Binary Arithmetic and Floating point

1 Binary Addition

$$\begin{array}{r} 111 \\ 11100 \\ +\ \ 01110 \\ \hline 101010 \end{array}$$

2 Overflow

Suppose the accumulator in your CPU is an 8 bit register
It has 11010010 in it
You execute the instruction ADD 01010000
What happens?

$$\begin{array}{r} 11010010 \\ +\,01010000 \\ \hline 100100010 \end{array}$$

The answer **doesn't fit** in the register. This should trigger a flag in the **status register**, but can cause errors

3 Binary Multiplication

$$\begin{array}{r} 11100 \\ \times\,01110 \\ \hline 00000 \\ 111000 \\ 1110000 \\ 11100000 \\ 000000000 \\ \hline 110001000 \end{array}$$

This can be efficiently accomplished with **left-shift** and **add** operations

4 Negative Numbers

How can we represent ***negative numbers using only bits?***
Common Solutions:

4.1 Signed Magnitude Representation

- Add a single bit flag: 0 for positive or 1 for negative
  - **0**000 0110=6
  - **1**000 0110=-6 (Not 134)
- Similar in concept to a minus sign
- Have two values for 0: 1000 0000 and 0000 0000
- Makes binary arithmetic messy

4.2 Ones Compliment

- The negative of a number is represented by flipping each bit
- For example 0100 1001=65 becomes 1011 0110=-65
- The higher order bit still indicates the sign of the number
- Still has two representations for zero: 00000000 and 11111111
- Makes binary addition a bit simpler
- Due to this method, only get 7 bits of data in a byte

4.3 Twos Compliment

- A negative number is obtained by flipping each bit and adding 1
- For example 0100 1001=65 becomes 1011 0111=-65
- The higher order bit still indicates the sign of the number
- One representation for 0: 00000000
- Makes binary arithmetic much simpler
- Allows counting by addition in the way you would expect

4.4 Add a bias

- For k-bit numbers add a bias of $2^{k-1}-1$ then store in normal binary (So for 8-bit add $2^7-1=127$
- Can store numbers between $-(2^{k-1}-1)$ and 2^{k-1} (-127 and 128)
- For example -65 stored as -65+127=62 becomes 0011 1110
- The higher order bit **does not indicate** the sign of the number in the normal way
- Used in storing floating point numbers for some reason

4.5 More on Twos compliment

We will stick with **twos compliment**
We need to be careful about how many bits we are using to represent a number:
In this method, the leading zeros are important, and so cannot be ignored

4 Bits:	$3_{10}=0011_2$	$-3_{10}=1101_2$
8 Bits:	$3_{10}=00000011_2$	$-3_{10}=11111101_2$

Subtracting is now the same as adding: 10-3=10+(-3)
$10_{10}=00001010_2, 3_10=00000011_2$
$00001010-00000011=00001010+11111101=100000111$ (Overflow so 00000111)

Note that 10000000 is it's own negative, but is taken to be -128

5 Floating point representation

Sometimes we need to deal with numbers outside the usual range:

Floating point is very like scientific notation
The typical floating point representation has three fields:

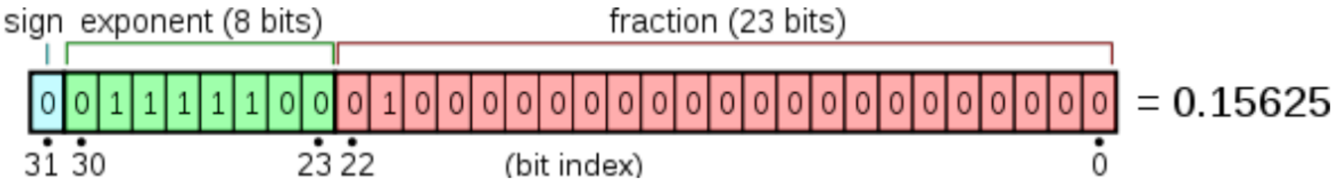
The sign bit	S
The exponent	e
The mantissa	M (Also called the significand)

These are used to represent the number:

$$+ \text{ or } - M\times 2^e$$

Single precision (32 bit) floating point numbers have:

- 1 bit sign
- 8 bit exponent
- 23 bit mantissa



5.1 The sign bit, S

0 indicates a positive number
1 indicates a negative number

5.2 The exponent, e

Value in range -126 to 127
Stored with a **bias**: 127 is added giving a number between 1 and 254

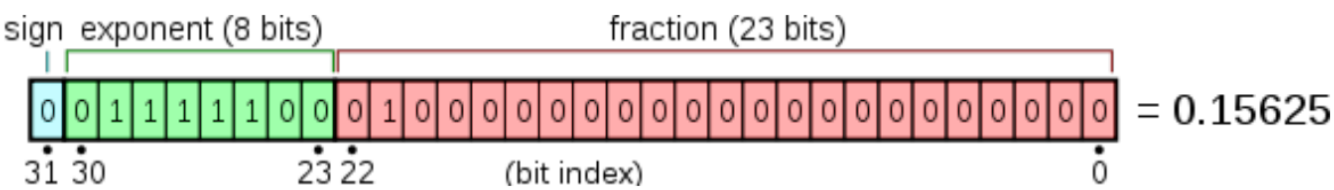
The 8 bit exponent field can store values in the range 0 to 255, but 0 and 255 have **special meanings**:

- Exponent field 0 with mantissa 0 gives the number 0
- Exponent field 0 with a non-zero mantissa: ”subnormal numbers” - below the threshold of numbers normally dealt with in floating point representation
- Exponent field 255 with mantissa 0 gives + or - infinity
- Exponent field 255 with non-zero mantissa: not a number

5.3 The mantissa, M

Some binary number like 1.10101010110
Always scaled so that the radix point is after the leading 1
Hence we need not store the leading 1 (we can assume it is there)
We only store 23 bits of the fractional part

5.4 Example



- Sign 0 - a positive number
- Exponent field is 124, so e is 124-127=-3
- Mantissa field is 010... so the actual mantissa is calculated to 1.25
- $1.25 \times 2^{-3} = 1.25/8 = 0.15625$

5.5 Example 2

-12.375
$12.375_{10} = 1100.011_2$
$1100.011 = 1.100011 \times 2^3$

- Sign is 1 to represent a negative
- Mantissa is 1.100011, we will store 100011000...
- Exponent is 3, we will store $130_{10} = 10000010_2$ after adding the bias of 127. Exponent is 3 to shift the radix point 3 places to the left so that the number starts 1. etc

11000001010001100000000000000000

5.6 More on floating point

5.6.1 Error in truncation

What is the binary FP representation of 0.1_{10}?
$0.1_{10} = 0.00011001100110011001100110011\ldots_2$
So the FP has e = -4; M = 1.10011001100110011001101 (limited to 23 digits)
which is actually 0.100000001490116119384765625, this is a **rounding error**.

5.6.2 Error in overflow/underflow

Minimum positive number is 2^{-126}, the **underflow level**.
Maximum positive number is $(2 - 2^{-23}) \times 2^{127}$, the **overflow level**.

Floating Point Operations should return the closest FP number to the answer. E.g. $1.1 \times 2^{123} - 1.10101 \times 2^{-23} = 1.1 \times 2^{123}$. In this the number being subtracted is too small to make a difference to such a large number