Computer Fundamentals: Number Systems

Dr Robert Harle

Today's Topics

- The significance of the bit and powers of 2
- Data quantities (B, kB, MB, GB, etc)
- Number systems (decimal, binary, octal , hexadecimal)
- Representing negative numbers (sign-magnitude, 1's complement, 2's complement)
- Binary addition (carries, overflows)
- Binary subtraction

So…

- What is a bit?

Binary Digit

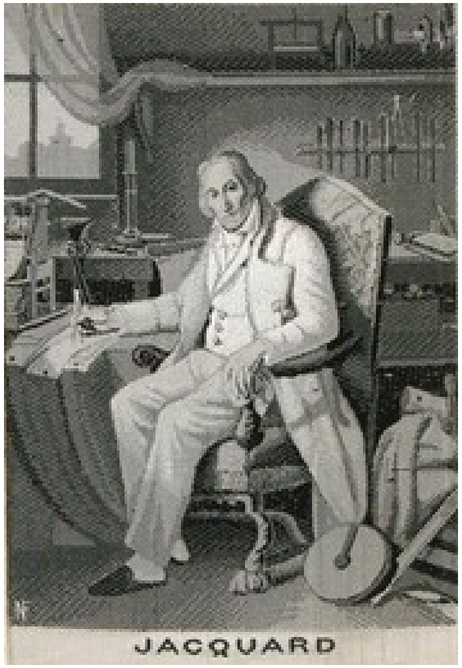
1 or 0

True or False

unit of information

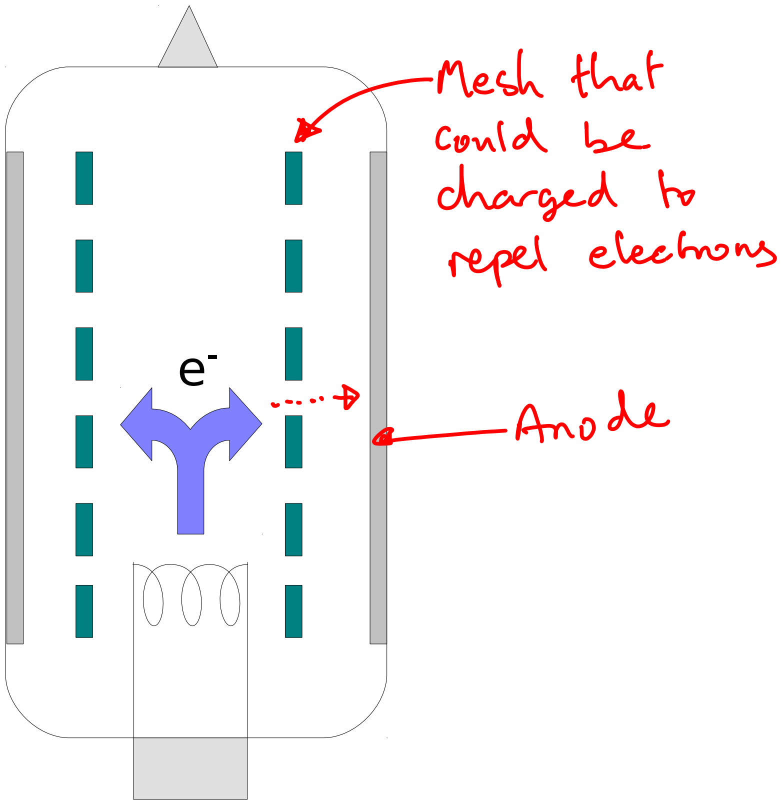
The Significance of the Bit

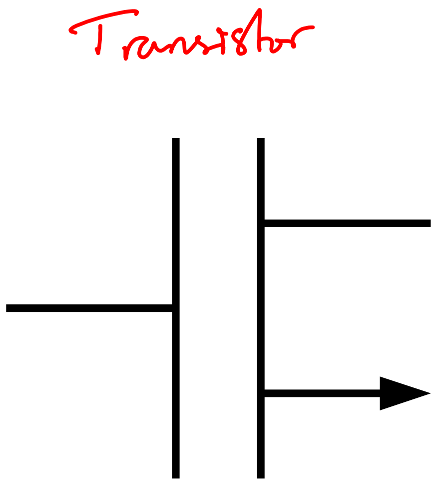
- A bit (Binary digIT) is merely 0 or 1
- It is a unit of ***information*** since you cannot communicate with anything less than two states
- The use of binary encoding dates back to the 1600s with Jacquard's loom, which created textiles using card templates with holes that allowed needles through



