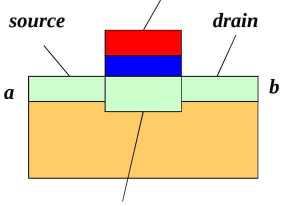

***If the gate is neutral, so that the capacitor is discharged, then current cannot flow between the source and drain, and we just have a wire containing a gap***

***The source and drain are disconnected!***

## Closed N-Channel

*This is called the field effect, using MOS devices—hence MOSFET*

*Gate is positively charged*



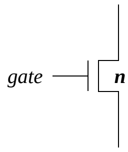

*Temporary N-type region*

***If the gate is positive, so the capacitor is charged, then the temporary N-type region under the gate closes the connection.***

***Current flows between source and drain!***



## A Controllable Switch




The N-channel pass transistor is a switch controlled by the gate



## P-Channel Pass Transistors

The same design, with the N/P and –/+ polarity reversed, is a P-channel pass transistor

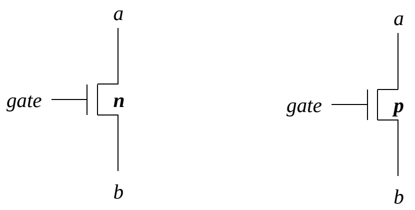
- If the gate is neutral, the wire has a gap
- If the gate is negatively charged, the source and drain are temporarily connected



## CMOS: Complementary MOS

- A circuit that contains both P-channel and N-channel devices is called CMOS
- Notice that we need both a positive voltage (to control the N-channel transistors) and a negative voltage (to control the P-channel transistors).
- CMOS is currently the dominant technology

# N and P Channel Transistors




The N-channel transistor makes connection if gate is +

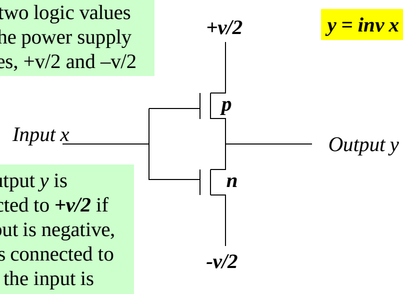
The P-channel transistor makes connection if gate is –



# CMOS Inverter

The two logic values are the power supply values, +v/2 and –v/2

**$y = inv\ x$**




The output *y* is connected to ***+v/2*** if the input is negative, and it's connected to ***–v/2*** if the input is positive



# Synthesis of Logic Gate Circuits

- The aim: design a circuit that implements a logic function f taking some inputs x, y, …
- The method:
  - Build a network of pass transistors that connect the output to High exactly when f x y = True
  - Build another network that connects the output to Low exactly when f x y = False



# Algebra of Logical And2

$z = and2\ x\ y$

= *True* if and only if $xy$ (obviously!)

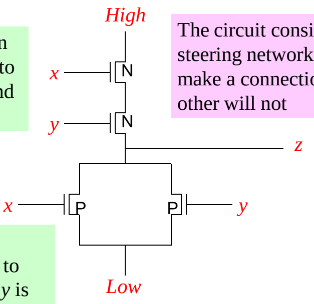
= *False* if and only if $\overline{xy} = \overline{x} + \overline{y}$

These two expressions are used directly to construct the steering logic…

## The Logic Gate *z = and2 x y*

The transistors in series connect *z* to *High* if both *x* and *y* are *High*

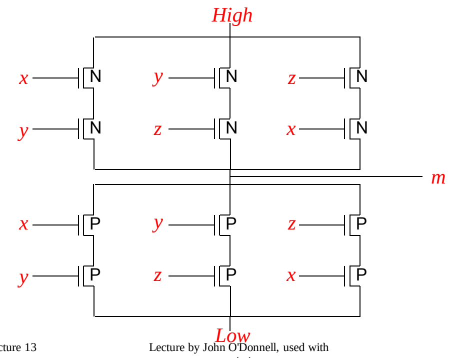
The circuit consists of two steering networks: one will make a connection, the other will not




The transistors in parallel connect *z* to *Low* if either *x* or *y* is *Low*



## Majority Voting






## Integrated Circuits

- We can fabricate many transistors on the same chip of silicon
- We can also fabricate wires connecting them directly on the chip
  - Short wires can simply be N or P type paths that are surrounded by the opposite type
  - Long wires are implemented by placing paths of aluminium on top of the surface



## IC Fabrication

- A photographic process – efficient because all the devices on the chip are manufactured in parallel
- The chip is built up in stages (e.g. doping selected regions to change them from P to N type)
- Photoresist is used to mask off the portions that are to be left unchanged

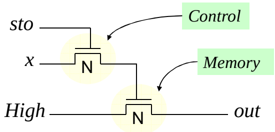
## Dynamic Storage

An inverter can be used to store a bit:

- Connect the gate to either High or Low
  - This will either attract charge carriers into the channel, or it will release them allowing the channel to revert to its default type
- Then disconnect the gate
  - The state of the channel will stay the same, because of the capacitor effect



## Dynamic Register Bit




The stored bit is always available on *out*

When sto is High, the value of x (which must be strong) goes onto the gate of the memory bit. When sto then goes Low, this charge remains isolated (because of the capacitor effect) and the memory remains.



## Refresh

- An charge that is isolated on the gate of a pass transistor will gradually dissipate.
- Eventually, the gate will not have a strong enough charge to control the channel, and the register bit become unreadable.
- To prevent this, the bit must be refreshed periodically: the gate must be reconnected to power (low or high) to restore the stored charge to its full strength.



## Dynamic RAM

- In DRAM chips, the memory cells are organized as a matrix: a row of columns. The address is used to activate a column, and to select a row to choose the right bit.
- In addition to serving store and fetch requests, the DRAM also does a periodic refresh operation: each bit in a column is read out and put back

# CS1Q Computer Systems
# Lecture 14

## Where we are

Global computing: the Internet
Networks and distributed computing
Application on a single computer
Operating System ← compilers and interpreters
Architecture
Digital Logic
Electronics
Physics



## High Level Languages

Most programming is done in high-level languages, such as Ada:
- their syntax is easier for humans to read and write
- the programming style is close to the way we think about problems, rather than close to the structure of the CPU
- we don't need to think about details such as register allocation and memory allocation
- programs are not specific to a particular design of CPU

How do we bridge the gap between high-level languages and the CPU?



## Compilers and Interpreters

There are two basic approaches: *compilation* and *interpretation*.

A *compiler* translates a program written in a high-level language (for example, Ada) into a program written in assembly language or machine language for a particular CPU (for example, Pentium).

This translation produces an independent, self-sufficient machine language program, which can be executed without reference to the original Ada program or to the compiler.

The Ada system which you use in CS1P is a compiler.

## Compilers and Interpreters

An *interpreter* analyses a high-level language program one statement at a time, and for each statement, carries out operations which produce the desired effect.

For example, if an Ada interpreter encounters the statement

```
x := 5;
```

then it must work out where in memory the variable x is stored, and put the value 5 in that location. If it encounters the statement

```
write(x);
```

then it must find the value of x in memory, and cause that value to be displayed on the screen.



## Compilers and Interpreters: Analogy

On holiday in Norway, I want to buy a sandwich for lunch. I do not speak Norwegian. I have two possible strategies:

1. Find someone who speaks English, and ask them to teach me how to ask for a sandwich in Norwegian. I can then buy a sandwich every day. This is like using a *compiler*.

2. Find someone who speaks English, and ask them to go into the shop and buy a sandwich for me. Every day I need to find another English speaker and repeat the process. I never find out how to ask for a sandwich in Norwegian. This is like using an *interpreter*.



## Compilers and Interpreters: Comparison

The compilation process takes time, but the compiled program runs quickly. With an interpreter, the overhead of compilation is avoided, but the program runs more slowly.

Traditionally, compilers are used for development of software which will be used many times and which must be efficient. Interpreters are used for simpler languages, more suited for quick programming tasks (e.g. *scripting* languages).

An interpreter supports a more flexible style of program development, e.g. for prototyping.



## Tombstone Diagrams

We can use *tombstone diagrams* to represent programs, compilers, and interpreters, and to keep track of the fact that three languages are involved in the compilation process:

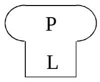
The *source language* is the language being compiled: e.g. in an Ada compiler, the source language is Ada.

The *target language* is the language produced by the compiler: e.g. Pentium machine language, or JVM instructions.

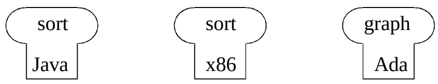
The *implementation language* is the language in which the compiler is written (remember that a compiler is just a program). This is not relevant to a user of the compiler, but is significant to compiler developers.

# Tombstone Diagrams

The following diagram represents a program called P, expressed in a language called L (the *implementation language*).




Examples:




A program called sort, expressed in Java.

A program called sort, expressed in x86 machine language (i.e. for Intel CPUs).

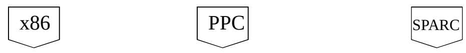
A program called graph, expressed in Ada.



# Tombstone Diagrams: Machines

A machine that executes machine language M is represented like this:


Examples:




An x86 machine (e.g. Pentium 4 PC)

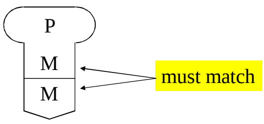
A Power PC (PPC) machine (e.g. Mac)

A SPARC machine (e.g. Sun workstation)

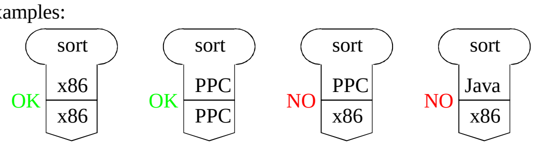


# Tombstone Diagrams: Execution

A program can run on a machine only if it is expressed in the correct machine language. Here's how we represent this:



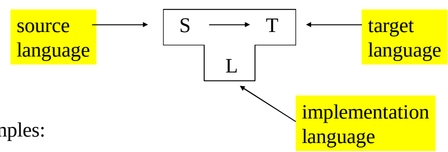

Examples:



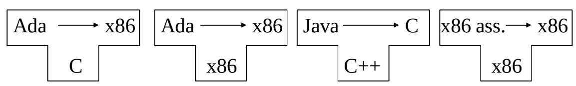



# Tombstone Diagrams: Compilers

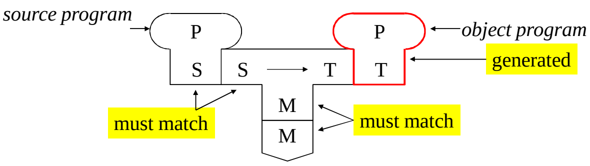
A compiler or *translator* is represented by this diagram:



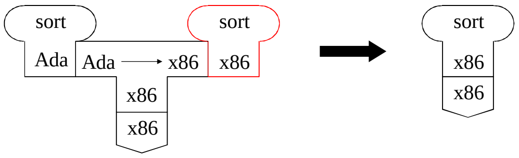

Examples:



# The Compilation Process




Example:






# Exercise

Imagine that we have:

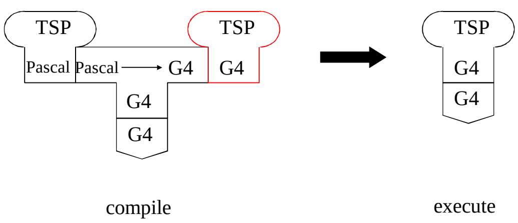
a program called TSP, written in the high-level language Pascal

a Pascal compiler, running on a G4 microprocessor (e.g. in an iMac), which produces object code for the G4

Draw diagrams showing how TSP can be compiled and executed.



# Solution

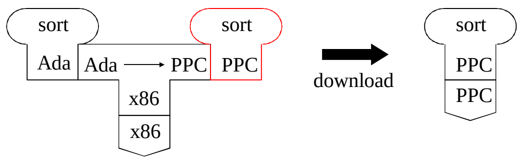





# Cross-Compilation

A *cross-compiler* runs on one machine (the *host* machine) but generates code for a different machine (the *target* machine).

Example:

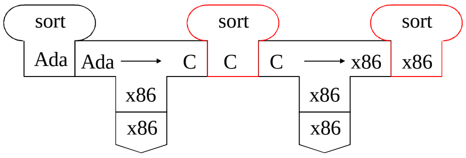



Examples: software development for video games, or embedded computing applications.

# Two-Stage Compilation

An S-into-T translator and a T-into-U translator can be composed to make a two-stage S-into-U translator.

