CSC290/420
Data Formats

Sreepathi Pai

September 10/15, 2025

URCS

Outline

Outline

Real World Data

- Numbers
- Text
- Pictures
- Audio
- Scents
- ...

Most can be encoded as numbers

Building Blocks of the Digital Universe

And most numbers can be encoded as binary digits (or *bits*), consisting of the values 0 and 1.

Bits in the Physical World

- In classical computers, usually voltages
- HIGH voltage indicates 1, LOW voltage indicates 0
- Actual voltages depend on *logic family*
  - for TTL, (V_{CC}) 5V: 0-0.8V is LOW, and 2V-5V is HIGH
  - for CMOS, much wider range, but 5V and 3.3V common
- In quantum computers, other weird phenomena
  - Read *The Talk*, if interested

Outline

AND

- *AND* outputs 1 only when both inputs are 1

a	b	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR

- *OR* outputs 1 if either input is 1
  - hence, "inclusive or"
  - *not* how it is used in English!

a	*b*	Output
0	0	0
0	1	1
1	0	1
1	1	1

XOR

- *XOR*, 1 only when exactly one of its input is 1
  - hence, "exclusive or"
  - pronounced "ecks-or" (i.e. x-or) or "zor"
  - I prefer the latter...

a	b	Output
0	0	0
0	1	1
1	0	1
1	1	0

NAND and NOR

- $NAND = NOT(AND(a, b))$

a	b	Output
0	0	1
0	1	1
1	0	1
1	1	0

- $NOR = NOT(OR(a, b))$

a	b	Output
0	0	1
0	1	0
1	0	0
1	1	0

- *NAND* and *NOR* are universal gates
  - Can be used to implement any boolean function

Examples of NAND

- What should ? be in the following examples to make LHS = RHS?
  - $NOT(a) = NAND(a, ?)$
  - $AND(a, b) = NAND(NAND(a, b), ?)$
  - $OR(a, b) = ?$

Generalizing to inputs longer than one bit

- Inputs longer than one bit are called:
  - bit vectors
  - bit strings
  - or more specific names for particular names (e.g. 8 bits = byte)

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
	0	1	0	1	1	0	0	1
AND	0	1	1	1	0	1	1	0
	0	1	0	1	0	0	0	0

- Each bit in the first 8-bit input is ANDed to its corresponding bit in the second input
- The AND operates on each pair of bits separately

Logic and Boolean Algebra

- Logical variables take only values TRUE and FALSE
- Logical operations are operations on these values
  - e.g., "Not True = False"
- Systematized by George Boole in 1847
  - Later expounded in *The Laws of Thought*, 1854
- Claude Shannon connected boolean algebra to digital circuit design
  - Originally, to design circuits that used electromechanical relays as switches
  - Now digital circuits use transistors, but principles are the same
  - Also coined the word "bit" later...

Outline

Bits, Bytes, Words, ...

- Almost no machine allows manipulation of single bits directly
- Bits are handled as aggregations

Size (bits)	Common Name
8	byte
16	word, halfword
32	word, doubleword
64	word, doubleword, quadword
128	?

- A *machine* word (sometimes the word "machine" is omitted) is the size (in bits) of data that a machine can manipulate at once.
    - Hence 16-bit machines, 32-bit machines, 64-bit machines, etc.

Reading a byte

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	1	1	0	1	1	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

- In place-value notation, $b_0 = 1$ and $b_7 = 2^7 = 128$
  - Hence, this is $1 \times 2^6 + 1 \times 2^5 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = 109$
- The grouping of 4 bits together is called a nybble (i.e. half a byte)
  - Primarily improves readability
  - But can also be used to easily convert to base-16 (i.e. hexadecimal)
- b_0 (i.e. rightmost bit) is called the least significant bit (LSB)
  - contributes the smallest value (2^0)
- b_7 (i.e. leftmost bit) is called the most significant bit (MSB)
  - contributes the most value (2^7)

Hexadecimal

- Numbers in base 16
  - 0 to 9 and A to F
  - Usually indicated by a $0x$ prefix, or a 16 subscript
  - e.g., $0xA = A_{16} = 10_{10} = 1010_2$
- $109_{10} = 0110\,1101_2 = 0x6D$
- Hexadecimal is widely used in low-level code

Multibyte Data Types and Memory Layout

- The 16-bit value 51996_{10} has hexadecimal representation $0xCAFE$
  - Its binary representation is $1100\,1010\,1111\,1110_2$
  - The value $0xCA$ is its most significant *byte*
  - The value $0xFE$ is its least significant *byte*
- RAM is byte addressable
  - Can read individual bytes of a multibyte value
  - How should we order each byte of a multibyte value?

