Base 10 Number Representation
- Natural representation for human transactions

Binary
- Hard to implement electronically
- Hard to store
- Hard to transmit

To transform a into as 11.100000000000000...

Reliably transmitted on noisy and inaccurate wires

- Easy to store with bistable elements
- Reliably transmitted on noisy and inaccurate wires

Examples
- Represent 3.5 as 1.0011001100110011...
- Binary floating point number cannot exactly represent 1.20

Leading bit is the sign bit
- Transformation
  - To transform a into -a, invert all bits in a and add 1 to the result

Range is: $-2^{N-1} < i < 2^{N-1} - 1$

Tmin = $2^{N-1} - 1$

Tmax = $2^{N-1} - 1$

Advantages:
- Operations need not check the sign
- Only one representation for zero
- Efficient use of all the bits

Problems:
- How do we do addition/subtraction?
- We have two numbers for zero (+/-)?
Manipulating Binary numbers - 1

- Binary to Decimal conversion & vice-versa
  - A 4-bit binary number $A = a_3a_2a_1a_0$ corresponds to:
    
    $a_2*2^2 + a_1*2^1 + a_0*2^0$.
  
  - A decimal number can be broken down by iteratively determining the highest power of two that "fits" in the number.

- Overflow
  - If we add the two (2's complement) 4 bit numbers representing $7$ and $5$ we get:
    
    $0111 + 0101 = 10000$.
  
  - We have overflowed the range of 4 bit 2's comp. (-8 to +7), so the result is invalid.

- X86 & Patel, Chapter 2:

  - Decide if the whole number fits in the number:
    
    $a_2*2^2 + a_1*2^1 + a_0*2^0$.
  
  - Note that if we add 16 to this result we get back 16.

- In general, if the sum of two positive numbers produces a negative result, or vice versa, an overflow has occurred, and the result is invalid in that representation.

CS Reality #1

- The first place where the underlying hardware abstraction can’t be treated as a magic black box results in the need for data types.
- You’ve got to understand binary encodings.

- Can x = 20,000,000,000?

- Assume machine with 32 bit word size, two’s complement integers.
- For each of the following C++ expressions, either:
  
  - Argue that is true for all argument values.
  
  - Give example where not true.

- Problems

  - Patt & Patel, Chapter 2:
    
    2.4, 2.8, 2.10, 2.11, 2.17, 2.21

C++ Integer Puzzles

- Assume machine with 32 bit word size, two’s complement integers.
- For each of the following C++ expressions, either:
  
  - Argue that is true for all argument values.
  
  - Give example where not true.

- Initialization

- Unsigned & Signed int’s: Yes!

- Float’s: Yes!

- Hex and other special characters vary

- Representing Strings

  - Strings in C/C++
    
    - Represented by array of characters
    
    - Each character encoded in ASCII format
    
    - Standard 7-bit encoding of character set
    
    - Other encodings exist, but uncommon
    
    - Uni1616 16-bit variant of ASCII that includes characters for non-English alphabets.
    
    - Character 'T' has code $54$.
    
    - Hex: 160000 = 65

  - Strings are really just arrays of bytes!

  - How can we represent numbers using bits?

  - "Hey, how do I know it’s a string in the first place?"
**Real numbers**

- Most numbers are not integer!
- Range:
  - The magnitude of the numbers we can represent is determined by how many bits we use.
  - E.g. with 32 bits, the largest number we can represent is about $2^{31}$, far too small for many purposes.
- Precision:
  - The exactness with which we can specify a number.
  - E.g. a 32-bit number gives us 31 bits of precision, or roughly 9 figure precision in decimal representation.
- Our decimal system handles non-integer real numbers by adding yet another symbol: the decimal point (.) to make a fixed point notation:
  - E.g. 3.45678 $\approx 3.10^1 + 4.10^{-1} + 5.10^{-2} + 6.10^{-3} + 7.10^{-4} + 8.10^{-5}$
- The floating point, or scientific, notation allows us to represent very large and very small numbers (integer or real), with as much or as little precision as needed.

**Real numbers in a fixed space**

- What if you only have 8 digits to represent a real number?
- Do you prefer...
  - A low-precision number with a large range?
  - Or a high-precision number with a smaller range?

Wouldn’t it be nice if we could “float” the decimal point to allow for either case?