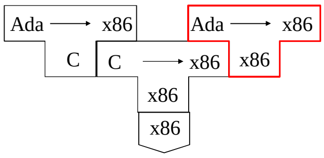
Example:






# Compiling a Compiler

Suppose we have an Ada-into-x86 compiler, expressed in C. We cannot run this compiler, because it is not expressed in machine language. But we can treat it as an ordinary source program to be translated by a C-into-x86 compiler:

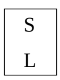



The object program is an Ada-into-x86 compiler expressed in x86 machine language, so it can be used as in the example on Slide 13.

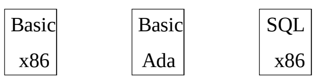


# Interpreters

To represent an interpreter, we show the source language S at the top and the implementation language L at the bottom.



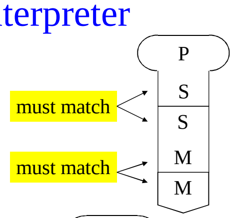

Examples:



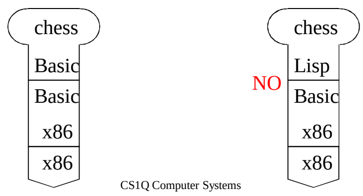



# Using an Interpreter

The combination of an interpreter expressed in M machine language, and a machine M, behaves like a machine which can execute the source language S. It is an *abstract* or *virtual machine*.



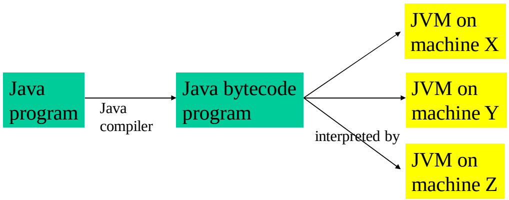

Examples:



# Interpretive Compilers:A Hybrid

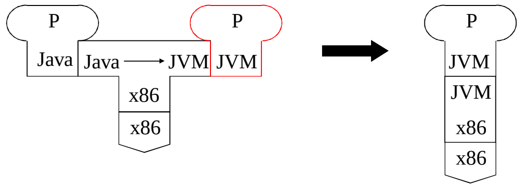
Combine compilation and interpretation:
- *compile* a high-level language into an *intermediate* language, often known as *bytecodes*
- *interpret* the bytecode program

This is the approach used by Java, to achieve *portability*.






# Interpretive Compilation of Java






# Virtual Machines

Java bytecode programs consist of instructions for the JVM (Java virtual machine). The JVM is a CPU which is implemented in software rather than hardware.

Advantages:
- "write once, run anywhere" - just need a JVM implementation for each type of computer, and a standard Java compiler.
- possibility of compiling other languages to JVM code

Disadvantages: interpreting a JVM program is slower than executing a truly compiled program.



# Refinements

Executing Java programs via the JVM leads to relatively poor performance, so some refinements of the idea have been developed.

*Native code compilers* compile Java into real machine language, for greater speed; of course, a different Java compiler is needed for each CPU.

*Just in time (JIT)* compilers interpret JVM instructions, but also compile them into native code. If a section of a program is executed more than once (e.g. in a loop) then the compiled version is used. Some of the efficiency of compiled code is achieved, without the initial cost. (But it is difficult for a JIT compiler to do as much optimisation as a conventional compiler.)

## Levels of Compilation

It's not always straightforward to say whether a language is being compiled or interpreted.

Java is *compiled* into JVM instructions which are then *interpreted*.

If a language is *compiled* into machine language, the individual machine language instructions are *interpreted* by the CPU.

In a sense, true compilation consists of translating a high level language program into a circuit, which directly carries out the desired function. Compilation to hardware is an active research area.

What we mean by a *machine language* is a language for which a *hardware* interpreter is available.



# CS1Q Computer Systems
# Lecture 15

## Where we are

Global computing: the Internet

Networks and distributed computing

Application on a single computer

Operating System

Architecture

Digital Logic

Electronics

Physics

How does the operating system support applications?



## Software and the Operating System

The hardware alone can't do anything; software is necessary.

Software is stored on disk, but must be in memory to run. Therefore:
- there must be a way of getting a program into memory

This is because we don't have a single memory technology which is big, cheap, fast and non-volatile.

We want a general-purpose machine. Therefore:
- there must be a way of changing the program which is running

It's nice to be able to run several programs at once (multi-tasking)
- there must be a way of sharing resources between programs

## Software and the Operating System

In a general-purpose computer, the software is split into the *operating system (OS)* and *application programs*. Assume for the moment that the OS is fixed and built into the computer. To change the function of the computer, different application programs can be loaded while the OS remains the same.

Examples of operating systems: Windows (in various forms), Unix (several versions including Linux), MSDOS (older), CP/M (older still), VMS, MacOS, BeOS, OS/2, PalmOS, etc. etc.

(Of course the OS is a piece of software, and it is possible to change the operating system: for example, dual-boot PCs, which can run both Windows and Linux, are common.)



## Multi-Tasking

*Multi-tasking* refers to the *appearance* that a computer is executing more than one program at a time. In a computer with a single CPU, this appearance must be created by switching rapidly between programs.

Example: Internet Explorer could be loading a web page, while you are typing into a Word document at the same time.

In a multi-tasking OS, many OS tasks (processes) run alongside user processes.



## Parallel Processing

*Parallel processing* refers to the use of several CPUs in order to actually execute many instructions simultaneously.

Example: weather forecasting is done by dividing the Earth's atmosphere into many cells, and applying the same calculations to each cell. This can be sped up by assigning a CPU to each cell.

Combinations of parallel processing and multi-tasking are possible: for example, a computer with 2 CPUs might also use multi-tasking to give the appearance of executing more than 2 programs at a time.

Example: it is straightforward to buy a PC with 2 or 4 CPUs.



## Operating System Functions

*Controlling the loading and execution of application programs*

The OS itself is a program, but it has a special role. A multi-tasking OS must share resources among several executing programs.

*Organising disk storage*

Information stored on disk is organised into *files* and *directories* (or *folders*), in a structure determined by the OS.

*Providing services to application programs*

The OS provides many common software functions, and insulates applications from hardware details.

*Providing a uniform user interface*

Applications use OS services to interact with the user; this gives a consistent "look and feel".

## Controlling Application Programs

A very simple OS might control applications as follows:
• user indicates that program P should be started
• OS transfers P from disk to memory
• OS executes a jump instruction, to the address of the beginning of P
• P does its stuff, eventually (we hope!) terminating and jumping back to an instruction within the OS
• OS removes P from memory and awaits another user instruction.

Problems with this approach:
• there is no way for the OS to stop a runaway program
• multi-tasking is difficult to support, unless it is assumed that every application jumps back into the OS at regular intervals.



## Interrupts

The solution to these problems is to use *interrupts*.
• The CPU is designed so that an interrupt signal (on a particular input) causes execution to jump to a particular address (either fixed, or determined by additional inputs associated with the interrupt).
• The hardware is designed so that interrupts are generated at regular intervals, automatically, many times per second.
• Interrupts cause jumps into the OS.

No matter what an application program is doing, the OS regains control of the computer regularly. On an interrupt, the OS saves the state of the application which was executing. It can then decide whether to continue executing the same application, switch to another application, shut down an application which is behaving badly, etc.



## Organising Disk Storage

The *file* (sometimes called a *document*) is the basic unit of long-term information storage. Ultimately a file contains binary data, but the data in different files is interpreted in different ways:
• human-readable text
• data produced by a software package, e.g. Microsoft Access
• the source code of a program, e.g. a .ada file
• an executable program, i.e. a .exe file (for Windows)
• an image, etc

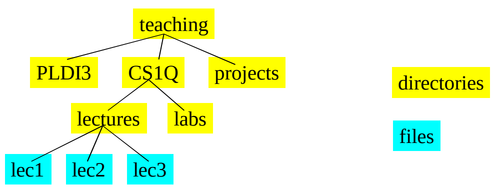
Sometimes the same file can be interpreted in different ways:
• an HTML file is treated as text if opened in an editor, but if it is opened by a web browser then it is interpreted as instructions for generating a web document
• a .exe file is interpreted as an executable program by the OS, but to a compiler it is a file containing data which is being generated.



## Organising Disk Storage

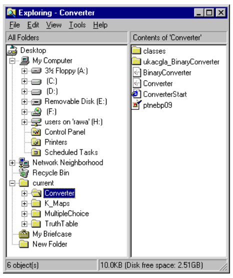
A *directory* (sometimes called a *folder*) is a collection of files and other directories. (This is an example of a *recursive* definition.)

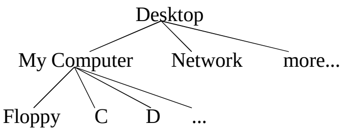
This leads to a *hierarchical filing system* with a *tree structure*.



# Windows Explorer

When viewing a folder, click on the “Folders” button to show the tree structure of the folders within it.






# Hierarchical Filing System

A hierarchical filing system is very natural, and has many advantages over a flat list of files.

*organisation of information*

*support for multiple users*
give them a directory each

*security*
convenient to restrict access to whole directories

*suppression of irrelevant information*
OS files can be kept separate from user files



# The Filing System as a Database

Each file or directory has several attributes:
• name, size, date created or modified
• security information: who can access or modify it
• for files: the data itself
• for directories: the files or directories which it contains

Exercise: think about the entity-relation structure of this data.

The hierarchical structure is very fundamental, and information on files and directories is usually accessed in relation to the directory tree. Global queries such as “how many directories contain just one file” are not the norm.



# Operating System Services

The most obvious OS functions are related to the user interface.

*Mouse/keyboard*: mouse movements and clicks are monitored by the OS. If something happens with the mouse that an application needs to know about, the OS informs that application. Similarly, keyboard input is directed to one of the running applications.

*GUI*: making a pop-up menu (for example) work is complex. The details are handled by the OS; an application is just told when a menu item has been selected.

Some of these functions *must* be in the OS: e.g. tracking the mouse must be done even when no applications are running. Others are there so that they can be used by many applications: e.g. controlling a pop-up menu.

# Operating System Services

Another important function of the OS is to provide a layer of abstraction which hides the specific details of the hardware. An application does not need to know exactly how bits are stored on the disk; it just needs to know that there are OS functions for reading and writing files. The OS communicates with the disk controller (a specific piece of hardware) in order to transfer the data. Within the OS, appropriate *device drivers* provide a uniform interface to particular types of disk controller (say).

The OS also provides higher-level services related to the filing system. For example, many applications make use of a file browser; this is provided by the OS.

Similar comments apply to network interfaces, printers, video displays, and any other peripheral devices.



# Example: Starting an Application

The user double-clicks on an icon.

The OS notices the double-click, sees that the mouse is on an area of the screen corresponding to an application icon, and launches the application. This involves finding the application code on disk, loading it into memory, and adding it to the list of executing applications so that it can be scheduled by the multi-tasking controller.

One of the first things the application does, is set up its user interface. This involves asking the OS to create windows, menus, buttons etc., and creating associations between UI elements and sections of program code which will process *events* involving those elements.



# Example: Saving a File

The user clicks on the "File" menu and selects "Save". This is all handled by the OS: detecting the first mouse click, realising that the mouse was on the menu heading, drawing the menu (its options have been defined previously by the application), detecting the mouse click on the "Save" option, removing the menu and restoring the original screen contents.

The OS tells the application that "Save" has been selected. Technically this is done by generating a "Menu item select" *event* which is detected by the application's *event handler*.

The application chooses what to do in response to this event. Presumably it will ask the OS to open the current file and store some data in it; perhaps the file browser (another OS function) will be used by the application to obtain a file name from the user.



# Example: Closing an Application

Normally an application shuts itself down in response to a "terminate" event. This event might be generated from within the application itself, in response to the user selecting "Exit" from a menu, or it might be generated by the OS if the user clicks on the close button on the application's window.

In either case, the application has a chance to execute some of its own code: tidy up files, ask the user whether to save unfinished work, etc.

When the application is ready to stop, it tells the OS. The OS removes it from memory and from the list of active tasks, removes its window from the screen, and deletes all of its UI elements.

In abnormal situations, such as a malfunctioning application, the OS has the power to shut down the application without going through its normal shutdown code.

## The OS as a Program

The OS is a program, not part of the hardware of the computer. How does the OS itself get loaded and start to execute? Two ideas:

Put the OS in *read only memory* (ROM), and build the computer so that it executes instructions from ROM when it is first turned on.

*This is rather inflexible.*

Store the OS on disk.

*But then how does it get from disk to memory?*

In practice an intermediate approach is used…



## Bootstrapping

The computer has a small amount of ROM, containing a program which is executed when the computer is first switched on. This program is able to find the OS on disk, load it into memory, and start executing it.

This approach is called *bootstrapping*, because the problem of loading the OS into memory seems similar to the impossible task of lifting yourself up by pulling on your own bootstraps. This is the origin of terms such as *booting*, *rebooting*, etc.

Very early programmable computers had, instead of ROM, a hardware mechanism allowing programs to be entered in binary by means of switches. To start using the machine, an operator would enter a small program by hand, which would then act as a simple OS, probably just reading another program from paper tape and executing it.



## Some OS History

1970 (approx): development of Unix at AT&T Bell Labs. The C programming language (ancestor of C++, Java, C#) was developed.

1975: CP/M developed for use on small computers; designed to be portable by separating the BIOS (Basic Input Output System) from the rest of the OS. (The name A: for the floppy disk drive in Windows is a relic of CP/M.)

1981: MS-DOS was supplied with the original IBM PC. The origin of Microsoft's success.

All of these were *command line* systems, influenced by Unix.



## Command-Line Interfaces

In a command-line system, the main (or only) means of interaction with the application / operating system / etc. is by typing textual commands. For example (from Unix):

```
$ cd /users/fda/simon/teaching
/users/fda/simon/teaching
$ ls
CS1Q      CF1       PLDI3
$ cd CS1Q
/users/fda/simon/teaching/CS1Q
$ ls
lectures  tutorials labs
```

Try "Command Prompt" in "Start -> Programs" and use `cd`, `dir`.

## Some OS History

1984: Apple Macintosh, with an OS entirely based on a GUI.
A smaller and cheaper version of the Lisa, in turn based on ideas from Xerox PARC in the 1970s. Apple's OS has evolved: now MacOS X.

1984: development of the X Windows System at MIT, a GUI for Unix. Unix with its command-line interface is still under the surface.

1985: development of Microsoft Windows as a response to the success of Macintosh. Initially running on top of MS-DOS (up to Windows 3.1)

1991: development of Linux, Unix for PCs.

1995: Windows 95 (later Windows 98) becoming more like a complete operating system. Meanwhile, Windows NT developed in parallel.

2000: Windows 2000 based on NT rather than 95/98; no longer based on MS-DOS.



# CS1Q Computer Systems
# Lecture 16

## Where we are

Global computing: the Internet

Networks and distributed computing

Application on a single computer

Operating System

Architecture

Digital Logic

Electronics

Physics

from small to large networks

interaction



## Networks

In computer terms:

*A group of computers connected together so that they can exchange information.*

More generally:

*Any system for exchanging information among physically separate components.*

Networks make *distributed computing* and *distributed applications* possible.

# Examples

bank cash machines

credit card payment machines

voice telephone
fax } *two different applications for the telephone network*

local area networks *e.g. within the CS department*

wide area networks

the Internet *a network of networks*



# Network vs. Application

Distinguish between a distributed application and the network which it uses.

Example: the World Wide Web is a particular application which uses the Internet for its data transfer. Electronic mail is another.

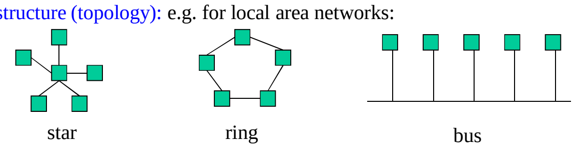
Example: home banking initially used dialup connections and special software; now it is usually done via the Internet.



# Issues in Networks & Distributed Systems

physical implementation: wire? optical fibre? undersea cables? satellites? radio? microwaves? analogue or digital?

structure (topology): e.g. for local area networks:




for wide area networks: mixture of structures

routing: must be a route between any two points. Perhaps several: how to choose one?



# Issues in Networks & Distributed Systems

naming: where is the data going? What does www.dcs.gla.ac.uk mean?

data formats and error correction

security: physical security? information security - encryption?
more issues: key exchange, authentication, non-repudiation, …

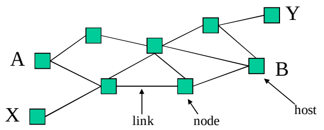
design of distributed systems:
- which part is in control?
- all the problems of programming, plus: synchronisation, security, reliability of networks, …
- issues beyond correctness: performance, quality of service, failure of components, failure of network, …
- how can we understand all this complexity?

# Hosts, Nodes and Links

Hosts are the computers that are running applications, including clients (e.g. your workstation running a web browser) and servers (e.g. a computer running a web server).

Nodes are computers at intermediate points in the networks, e.g. routers (specialised computers which deal with sending data to the correct destination).

Links are data connections between nodes, or between nodes and hosts.






# Circuit Switching, Packet Switching

Two fundamentally different ways to organise communication.

Circuit switching

for example, the telephone system

Packet switching

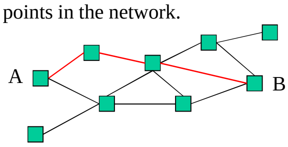
for example, the Internet



# Circuit Switching

Circuit switching: the same route is used for the whole of a "conversation" between two points in the network.

e.g. telephone system.




To establish a connection from A to B, a complete path must be determined and each node on the path remembers its role in the connection. Data can then be transferred at high speed.



# Circuit Switching: Assessment

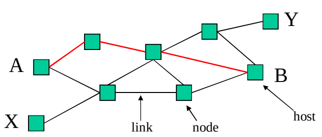
Advantages:
- Can carry very high volume messages efficiently.
- Works for both analogue and digital messages

Disadvantages
- It takes resources to establish the circuit in the first place, and remove it at the end.
- Bandwidth (transmission capacity) is wasted if a transmission comes in bursts.

# Packet Switching

Data is split into *packets*, and a route is found for each packet separately; packets are reassembled at destination. Only used for digital data.

e.g. the Internet




each node can contribute to several routes; similar idea to multi-tasking



# Client-Server Systems

A very common design for distributed applications is the *client-server* architecture.

The idea:

- the *server* is able to provide *services* to *clients*. The server responds to requests from clients.
- a *client* makes use of these services to accomplish some task.

Examples:

- a web browser acts as a client of web servers
- an email application acts as a client of an email server
- any application can act as a client of a printer server
- in a networked file system, individual computers are clients

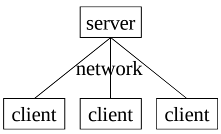
The client is in control.



# Client-Server Systems

Usually the client and the server are at different places on the network. Think of them as two different computers (actually, each is an application running on a computer).

Many clients may use the same server (this is the whole point!).



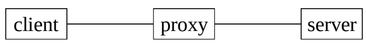

A particular application may be a client of several different types of server (e.g. printing from a web browser), or a client of several different servers of the same type (e.g. browsing different web sites).



# Client-Server Systems

An application may be both a client and a server.

Example: proxy web server




client sends requests to server, which are intercepted by the proxy server

proxy server responds using local data if possible, otherwise passes on the requests to the server

# Protocols

In order to communicate effectively, there must be standard conventions about:
• possible types of message
• the precise format and meaning of each message
• situations in which messages are sent

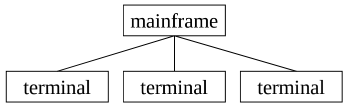
A *protocol* is a set of conventions for a particular area of networking.

Examples:
• protocols used by distributed applications, e.g. HTTP, FTP, …
• protocols used for detailed routing of packets between nodes



# Local Area Networks

In the 1960s and 1970s, multi-user computing was based on a central *mainframe* or *minicomputer* with a number of *terminals* directly connected to it by wires.




The terminals had minimal computing power.

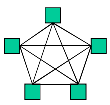


# Local Area Networks

Personal computers became widespread during the 1980s. An office full of independent computers makes it difficult to share resources (e.g. printers) and data, so networking was an obvious and necessary development (many ideas go back to the 1970s).

How best to organise the connections between computers?

One idea: point-to-point connections



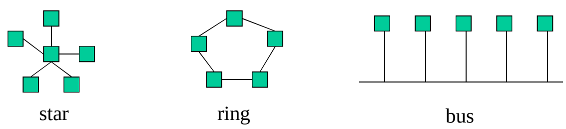
some benefits, but a huge drawback is the large number of connections



# Local Area Networks

Instead of point-to-point connections, LANs are based on cheaper (less wiring) interconnection schemes, and *broadcasting*. Every computer receives every message, but discards all the messages which are not addressed to it. (The details are handled by the hardware: a networked application is only aware of relevant messages.)

There is a variety of LAN technologies which use different network structures or *topologies*: (also, extensions and variations of these)



# Local Area Networks

Broadcasting means that all hosts are sharing the transmission medium. Data is organised into packets, so that long messages do not hog the resource.

Networked computers must be *named*, so that messages can be sent to the correct destination. There are three naming systems (in the Internet as well as in LANs): hardware addresses (48 bit numbers), IP addresses (32 bit numbers), host names (e.g. www.dcs.gla.ac.uk).

LANs generally involve relatively small numbers of computers in a local area (of course!) such as a single building.



# Local Area Networks

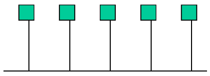
An obvious potential problem with broadcasting is: what happens if two or more computers broadcast at the same time? This must be avoided.

Also, in configurations other than a bus topology (where all computers are connected to a single wire), how does the broadcast happen?

Let's have a brief look at two systems: Ethernet (a bus topology) and token rings.



# Ethernet



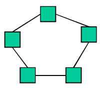
Ethernet is a widely used LAN technology based on a bus topology. Only one computer can transmit data at a time (otherwise there would be electrical conflicts).

Each computer is able to detect *collisions* when transmitting. If a collision occurs, each sender waits for a random time (up to 10ms, say) before trying again. If there is a second collision, the upper bound for the random delay is doubled (and so on, for subsequent collisions).

This is called *binary exponential backoff*. It is handled by the network interface hardware.



# Token Ring



Messages must be passed from computer to computer until they reach their destination.

In order to transmit a new message, a computer must wait for permission. Permission is obtained by receiving a special message, the *token*, which circulates around the ring.

After receiving the token, a computer is allowed to transmit one packet, then it passes the token on. Any computer not holding the token just passes messages on.

Handling the token, and passing on messages, is done by the network interface hardware; higher levels of software are just able to send and receive messages.

# Wide Area Networks

This really means large-scale networks, which generally also means that a large geographical area is involved.

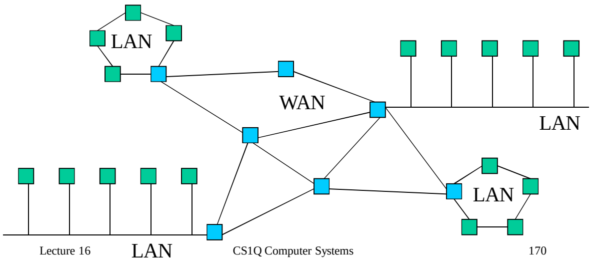
Different technology is needed. Broadcasting is no longer feasible:
- with a large number of hosts, so many messages are generated that broadcasting them all to the whole network would swamp it
- even in a wide area network there is still a lot of local communication, which it would be pointless to broadcast
  - e.g. emails within the university are sent via the Internet but there is no need for them to leave Glasgow

Scalability is important: the network structure must allow arbitrary numbers of hosts to be added.



# Wide Area Networks

A WAN consists of a number of LANs connected together by nodes which take care of long-distance routing. Messages from a LAN are routed through the network to their destination LAN, and then broadcast so that they can be picked up by the intended recipient.






# History of the Internet

The Internet originated in the ARPAnet project of the 1960s (USA).

This spread to become a more general academic, and later commercial, network during the 1970s and 1980s.

Various internet applications developed: email, FTP, telnet, usenet,...

The World Wide Web was developed in about 1993, primarily by Tim Berners-Lee at CERN. Its essence is a combination of a GUI based on *hypertext* (the browser) and a file transfer mechanism, using HTML to define hypertexts and HTTP as a protocol.

The web has become almost synonymous with the internet; in reality, the web is just one distributed application which uses the internet for data transfer. Of course, it's become the *killer app* for the internet.