Little and Big-endian

- Storing a 32-bit value 0x*DEADCAFE* in memory
- Big endian: Most significant byte at lower addresses
- Little endian: Least significant byte at lower addresses

address	x	$x+1$	$x+2$	$x+3$
big-endian	0xDE	0xAD	0xCA	0xFE
little-endian	0xFE	0xCA	0xAD	0xDE

- Different machines use different conventions
  - Intel/AMD usually little endian
  - SPARC/PowerPC usually big endian
  - ARM can switch between the two
- Big endian is sometimes called network byte order
  - Similar problem: which byte of a word gets on the wire first?

The Interpreter of Bits

- Does the byte $0x55$ in memory indicate:
  - The integer value 85?
  - The Intel assembly language instruction `push %rbp`
- There is nothing in $0x55$ that can distinguish between these two interpretations
  - Very powerful idea
  - Code can be data and data can be code

Outline

Integers

- The most common interpretation of bytes, words, etc. is that as "integers"
  - Whole numbers (no fractional part)
  - Can be positive or negative
- Examples: -3, -2, -1, 0, 1, 2, 3

How many bits are required?

- The number of bits required to store N distinct values is $\lceil log_2(N) \rceil$
  - i.e. logarithm of N to the base 2
  - i.e. find x such that $2^x = N$, and round it up
- Example #1: There are two possible values for sign, so $N = 2$
  - $log_2(2) = 1$, so one bit is required to store sign
- Example #2: If N is 200, then $x = log_2(200) = 7.644$, so 8 bits are required

Stuffing numbers into a byte: Sign-Magnitude

- A byte has 8 bits
- One bit is used for the sign, 7 bits left
- Can store magnitudes from 0 to $2^7 = 127$
- Let MSB be sign bit
- Let other bits store magnitude
- Can store numbers from -127 to +127

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$+89_{10}$	0	1	0	1	1	0	0	1
-89_{10}	1	1	0	1	1	0	0	1
0_{10}	0	0	0	0	0	0	0	0
-0_{10}	1	0	0	0	0	0	0	0

Stuffing numbers into a byte: One's Complement

- Can store magnitudes from 0 to $2^7 = 127$
- Let MSB be sign bit
- Let other bits store magnitude
  - except if sign bit is set, magnitude must be *complemented* (i.e. inverted) to get actual value
  - one's complement of bit value x is $1 - x$, i.e. the same as $NOT(x)$
- Represents numbers from -127 to $+127$

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$+89_{10}$	0	1	0	1	1	0	0	1
-89_{10}	1	0	1	0	0	1	1	0
0_{10}	0	0	0	0	0	0	0	0
-0_{10}	1	1	1	1	1	1	1	1

Stuffing numbers into a byte: Two's Complement

- Can store magnitudes from 0 to $2^7 = 127$
- Let MSB be sign bit
- Let other bits store magnitude
  - To negate a number, complement all its bits and add 1
- Can store numbers from -128 to 127

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
$+89_{10}$	0	1	0	1	1	0	0	1
-89_{10}	1	0	1	0	0	1	1	1
0_{10}	0	0	0	0	0	0	0	0
-0_{10}	0	0	0	0	0	0	0	0
-128_{10}	1	0	0	0	0	0	0	0

Integer Representations

- There is more than one way to represent the same integer
  - Sign-magnitude
  - One's complement
  - Two's complement
- Some of them are non-intuitive
  - negative and positive zeroes
  - asymmetric ranges [-128, 127]
- All of them have different hardware implications
  - Addition and subtraction circuits differ
- Generally, most computers you will encounter use two's-complement arithmetic

Integers in C

- Basic C types:

```
char a;
short b;  /* alternative form: short int */
int c;
long d;   /* alternative form: long int */
long long e;
```

- C implementations are required to provide a minimum size for each type
  - `char` must be at least 8 bits
  - `int` must be at least 16 bits
  - `long` must be at least 32 bits
  - `long long` must be at least 64 bits
- The prefix `unsigned` (e.g. `unsigned char`) allows all bits to be used to store the magnitude (i.e. there is no sign bit).
  - `char` must be able to store $[-127, 127]$
  - `unsigned char` must be able to store $[0, 255]$
  - C23 requires support for two's complement

