MLDS-413 Introduction to Databases and Information Retrieval

Lecture 2
Fixed-point and Floating-point Representations

Instructor: Nikos Hardavellas

Slides adapted from Steve Tarzia

Last week we talked about Integers

- Integers can be stored in a base-two positional notation in binary
- Addition and subtraction follow the familiar mechanics
  - **IMPORTANT: overflow results in "wrap-around" result value**
- Learned some tricks (e.g., $2^{10} \approx 1000$, $2^{20} \approx 1$ million)
- Signed integers use 2's complement representation
  - Two's complement makes subtraction just as easy as addition: $x - y = x + (- y)$
  - Positive numbers are represented in the same way whether you're using a signed or unsigned data type, but
  - **Small negatives and huge positives can be confused if you misinterpret the type**

Signed:	Unsigned:
	7: 111
	6: 110
	5: 101
	4: 100
3: 011	3: 011
2: 010	2: 010
1: 001	1: 001
0: 000	0: 000
-1: 111	
-2: 110	
-3: 101	
-4: 100	

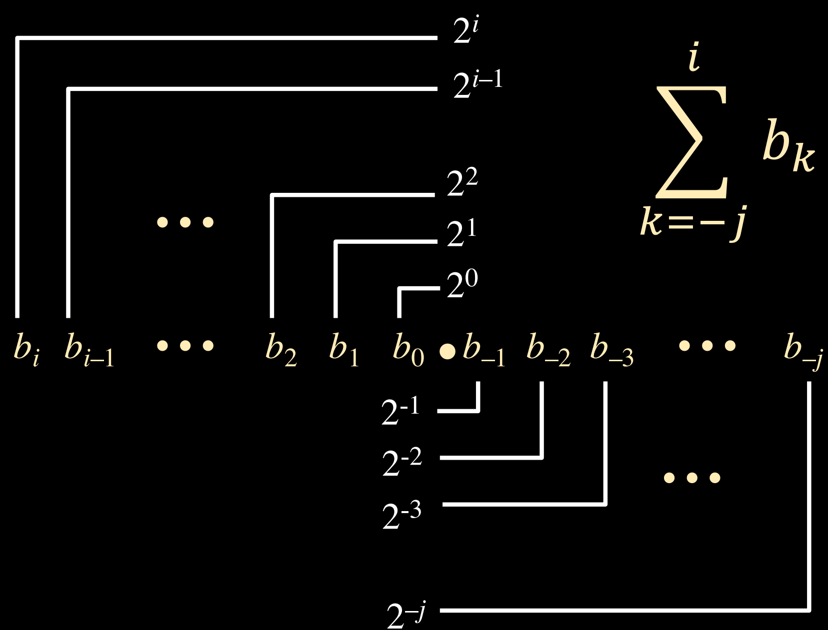
A few more things about integers

- Multiplication: two's complement works magically here too
- Positive division works as expected
- "*Sign extension*:" when increasing the "bit size" of a negative number, add leading ones
  - Eg., -2 is `1110` as a 4-bit signed integer and `11111110` in 8 bits
- Computers typically use 32 or 64 bit integers

Any questions on last week's material?

Fractional Binary Numbers

- Representation
  - Bits to right of “binary point” represent fractional powers of 2
  - Represents rational number:



$$\sum_{k=-j}^{i} b_k \cdot 2^k$$

Integers are great for **counting**, but sometimes we need to **measure** fractional quantities

Binary numbers can have "decimal" places, too

- `0.1111111111`$_{\text{two}}$ is slightly smaller than 1
- `0.0000000001`$_{\text{two}}$ is slightly larger than 0
- `0.1`$_{\text{two}}$ is one half

- `10.101`$_{\text{two}}$ $= \mathbf{1} \times 2^{1} + \mathbf{0} \times 2^{0} + \mathbf{1} \times 2^{-1} + \mathbf{0} \times 2^{-2} + \mathbf{1} \times 2^{-3}$
  $= 2 \quad + 0 \quad + 1/2 \quad + 0 \quad + 1/8 \quad = \; 2$ = `2.625`$_{\text{ten}}$

How shall we represent fractional number in the computer?

