Binary numbers

Internally, computers rely on electrical current to represent data and execute instructions to manipulate that data

As we will see in later lectures computers that store and manipulate data can do so through combinations of simple electronic circuits

These circuits and data storage are controlled by provision of current. This is not a particularly sophisticated mechanism

The current is either ON or OFF

This is a choice of two states - it is *binary*

Binary numbers

Rather than discuss the values stored in the computer in terms of ON/OFF we use a convenient number system

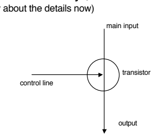
We conceptually think of the smallest manipulable value as one instance of an ON/OFF value

We call this a *bit*, the value of which can be ON or OFF

Binary numbers

Imagine that we want to store the value 1 (ON) or the value 0 (OFF) in the computer

We could represent this in terms of the required circuitry (we'll return to this circuit later - don't worry about the details now)



But such representations are difficult to deal with!

Binary numbers

We use an *abstraction*

The *binary* number system contains only two symbols: 1 and 0

In place of electrical current we can use these symbols to *abstractly* represent some internal state of the computer achieved through combinations of ON/OFF values

We can represent
- data
- somewhere to store the data
- instructions

by combining individual bits

Binary numbers

The binary number system is a *positional* number system

We are practiced at using the decimal number system which is also positional

eg in the number 333 the digits are the same but are interpreted based on their position

(3x100) + (3x10) + (3x1)

or

$(3x10^2) + (3x10^1) + (3x10^0)$

based on

100000	10000	1000	100	10	1
10^5	10^4	10^3	10^2	10^1	10^0

Binary numbers

Whereas the decimal system has ten digits 0,1,2,3,4,5,6,7,8,9 binary only has two 0 and 1

With positional values

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

so the binary number 10101 can be converted to a decimal value

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	0	1	0	1	0	1

$(1x2^4) + (0x2^3) + (1x2^2) + (0x2^1) + (1x2^0)$

= 16 + 0 + 4 + 0 + 1

= 21

Binary numbers

Now we can describe

- data
- locations for storing data
- instructions

by something other than circuits, yet retain a close relationship with the underlying notion of ON/OFF values used within the machine

Data representation

How is the data and instructions of our programs stored?

- in main memory
- in compartments called *bytes*

byte is basic unit of memory

- holds 1 piece of data or 1 piece of an instruction
- has a unique *address* to say where it is in memory
- byte is just a binary (base 2) number
- contains 8 bits where each bit is either 0 or 1

So, how is data stored using only bytes?

What can one byte store?

Eight bits

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
0	0	0	1	1	0	0	1

$1 \times 2^4 + 1 \times 2^3 + 1 \times 2^0$

16 + 8 + 1

25

Just like

9045 in decimal (base 10)

$9 \times 10^3 + 0 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$

9000 + 0 + 40 + 5

9045

What can one byte store?

From

00000000 (0 base 10)

to 11111111 (255 base 10)

So how can we have values > 255 in our programs?

Use more bytes!

k	k+1	← addresses
00000001	00100101	

$1 \times 2^8 + 1\times2^5 + 1\times2^2 + 1\times2^0$

so using *n* bits we can represent numbers from (check it out)

0 to 2^n-1

Binary addition

What happens when 2 numbers are added together?

- shall we convert to decimal, add them then convert back to binary?
- no need
- binary addition follows same principles as decimal, using carry digits/bits

eg carry when sum > 9

```
    1 6 3
+   1 5 9
    1 1
    -----
    3 2 2
```

eg carry when sum > 1

```
    0 1 0 1 0 0 1 1
+   0 0 0 1 0 1 0 1
        1   1 1 1
    ---------------
    0 1 1 0 1 0 0 0
```


Negative integers

So far we have only non-negative (*unsigned*) integers

In base 10 we use a minus sign

-213

In binary we use the most significant bit (MSB)

- if MSB is 1 number is negative
- if MSB is 0 number is positive
- other 7 bits represent magnitude

	2^6	2^5	2^4	2^3	2^2	2^1	2^0
	64	32	16	8	4	2	1
↑	0	0	1	1	0	0	1

- if this is 0 value is 25
- if this is 1value is -25

Called “sign-magnitude” form

Can represent -127 to 127

Sign-magnitude problems

Representing zero

0 0 0 0 0 0 0 0 = ?

1 0 0 0 0 0 0 0 = ?

Adding negative to positive

	0 0 1 0 1 1 0 1		45
+	1 0 0 1 1 0 0 0	+	- 24
	1 1 0 0 0 1 0 1		- 69

Need a better solution………..

2's complement

positive integers are in sign-magnitude form

negative numbers are equivalent positive number with all bits "flipped" (complemented), and 1 added to result

eg

	0 0 0 0 0 1 1 1	+7
flip	1 1 1 1 1 0 0 0	
add 1	1 1 1 1 1 0 0 1	-7

2's complement

45 + (-24)

+24	0 0 0 1 1 0 0 0	
flip	1 1 1 0 0 1 1 1	
add 1	1 1 1 0 1 0 0 0	-24

	0 0 1 0 1 1 0 1		45
+	1 1 1 0 1 0 0 0	+	- 24
	0 0 0 1 0 1 0 1		21

How do we know if result is negative or positive?

Interpreting 2's complement

If MSB is 0 number is positive, intepret normally

If MSB is 1 number is negative
- complement all bits
- add 1
- interpret as unsigned integer but remember value is negative

Try

4 - 7

10 - 18

What else?

What range of values can we represent in 2's complement?
- read Appendix G of the manual

Do we need to think about any other number bases?
- yes, in many situates base 8 (octal) and base 16 (hexadecimal) are important to consider
- Lecture 5
- read Appendix G of the manual