Fixed-width Integers in C99

```
 #include <stdint.h>
int8_t a;   /* signed 8-bit integer */
uint8_t ua;  /* unsigned 8-bit integer */

int32_t b;   /* signed 32-bit integer */
uint32_t ub;  /* unsigned 32-bit integer */

 ...
```

- C99 is the C standard "version" 1999.
  - Finally allowed fixed-width types
  - Still does not mandate any particular representation (C23 does!)
- The variables `INT8_MIN` and `INT8_MAX` contain the range for `int8_t`
  - similarly, `UINT8_MIN` and `UINT8_MAX` contain the range `uint8_t`

Outline

Bitpacking

- Sometimes space is at a premium
- Want to use as few bits as possible
- Bitfields:
    - Partition a machine word into distinct fields

Example

- Suppose we want to store day and day of the week using as little space as possible

```
uint8_t day = 9;
uint8_t dow = MON;
```

- Day takes value 1–31
  - Bits required: ?
- Day of week takes 0 (SUN)– 6(SAT)
  - Note here we want to store 1 of 7 values
  - Bits required: ?
- Total storage used by two `uint8_t` variables: 16 bits
- Bits wasted: ?

Minimum Bits using Bitfields

```
enum days_of_week {
  SUN = 0,
  MON = 1,
  TUE = 2,
  WED = 3,
  THU = 4,
  FRI = 5,
  SAT = 6
};
```

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Mon 9	0	1	0	0	1	0	0	1
	Day of Month					Day of Week		

Constructing a bitfield

```
uint8_t daydow;
daydow = (9 << 3) | MON;
```

- Here, 9 is the day
- It is *left-shifted* by 3 bits using the left-shift (<<) operator
  - Empty positions at right are filled with zeroes
  - Bits at left are *discarded*
  - Like multiplying by 10^3 in the metric system, except here we're multiplying by 2^3
  - `0x9` (binary 1001) becomes `0x48` (binary 0100 1000)
- Then we OR the day of the week into the freshly created lower 3 zero bits

Getting the Day of the Week

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Mon 9	0	1	0	0	1	0	0	1
	Day of Month					Day of Week		
result	0	0	0	0	0	0	0	1

- We want to force the day (of month) field to zero.
- What are OP and MASK?

```
dow = daydow OP MASK;
```

Getting the Day of the Week (Solution)

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Mon 9	0	1	0	0	1	0	0	1
	Day of Month					Day of Week		
mask	0	0	0	0	0	1	1	1
result	0	0	0	0	0	0	0	1

- We want to force the day (of month) field to zero.

```
dow = daydow & 0x7;
```

Getting Day of Month

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
Mon 9	0	1	0	0	1	0	0	1
	Day of Month					Day of Week		
mask	1	1	1	1	1	0	0	0
result	0	1	0	0	1	0	0	0

- C code, all bits except lower 3

```
day = daydow & (0x1f << 3);
```

- Is the result what we want?

Undoing the left shift

```
day = (daydow & (0x1f << 3)) >> 3;
```

- We got `0x48` because we had shifted it left
- We can undo it by doing a right shift by 3 bits using the right-shift >>) operator
  - Bits at right are *discarded*
  - Like dividing by 2^3, and throwing away the remainder/fractional part
  - `0x48` (binary 0100 1000) becomes `0x9` (binary 0000 1001)

Even shorter ...

```
day = (daydow >> 3) & 0x1f;
```

- Since we're using `uint8_t`, the masking is superfluous
- For `unsigned` integers, right shifting will fill in bits at left with 0
- Since all bits to the left of the Day of Month field are zero, we can eliminate the mask and the AND
  - But recommend always using a mask, as good programming practice

Real-life bitfields and bitsets: Unix file permissions

- Basic File permissions (can be simultaneously enabled)
  - READ
  - WRITE
  - EXECUTE
- Permissions for
  - User/Owner
  - User's Group
  - Others

	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
`rwxr-x---`	1	1	1	1	0	1	0	0	0
	User/Owner			Group			Others		

Outline

Real Numbers

- $\mathbb{R}$
  - infinite (just like integers)
  - but they are different infinity (uncountable)
- There are infinite real numbers between any two real numbers
- How do we represent these using a finite, fixed number of bits?
  - Say, 32 bits

The problem

- Assume 5 bits are available
- Consider 17: 10001
- Consider 18: 10010
- Where shall we put 17.5?
    - No bit pattern "halfway" between 10001 and 10010

One option

- Consider only deltas of 0.25, 0.5, 0.75
- Then
  - 17.00: 10001
  - 17.25: 10010
  - 17.50: 10011
  - 17.75: 10100
  - 18.00: 10101
- This is the basis of the idea of *fixed point*
  - Can't represent all numbers
  - Fixed accuracy
- Used widely in tiny computers

Representing Real Numbers

- We cannot represent real numbers accurately using a finite, fixed number of bits
  - But do we need infinite accuracy?
- How many (decimal) digits of precision do we use?
  - In our bank accounts (before and after the decimal point?)
  - In engineering?
  - In science?

On magnitudes

- Smallest length
  - Planck length, on the order of 10^{-35} (would require 35 decimal digits)
- Smallest time
  - Planck time, on the order of 10^{-44}
- Width of visible universe
  - On the order of 10^{24}
  - Lower bound on radius of universe: 10^{27}

On precision

- Avogadro's number: $6.02214076 \times 10^{23}$
  - So, actually: 602214076000000000000000
- $\pi = 3.1415... \times 10^{0}$
  - NASA requires about 16 decimal digits of π[1]
  - We know about a trillion

[1] https://blogs.scientificamerican.com/observations/how-much-pi-do-you-need/

Scientific notation for numbers

- The scientific notation allows us to represent real numbers as:

$$\text{significand} \times \text{base}^{\text{exponent}}$$

- For Avogadro's number:
  - Significand: 6.02214076
  - Significand is scaled so always only one digit before the decimal point
  - Base: 10
  - Exponent: 23

Binary Scientific Notation

- We can use scientific notation for binary numbers too:

$$1.011 \times 2^3$$

- Here, the number is:
  - $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^3$
  - $(1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) = 11_{10}$
- Components:
  - Significand: 1.011
  - Base: 2
  - Exponent: 3

Binary Scientific Notation: Example #2

- Now with a negative exponent:

$$1.011 \times 2^{-3}$$

- Here, the number is:
  - $(1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3}) \times 2^{-3}$
  - $(1 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6})$
  - $(0.125_{10} + 0 + 0.0625_{10} + 0.03125_{10}) = 0.171875$
- Components:
  - Significand: 1.011
  - Base: 2
  - Exponent: -3

Some design notes

- Significand contains a radix point (i.e. decimal point or binary point)
  - But it's position is fixed: only one digit before the radix point
  - In binary scientific notation, this is always 1 (why?)
  - We don't need to store the radix point
  - So significand can be treated as an integer with an implicit radix point
- Base is always 2 for binary numbers
  - No need to store this
- Exponent is also an integer
  - Could be negative or positive or zero

Design notes (continued)

- So (binary) real numbers can be expressed as a combination of two fields:
  - significand (possibly a large number, say upto 10 decimal digits)
  - exponent (possibly a smallish number, say upto 44_{10})
  - would allow us to store numbers with at least 10 decimal digits of precision, upto 44 decimal digits long
- We'll also need to store sign information for the significand and the exponent
- How many bits?
  - for 10 significant decimal digits? e.g. 9,999,999,999
  - for max. exponent 50_{10}?
  - plus two bits for sign (one for significand, one for exponent)

Design notes (continued)

- How many bits?
  - for 10 significant decimal digits? e.g. 9,999,999,999: about 34 bits
  - for max. exponent 50? about 6 bits
  - plus two bits for sign (one for significand, one for exponent)
- Total: $34 + 6 + 2 = 42$ bits
  - **Could be implemented as a bitfield**
  - But 42 is between 32 and 64, not efficient to manipulate
- What format should we use to store negative significands and exponents?
  - sign/magnitude
  - one's complement
  - two's complement
  - other?