# CS1Q Computer Systems
# Lecture 17

# The Layered Model of Networks

It is useful to think of networks in terms of a number of *layers*, or levels of abstraction. When working at one layer, all we need to know about is the *functionality* provided by the next layer down, not how that functionality is implemented.

This is just another example of the use of abstraction as the essential technique for understanding computer systems: for example, the hierarchical view of computing from Slide 2.

The *ISO 7 layer model* is a standard view of networks; we will look at the *TCP/IP layering model*, also known as the *Internet layering model*, which is more specific and is used by the Internet.



# The Internet Layering Model

5 *application layer*: specific network applications use specific types of message

4 *transport layer*: how to send a *message* between *any two points in the network* (two versions: TCP and UDP)

3 *internet layer*: how to send a *packet* between *any two points in the network* (also known as the IP layer)

2 *network interface layer*: how to send a *packet* along a *single physical link.* A packet is a frame containing routing information

1 *physical layer*: how to send *binary data* along a *single physical link.* Bits are organised into blocks called *frames*



# Naming

There are three different naming schemes in the Internet, used at different layers.

*hardware addresses*, also known as *MAC addresses*: each hardware device connected to the Internet has a 48 bit address, commonly written as 6 two-digit hex numbers, e.g. 00:d0:b7:8f:f2:0e

*internet addresses*, also known as *IP addresses*: each hardware device is given a 32 bit IP address when it is attached to the Internet. (Analogy: IP address is like a telephone number, MAC address is like the identity of a particular physical telephone.) IP addresses are commonly written in *dotted decimal* form, e.g. 130.209.241.152
IP addresses are running out and will be replaced by a 128 bit scheme known as IPv6.



# Naming

*domain names*: a more human-friendly naming scheme, e.g. marion.dcs.gla.ac.uk, www.dcs.gla.ac.uk
Note that there is not a direct correspondence between the fields in a domain name and the fields in an IP address.

Software in layer 5 uses domain names (so you can type a domain name into a web browser). Software in layer 4 translates between domain names and IP addresses. Software in layer 3 translates between IP addresses and hardware addresses.

## Uniform Resource Locator (URL)

A significant part of the development of the World Wide Web was the introduction of the URL: a standard way of specifying a host (computer), a protocol to use for communication with the host, and the name of a resource belonging to the host.

Example: http://www.dcs.gla.ac.uk:80/index.html

- http — the protocol, in this case HyperText Transmission Protocol
- www.dcs.gla.ac.uk — the host name
- 80 — the port to use when connecting to the host
- index.html — the resource name



## Example: Following a Web Link

Suppose we are using a web browser, and viewing a page which has a link to `www.dcs.gla.ac.uk/index.html`

What happens if we click on this link? The browser must:

- send a request to the server at `www.dcs.gla.ac.uk` asking for the file `index.html`
- receive the contents of this file from the server
- use the information in the file to display a web page

The server must receive the request and send the data.

There's a huge amount of complexity under the surface, but we'll look at the main points at each layer.



## View from Layer 5

| web browser | 5 | | 5 | web server at www.dcs.gla.ac.uk |
|---|---|---|---|---|
| | | GET /index.html HTTP/1.0 → | | |
| | | ← HTTP/1.0 200 OK *contents of index.html* | | |
| opens a connection to www.dcs.gla.ac.uk | | | | |
| sends request | | | | |
| | | | | sends response and data |
| closes connection | | | | closes connection |
| displays web page | | | | |



## View from Layer 5: Notes

The browser and server work with the idea of a *connection*, which is like the idea of circuit switching. The underlying network uses packet switching, not circuit switching, so a connection at layer 5 is a *virtual circuit* provided by software in layer 4.

The browser and server work with *messages* (in this case, text) of arbitrary length. Software in layer 4 breaks long messages into packets if necessary.

Software at layer 5 can assume that communication is reliable, even though the physical network may be unreliable. Layer 4 software monitors success/failure of packet transmission, resending if necessary.

Software at layer 5 is unaware of the details of the network between the client and the server.

## Interaction between L5 and L4

web browser 5 — 5 web server at www.dcs.gla.ac.uk

TCP/IP software 4 — 4 TCP/IP software



## Interaction between L5 and L4

web browser 5 — 5 web server at www.dcs.gla.ac.uk

function call

TCP/IP software 4 — 4 TCP/IP software

The server calls a function `listen(80)` to indicate that it is willing to accept connections on port 80.



## Interaction between L5 and L4

web browser 5 — 5 web server at www.dcs.gla.ac.uk

function call

TCP/IP software 4 — 4 TCP/IP software

The browser calls a function `connect("www.dcs.gla.ac.uk",80)` to open a connection to the server.



## Interaction between L5 and L4

web browser 5 — 5 web server at www.dcs.gla.ac.uk

TCP/IP software 4 — create connection → 4 TCP/IP software

The browser calls a function `connect("www.dcs.gla.ac.uk",80)` to open a connection to the server.

L4 software translates www.dcs.gla.ac.uk into 130.209.240.1 (how? later) and sends a message to create the connection.

## Interaction between L5 and L4

web browser 5 | 5 web server at www.dcs.gla.ac.uk

TCP/IP software 4 | 4 TCP/IP software

The functions `connect` and `listen` return objects which the browser and server can use to access the connection.

Messages can now be exchanged without specifying the destination address each time.



## View from Layer 4

TCP/IP software (on client machine) 4 ← ack packet 2 ← 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary



## View from Layer 4

TCP/IP software (on client machine) 4 | 4 TCP/IP software (on server machine)

packets are received & acknowledged
packets are reassembled so that when L5 software calls `receive` a complete message can be returned

if the L5 server software calls `send` then the reverse occurs



## View from Layer 4: Notes

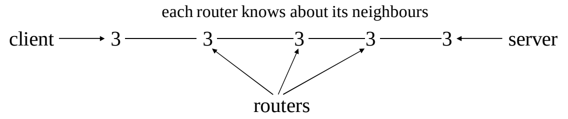
Software at layer 4 is unaware of the details of the network between the endpoints of a connection: it just calls functions provided by layer 3 software to send packets to an IP address.

Converting unreliable packet communication into reliable message communication is complicated:
- packets might not arrive in the same order in which they are sent
- acknowledgement packets might be lost
- next packet can be sent before previous packet is acknowledged
- how long should we wait for an acknowledgement?

There is an alternative version of layer 4, called UDP, which is based on individual messages rather than connections.

# View from Layer 3

Layer 3, the IP layer, is where the structure of the intermediate network becomes visible. It is where routing takes place.

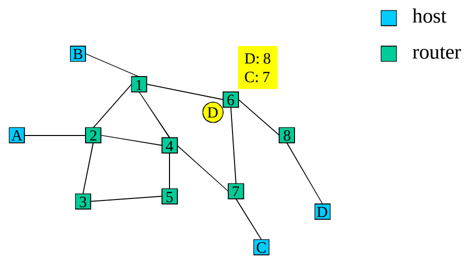



Routers do not have software from layers 4 or 5.
Packets are processed independently and are not acknowledged.

Basic operation of a router: receive a packet, labelled with its intended destination, and send it one step further towards that destination. The *local routing table* specifies which direction that step should be in.



# View from Layer 3: Routing




In each router, the local routing table specifies the next hop destination, depending on the final destination of the packer.



# Static and Dynamic Routing

Typically a host is part of just one local network, which is connected to the rest of the Internet by a single router. The local routing table in a host simply sends all packets to that router, and this never changes. Hosts use *static routing*.

Routers in general are connected to more than one network and have connections to several other routers. Each router has an initial local routing table, set up when it is first switched on, but the table can change. Routers use *dynamic routing*.

Local routing tables can change, as a result of changes in the structure of the network, or failures. Routers constantly check their own physical links, and exchange routing information with neighbouring routers. A distributed algorithm adjusts the routing tables so that, together, they specify the best overall routes for packets.



# Layer 2: IP to MAC Translation

Layer 2 functions are called from layer 3 in order to send a packet to the next hop destination. Layer 2 must translate the IP address of the next hop into the corresponding MAC (hardware) address.

Translation from IP addresses to MAC addresses is called *address resolution*. It only takes place for addresses on the same local network, and the details depend on the type of local network: either

- table lookup, or
- calculation, if the local network allows IP addresses and MAC addresses to be chosen in a related way, or
- by broadcasting a request to the local network, asking which machine has a certain IP address.

# View from Layer 1

Layer 1 consists of the network interface hardware (e.g. an Ethernet card plugged into a PC) and the low-level driver software.

In response to a call from layer 2, to send a packet to another machine on the local network, the layer 1 hardware generates the correct electrical signals and deals with collision detection.

In other forms of network (e.g. radio, optical, etc) the correct physical signals of whatever type are handled.



# Overall Progress of a Message

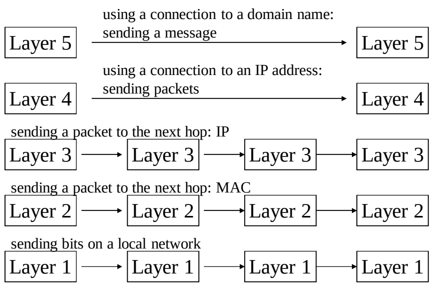
Messages are often described as moving down the layers of the network model, then across the network at layer 1, then back up the layers to the receiving application. What does this mean?


Downwards movements are calls to `send` functions in lower layers.
Upwards movements are responses to `receive` functions in higher layers.



# Overall Progress of a Message

As we have seen, each layer has its own view of how data moves across the network.






# More about HTTP

The basic interaction in HTTP, after a connection has been opened, is a request like GET /index.html HTTP/1.0 from the client, followed by a response like PUT HTTP/1.0 200 OK *data* from the server. The connection is then closed.

Other responses are possible, some of which are frequently seen while browsing: e.g. 404 FILE NOT FOUND.

A new connection is opened for every request, even if there is a series of requests to the same server. Not all protocols work in this way: e.g. FTP (file transfer protocol) keeps a control connection open, allowing several requests to be sent in sequence; for each request, a data connection is opened to transfer a file.

## More about Routing

The internet is designed to be *scalable* and *fault-tolerant*.

Fault-tolerance requires dynamic routing, so that broken connections or routers can be avoided. Scalability means that it is essential for routes to be chosen *locally*, using a *distributed algorithm*.

A global routing table for the whole internet would be huge. If copied to every router, it would overload them; if stored centrally, the network would be flooded with requests for routing information.

Next-hop routing using the local routing tables is an effective solution.



# CS1Q Computer Systems
# Lecture 18

## The Layered Model of Networks

It is useful to think of networks in terms of a number of *layers*, or levels of abstraction. When working at one layer, all we need to know about is the *functionality* provided by the next layer down, not how that functionality is implemented.

This is just another example of the use of abstraction as the essential technique for understanding computer systems: for example, the hierarchical view of computing from Slide 2.

The *ISO 7 layer model* is a standard view of networks; we will look at the *TCP/IP layering model*, also known as the *Internet layering model*, which is more specific and is used by the Internet.



## The Internet Layering Model

5 *application layer*: specific network applications use specific types of message

4 *transport layer*: how to send a *message* between *any two points in the network* (two versions: TCP and UDP)

3 *internet layer*: how to send a *packet* between *any two points in the network* (also known as the IP layer)

2 *network interface layer*: how to send a *packet* along a *single physical link*. A packet is a frame containing routing information

1 *physical layer*: how to send *binary data* along a *single physical link*. Bits are organised into blocks called *frames*

# Naming

There are three different naming schemes in the Internet, used at different layers.

*hardware addresses*, also known as *MAC addresses*: each hardware device connected to the Internet has a 48 bit address, commonly written as 6 two-digit hex numbers, e.g. 00:d0:b7:8f:f2:0e

*internet addresses*, also known as *IP addresses*: each hardware device is given a 32 bit IP address when it is attached to the Internet. (Analogy: IP address is like a telephone number, MAC address is like the identity of a particular physical telephone.) IP addresses are commonly written in *dotted decimal* form, e.g. 130.209.241.152
IP addresses are running out and will be replaced by a 128 bit scheme known as IPv6.



# Naming

*domain names*: a more human-friendly naming scheme, e.g. marion.dcs.gla.ac.uk, www.dcs.gla.ac.uk
Note that there is not a direct correspondence between the fields in a domain name and the fields in an IP address.

Software in layer 5 uses domain names (so you can type a domain name into a web browser). Software in layer 4 translates between domain names and IP addresses. Software in layer 3 translates between IP addresses and hardware addresses.