Fixed point: Integers 2.0

- Simplest solution is to just stick an implicit **binary (radix) point** somewhere (We don't call it a decimal point because we're not in base ten)
- Examples of fixed point numbers in base ten:
  - Represent the cost of a purchase with an integer number of cents
    - The cost of a sandwich is 625 cents ($6.25)
  - Represent the distance between cities by counting the hundredths of a mile
    - Evanston is 1321 hundredths of a mile from Chicago (13.21 miles)
    - and 79,543 hundredths of a mile from Philadelphia

Fixed point example in 16 bits

Let's store the chemical elements' atomic weights

- Smallest value (hydrogen) is 1.00784
- Largest value (uranium) is 238.02891
- Negative values are not possible
- We can reserve 8 bits for the fractional part and 8 bits for the part > 1
- In this particular binary fixed point representation, the weight of uranium is:
  $\mathtt{1110111000000111}_2$ Remember that the radix point is implicit. This represents the value
  $\mathtt{11101110.00000111}_2 = 238_{10} = 238.02734375_{10}$
  (We had to round off, so this is not precisely accurate)
- And the weight of hydrogen is:
  `0000000100000010,`
  i.e., `00000001.00000010` = 1 = 1.0078125

Fixed point is simple & efficient but it has its limitations

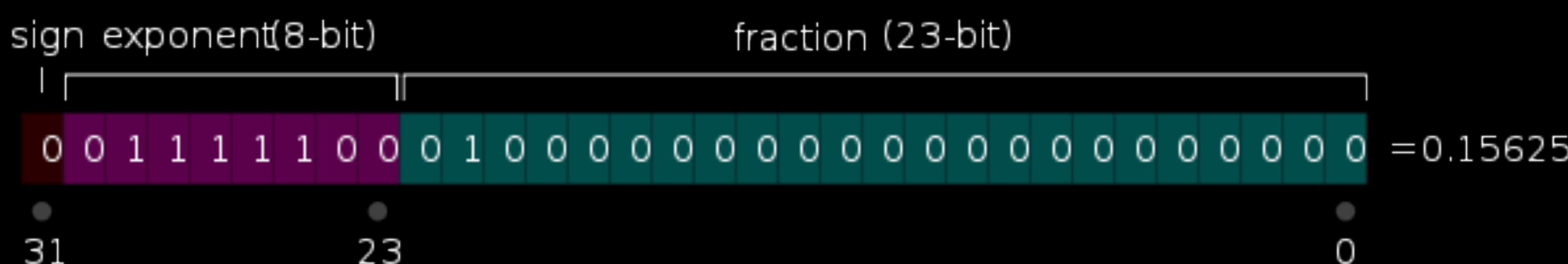
- Range is very limited
  - Multiplication overflows easily – can double the number of bits
    - e.g., multiplying two 32-bit values may give a 64-bit result
  - Division **underflows** easily (small values are rounded to zero)
- Precision varies across the range:
  - Small numbers have few significant figures

Floating point

- Based on scientific notation:
  - $10{,}340 = 1.034 \times 10^4$
  - $0.00424 = 4.24 \times 10^{-3}$
- Gives a compact representation of extreme values:
  - $1{,}000{,}000{,}000{,}000{,}000{,}000{,}000{,}000 = 1.0 \times 10^{24}$
  - $0.000\ 000\ 000\ 000\ 000\ 000\ 000\ 001 = 1.0 \times 10^{-24}$
- In binary:
  - $100010_{two} = 1.0001_{two} \times 2^5{}_{ten} = 1.0001 \times 10^{101}{}_{two}$
  - $0.00101_{two} = 1.01_{two} \times 2^{-3}{}_{ten} = 1.01 \times 10^{-11}{}_{two}$

