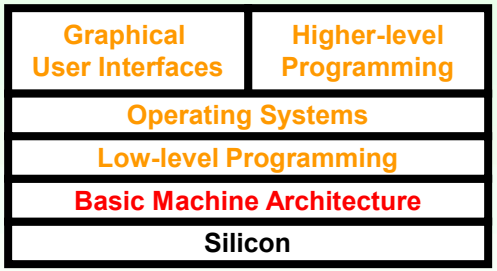
CSCU9V4 Systems

Systems Lecture 1
Binary Representation



Motivation…

- *Why study computer systems?*
  - Do you care about what's actually happening inside the box?
  - Should you?

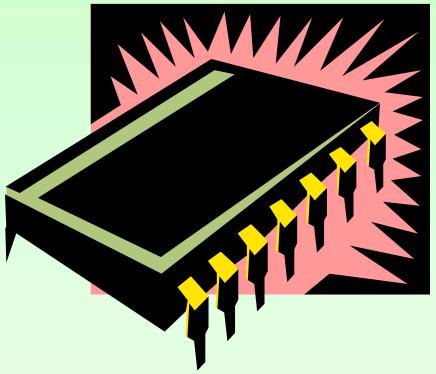
Many computer applications require knowledge of…

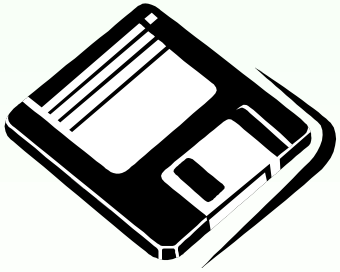
- precision of calculations
- colour and detail of pictures
- speed of calculation

- data transfer speeds
- what you control

Binary Representation

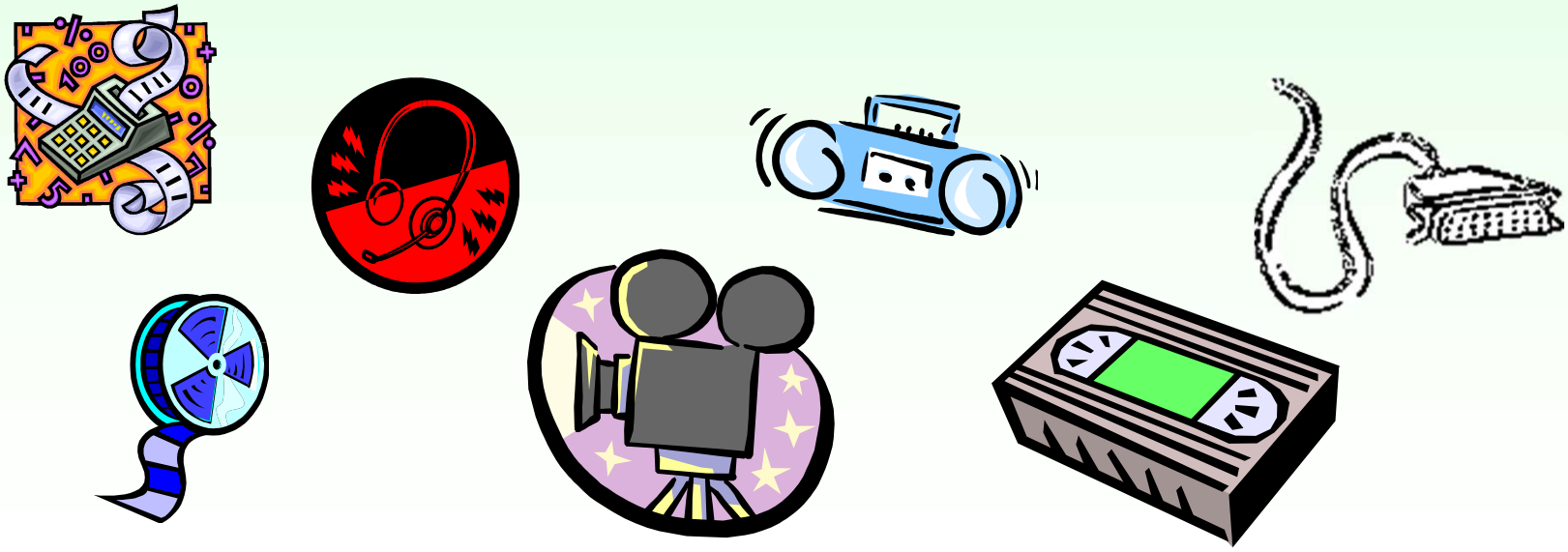
- In digital computers **everything** is coded as **binary digits, or bits**
- A bit is either
  - 1, or 0
- Alternative ways of thinking about bits:
  - 1 or ("on", "high", "true", "present", "yes", ...), *or*
  - 0 or ("off", "low", "false", "absent", "no", ...)
- reflects the available technology
  - **electronic: an electrical voltage at 1 of 2 voltage levels**
  - **magnetic: magnetic field is polarised in 1 of 2 directions**
  - **optical: a surface is reflective (light) or not (dark)**