# Uniform Resource Locator (URL)

A significant part of the development of the World Wide Web was the introduction of the URL: a standard way of specifying a host (computer), a protocol to use for communication with the host, and the name of a resource belonging to the host.

Example: http://www.dcs.gla.ac.uk:80/index.html

- the protocol, in this case HyperText Transmission Protocol
- the host name
- the port to use when connecting to the host
- the resource name



# Example: Following a Web Link

Suppose we are using a web browser, and viewing a page which has a link to `www.dcs.gla.ac.uk/index.html`

What happens if we click on this link? The browser must:
- send a request to the server at `www.dcs.gla.ac.uk` asking for the file `index.html`
- receive the contents of this file from the server
- use the information in the file to display a web page

The server must receive the request and send the data.

There's a huge amount of complexity under the surface, but we'll look at the main points at each layer.

## View from Layer 5

web browser 5 5 web server at www.dcs.gla.ac.uk

Lecture 18 CS1Q Computer Systems 205

## View from Layer 5

web browser 5 5 web server at www.dcs.gla.ac.uk

opens a connection to www.dcs.gla.ac.uk

Lecture 18 CS1Q Computer Systems 206

## View from Layer 5

web browser 5 —— 5 web server at www.dcs.gla.ac.uk

opens a connection to www.dcs.gla.ac.uk

Lecture 18 CS1Q Computer Systems 207

## View from Layer 5

GET /index.html HTTP/1.0 →

web browser 5 —— 5 web server at www.dcs.gla.ac.uk

opens a connection to www.dcs.gla.ac.uk

sends request

Lecture 18 CS1Q Computer Systems 208

## View from Layer 5

GET /index.html HTTP/1.0

web browser 5 —— 5 web server at www.dcs.gla.ac.uk

opens a connection to www.dcs.gla.ac.uk

HTTP/1.0 200 OK
*contents of index.html*

sends request

sends response and data



## View from Layer 5

GET /index.html HTTP/1.0

web browser 5 —— 5 web server at www.dcs.gla.ac.uk

opens a connection to www.dcs.gla.ac.uk

HTTP/1.0 200 OK
*contents of index.html*

sends request

sends response and data

closes connection

closes connection

displays web page



## View from Layer 5: Notes

The browser and server work with the idea of a *connection*, which is like the idea of circuit switching. The underlying network uses packet switching, not circuit switching, so a connection at layer 5 is a *virtual circuit* provided by software in layer 4.

The browser and server work with *messages* (in this case, text) of arbitrary length. Software in layer 4 breaks long messages into packets if necessary.

Software at layer 5 can assume that communication is reliable, even though the physical network may be unreliable. Layer 4 software monitors success/failure of packet transmission, resending if necessary.

Software at layer 5 is unaware of the details of the network between the client and the server.



## Interaction between L5 and L4

web browser 5 5 web server at www.dcs.gla.ac.uk

TCP/IP software 4 4 TCP/IP software

## Interaction between L5 and L4

web browser 5 | 5 web server at www.dcs.gla.ac.uk

function call

TCP/IP software 4 | 4 TCP/IP software

The server calls a function `listen(80)`to indicate that it is willing to accept connections on port 80.



## Interaction between L5 and L4

web browser 5 | 5 web server at www.dcs.gla.ac.uk

function call

TCP/IP software 4 | 4 TCP/IP software

The browser calls a function `connect("www.dcs.gla.ac.uk",80)` to open a connection to the server.



## Interaction between L5 and L4

web browser 5 | 5 web server at www.dcs.gla.ac.uk

TCP/IP software 4 → 4 TCP/IP software

create connection

The browser calls a function `connect("www.dcs.gla.ac.uk",80)` to open a connection to the server.

L4 software translates www.dcs.gla.ac.uk into 130.209.240.1 (how? later) and sends a message to create the connection.



## Interaction between L5 and L4

web browser 5 | 5 web server at www.dcs.gla.ac.uk

TCP/IP software 4 | 4 TCP/IP software

The functions `connect` and `listen` return objects which the browser and server can use to access the connection.

Messages can now be exchanged without specifying the destination address each time.

## View from Layer 4

TCP/IP software (on client machine) 4

4 TCP/IP software (on server machine)

L5 software calls
`send(message)`



## View from Layer 4

TCP/IP software (on client machine) 4

4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary



## View from Layer 4

TCP/IP software (on client machine) 4 → packet 1 → 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary



## View from Layer 4

TCP/IP software (on client machine) 4 ← ack packet 1 ← 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary

# View from Layer 4

TCP/IP software (on client machine) 4 —packet 2→ 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary



# View from Layer 4

TCP/IP software (on client machine) 4 (no ack packet 2) 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary



# View from Layer 4

TCP/IP software (on client machine) 4 —resend packet 2→ 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary



# View from Layer 4

TCP/IP software (on client machine) 4 ←ack packet 2— 4 TCP/IP software (on server machine)

L5 software calls
`send(message)`

message is split into packets
packets are sent to destination IP address (by calling L3 functions)
wait for acknowledgement and resend if necessary

# View from Layer 4

TCP/IP software 4
(on client machine)

4 TCP/IP software
(on server machine)

packets are received & acknowledged
packets are reassembled so that when L5 software calls `receive` a complete message can be returned

if the L5 server software calls `send` then the reverse occurs



# View from Layer 4: Notes

Software at layer 4 is unaware of the details of the network between the endpoints of a connection: it just calls functions provided by layer 3 software to send packets to an IP address.

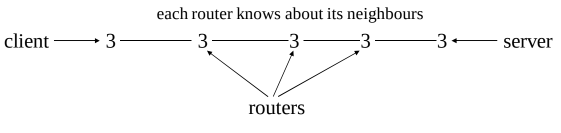
Converting unreliable packet communication into reliable message communication is complicated:
• packets might not arrive in the same order in which they are sent
• acknowledgement packets might be lost
• next packet can be sent before previous packet is acknowledged
• how long should we wait for an acknowledgement?

There is an alternative version of layer 4, called UDP, which is based on individual messages rather than connections.



# View from Layer 3

Layer 3, the IP layer, is where the structure of the intermediate network becomes visible. It is where routing takes place.

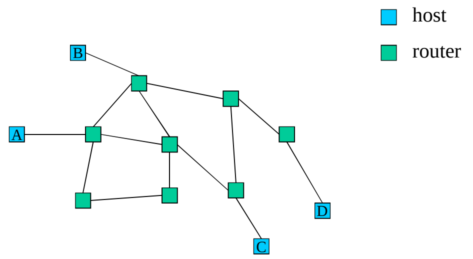



Routers do not have software from layers 4 or 5.
Packets are processed independently and are not acknowledged.

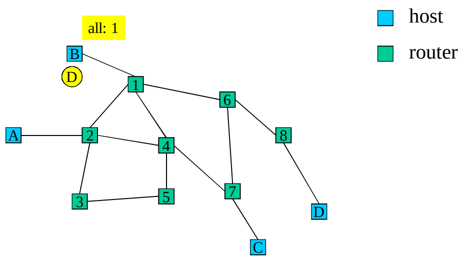
Basic operation of a router: receive a packet, labelled with its intended destination, and send it one step further towards that destination. The *local routing table* specifies which direction that step should be in.



# View from Layer 3: Routing



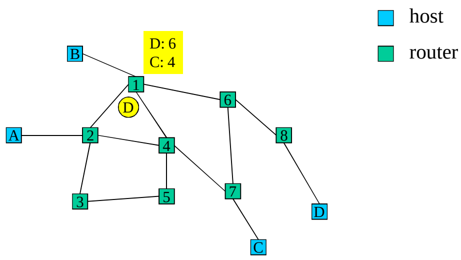
# View from Layer 3: Routing




A call from layer 4 in B produces a packet to be sent to D.
The local routing table (layer 3) in B specifies the *next hop* on the way to D. A function from layer 2 is called to send the packet.



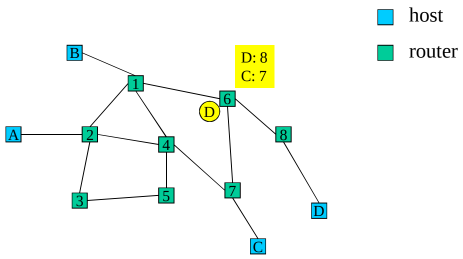
# View from Layer 3: Routing




In each router, the local routing table specifies the next hop destination, depending on the final destination of the packet.



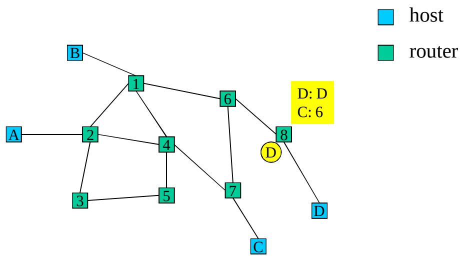
# View from Layer 3: Routing




In each router, the local routing table specifies the next hop destination, depending on the final destination of the packet.

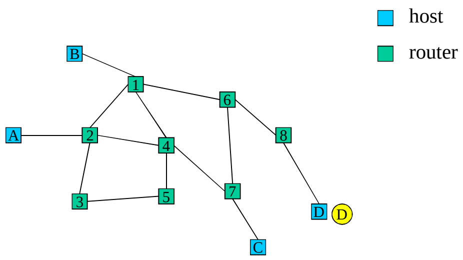


# View from Layer 3: Routing




In each router, the local routing table specifies the next hop destination, depending on the final destination of the packet.

## View from Layer 3: Routing






## Static and Dynamic Routing

Typically a host is part of just one local network, which is connected to the rest of the Internet by a single router. The local routing table in a host simply sends all packets to that router, and this never changes. Hosts use *static routing*.

Routers in general are connected to more than one network and have connections to several other routers. Each router has an initial local routing table, set up when it is first switched on, but the table can change. Routers use *dynamic routing*.

Local routing tables can change, as a result of changes in the structure of the network, or failures. Routers constantly check their own physical links, and exchange routing information with neighbouring routers. A distributed algorithm adjusts the routing tables so that, together, they specify the best overall routes for packets.



## Layer 2: IP to MAC Translation

Layer 2 functions are called from layer 3 in order to send a packet to the next hop destination. Layer 2 must translate the IP address of the next hop into the corresponding MAC (hardware) address.

Translation from IP addresses to MAC addresses is called *address resolution*. It only takes place for addresses on the same local network, and the details depend on the type of local network: either

- table lookup, or
- calculation, if the local network allows IP addresses and MAC addresses to be chosen in a related way, or
- by broadcasting a request to the local network, asking which machine has a certain IP address.



## View from Layer 1

Layer 1 consists of the network interface hardware (e.g. an Ethernet card plugged into a PC) and the low-level driver software.

In response to a call from layer 2, to send a packet to another machine on the local network, the layer 1 hardware generates the correct electrical signals and deals with collision detection.

In other forms of network (e.g. radio, optical, etc) the correct physical signals of whatever type are handled.

## Overall Progress of a Message

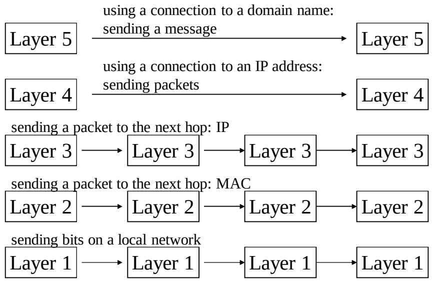
Messages are often described as moving down the layers of the network model, then across the network at layer 1, then back up the layers to the receiving application. What does this mean?


Downwards movements are calls to `send` functions in lower layers.
Upwards movements are responses to `receive` functions in higher layers.



## Overall Progress of a Message

As we have seen, each layer has its own view of how data moves across the network.






## More about HTTP

The basic interaction in HTTP, after a connection has been opened, is a request like GET /index.html HTTP/1.0 from the client, followed by a response like PUT HTTP/1.0 200 OK *data* from the server. The connection is then closed.

Other responses are possible, some of which are frequently seen while browsing: e.g. 404 FILE NOT FOUND.

A new connection is opened for every request, even if there is a series of requests to the same server. Not all protocols work in this way: e.g. FTP (file transfer protocol) keeps a control connection open, allowing several requests to be sent in sequence; for each request, a data connection is opened to transfer a file.



## More about Routing

The internet is designed to be *scalable* and *fault-tolerant*.