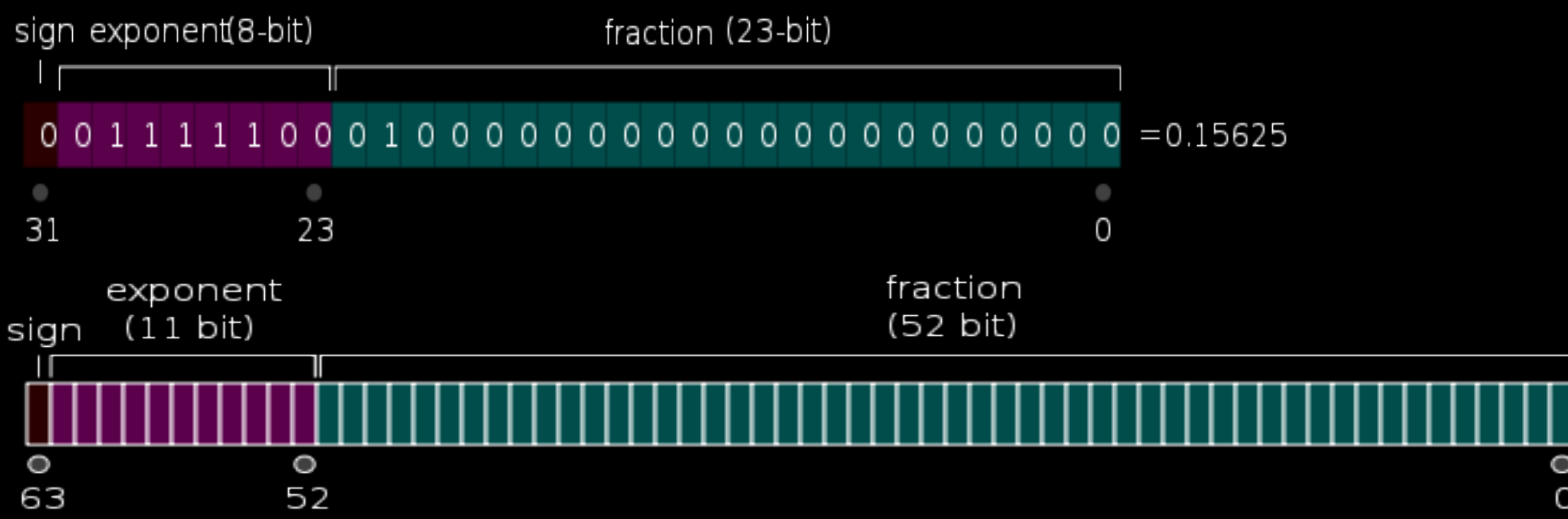
Representing floating point in bits

- $0.15625_{ten} = 0.00101_{two} = 1.01_{two} \times 2^{-3}{}_{ten} = 1.01 \times 10^{-11}{}_{two}$
- Three essential parts are the **sign**, **fraction**, & **exponent**
  - Notice that the first significant figure is always “1” so we don’t have to store it
- In the mid 1980s, the IEEE standardized the floating point representation of 32 and 64 bit numbers:
  - The exponent has a sign too, but the standard says for 32-bit FP to add a “bias” of 127

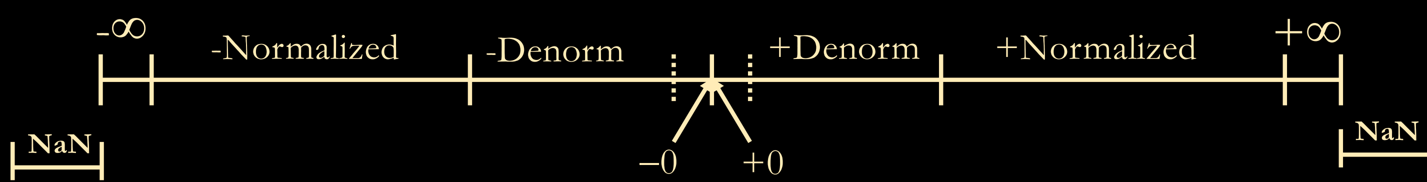


64-bit floating point

- Similar to 32-bit, but we have more precision in the fraction and larger exponents are possible
- 32-bit is called **single precision** and 64-bit is called **double precision**
- Double precision can represent larger, smaller, and more precise numbers