Bits and Computers

- The nice thing about a bit is that, with only two states, it is easy to embody in physical machinery
- Each bit is simply a switch and computers moved from vacuum tubes to transistors for this





Decimal Number System

- Most computers count in binary, which we can easily understand from the decimal so ingrained in us

35462

Decimal Number System

- Most computers count in binary, which we can easily understand from the decimal so ingrained in us

$2x10^0$

35462

Decimal Number System

- Most computers count in binary, which we can easily understand from the decimal so ingrained in us

$$6 \times 10^1 + 2 \times 10^0$$

35462

Decimal Number System

- Most computers count in binary, which we can easily understand from the decimal so ingrained in us

$$4x10^2 + 6x10^1 + 2x10^0$$

35462

Decimal Number System

- Most computers count in binary, which we can easily understand from the decimal so ingrained in us

$$5x10^3+4x10^2+6x10^1+2x10^0$$

$$35462$$

Decimal Number System

- Most computers count in binary, which we can easily understand from the decimal so ingrained in us

$$3\times10^4 + 5\times10^3 + 4\times10^2 + 6\times10^1 + 2\times10^0$$

$$35462$$

Binary

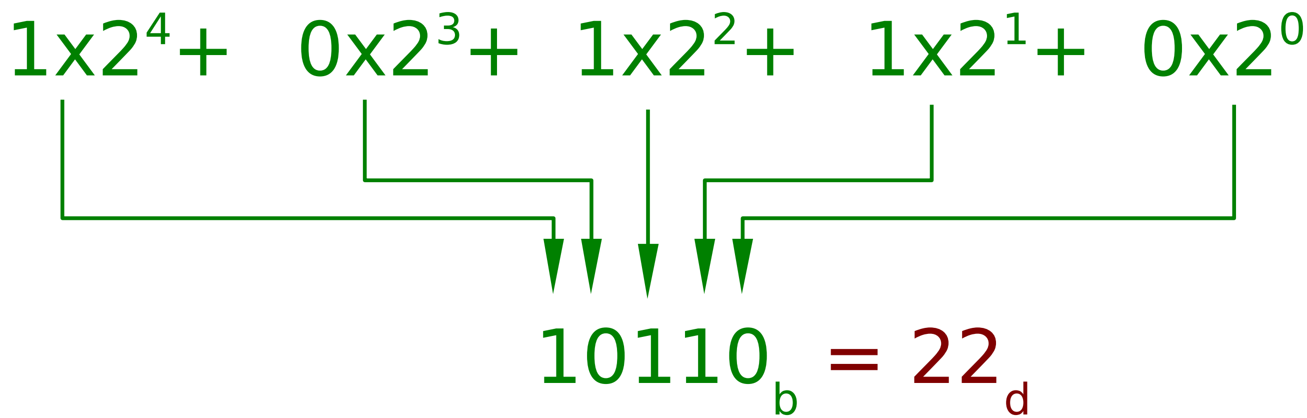
- Binary is exactly the same, only instead of ten digits/states (0 to 9) we have just two, so the base becomes 2:

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

$$10110$$

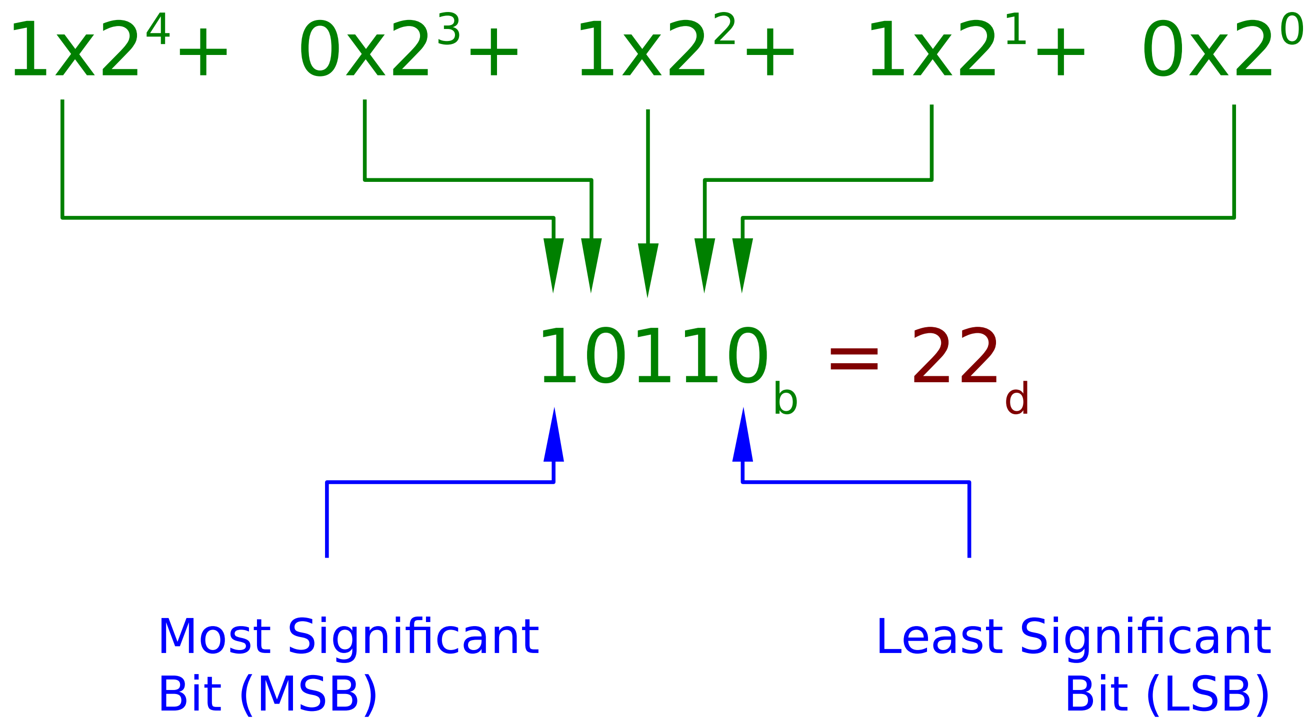
Binary

- Binary is exactly the same, only instead of ten digits/states (0 to 9) we have just two, so the base becomes 2:



Binary

- Binary is exactly the same, only instead of ten digits/states (0 to 9) we have just two, so the base becomes 2:



Works for Fractional Numbers too...

$$3x10^{1} + 5x10^{0} + 4x10^{-1} + 6x10^{-2} + 2x10^{-3}$$

$$35.462$$

Works for Fractional Numbers too...

$$3 \times 10^{1} + 5 \times 10^{0} + 4 \times 10^{-1} + 6 \times 10^{-2} + 2 \times 10^{-3}$$

$$35.462$$

$$1 \times 2^{1} + 0 \times 2^{0} + 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3}$$

$$10.110_{b} = 2.75_{d}$$

Check

11.011_b

Check

$$1 \times 2^{1} + \ 1 \times 2^{0} + \ 0 \times 2^{-1} + \ 1 \times 2^{-2} + \ 1 \times 2^{-3}$$

$$11.011_{b} = 3.375_{d}$$

Representable Numbers

- With d decimal digits, we can represent 10^d different values, usually the numbers 0 to (10^d-1) inclusive
- In binary with n bits this becomes 2^n values, usually the range 0 to (2^n-1)

- Computers usually assign a set number of bits (physical switches) to an instance of a type.
  - An integer is often 32 bits, so can represent positive integers from 0 to 4,294,967,295 incl.
  - Or a range of negative and positive integers...