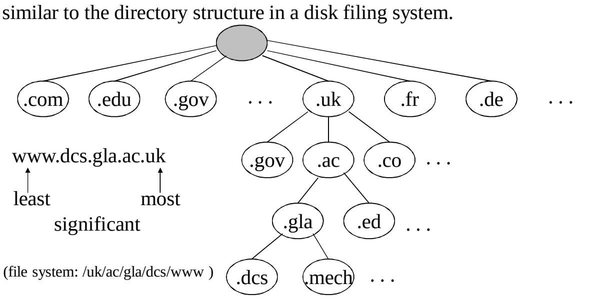
Fault-tolerance requires dynamic routing, so that broken connections or routers can be avoided. Scalability means that it is essential for routes to be chosen *locally*, using a *distributed algorithm*.

A global routing table for the whole internet would be huge. If copied to every router, it would overload them; if stored centrally, the network would be flooded with requests for routing information.

Next-hop routing using the local routing tables is an effective solution.

## Domain Names

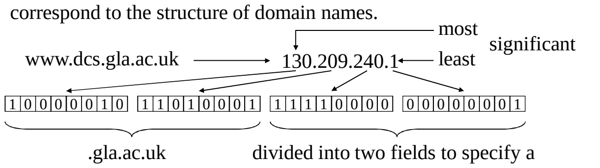
Domain names such as www.dcs.gla.ac.uk have a hierarchical structure similar to the directory structure in a disk filing system.






## Domain Names / IP Numbers

IP numbers also have a hierarchical structure but it does not directly correspond to the structure of domain names.




divided into two fields to specify a *subnet* and a *host* (dcs has several subnets)

e.g. the Level 1 lab is a subnet



## Name Resolution

Translation between domain names and IP addresses is necessary. This is called *name resolution*: a domain name is *resolved* to an IP address.

Essentially we need a database: think of a single table with two fields, the domain name and the IP address.

Properties of this database:
- very large: there are up to 4 billion IP addresses, so perhaps a complete database might require 64 gigabytes.
- constantly growing

It's not practical to have a copy of this database on every host:
- too big
- how would updates be managed?



## Name Resolution

It's not practical to have a central copy of the database: it would generate too much network traffic to and from the computer which stored it.

What is the solution? There is a *distributed* database called the *domain name system* (DNS), organised around a hierarchy of domain name servers.

To find out the IP address of a domain name, send a request to a domain name server. The server will either find the IP address in a local database, or find it by asking another server.

## DNS Example

A particular computer within the department, hawaii.dcs.gla.ac.uk, is the domain name server (strictly speaking, it runs a piece of software which is the domain name server) for the department's networks.

hawaii knows the IP numbers of all domain names within dcs.gla.ac.uk It is an *authority* for those domain names: if hawaii does not know about a domain name, say foo.dcs.gla.ac.uk, then it does not exist.

If hawaii is asked to resolve a domain name outside dcs, then it can pass on the request to another server, for example dns0.gla.ac.uk (or, it can give the requester a list of other servers to try; there are two modes)



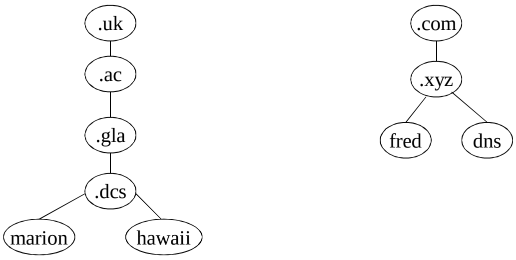
## DNS Example

A request for resolution of a name in a distant part of the domain name tree will ultimately be passed to a *root server*, which is an authority for the top-level domains (e.g. .com, .uk etc).

The root server does not contain all possible domain names, but it knows about servers which are authorities for parts of the domain name hierarchy. For example, it can pass on a request for www.ibm.co.uk to a server which is an authority for .co.uk ; eventually the request will reach a domain name server within IBM.



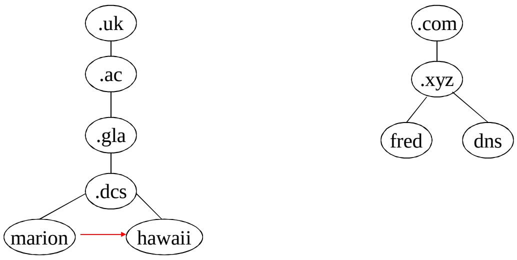
## DNS Example




An application on marion wants to resolve fred.xyz.com

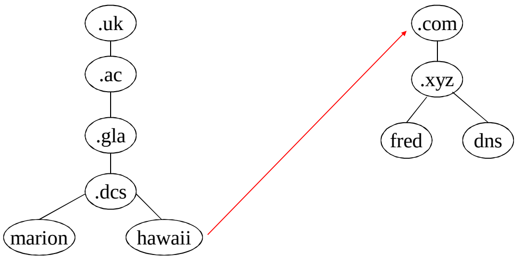


## DNS Example




hawaii is the local name server

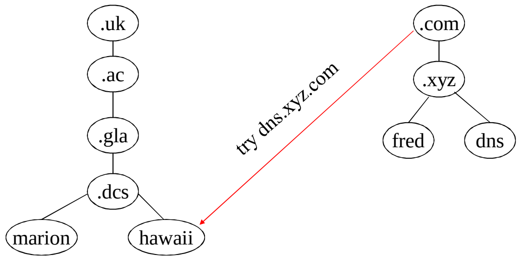
## DNS Example




hawaii is not an authority for .xyz.com, so it asks the root server for .com



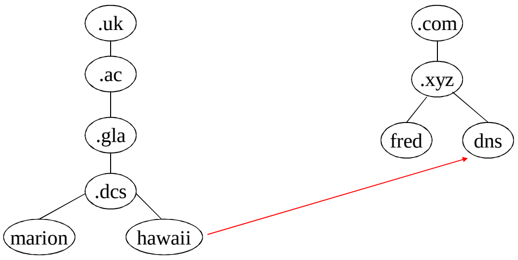
## DNS Example




the root server is not an authority for .xyz.com, but it knows who is: dns.xyz.com



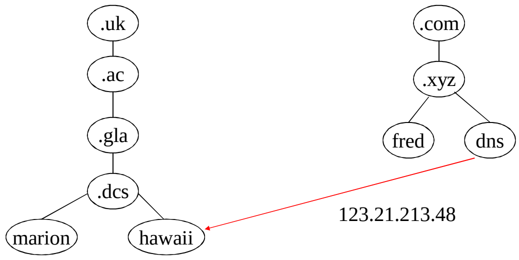
## DNS Example




hawaii sends a request to dns.xyz.com

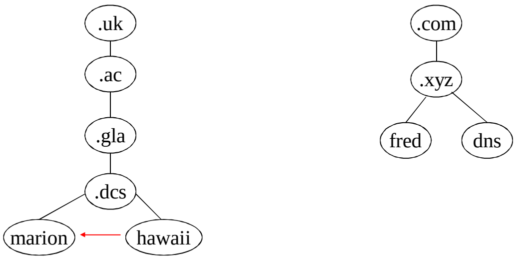


## DNS Example




dns.xyz.com responds with an IP address

# DNS Example




which hawaii returns to marion



# DNS Example

This illustrates the two styles of name resolution:

*iterative query resolution* describes the way a name server uses DNS: the result of a query is either an IP address or the name of another server.

*recursive query resolution* describes the way an application on a host uses DNS: the result of a query is an IP address, obtained by querying various servers until the appropriate authority is found.



# DNS Optimisations

The domain name system as described would be very inefficient, because there would be too much traffic at the root servers: too many requests (every time someone mentions the name of a computer in a different part of the DNS tree) would be passed to the root servers.

Two mechanisms are used to solve this problem: *replication* and *caching* (most important).

*Replication*: there are several copies of each root server, in different parts of the world; a local DNS server will use a root server which is geographically nearby.



# Caching

Caching is an important technique in many areas of computing. A *cache* (from the French *cacher*, to hide) is a local copy of recently-used or frequently-used data, which can be accessed more easily than the original source of that data.

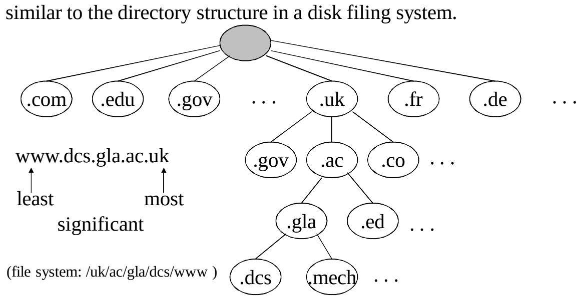
In DNS caching, each server maintains a local table of names and their resolutions, including names for which the server may not be an authority. Requests are answered from the cache if possible; if not, the cache is extended when the unknown name has been resolved.

Caching works well because name resolution exhibits *temporal locality of reference*: within a period of time, a user is likely to look up the same name repeatedly.

# CS1Q Computer Systems
# Lecture 19

## Domain Names

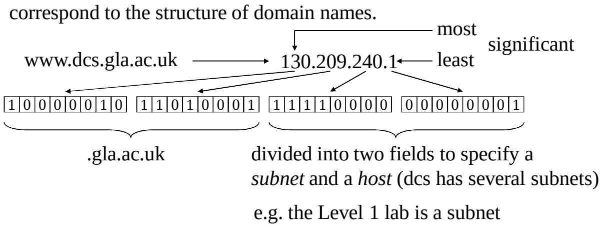
Domain names such as www.dcs.gla.ac.uk have a hierarchical structure similar to the directory structure in a disk filing system.






## Domain Names / IP Numbers

IP numbers also have a hierarchical structure but it does not directly correspond to the structure of domain names.




e.g. the Level 1 lab is a subnet



## Name Resolution

Translation between domain names and IP addresses is necessary. This is called *name resolution*: a domain name is *resolved* to an IP address.

Essentially we need a database: think of a single table with two fields, the domain name and the IP address.

Properties of this database:
- very large: there are up to 4 billion IP addresses, so perhaps a complete database might require 64 gigabytes.
- constantly growing

It's not practical to have a copy of this database on every host:
- too big
- how would updates be managed?

## Name Resolution

It's not practical to have a central copy of the database: it would generate too much network traffic to and from the computer which stored it.

What is the solution? There is a *distributed* database called the *domain name system* (DNS), organised around a hierarchy of domain name servers.

To find out the IP address of a domain name, send a request to a domain name server. The server will either find the IP address in a local database, or find it by asking another server.



## DNS Example

A particular computer within the department, hawaii.dcs.gla.ac.uk, is the domain name server (strictly speaking, it runs a piece of software which is the domain name server) for the department's networks.

hawaii knows the IP numbers of all domain names within dcs.gla.ac.uk
It is an *authority* for those domain names: if hawaii does not know about a domain name, say foo.dcs.gla.ac.uk, then it does not exist.

If hawaii is asked to resolve a domain name outside dcs, then it can pass on the request to another server, for example dns0.gla.ac.uk (or, it can give the requester a list of other servers to try; there are two modes)



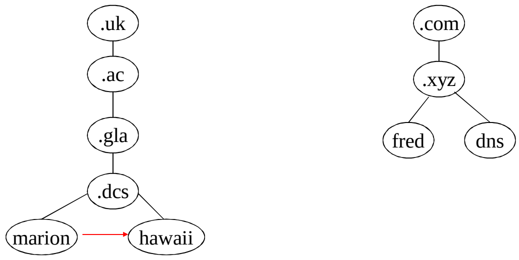
## DNS Example

A request for resolution of a name in a distant part of the domain name tree will ultimately be passed to a *root server*, which is an authority for the top-level domains (e.g. .com, .uk etc).

The root server does not contain all possible domain names, but it knows about servers which are authorities for parts of the domain name hierarchy. For example, it can pass on a request for www.ibm.co.uk to a server which is an authority for .co.uk ; eventually the request will reach a domain name server within IBM.