FP Real Number Encodings



$$V = (-1)^s \times M \times 2^E$$

	Normalized	Denormalized
s	0/1 means +/-	0/1 means +/-
exp	$\text{exp} \neq 000\ldots0_2$ and $\neq 111\ldots1_2$	$\text{exp} = 000\ldots0_2$
frac	$x_1x_2x_3\ldots x_j$	$x_1x_2x_3\ldots x_j$
Bias=	$2^{(k-1)} - 1$, k exponent bits	$2^{(k-1)} - 1$, k exponent bits
E=	exp – Bias	1 – Bias
M=	1. $x_1x_2x_3\ldots x_j$ a.k.a. 1.frac	0. $x_1x_2x_3\ldots x_j$ a.k.a. 0.frac
V=	$\mathbf{(-1)^s \times (1.frac) \times 2^{(exp - Bias)}}$	$\mathbf{(-1)^s \times (0.frac) \times 2^{(1 - Bias)}}$

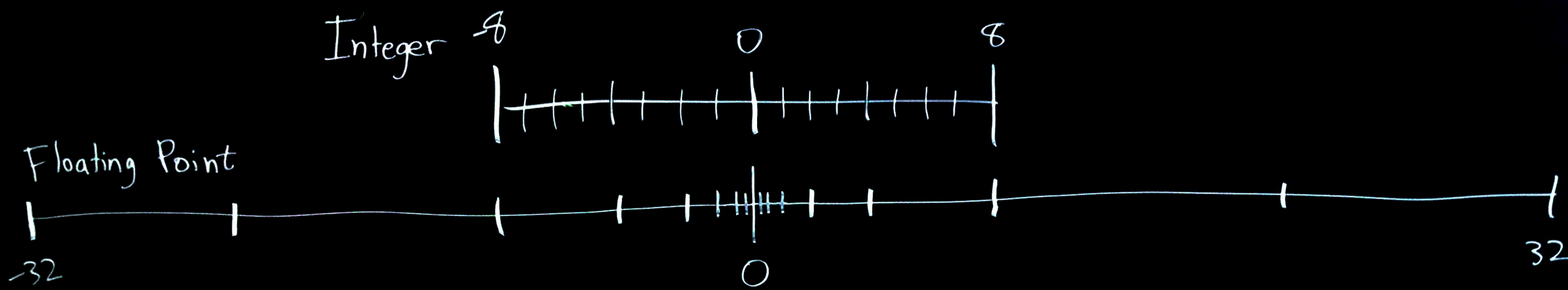
A few special floats

- The IEEE standard allows for a few special values to be stored
  - Positive and negative zero (remember that we normally start with an implied "1")
    - All exponent bits set to zeros
  - Positive and negative infinity (e.g., the result of divide by zero)
  - Not a number – NaN (e.g., the result of zero divided by zero)
    - These all have the exponent bits set to all ones

The Flexibility and Flaws of Floats

- A 32-bit signed integer can represent all the whole numbers between -2,147,483,648 and 2,147,483,647
- A 32-bit floating point number can be as large as $\pm 3.402823 \times 10^{38}$ = 340,282,300,000,000,000,000,000,000,000,000,000,000
- or as tiny as $5.8774718 \times 10^{-39}$ = 0.000 000 000 000 000 000 000 000 000 000 000 000 005 877 471 8
- But, single-precision floats have only 24 bits of precision:
  - Can only precisely store integers up to 2^{24} = 16,777,216
- Floats can store larger numbers than integers of the same bit-length, but with less precision because 8 bits are set aside for the exponent