**Scientific notation**

- In binary, we only have 0 and 1, how can we represent the decimal point?
- In scientific notation, it’s implicit. In other words:
  - $4250000 = 4.25 \times 10^6$
  - $42.5 = 4.25 \times 10^1$
  - $4.25 = 4.25 \times 10^0$
- We can represent any number with the decimal point after the first digit by “floating” the decimal point to the left (or right)
  - 0.0000125 = $1.25 \times 10^{-5}$

**Real numbers in binary**

- We mimic the decimal floating point notation to create a “hybrid” binary floating point number:
  - We first use a “binary point” to separate whole numbers from fractional numbers to make a fixed point notation:
    - E.g. 25.75 = $1.210^3 + 0.10^2 = 1.210^3 + 0.10^2 = 11001.110 = 10101.110$
  - We then “float” the binary point:
    - 00011001.110 => 1.1001110 x 2
  - Now we have to express this without the extra symbols (e.g., 2, .)
    - By convention, we divide the available bits into three fields: sign, mantissa, exponent

**IEEE-754 fp numbers – 32 bit “float”**

<table>
<thead>
<tr>
<th>32 bits</th>
<th>8 bits</th>
<th>23 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s</strong> signed</td>
<td><strong>e</strong> biased exp.</td>
<td><strong>f</strong> fraction</td>
</tr>
</tbody>
</table>

N = (-1)^s \times 1.fraction \times 2^{(biased exp. - 127)}

**IEEE-754 fp numbers - example**

- Example:
  - 25.75 $\Rightarrow$ 00011001.110 $\Rightarrow$ 1.10101 x 2^3
  - -4.125 $\Rightarrow$ 10101100000000000000000 $\Rightarrow$ -1.000111110 x 2^-3
  - vs: 50.1101110010000000000000 $\Rightarrow$ +55.1E30000

**Special values represented by convention:**

- Infinity (+ and –): exponent = 255 (11111111) and mantissa = 0
- NaN (not a number): exponent = 255 and mantissa $\neq$ 0
- Zero (0): exponent = 0 and mantissa = 0
- Not a special case: fraction is in-de-normalized, i.e., no leading 1
IEEE-754 fp numbers – 64-bit “double“

- **Double** precision (64-bit) floating point

<table>
<thead>
<tr>
<th>1</th>
<th>11 bits</th>
<th>52 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s</strong> biased exp.</td>
<td>fraction</td>
<td></td>
</tr>
</tbody>
</table>

N = (-1)^s \times 1.\text{fraction} \times 2^{(\text{biased exp.} – 1023)}

- **Range & Precision:**
  - 32-bit:
    - mantissa of 23 bits + 1 ➞ approx. 7 digits decimal
    - \(2^{-1022} \gg \text{approx.} \ 10^{-38}\)
  - 64-bit:
    - mantissa of 52 bits + 1 ➞ approx. 15 digits decimal
    - \(2^{-1022} \gg \text{approx.} \ 10^{-306}\)

Floating point in C/C++

- **C Guarantees Two Levels**
  - **float** single precision
  - **double** double precision

- **Conversions**
  - Casting between **int**, **float**, and **double** changes numeric values
  - Double or float to int:
    - Truncates fractional part
    - Like rounding toward zero
    - Not defined when out of range
    - Generally saturates to **TMin** or **TMax**
  - int to double:
    - Exact conversion, as long as int has \(\leq 53\) bit word size
    - int to float:
      - Will round according to rounding mode

Floating point puzzles in C++

```cpp
int x = ...;
float f = ...;
double d = ...;

' x == (int)(float) x
' x == (int)(double) x
' f == (float)(double) f
' d == (float) d
' f = -(f);
' 2/3 == 2/3.0
' d < 0.0 ➞ (d*2 < 0.0)
' d + d == 0.0
' (d+f)== f
```

Floating point number line

- 32 bits can represent \(2^{32}\) unique values
- How are those values distributed differently between integers and floats?
- What are the implications?

Ariane 5

- Payload rocket
- 7 billion (Euro) in design alone
- Danger:
  - Computed horizontal velocity as floating point number
  - Converted to 16-bit integer
  - Worked OK for Ariane 4
  - Overflowed for Ariane 5
  - same software
- Result:
  - Exploded 37 seconds after liftoff
  - Cargo worth $500 million
Other Data Types

- Other numeric data types
  - e.g. BCD
- Bit vectors & masks
  - sometimes we want to deal with the individual bits themselves
- Logic values
  - True (non-zero) or False (0)
- Instructions
  - Output as "data" from a compiler
- Misc
  - Graphics, sounds, ...

Practice Problems

- Patt & Patel
  - 2.39, 2.40, 2.41, 2.42, 2.56