Other Common Bases

- Higher bases make for shorter numbers that are easier for humans to manipulate. e.g. $6654733_d = 11001011000101100001101_b$
- We traditionally choose powers-of-2 bases because this corresponds to whole chunks of binary

Other Common Bases

- Higher bases make for shorter numbers that are easier for humans to manipulate. e.g. 6654733_d=$11001011000101100001101_b$
- We traditionally choose powers-of-2 bases because this corresponds to whole chunks of binary
- **Octal** is base-8 ($8=2^3$ digits, which means 3 bits per digit)
  - 6654733_d=$011\text{-}001\text{-}011\text{-}000\text{-}101\text{-}100\text{-}001\text{-}101_b$= 31305415_o

Other Common Bases

- Higher bases make for shorter numbers that are easier for humans to manipulate. e.g. 6654733_d=$11001011000101100001101_b$
- We traditionally choose powers-of-2 bases because this corresponds to whole chunks of binary
- **Octal** is base-8 ($8=2^3$ digits, which means 3 bits per digit)
  - 6654733_d=$011\text{-}001\text{-}011\text{-}000\text{-}101\text{-}100\text{-}001\text{-}101_b$= 31305415_o
- **Hexadecimal** is base-16 ($16=2^4$ digits so 4 bits per digit)
  - Our ten decimal digits aren't enough, so we add 6 new ones: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - 6654733_d=$0110\text{-}0101\text{-}1000\text{-}1011\text{-}0000\text{-}1101_b$=$658B0D_h$

Other Common Bases

- Higher bases make for shorter numbers that are easier for humans to manipulate. e.g. $6654733_d = 11001011000101100001101_b$
- We traditionally choose powers-of-2 bases because this corresponds to whole chunks of binary
- **Octal** is base-8 ($8=2^3$ digits, which means 3 bits per digit)
  - 6654733_d=011-001-011-000-101-100-001-101$_b$= 31305415_o
- **Hexadecimal** is base-16 ($16=2^4$ digits so 4 bits per digit)
  - Our ten decimal digits aren't enough, so we add 6 new ones: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
  - 6654733_d=0110-0101-1000-1011-0000-1101$_b$=658B0D$_h$
- Because we constantly slip between binary and hex, we have a special marker for it
  - Prefix with '0x' (zero-x). So 0x658B0D=6654733_d, 0x123=291_d

Bytes

- A byte was traditionally the number of bits needed to store a character of text
- A de-facto standard of 8 bits has now emerged
  - 256 values
  - 0 to 255 incl.
  - Two hex digits to describe
    - 0x00=0, 0xFF=255
- Check: what does 0xBD represent?

Bytes

- A byte was traditionally the number of bits needed to store a character of text
- A de-facto standard of 8 bits has now emerged
  - 256 values
  - 0 to 255 incl.
  - Two hex digits to describe
    - 0x00=0, 0xFF=255
- Check: what does 0xBD represent?
  - B → 11 or 1011
  - D → 13 or 1101
  - Result is $11 \times 16^1 + 13 \times 16^0$ = **189** or 10111101

Larger Units

- Strictly the SI units since 1998 are:
- Kibibyte (KiB)
  - 1024 bytes (closest power of 2 to 1000)
- Mebibyte (MiB)
  - 1,048,576 bytes
- Gibibyte (GiB)
  - 1,073,741,824 bytes

Larger Units

- Strictly the SI units since 1998 are:
- Kibibyte (KiB)
  - 1024 bytes (closest power of 2 to 1000)
- Mebibyte (MiB)
  - 1,048,576 bytes
- Gibibyte (GiB)
  - 1,073,741,824 bytes
- but these haven't really caught on so we tend to still use the SI Kilobyte, Megabyte, Gigabyte. This leads to lots of confusion since technically these are multiples of 1,000.

The Problem with Ten

Decimal Division

$\frac{9}{11}$

$$
\begin{array}{r}
0.818181 \\
11\,\overline{)\,9.000000000} \\
90 \\
20 \\
90 \\
20 \\
90 \\
20 \\
\ldots
\end{array}
$$

$= 0.\overline{81}$

Binary Division

$\frac{1}{10_d}$

$$
\begin{array}{r}
0.000110011 \\
1010\,\overline{)\,1.0000000000} \\
10000 \\
1100 \\
10000 \\
1100
\end{array}
$$

$= 0.0\overline{0011}$

$\Rightarrow$ it takes ∞ bits to represent $\frac{1}{\text{ten}}$ exactly!

