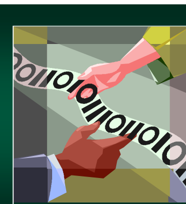
Set Theory in Computer Science

Part 3B

Binary Numbers



Bit of This and a Bit of That

What is a Number?

- We use the Hindu-Arabic Number System
  - positional grouping system
  - each position is a power of 10
- Binary numbers
  - based on the same system
  - use powers of **2** rather than 10
  - each digit is in the set { 0, 1 }

Base 10 Number

The number 1783 is ...

10^4	10^3	10^2	10^1	10^0
10000	1000	100	10	1
0	1	7	8	3

1000 + 700 + 80 + 3 = 1783

Binary Number Example

The number 0110 1001 is ...

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	1	1	0	1	0	0	1

64 + 32 + 8 + 1 = 105

Binary Number Example

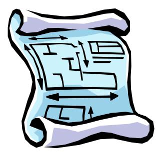
The number 1101 1011 is ...

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

128 + 64 + 16 + 8 + 2 + 1 = 219

Numbers are Tuples

- In Hindu-Arabic system, the order of the symbols is important – so they are tuples
- e.g. 123 ≠ 321
- Other number styles use sets – i.e. the ancient Egyptian system



Looking at Numbers

- Numbers are tuples 1947 ≠ 1974
- Members of the decimals number are also members of the set {0, 1, 2, … 9}

```
1947 → (1,9,4,7)
```


Looking at Binary Numbers

- Binary numbers are tuples 10010100 ≠ 11100000
- Members of the binary number are also members of the set {0, 1}

```
10100111 → (1,0,1,0,0,1,1,1)
```


Looking at Binary Numbers

- So, for a binary number B, all $x \in B$ holds the following: $x \in \{0, 1\}$

```
10100111 → (1,0,1,0,0,1,1,1)
```

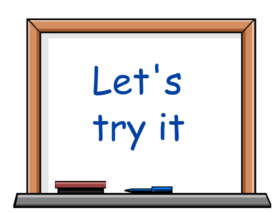

So….

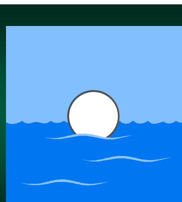
```
{1776, 1846, 1947} →

{ (1,7,7,6), (1,8,4,6)
  (1,9,4,7) }
```


Let's Make a Set-Based System

- We are mostly used to tuple-based number systems
- But, for most of history, people used sets
- Let's create one



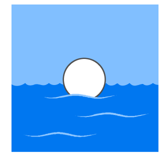


Floating Point Numbers

Real numbers are *real* complex

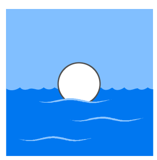
Floating Point Numbers

- Often, programs need to perform mathematics on *real* numbers
- *Floating point numbers* are used to represent quantities that cannot be represented by integers



Floating Point Numbers

- Why?
  - regular binary numbers can only store whole positive and negative values
  - many numbers outside the range representable within the system's bit width (too large/small)



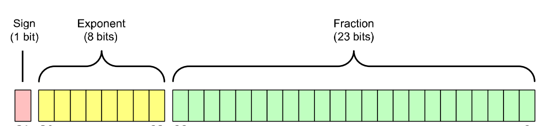
IEEE 754

- Practically modern computers use the *IEEE 754 Standard* to store floating-point numbers
- Represent by a mantissa and an exponent
  - similar to scientific notation
  - the value of a number is: $mantissa \times 2^{exponent}$
  - uses signed magnitude

IEEE 754

- Comes in three forms:
  - single-precision: 32-bit
  - double-precision: 64-bit
  - quad-precision: 128-bit
- Also supports special values:
  - negative and positive *infinity*
  - and "not a number" for errors (e.g. 1/0)

IEEE 754 Single Precision (32 bit)



Fractional Field

- The fraction field number that represents part of the mantissa
- If a number is in proper scientific notation…
  - it always has a single digit before the decimal place
  - for decimal numbers, this is 1..9 (never zero)
  - for base-2 numbers, it is always 1

Fractional Field

- So, do we need to store the leading 1? It will always be a 1
- The faction field, therefore…
  - only represents the fractional portion of a binary number
  - the integer portion is assumed to be 1
  - this increases the number of significant digits that can be represented (by not wasting a bit)

Exponent Field

- The exponent field supports negative and positive values but does not use sign-magnitude or 2's complement
- Uses a "biased" integer representation
  - fixed value is added to the exponent *before* storing it
  - when interpreting the stored data, this fixed value is then subtracted

Exponent Field

- Bias is different depending on precision
  - single precision: 127
  - double precision: 1023
  - quad precision: 16383
- For example, for single precision…
  - exponent of 12 stored as: (+12 + 127) = 139
  - exponent of -56 stored as: (-56 + 127) = 71

Interpretation: Normal Case

- Exponent Field: not all 0's or all 1's
- Fraction Field: Any

$$\pm\ (1.\mathit{fraction}) \times 2^{(\text{exponent} - \text{bias})}$$

Interpretation: Zero

- Exponent Field: all 0's
- Fraction Field: all 0's

0

Interpretation: Tiny Numbers

- Exponent Field: all 0's
- Fraction Field: Any

$\pm\ (0.\mathit{fraction}) \times 2^{(1\ -\ \text{bias})}$

Interpretation: Infinity

- Exponent Field: All 1's
- Fraction Field: 0

± ***infinity***

Interpretation: Invalid Numbers

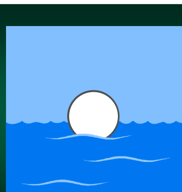
- Exponent Field: All 1's
- Fraction Field: Not 0

Not a number (NaN)

Interpretation: Invalid Numbers

NaN → **1 / 0**

Naan → 



Let's Encode Some Numbers!