Binary Representation

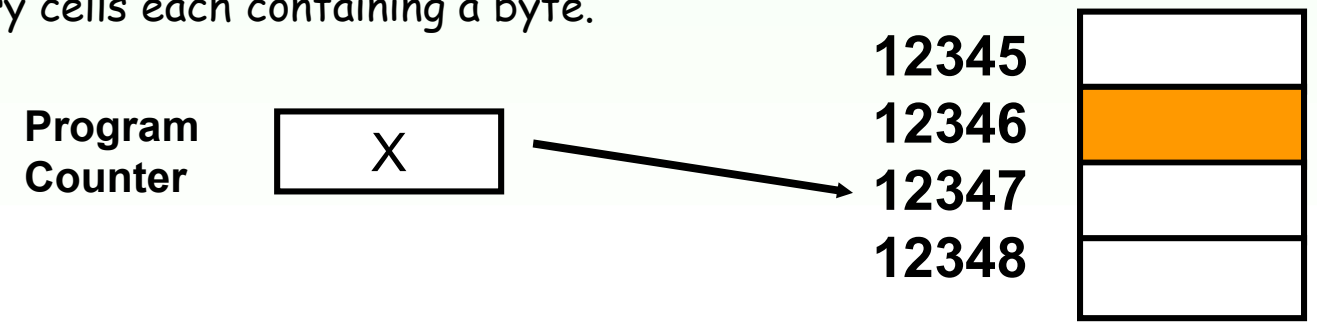
- **All** data (whether numbers, text, pictures, sounds, etc.) handled by digital computers is stored using bits.
- Binary representation seems initially unpromising (can we really store, say, music in on/off form?) but it has conquered the world:
  - Analogue LPs replaced by digital CDs and now files e.g. mp3
  - DVDs (Digital Video Discs) and files (eg mp4) have replaced video tapes
  - …and other examples too: phones, TV, radio, modems,…



Machine Data: nibbles, bytes & words

- We usually consider bits grouped together:
  - A *nibble* is 4 bits (half a byte)
  - A *byte* is 8 bits
  - A *word* is some number of bytes (usually 2,4, or 8)

    A word is the sequence of bits that the computer operates on at a time.
- Historically, different computers had different word lengths, and worked on different numbers of bits at a time: but now they operate almost universally on multiples of 8 bits, and we almost invariably think of a word as two, four or eight bytes
  - Most PCs in the university labs have 4-byte words (4x8 = 32 bits) or 8 byte (64-bit) words
- A good mental picture of computer memory is to imagine lots of numbered memory cells each containing a byte.



Representing Numbers

- *Very important*: how accurate is the result of your program?
- hence need to understand binary – so it helps to revisit decimal!
- How numbers are stored: think of an odometer (i.e. digital display) having only 0 and 1 on each wheel
- Comparing binary to decimal:

00	0000	08	1000
01	0001	09	1001
02	0010	10	1010
03	0011	11	1011
04	0100	12	1100
05	0101	13	1101
06	0110	14	1110
07	0111	. . .	

Understanding Decimal Numbers

- Think first about decimal ("base 10" numbers). Consider the decimal number **543**:
- This is shorthand for the sum:

$$\begin{aligned} 3 &= 3 \times 1 \\ +\,40 &= 4 \times 10 \\ +\,500 &= 5 \times 100 \end{aligned}$$

- But

$$1 = 10^0 \qquad (x^0 \text{ is always } 1)$$
$$10 = 10^1$$
$$100 = 10^2 \quad \text{etc}$$

- So we could write the sum as

$$\begin{aligned} 3 &= 3 \times 10^0 \\ +\,40 &= 4 \times 10^1 \\ +\,500 &= 5 \times 10^2 \end{aligned}$$

Understanding Binary Numbers

- We can view numbers in any base in this way. Consider binary ("base 2")
- Instead of numbers being written using 10^0,10^1 , 10^2 and so on, we use powers of 2

- Quick reference for powers of 2, converted to decimal:

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

➢ Memorise these... (preferably up to 2^{16}=65536)

A Binary Example

- Consider the binary number 1101

$$
\begin{aligned}
1 &= 1 \times 2^0 = 1 \times 1 \\
+\ 00 &= 0 \times 2^1 = 0 \times 2 \\
+\ 100 &= 1 \times 2^2 = 1 \times 4 \\
+\ 1000 &= 1 \times 2^3 = 1 \times 8
\end{aligned}
$$

- So 1101 binary is (8 + 4 + 1) decimal, i.e. 13

- A useful trick is to write the positions above the number: consider the binary number 10010100

  76543210
  10010100

- So this binary number is $2^7 + 2^4 + 2^2 = 128 + 16 + 4 = 148$ decimal
  - See why it's useful to memorise powers of 2?

Converting Decimal to Another Base

- To convert decimal to another base:
  - Repeatedly divide number by base
  - Then write down the remainders in reverse order

- For example, convert 13 to binary:

```
13 / 2 -> 6 remainder 1
 6 / 2 -> 3 remainder 0
 3 / 2 -> 1 remainder 1
 1 / 2 -> 0 remainder 1
```

- So 13 decimal is 1101 binary

All Bits Set

- If all (or nearly all) the digits in a binary number are 1 ("all bits set") it looks as though conversion to decimal is going to be hard
  - E.g. 1111 1111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
- But we can save a lot of work if we use the fact that

  1111 1111 + 1 = 1 0000 0000
- The number on the right is 2^8 (which is 256 decimal)
- In general, if we have all *n* bits set, then the value is $(2^n - 1)$
- This gives us an important fact: the largest number we can store in a given number of bits:

  in 8 bits : numbers up to $2^8 - 1 = 255$
  in 16 bits : numbers up to $2^{16} - 1 = 65,535$
  in 32 bits : numbers up to $2^{32} - 1 = 4,294,967,295$

Binary Usage in Programming Languages

Datatypes that you have met already in tutorials and practicals:

- **boolean** - 1 bit
  - Set to either true or false, depending on whether the bit is 1 or 0
- **int** - 32 bits
  - Can represent 2^{32} different integers, from -2,147,483,648 to 2,147,483,647
  - (We will see how to represent negative numbers later in the course using twos complement)
- **float** - 32 bits
  - Can represent floating point numbers (fractions) over a wide range
  - (We will see how to represent fractions later on too)

Binary Usage in Programming Languages

Some of the other primitive data types available in Java:

- **byte** - 8 bits (*of course*!)
- **short** - 16 bits
  - This is an integer type, with fewer integers representable
  - Can represent 2^{16} different integers, from -32768 to 32767
- **long** - 64 bits
  - An integer type for very big integers, if int won't do!
  - 2^{64} different integers, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
- **double** - 64 bits
  - Even more floating point numbers with even more accuracy

Aside: large number of bytes

Commonly used terms for counting bytes (and other things) are

- kilo (K) - meaning a thousand
- mega (M) - meaning a million
- giga (G) - meaning a thousand million (US: Billion)
- tera (T) – meaning a million million (US: Trillion)
- peta (P) – meaning a thousand million million

- However in computing as 2^{10} = 1,024, 1K means 1024, not 1000

- Similarly 1M is 2^{20} = 1,048,576 (not 1,000,000)
  1G is $2^{30} \cong$ 1,073 million (not 1,000 million)

1T is $2^{40} \cong$ 1,099,511 million (not 1 million million)

End of Lecture