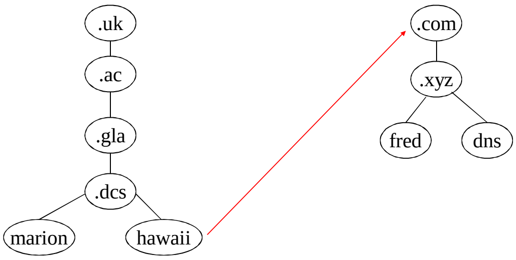
## DNS Example




An application on marion wants to resolve fred.xyz.com

hawaii is the local name server

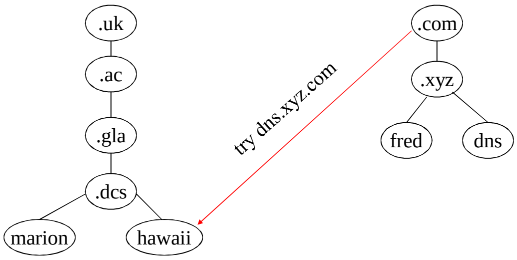
# DNS Example




hawaii is not an authority for .xyz.com, so it asks the root server for .com



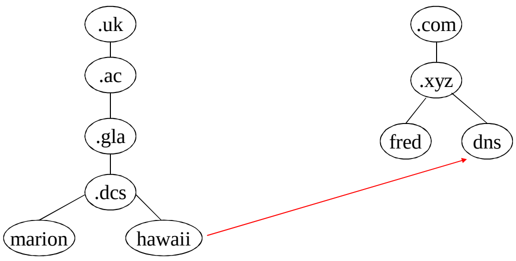
# DNS Example




the root server is not an authority for .xyz.com, but it knows who is: dns.xyz.com



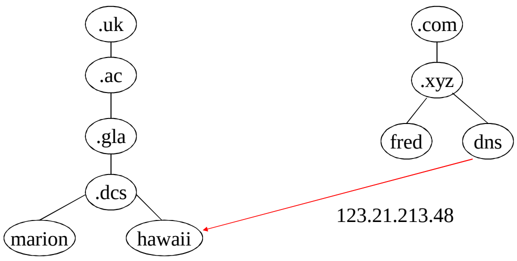
# DNS Example




hawaii sends a request to dns.xyz.com

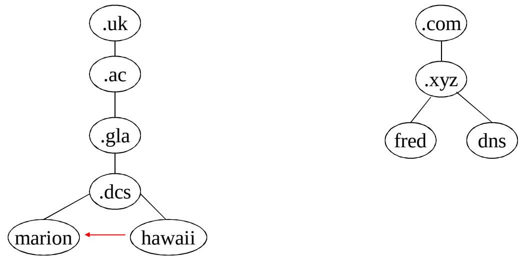


# DNS Example




dns.xyz.com responds with an IP address

# DNS Example




which hawaii returns to marion



# DNS Example

This illustrates the two styles of name resolution:

*iterative query resolution* describes the way a name server uses DNS: the result of a query is either an IP address or the name of another server.

*recursive query resolution* describes the way an application on a host uses DNS: the result of a query is an IP address, obtained by querying various servers until the appropriate authority is found.



# DNS Optimisations

The domain name system as described would be very inefficient, because there would be too much traffic at the root servers: too many requests (every time someone mentions the name of a computer in a different part of the DNS tree) would be passed to the root servers.

Two mechanisms are used to solve this problem: *replication* and *caching* (most important).

*Replication*: there are several copies of each root server, in different parts of the world; a local DNS server will use a root server which is geographically nearby.



# Caching

Caching is an important technique in many areas of computing. A *cache* (from the French *cacher*, to hide) is a local copy of recently-used or frequently-used data, which can be accessed more easily than the original source of that data.

In DNS caching, each server maintains a local table of names and their resolutions, including names for which the server may not be an authority. Requests are answered from the cache if possible; if not, the cache is extended when the unknown name has been resolved.

Caching works well because name resolution exhibits *temporal locality of reference*: within a period of time, a user is likely to look up the same name repeatedly.

## Electronic Mail

Originally used for communication within an office or organisation, first between users of a single computer system, then over a LAN, then over the Internet.

Email is now a very widely used Internet application. We'll look at some details of

- the format and content of email messages
- the operation of email clients and servers
- protocols and data formats



## Email: Basic Concepts

A *mailbox* is an area of storage ( a particular file or directory on disk) which contains messages sent to a particular email user.

An *email address* is a combination of a *mailbox* and a *domain name*.

Example:

simon@dcs.gla.ac.uk

the name of a mailbox (often, but not necessarily, the same as a login name)

the domain name of a computer which is able to receive email



## Email Addresses

Because it is convenient for all email addresses within an organisation to have a uniform format, it is usual for one particular computer to be designated as the *mail gateway*.

For example, my email address is simon@dcs.gla.ac.uk rather than simon@marion.dcs.gla.ac.uk

Our mail gateway is a computer called iona.dcs.gla.ac.uk, also known as mailhost.dcs.gla.ac.uk or just dcs.gla.ac.uk. All of these domain names resolve to the same IP address (130.209.240.35).

Email addressed to simon@dcs.gla.ac.uk is received by iona and placed in my mailbox, which is a directory called /users/fda/simon/Mail/inbox

(In fact, simon@marion.dcs.gla.ac.uk also works.)



## Format of an Email Message

An email message consists of *ASCII text* (see next slide). There are a number of *header lines* of the form *keyword* : *information* , then a blank line, then the *body* of the message.

The header lines *To: From: Date: Subject* are always included. Others are optional and many of them may be suppressed by email reading software.

Email software is free to add header lines which may or may not be meaningful to the software at the receiving end (e.g. some email software is able to request an acknowledgement message when a message is read, but not all receiving software understands this).

```
Return-Path: vv@di.fc.ul.pt
Return-path: <vv@di.fc.ul.pt>
Envelope-to: simon@dcs.gla.ac.uk
Delivery-date: Wed, 24 Apr 2002 09:48:42 +0100
Received: from mail.di.fc.ul.pt ([194.117.21.40] helo=titanic.di.fc.ul.pt)
          by iona.dcs.gla.ac.uk with esmtp (Exim 3.13 #1)
          id 170ISA-00001y-00
          for simon@dcs.gla.ac.uk; Wed, 24 Apr 2002 09:48:42 +0100
Received: from di.fc.ul.pt (dialup06.di.fc.ul.pt [194.117.22.76])
          by titanic.di.fc.ul.pt (8.9.3/8.9.3) with ESMTP id JAA17140
          for <simon@dcs.gla.ac.uk>; Wed, 24 Apr 2002 09:48:40 +0100
Received: (from vv@localhost)
          by di.fc.ul.pt (8.11.6/8.11.6) id g3O8oJM17793;
          Wed, 24 Apr 2002 09:50:19 +0100
Date: Wed, 24 Apr 2002 09:50:17 +0100
From: Vasco Thudichum Vasconcelos <vv@di.fc.ul.pt>
To: simon gay <simon@dcs.gla.ac.uk>
Subject: Re: Titles and Abstracts
Message-Id: <20020424095017.49de0cdc.vv@di.fc.ul.pt>
In-Reply-To: <200204230908.KAA11301@marion.dcs.gla.ac.uk.dcs.gla.ac.uk>
References: <20020320102141.273e7e41.vv@di.fc.ul.pt>
          <200204230908.KAA11301@marion.dcs.gla.ac.uk.dcs.gla.ac.uk>
Organization: DI/FCUL
X-Mailer: Sylpheed version 0.7.0 (GTK+ 1.2.10; i586-pc-linux-gnu)
Mime-Version: 1.0
Content-Type: text/plain; charset=US-ASCII
Content-Transfer-Encoding: 7bit

thanks.

vasco
```

Received … lines: information about the mail system of the sender

Date / From / To / Subject: useful to the human reader

thanks. / vasco: the body of the message (in this case, dwarfed by the header!)



# Emails are Text

The body of an email is *text*: specifically, characters from the ASCII (American Standard Code for Information Interchange) character set, which uses a 7 bit format to represent 128 characters including upper and lower case letters, digits, punctuation symbols, and 33 control characters such as *newline*.

If non-text data (e.g. images, or binary data files produced by applications) are to be sent by email, they must be *encoded* using only printable ASCII characters.

(As email is a lowest common denominator of Internet access, it's convenient to be able to transmit data of all types by email, despite the existence of special-purpose applications/protocols such as FTP.)



# Encoding Non-Text Data

One way of encoding arbitrary binary data into printable ASCII would be to encode binary 0 as the character '0' (ASCII code 48) and binary 1 as the character '1' (ASCII code 49). This would have the unfortunate effect of multiplying the size of the data by 8, because each ASCII character is represented by one byte (8 bits) in a data packet.

A better way would be to think of a byte as a two digit hex number, and use the characters '0'… '9' and 'A'… 'F' to encode it. This would multiply the size of the data by 2, which is still not very good.

Based on the same idea, but using a greater proportion of the printable ASCII range, we get *base 64 encoding* which is commonly used with email.



# Base 64 Encoding

| | | | | |
|---|---|---|---|---|
| 10010010 | 01010001 | 11110100 | | 3 bytes = 24 bits |
| 100100 | 100101 | 000111 | 110100 | 4 blocks of 6 bits |
| 36 | 37 | 7 | 52 | numbers 0…63 = "digits" in base 64 |
| D | E | ' | T | add 32 to get the ASCII code of a printable character |

Each character is represented by 1 byte, so the data expansion is 8/6 = 33% larger (actually it's slightly worse because extra information is introduced in relation to line breaks).

# MIME and Attachments

Modern email applications provide *attachments*, a convenient way of combining text and non-text data (generally, several pieces of data, perhaps of various types and encoded in different ways) in a single message. This system is called MIME (Multipurpose Internet Mail Extensions).

The email application which creates the message uses the MIME format to specify what data is present and how it is encoded. The receiving email application has the opportunity to decode it (although there is no guarantee that it will know how to do so).

The next slide shows an example of the text form of a MIME-encoded message. Most email applications will hide the non-text data, and just present a button (for example) allowing the attached data to be extracted.



```
Mime-Version: 1.0
Content-Type: multipart/mixed; boundary="============_-1192028076==_============"
Date: Mon, 29 Apr 2002 16:33:39 +0100
To: simon@dcs.gla.ac.uk
From: Tania Sprott <tania@dcs.gla.ac.uk>
Subject: CS1Q Questionnaire

--============_-1192028076==_============
Content-Type: text/plain; charset="us-ascii"

Hi Simon

Attached list of CS1Q tutors for PowerPoint as discussed.  I'll be along
shortly with the questionnaires.

Tania

--============_-1192028076==_============
Content-Type: multipart/appledouble; boundary="============_-1192028076==_D============"

--============_-1192028076==_D============
Content-Transfer-Encoding: base64
Content-Type: application/applefile; name="%CS1Q-Groups.doc"
Content-Disposition: attachment; filename="%CS1Q-Groups.doc"

AAUWBwACAAAAAAAAAAAAAAAAAAAAAAAAAAQAAAADAAAASgAAAA8AAAAJAAAAWQAAACAA
AAAIAAAAeQAAABAAAAACAAAAiQAAAR5DUzFRLUdyb3Vwcy5kb2NXOEJOTVNXRAEAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
HlNPUlQHAgCAABwAHv//
--============_-1192028076==_D============
Content-Type: application/octet-stream; name="CS1Q-Groups.doc"
 ; x-mac-type="5738424E"
 ; x-mac-creator="4D535744"
Content-Disposition: attachment; filename="CS1Q-Groups.doc"
Content-Transfer-Encoding: base64

0M8R4KGxGuEAAAAAAAAAAAAAAAAAAAAAPgADAP7/CQAGAAAAAAAAAAAAAAAADAAAANgAA
AAAAAAAAEAAAOQAAAAEAAAD+////AAAAADcAAAA8AAAAPQAAAP/////////////////
////////////////////////////////////////////////////////////////////
```



# Sending Email

Suppose I send an email to my colleague Vasco, vv@di.fc.ul.pt

There are several stages:

- using my email application, I compose a message and press "send"
- my email application becomes a client of the mail server on the department's mail gateway machine, iona.dcs.gla.ac.uk, and specifies the destination address and the content of the message (*)
- iona (actually, the `sendmail` program running on iona) becomes a client of the mail gateway in Vasco's department, mail.di.fc.ul.pt (194.117.21.40) and specifies the mailbox (vv) and the content (*)
- the message is delivered to Vasco's mailbox directory

Steps (*) use SMTP, the Simple Mail Transfer Protocol.



# SMTP

The email application and the `sendmail` program live in layer 5 of the network hierarchy. Opening a connection to the destination mail gateway uses the same mechanism as opening a connection to a web server: TCP functions (layer 4) are called, which must resolve the domain name of the destination and send messages to establish the connection. The difference at this stage is that port 25 is used, instead of port 80.