This is actually fun!

Something Else About Numbers…

The number 36.74 is ...

10^2	10^1	10^0	10^{-1}	10^{-2}
100	10	1	1/10	1/100
0	3	6	7	4

$= (3\times10) + (6\times1) + 7/10 + 4/100$

Binary Fractions!

5 and 3 / 8 is ...

2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
16	8	4	2	1	1/2	1/4	1/8
0	0	1	0	1	0	1	1

= 4 + 1 + 1/4 + 1/8 = 101.011

Let's Encode 14.25 (32 bit)

- First, we need to convert 14.25 to its binary equivalent
- So, we need to calculate the integer part and fraction part of the number
- Everything after the decimal is a fraction
  - 0.25 is actually 25 / 100
  - we need to find the base 2 equivalent (1/4)

Step 1: Convert to binary

```
14 → 1110

0.25 → 1/4 → 0.01

Hence:
14.25 → 1110.01
```

binary 01 / 100

Step 2: Scientific Notation

- IEEE stores the data in scientific notation
- So we move the "binary point" over

$1110.01 \rightarrow 1.11001 \times 2^3$

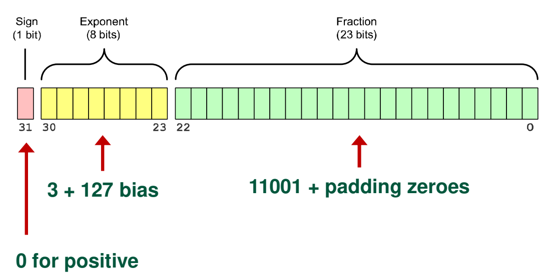
Step 2: Scientific Notation

- In binary scientific notation, the leading digit is always going to be 1
- Why store it? IEEE doesn't.
- Only data after the point is encoded

Fraction

$1.11001 \times 2^3 \rightarrow (1 + .11001) \times 2^3$

Step 3: Encode



Result: 14.25 (32 bit)

- The following is the encoded version of 14.25
- The rules are similar for double-precision

```
0 10000010 11001000000000000000000
```


Result: 14.25 (32 bit)

- We can also convert this into bytes
- Note: the floating point fields don't really "fit" cleanly into actual bytes

```
01000001 01100100 00000000 00000000
   41       64       00       00
```


Example 2: Encode 13.75 (32 bit)

- First, we need to convert 13.75 to its binary equivalent
- So, we need to calculate the integer part and fraction part of the number
- Everything after the decimal is a fraction
  - 0.75 is actually 75 / 100
  - we need to find the base 2 equivalent (3/4)

Step 1: Convert to binary

```
13 → 1101

0.75 → 3/4 → 0.11
```

binary 11 / 100

```
Hence:
13.75 → 1101.11
```


Step 2: Scientific Notation

- IEEE stores the data in scientific notation
- So we move the "binary point" over

$1101.11 \rightarrow 1.10111 \times 2^3$

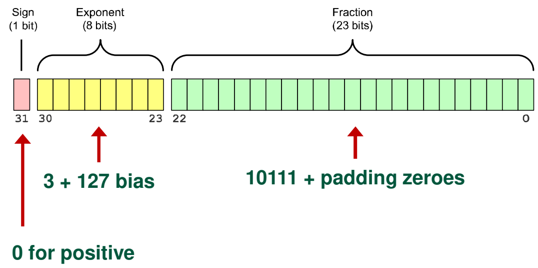
Step 2: Scientific Notation

- In binary scientific notation, the leading digit is always going to be 1
- Why store it? IEEE doesn't.
- Only data after the point is encoded

Fraction

$1.10111 \times 2^3 \rightarrow (1 + .10111) \times 2^3$

Step 3: Encode



Result: 13.75 (32 bit)

- The following is the encoded version of 13.75
- The rules are similar for double-precision

```
0 10000010 10111000000000000000000
```


Result: 13.75 (32 bit)

- We can also convert this into bytes
- Note: the floating point fields don't really "fit" cleanly into actual bytes

```
01000001 01011100 00000000 00000000
   41       5C       00       00
```


More Single-Precision Examples

```
Zero:
0 00000000 00000000000000000000000

Positive Infinity:
0 11111111 00000000000000000000000

Negative Infinity:
1 11111111 00000000000000000000000
```


Tuples and Floats

Its all set theory, folks!

Floats Are Tuples

- Just like regular binary numbers, floating-point numbers of tuples
- They consist of three fields making them 3-tuples

```
(sign, exponent, fraction)
```

Encoding of 14.25

- Sign is a 1-tuple
- Exponent is a 8-tuple
- Fraction is a 23-tuple

```
0 10000010 11001000000000000000000
```


Set Notation of 14.25

```
(  (0),
   (1,0,0,0,0,0,1,0),
   (1,1,0,0,1,0,0,0,
    0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0)    )
```