Unsigned Integer Addition

- Addition of unsigned integers works the same way as addition of decimal (only simpler!)
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 0 = 1
  - 1 + 1 = 0, carry 1
- Only issue is that computers have fixed sized types so we can't go on adding forever...

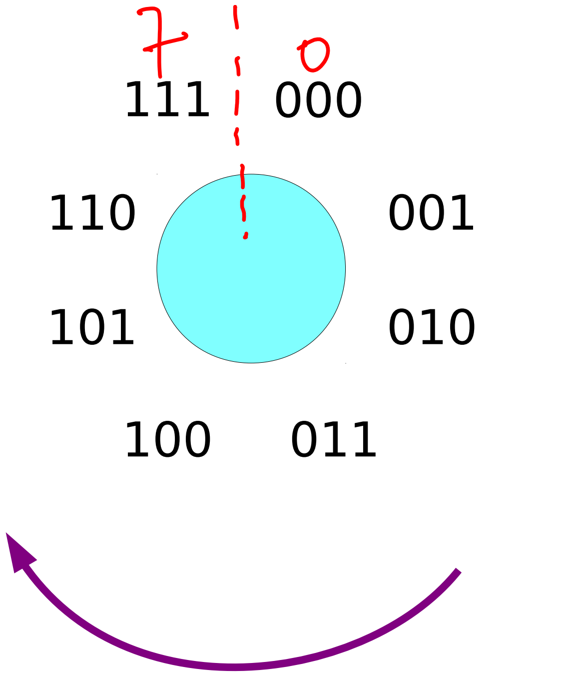
```
 0 0 1
   001
 + 001
 -----
   010

Carry flag: 0
```

```
 1 1 1
   111
 + 001
 -----
   000

Carry flag: 1
```

Modulo or Clock Arithmetic



000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

- Overflow takes us across the dotted boundary
  - So 7+1=0 (overflow)
  - We say this is (7+1) mod 8

Negative Numbers

- All of this skipped over the need to represent negatives.
- The naïve choice is to use the MSB to indicate +/-
  - 1 in the MSB → negative
  - 0 in the MSB → positive

$$-7 = 1111$$

Negative

Normal positive representation of 7

- This is the **sign-magnitude** technique

Difficulties with Sign-Magnitude

- Has a representation of minus zero (1000_2=-0) so wastes one of our 2^n labels
- Addition/subtraction circuitry must be designed from scratch

$$
\begin{array}{r}
1101 \\
+\ \underline{0001} \\
1110
\end{array}
$$

Our unsigned addition alg.

Difficulties with Sign-Magnitude

- Has a representation of minus zero (1000_2=-0) so wastes one of our 2^n labels
- Addition/subtraction circuitry must be designed from scratch

1101	+13
+ 0001	+1
1110	+14

Hardware output

Unsigned interpretation

Our unsigned addition alg.

Difficulties with Sign-Magnitude

- Has a representation of minus zero (1000_2=-0) so wastes one of our 2^n labels
- Addition/subtraction circuitry must be designed from scratch

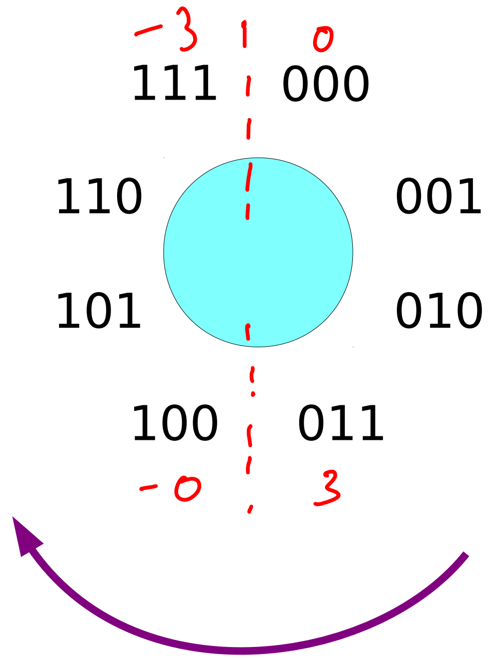
-5	1101	+13
+1	+ 0001	+1
-6	1110	+14

Sign-mag interpretation

Unsigned interpretation

Our unsigned addition alg.

Alternatively...



000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3

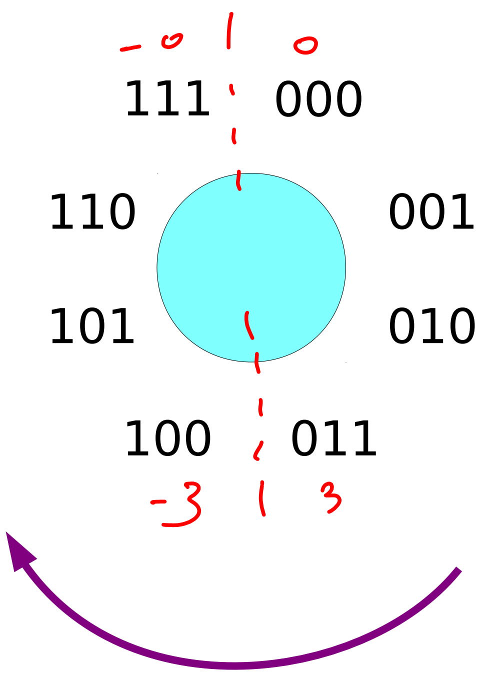
Increasing

Increasing

- Gives us two discontinuities and a reversal of direction using normal addition circuitry!!

Ones' Complement

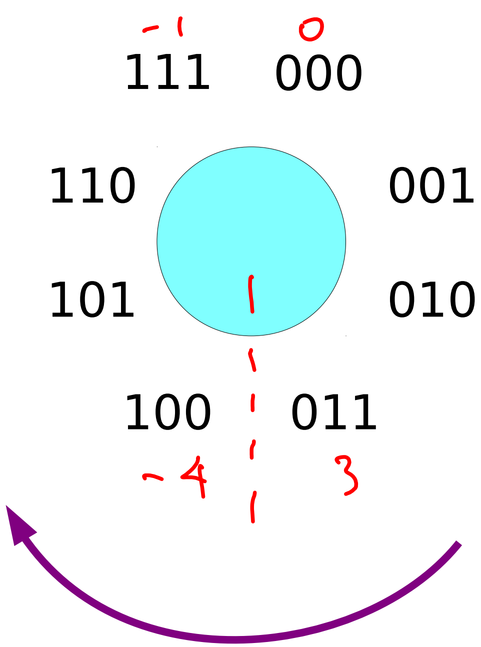
- The negative is the positive with all the bits flipped
- 7 → 0111 so -7 → 1000
- Still the MSB is the sign
- One discontinuity but still -0   :-(



000	0	Increasing
001	1	
010	2	
011	3	
100	-3	Increasing
101	-2	
110	-1	
111	-0	

Two's Complement

- The negative is the positive with all the bits flipped and 1 added (the same procedure for the inverse)
- 7 → 0111 so -7 → 1000+0001 → 1001
- Still the MSB is the sign
- One discontinuity and proper ordering



One discontinuity
addition moves round clockwise

⇒ Same as unsigned!

⇒ should be able to use the same circuitry

000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1

Two's Complement

- Positive to negative: Invert all the bits and add 1

  0101 (+5) → 1010 → 1011 (-5)

- Negative to positive: Same procedure!!

  1011 (-5) → 0100 → 0101 (+5)

Signed Addition

- ...it just works with our addition algorithm!

$$\begin{array}{rr} 1101 & +13 \\ +0001 & +1 \\ \hline 1110 & +14 \end{array}$$

Unsigned

Our unsigned addition alg.

Signed Addition

- ...it just works with our addition algorithm!

-3	1101	+13
+1	+0001	+1
-2	1110	+14

2's-comp

Unsigned

The result (in terms of bits) is the same. The <u>interpretation</u> of the result differs of course

Our unsigned addition alg.

Signed Addition

- ...it just works with our addition algorithm!

-3	1101	+13
+1	+0001	+1
-2	1110	+14

2's-comp

Unsigned

Our unsigned addition alg.

Signed Addition

- So we can use the same circuitry for unsigned and 2s-complement addition :-)
- Well, almost.

$$\begin{array}{rl} 0100 & +4 \\ +0100 & +4 \\ \hline 1000 & +8 \end{array}$$

Carry flag: 0

Unsigned

Signed Addition

- So we can use the same circuitry for unsigned and 2s-complement addition :-)
- Well, almost.

+4	0100	+4
+4	+0100	+4
-8	1000	+8
2's-comp		Unsigned

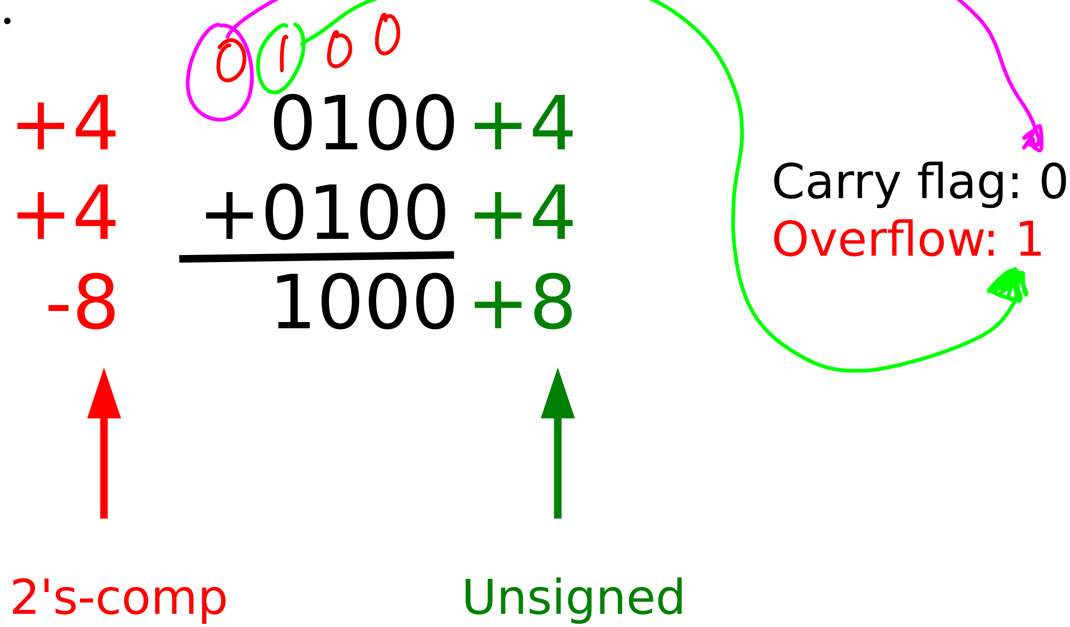
Carry flag: 0

This is because our overflow discontinuity occurs in a different place (i.e. 6 'o' clock not 12 'o' clock)

- The problem is our MSB is now signifying the sign and our carry should really be testing the bit to its right :-(

Signed Addition

- So we can use the same circuitry for unsigned and 2s-complement addition :-)
- Well, almost.



- The problem is our MSB is now signifying the sign and our carry should really be testing the bit to its right :-(
- So we introduce an overflow flag that indicates this problem

Integer subtraction

- Could implement the "borrowing" algorithm you probably learnt in school
- But why bother? We can just <u>add</u> the 2's complement instead.

```
  0100          1100
- 0011    →     0100
                +1101
                -----
                 0001
```

Flags Summary

- When adding/subtracting
  - **Carry** flag → overflow for **unsigned** integer
  - **Overflow** flag → overflow for **signed** integer
- The CPU does *not* care whether it's handling signed or unsigned integers
  - Down to our compilers/programs to interpret the result

Fractional Numbers

- Scientific apps rarely survive on integers alone, but representing fractional parts efficiently is complicated.
- Option one: **fixed point**
  - Set the point at a known location. Anything to the left represents the integer part; anything to the right the fractional part
  - But where do we set it??
- Option two: **floating point**
  - Let the point 'float' to give more capacity on its left or right as needed
  - Much more efficient, but harder to work with
  - Very important: more in Numerical Methods course