SMTP specifies a completely different (and more complex) exchange of messages than HTTP.

For example, the first SMTP message is sent by the server, not the client (but remember that the first TCP message comes from the client side, to open the connection).

# Example SMTP Session

```
220-Welcome to the dcs.gla.ac.uk mail relay.
220-
220 iona.dcs.gla.ac.uk ESMTP Exim 3.13 Mon, 29 Apr 2002 18:10:42 +0100
>>> EHLO marion.dcs.gla.ac.uk.dcs.gla.ac.uk
250-iona.dcs.gla.ac.uk Hello root at marion.dcs.gla.ac.uk [130.209.241.152]
250-SIZE 26214400
250-PIPELINING
250 HELP
>>> MAIL From:<simon@dcs.gla.ac.uk> SIZE=37
250 <simon@dcs.gla.ac.uk> is syntactically correct
>>> RCPT To:<simon@dcs.gla.ac.uk>
250 <simon@dcs.gla.ac.uk> verified
>>> DATA
354 Enter message, ending with "." on a line by itself
>>>
>>> Hi there!
>>>
>>> .
250 OK id=172Efi-0005Dz-00
simon... Sent (OK id=172Efi-0005Dz-00)
Closing connection to mailhost.dcs.gla.ac.uk.
>>> QUIT
221 iona.dcs.gla.ac.uk closing connection
```

from client



# Receiving Email

If a user directly logs in to the computer whose disk holds the mailbox, then the email application simply looks at the mailbox.

(Similarly in the department, although the mailbox may live on a different computer on the local network.)

Alternatively, another protocol called POP (Post Office Protocol) allows a mailbox to be inspected remotely; this is typically used in situations where a user dials up (using a modem) to an email service provider.



# CS1Q Computer Systems
# Lecture 20

# Electronic Mail

Originally used for communication within an office or organisation, first between users of a single computer system, then over a LAN, then over the Internet.

Email is now a very widely used Internet application. We'll look at some details of
• the format and content of email messages
• the operation of email clients and servers
• protocols and data formats

## Email: Basic Concepts

A *mailbox* is an area of storage ( a particular file or directory on disk) which contains messages sent to a particular email user.

An *email address* is a combination of a *mailbox* and a *domain name*.

Example:

simon@dcs.gla.ac.uk

the name of a mailbox (often, but not necessarily, the same as a login name)

the domain name of a computer which is able to receive email



## Email Addresses

Because it is convenient for all email addresses within an organisation to have a uniform format, it is usual for one particular computer to be designated as the *mail gateway*.

For example, my email address is simon@dcs.gla.ac.uk rather than simon@marion.dcs.gla.ac.uk

Our mail gateway is a computer called iona.dcs.gla.ac.uk, also known as mailhost.dcs.gla.ac.uk or just dcs.gla.ac.uk. All of these domain names resolve to the same IP address (130.209.240.35).

Email addressed to simon@dcs.gla.ac.uk is received by iona and placed in my mailbox, which is a directory called /users/fda/simon/Mail/inbox

(In fact, simon@marion.dcs.gla.ac.uk also works.)



## Format of an Email Message

An email message consists of *ASCII text* (see next slide). There are a number of *header lines* of the form *keyword* : *information* , then a blank line, then the *body* of the message.

The header lines *To: From: Date: Subject* are always included. Others are optional and many of them may be suppressed by email reading software.

Email software is free to add header lines which may or may not be meaningful to the software at the receiving end (e.g. some email software is able to request an acknowledgement message when a message is read, but not all receiving software understands this).



```
Return-Path: vv@di.fc.ul.pt
Return-path: <vv@di.fc.ul.pt>
Envelope-to: simon@dcs.gla.ac.uk
Delivery-date: Wed, 24 Apr 2002 09:48:42 +0100
Received: from mail.di.fc.ul.pt ([194.117.21.40] helo=titanic.di.fc.ul.pt)
          by iona.dcs.gla.ac.uk with esmtp (Exim 3.13 #1)
          id 170ISA-00001y-00
          for simon@dcs.gla.ac.uk; Wed, 24 Apr 2002 09:48:42 +0100
Received: from di.fc.ul.pt (dialup06.di.fc.ul.pt [194.117.22.76])
          by titanic.di.fc.ul.pt (8.9.3/8.9.3) with ESMTP id JAA17140
          for <simon@dcs.gla.ac.uk>; Wed, 24 Apr 2002 09:48:40 +0100
Received: (from vv@localhost)
          by di.fc.ul.pt (8.11.6/8.11.6) id g3O8oJM17793;
          Wed, 24 Apr 2002 09:50:19 +0100
Date: Wed, 24 Apr 2002 09:50:17 +0100
From: Vasco Thudichum Vasconcelos <vv@di.fc.ul.pt>
To: simon gay <simon@dcs.gla.ac.uk>
Subject: Re: Titles and Abstracts
Message-Id: <20020424095017.49de0cdc.vv@di.fc.ul.pt>
In-Reply-To: <200204230908.KAA11301@marion.dcs.gla.ac.uk.dcs.gla.ac.uk>
References: <20020320102141.273e7e41.vv@di.fc.ul.pt>
          <200204230908.KAA11301@marion.dcs.gla.ac.uk.dcs.gla.ac.uk>
Organization: DI/FCUL
X-Mailer: Sylpheed version 0.7.0 (GTK+ 1.2.10; i586-pc-linux-gnu)
Mime-Version: 1.0
Content-Type: text/plain; charset=US-ASCII
Content-Transfer-Encoding: 7bit

thanks.

vasco
```

information about the mail system of the sender (Return-Path through the Received lines)

useful to the human reader (Date, From, To, Subject)

the body of the message (in this case, dwarfed by the header!)

## Emails are Text

The body of an email is *text*: specifically, characters from the ASCII (American Standard Code for Information Interchange) character set, which uses a 7 bit format to represent 128 characters including upper and lower case letters, digits, punctuation symbols, and 33 control characters such as *newline*.

If non-text data (e.g. images, or binary data files produced by applications) are to be sent by email, they must be *encoded* using only printable ASCII characters.

(As email is a lowest common denominator of Internet access, it's convenient to be able to transmit data of all types by email, despite the existence of special-purpose applications/protocols such as FTP.)



## Encoding Non-Text Data

One way of encoding arbitrary binary data into printable ASCII would be to encode binary 0 as the character '0' (ASCII code 48) and binary 1 as the character '1' (ASCII code 49). This would have the unfortunate effect of multiplying the size of the data by 8, because each ASCII character is represented by one byte (8 bits) in a data packet.

A better way would be to think of a byte as a two digit hex number, and use the characters '0'… '9' and 'A'… 'F' to encode it. This would multiply the size of the data by 2, which is still not very good.

Based on the same idea, but using a greater proportion of the printable ASCII range, we get *base 64 encoding* which is commonly used with email.



## Base 64 Encoding

| | | | | |
|---|---|---|---|---|
| 10010010 | 01010001 | 11110100 | | 3 bytes = 24 bits |
| 100100 | 100101 | 000111 | 110100 | 4 blocks of 6 bits |
| 36 | 37 | 7 | 52 | numbers 0…63 = "digits" in base 64 |
| D | E | ' | T | add 32 to get the ASCII code of a printable character |

Each character is represented by 1 byte, so the data expansion is 8/6 = 33% larger (actually it's slightly worse because extra information is introduced in relation to line breaks).



## MIME and Attachments

Modern email applications provide *attachments*, a convenient way of combining text and non-text data (generally, several pieces of data, perhaps of various types and encoded in different ways) in a single message. This system is called MIME (Multipurpose Internet Mail Extensions).

The email application which creates the message uses the MIME format to specify what data is present and how it is encoded. The receiving email application has the opportunity to decode it (although there is no guarantee that it will know how to do so).

The next slide shows an example of the text form of a MIME-encoded message. Most email applications will hide the non-text data, and just present a button (for example) allowing the attached data to be extracted.

```
Mime-Version: 1.0
Content-Type: multipart/mixed; boundary="============_-1192028076==_============"
Date: Mon, 29 Apr 2002 16:33:39 +0100
To: simon@dcs.gla.ac.uk
From: Tania Sprott <tania@dcs.gla.ac.uk>
Subject: CS1Q Questionnaire

--============_-1192028076==_============
Content-Type: text/plain; charset="us-ascii"

Hi Simon

Attached list of CS1Q tutors for PowerPoint as discussed.  I'll be along
shortly with the questionnaires.

Tania

--============_-1192028076==_============
Content-Type: multipart/appledouble; boundary="============_-1192028076==_D============"

--============_-1192028076==_D============
Content-Transfer-Encoding: base64
Content-Type: application/applefile; name="%CS1Q-Groups.doc"
Content-Disposition: attachment; filename="%CS1Q-Groups.doc"

AAUWBwACAAAAAAAAAAAAAAAAAAAAAAAAAAQAAAADAAAASgAAAA8AAAAJAAAAWQAAACAA
AAAIAAAAeQAAABAAAAACAAAAiQAAAR5DUzFRLUdyb3Vwcy5kb2NXOEJOTVNXRAEAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
HlNPUlQHAgCAABwAHv//
--============_-1192028076==_D============
Content-Type: application/octet-stream; name="CS1Q-Groups.doc"
 ; x-mac-type="5738424E"
 ; x-mac-creator="4D535744"
Content-Disposition: attachment; filename="CS1Q-Groups.doc"
Content-Transfer-Encoding: base64

0M8R4KGxGuEAAAAAAAAAAAAAAAAAAAAAPgADAP7/CQAGAAAAAAAAAAAAAAADAAAANgAA
AAAAAAAAEAAAOQAAAAEAAAD+////AAAAADcAAAA8AAAAPQAAAP//////////////////
////////////////////////////////////////////////////////////////////
```



# Sending Email

Suppose I send an email to my colleague Vasco, vv@di.fc.ul.pt

There are several stages:

- using my email application, I compose a message and press “send”
- my email application becomes a client of the mail server on the department’s mail gateway machine, iona.dcs.gla.ac.uk, and specifies the destination address and the content of the message (*)
- iona (actually, the `sendmail` program running on iona) becomes a client of the mail gateway in Vasco’s department, mail.di.fc.ul.pt (194.117.21.40) and specifies the mailbox (vv) and the content (*)
- the message is delivered to Vasco’s mailbox directory

Steps (*) use SMTP, the Simple Mail Transfer Protocol.



# SMTP

The email application and the `sendmail` program live in layer 5 of the network hierarchy. Opening a connection to the destination mail gateway uses the same mechanism as opening a connection to a web server: TCP functions (layer 4) are called, which must resolve the domain name of the destination and send messages to establish the connection. The difference at this stage is that port 25 is used, instead of port 80.

SMTP specifies a completely different (and more complex) exchange of messages than HTTP.

For example, the first SMTP message is sent by the server, not the client (but remember that the first TCP message comes from the client side, to open the connection).



# Example SMTP Session

```
220-Welcome to the dcs.gla.ac.uk mail relay.
220-
220 iona.dcs.gla.ac.uk ESMTP Exim 3.13 Mon, 29 Apr 2002 18:10:42 +0100
>>> EHLO marion.dcs.gla.ac.uk.dcs.gla.ac.uk
250-iona.dcs.gla.ac.uk Hello root at marion.dcs.gla.ac.uk [130.209.241.152]
250-SIZE 26214400
250-PIPELINING
250 HELP
>>> MAIL From:<simon@dcs.gla.ac.uk> SIZE=37
250 <simon@dcs.gla.ac.uk> is syntactically correct
>>> RCPT To:<simon@dcs.gla.ac.uk>
250 <simon@dcs.gla.ac.uk> verified
>>> DATA
354 Enter message, ending with "." on a line by itself
>>>
>>> Hi there!
>>>
>>> .
250 OK id=172Efi-0005Dz-00
simon... Sent (OK id=172Efi-0005Dz-00)
Closing connection to mailhost.dcs.gla.ac.uk.
>>> QUIT
221 iona.dcs.gla.ac.uk closing connection
```

from client

## Receiving Email

If a user directly logs in to the computer whose disk holds the mailbox, then the email application simply looks at the mailbox.

(Similarly in the department, although the mailbox may live on a different computer on the local network.)

Alternatively, another protocol called POP (Post Office Protocol) allows a mailbox to be inspected remotely; this is typically used in situations where a user dials up (using a modem) to an email service provider.