Floats just distributed the same number of values differently – with exponential spacing



Know when to use integers, floating point, and fixed point

- When **counting** or labelling things, always use integers
- When **measuring** physical quantities, usually use floating point
  - May use fixed point if speed/simplicity is more important than accuracy
- If your machine does not support floating point (e.g., a toaster):
  - Use fixed point representation for fractional quantities
- If rounding is desired then use fixed point (but carefully)
  - U.S. currency values usually should be rounded to the nearest cent
- Use 64-bit integers when you need values > 2 billion
- Use unsigned integers **only when you need the extra range**
- Floating point rules of thumb:
  - Single precision gives ~7 decimal digits of precision
  - Double precision gives ~16 decimal digits of precision

One more point about fractions in binary: Base ten decimals usually have to be rounded

- We all know that 1/3 cannot be represented exactly in decimal
  - That's because 10^x not divisible by 3 (for any integer x)
- Similarly, 1/10 cannot be represented exactly in binary
  - Because 2^x is not divisible by 10 (for any integer x)
- In general, a rational number ***a/b*** can be ***exactly*** represented in binary only if ***b*** is a power of 2
  - Otherwise, there is some rounding error
- Most fractions cannot be stored exactly with a finite number of bits
  - Actually, this is also true in decimal!
- So, always expect small rounding errors when working in floating point

How do computers work with floats?

- It's complicated and slow!
- Have to manipulate both the fraction and the exponent
- Addition is no longer simple

Computer arithmetic can be tricky!

- USS Yorktown CG48 off the coast of Cape Charles, VA (1998)
  - Nuclear US Navy “smart ship”; assigned sailor’s jobs to a Windows NT system
  - A crew member entered a zero **into a database field**
  - Division by 0 in the ship's Remote Database Manager → buffer overrun
  - All systems crashed; no propulsion control; dead in the sea for 2.5 hours
  - Result: mighty nuclear ship brought to safety by a tugboat

Overflows are bad for your health!

- Ariane 5 (1996)
  - Inertial reference system converted a 64-bit float to a 16-bit integer
  - Had worked in the past in Ariane 4, but Ariane 5 was faster
  - Speed too large to fit in a 16-bit integer → overflow
  - Result: guidance system tries to adjust by 90° in supersonic speeds

Arithmetic approximation / rounding errors

- Sleipner-A offshore platform (1991)
  - Oil and gas exploration at North Sea
  - 16,000 m^2 base area; 57,000 ton deck; 200 people + 40,000 tons of equipment
  - Kept afloat by 24 hollow concrete cells
  - Finite elements analysis SW miscalculated concrete wall thickness by 47%
  - Cell cracked; pumps couldn't keep up with the leak
  - Platform sank; caused a seismic event of 3.0 Richter; $700M loss

Approximation / rounding errors redux

- Patriot missile failure (Gulf war, 1992)
  - Intervals of 0.1sec **approximated** as $0.00011001100110011001100_2$
  - 3.6 x 10^6 ticks later (100 hours), accumulated error is 0.3433 sec
  - Iraqi Scud travels ~0.6 km in 0.3433 sec; interception failed; 28 dead
- Vancouver Stock Exchange (1992)
  - Inception of new market index with initial value 1000.000
  - Index computations **truncated** to 3 decimal places (round-off error)
  - Accumulated truncations led to an erroneous loss of around 25 points per month
  - 22 months later, recomputed value is 524.881; but real value is 1009.811
- Error changes Germany's parliamentary makeup (1992)
  - The 5% clause: no party with less than 5% of the vote may be seated in parliament
  - Software counting votes **round up results** to 1 decimal place
  - Green party gets 4.97%, software prints it out as 5.0%
  - Green party gets seated, Social-Democrats (SPD) lose a seat in Schleswig-Holstein
  - Most unfortunate: the lost seat is of the candidate for minister-president