Bitfield Design Constraints

- Ideally should fit sign, significand and exponent in 32 bits or 64 bits
  - Easier to manipulate on modern systems
- Arithmetic operations should be fast and "easy"
- Comparison operations should be fast and "easy"
  - e.g. should not need to extract fields and compare separately
- Should satisfy application requirements
  - esp. with accuracy, precision and rounding
  - should probably be constraint #1

Outline

IEEE 754 32-bit floating point standard

- Total size: 32-bits
  - Also called "single-precision"
  - On most systems, the C type `float` is single-precision
- Significand: 24 bits, roughly 7 significant (decimal) digits of accuracy
  - Sometimes called (wrongly) the Mantissa
- Exponent: 8 bits, from 2^{-126} to 2^{127} (roughly 10^{-38} to 10^{38} (decimal))
- Sign bit: 1 sign bit for the significand
  - What about sign bit for the exponent?
- Also supports special representations:
  - for $+\infty$ and $-\infty$
  - For "not-a-number" *NaN*, e.g. for representing $(0/0)$
  - "denormals"
- Note: $24 + 8 + 1 = 33$, not 32

Representing the significand

$$1.100\,1001\,0000\,1111\,1101\,1011$$

- 24 bits of significand
- *Normalized* form, only one digit before the radix point
  - Change the exponent until this is achieved (normalization)
  - That digit must be non-zero
  - Always 1
- Hence, do not need to store it!
  - Only use 23 bits for the magnitude
  - In example, only $100\,1001\,0000\,1111\,1101\,1011$ is stored
- Uses sign/magnitude notation (not one's or two's complement)
  - 1 bit for sign (0 for $+$, 1 for $-$)
  - 23 bits for magnitude + one always 1 implicit bit (not stored)

Appreciating Precision

One weird trick to make money from banks:

```
#include <stdio.h>

int main(void) {
  float f;
  int i;

  f = 16777216.0;
  f = f + 3.0;

  printf("%f\n", f);
}
```

- Note that 16777216 is 2^{24}
- What is the value of `f` that is printed?
  - A: 16777216.0
  - B: 16777219.0
  - C: 16777220.0
  - D: something else
  - E: undefined

More Surprises

```
#include <stdio.h>

int main(void) {
  float f;
  int i;

  f = 16777216.0;

  for(i = 0; i < 2000; i++) {
    f = f + 1.0;
    // printf("%f\n", f) // uncomment to see what is happening
  }

  printf("%f\n", f);
}
```

- What is the value of `f` that is printed?
  - A: 1677**7**216.0
  - B: 1677**9**216.0
  - C: something else
  - D: undefined

This is not C specific!

- Integers in Python behave differently from C
  - Don't overflow, are always signed
  - Python's integers are like mathematical integers
- Very few languages implement mathematical reals though.
  - Most use IEEE 754
  - Python, Javascript, Java, etc.

Rounding

- IEEE floating point rounds numbers that cannot be exactly represented
- For an operation $\oplus$ (where $\oplus$ could be any of *mathematical* $+, -, /, \times$)
  - the standard says $x \oplus y \rightarrow \mathit{Round}(x \oplus y)$
- Four rounding modes
  - Round towards nearest (also known as round towards even, and default)
  - Round towards zero
  - Round towards $+\infty$
  - Round towards $-\infty$

What's happening

- $16777216.0 + 1.0$ is unrepresentable
  - By default, rounding mode is round to nearest
  - Nearest is 16777216.0
  - No change!
- Why it is also called round to even
  - If an unrepresentable value is equidistant between two representable values
  - It is not possible to say which is "nearest"
  - IEEE standard picks the *even* value between the two representable values
- This makes floating point arithmetic *non-associative*
  - $(a + b) + c \neq a + (b + c)$
  - $((a + 1.0) + 1.0) \neq (a + (1.0 + 1.0))$

Representing the Exponent

- 8-bit wide bitfield
  - Can store 256 values
  - Must store values from -126 to 127 (that's 254 values)
- Uses *biased* representation
  - To store x, we actually store $x + 127$ in 8 bits
  - So 127 is stored as 254
  - And -126 is stored as 1
  - No sign bit required!
  - So field actually contains values from 1 to 254 to represent -126 to 127
- The biased values 0 and 255 are used to indicate special numbers

Why biased? Comparing exponents

Which is greater?

$$1.011 \times 2^{-3}$$

Or:

$$1.011 \times 2^{+3}$$

- Note -3 in biased notation is $-3 + 127 = 124 = 0111\,1100_2$
- Note 3 in biased notation is $+3 + 127 = 130 = 1000\,0010_2$

Putting it altogether

- Three bit fields
  - s: Significand Sign (1 bit)
  - M: Significand (23 bits)
  - E: Biased Exponent (8 bits)
- 6 possible ways to order them
  - s, M, E
  - s, E, M
  - M, s, E
  - M, E, s
  - E, s, M
  - E, M, s
- Out of familiarity, let's only consider those where s occupies higher bits than M

Comparing Three Formats

- Suppose you have two numbers:
  - $a = 1.100... \times 2^3$
  - $b = 1.010... \times 2^5$
  - Which is greater?
- Representation
  - Significand: $100..._2$ for a and $010.._2$ for b
  - Exponent: $3 + 127 = 130 = 1000\,0010_2$ and $5 + 127 = 132 = 1000\,0100_2$
  - Sign is 0 for both

Comparing Three formats (contd.)

- s, M, E
  - `0 | 100 000 ... | 1000 0010`
  - `0 | 010 000 ... | 1000 0100`
- s, E, M
  - `0 | 1000 0010 | 100 000 ...`
  - `0 | 1000 0100 | 010 000 ...`
- E, s, M
  - `1000 0010 | 0 | 100 000 ...`
  - `1000 0100 | 0 | 010 000 ...`

IEEE 754 Single Precision Format

- Uses s, E, M format
- If a number $x > y$, then its bitwise representation $x > y$
  - When sign bit is same, scan from bit 30 to 0, looking for first different bit
  - When sign bit is different, 1 in sign bit indicates less than 0 (exceptions $+0$ and -0)
- Can thus compare floating point numbers without having to extract bitfields!

Representing Zero

$$0 \times 2^x$$

- Has no leading 1
- Special representation
  - Sign bit can be 0 or 1
  - Exponent is all zeroes (i.e. it appears to be -127 stored biased, hence -126 is lower limit)
  - Magnitude is all zeroes
- Hence:
  - $+0$: all 32 bits are zero
  - -0: sign bit is 1, but all other bits are zero

The smallest normalized single-precision number

$$+1.000\,0000\,0000\,0000\,0000 \times 2^{-126}$$

- In IEEE: 0 | 0000 0001 | 000 0000 0000 0000 0000
  - | just for visual separator
- That's just 2^{-126}
  - Approximately, $1.17549435 \times 10^{-38}$
- What should happen if we divide this by two?
  - I.e. $(0.00000000000000000000000000000000117549435/2)$
  - $(+1.000\,0000\,0000\,0000\,0000 \times 2^{-126})/2$
  - $(+1.000\,0000\,0000\,0000\,0000 \times 2^{-127})$

Let's make it zero!

- Default behaviour on many systems before IEEE754
  - Underflow to zero
- $a = 1.000\,0000\,0000\,0000\,0000 \times 2^{-126}$
- $b = 1.000\,0000\,0000\,0000\,0001 \times 2^{-126}$
- What is $a - b$?
  - Remember, $a \neq b$
- What would $x/(a - b)$?

Denormals

- $a = 1.000\,0000\,0000\,0000\,0000 \times 2^{-126}$
- $b = 1.000\,0000\,0000\,0000\,0001 \times 2^{-126}$
- $a - b = 0.111\,1111\,1111\,1111\,1111 \times 2^{-126}$
  - Numbers of this form are called *denormals* or *subnormals*
  - They have a 0 before the radix point
- IEEE 754 specifies how to store denormals:
  - s, sign as usual
  - E, exponent is zero
  - M, the significand is non-zero
- This allows "gradual underflow" to zero
  - Some systems detect denormals and perform arithmetic in software
  - Slow!

Representing Infinities

```
0 1111 1111 000 0000 0000 0000 0000
```

- In the above representation,
  - Sign: 0
  - Exponent: 255
  - Significand: 0
- Exponent
  - 0 indicates either zero or a subnormal
  - 1 to 254 indicates normalized exponent -126 to 127
  - 255 indicates either infinity or NaN
- With significand zero:
  - Exponent 255 indicates $+\infty$ or $-\infty$ (depending on sign)
  - $-\infty < x < +\infty$ where x is any representable number

Representing NaNs

```
0 1111 1111 xxx xxxx xxxx xxxx xxxx
```

- In the above representation,
  - Sign: 0
  - Exponent: 255
  - Significand: $\neq 0$ (i.e. the `x` bits are not all zero)
- With significand non-zero:
  - Exponent 255 indicates *NaN* (not-a-number)
  - Produced by operations like $0/0$, ∞/∞, etc.
- *NaN*s propagate:
  - Any operation involving a *NaN* results in a *NaN*

Addition in Floating Point

Add

$$a = 1.000\,0000\,0000\,0000\,0000 \times 2^3$$

to:

$$b = 1.000\,0000\,0000\,0000\,0000 \times 2^4$$

Equalizing exponents

- Exponents for a and b are different, so equalize them
  - Shift one of them
  - The shifted representation is *internal*
  - Only the result after addition is visible
- $b = 10.00\,0000\,0000\,0000\,0000 \times 2^3$
- $a + b = 11.00\,0000\,0000\,0000\,0000 \times 2^3$
- Normalized, $1.100\,0000\,0000\,0000\,0000 \times 2^4$

Double-precision

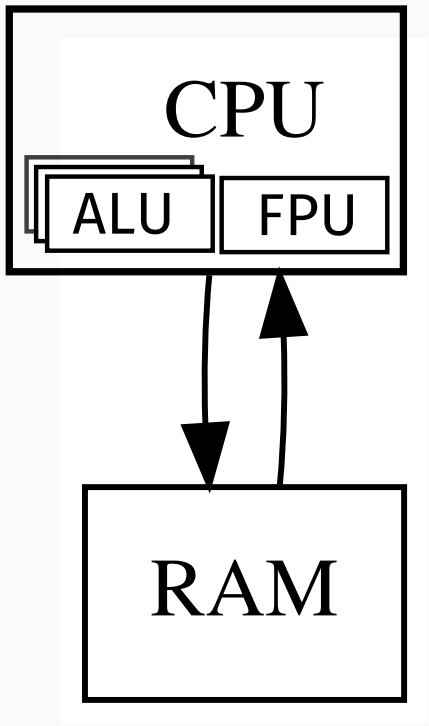
- 64-bit floating point format
  - Significand: 53 bits (52 stored), around 17 decimal digits of precision
  - Exponent: 11 bits (biased by 1023)
  - Sign: 1 bit
- In C, usually `double`
- Range: 2^{-1022} to 2^{1023} for normalized numbers
  - Roughly 10^{-308} to 10^{308}

A Programmer's View of Floating Point

- When translating algorithms from math to code, be wary
  - Computers use floats, not real numbers!
- Two major problems:
  - Non-termination (usually because exact $==$ is not possible)
  - Numerical instability (approximation errors are "magnified")
- If you deal with *computational* science or use numerics extensively, educate yourself
  - Resources at the end
  - Or take a Numerical Analysis class (primer at the end)

How the machine supports floating point

- A math co-processor called the "floating point unit"
  - Back in the day, a separate processor
  - The Intel 387 is a classic
- For machines without a coprocessor, everything was done in software
  - Sometimes called "softfloat"
  - Still used to handle denormals on some processors
- These days, integrated into the CPU as FPUs



Some excitement these days

- Intel, NVIDIA and AMD recently announced support for FP8
  - 8-bit floating point numbers
- Follows the introduction of FP16 a few years ago
  - 16-bit floating point numbers
- Driven by the needs of deep learning programs and memory constraints
  - Also, "correctness" is not number #1 priority for DL

Outline

Python Integers

- Python only has signed integers (like Java)

```
v = 1
for i in range(256):
    v = v * 2

print(v)
```

- What is the value of `v` that is printed?
  - A: Undefined
  - B: $2^{256} \mod 2^{64}$ (assuming 64-bit integers)
  - C: 2^{256}
- Reference: Python Numeric Types

Arbitrary Precision Floating Point

The bc calculator in Linux:

```
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
16777216.0+1.0
16777217.0

16777216.0+3.0
16777219.0

f = 16277216.0
for(i = 0; i < 2000; i++) { f += 1.0; }

f
16279216.0
```

Summary

- Take away: floating point numbers are NOT real numbers

For further study:

- Link to An Interview with the Old Man of Floating-Point
  - IEEE754 won William Kahan the Turing Award
- Definitely read:
  - Goldberg, What Every Computer Scientist should Know about Floating-Point Arithmetic, ACM 1991
  - Stadherr, High Performance Computing, Are we just getting wrong answers faster?
  - Trefethen, Numerical Analysis