

Section 13.3 - Floating-point Numbers, Representation and Manipulation

Layer 6: High-Order Language

Syllabus Content Section 13: Data Representation

S13.3.1 Describe the format of binary floating-point real numbers

- Use two's complement form
- Understand of the effects of changing the allocation of bits to mantissa and exponent in a floating-point representation

Floating-point representation: a representation of real numbers that stores a value for the mantissa and a value for the exponent

With floating point, the number of bits you use to store your number are the same, but where you put the binary point can move

| Some terms and formula for Floating Point Number

Sign: Positive or negative

Mantissa / **Significand**: The value of the number

Base / **Radix**: How many numbers are in the number system (binary base 2, decimal base 10)

Exponent / **Exrad**: Number used for power of.

$+3.16 \times 10^4$
 -1011×2^{-2}
 $\pm M \times R^E$

S13.3.2 Convert binary floating-point real numbers into denary and vice versa

| Conversion

Calculate the normalised binary number for -3.75. Show your working

- $-3.75 = 100.01000 // -4 + 1/4 // -4 + 0.25$
- 100.01000 becomes 1.0001000 Exponent=+2
- Answer: Mantissa=1.0001000 Exponent=0010

Calculate the normalised floating-point representation of +1.5625 in this system (12bit-mantissa, 4bit-exponent). Show your working


- Correct conversion to binary: 01.1001
- Correct calculation of the exponent: 1
- Answer: Mantissa=0110 0100 0000 | Exponent= 0001

S13.3.3 Normalise floating-point numbers

- Understand the reasons for normalisation

Normalisation

Why binary/floating-point numbers are stored in normalized form

- To store the maximum range of numbers in the minimum number of bits
- Normalisation minimizes the number of leading zeros/ones represented
- Maximizing the number of significant bits // maximizing the number of precision/accuracy with given number of bits
- Enable large/small numbers to be stored with accuracy
- Avoids the possibility that many numbers have multiple representation
- 
- There will be a unique representation for a number
- The format will ensure it will be represented with greatest possible accuracy
- Multiplication is performed more accurately

Problems that can occur when a floating-point number is not normalised

- Lost of precision
- Redundant leading zeros in the mantissa
- Lost of the least-significant bits (bits on the right-hand end)
- Multiple representation of a single number

S13.3.4 Show understanding of the consequences of a binary representation only being an approximation to the real number it represents (in certain cases)

- Understand how underflow and overflow can occur

OVERFLOW = When the number that one gets as a result of some arithmetic operation on two n-bit binary numbers (signed or unsigned) is larger in magnitude than the largest number one can represent using n-bits.

When your answer is bigger than what the number of bits you have can represent

UNDERFLOW = When the number you get is smaller than you can represent

S13.3.5 Show understanding that binary representations can give rise to rounding errors

Rounding Errors

Many fractional numbers can't be represented exactly using floating point format. After all, we're using a finite number of bits to try and represent an infinite amount of numbers. This will inevitably lead to rounding errors - your computer will always round numbers to the closest representable number.

For example 0.1 can't be exactly represented in binary (feel free to try and make 0.1 using floating point format). Python usually hides this fact for us out of convenience, but here is *"the real 0.1"*:

```
>>> f"{0.1:.60f}"  
'0.100000000000000005551115123125782702118158340454101562500000'
```

So 0.1 is actually represented as a number slightly bigger than 0.1! I wrote a function to work out if a number is stored exactly or inexactly, and to show the rounding error: