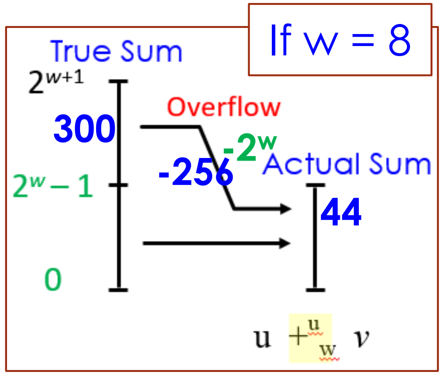
CMPT 295

Unit - Data Representation

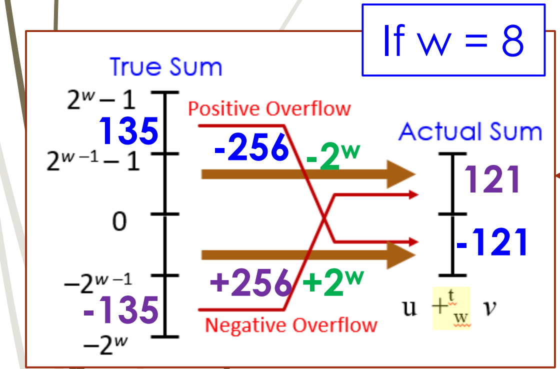
Lecture 5 – Representing fractional numbers in memory
– IEEE floating point representation

Last Lecture

- Demo of size and sign conversion in C: code and results posted!
- Addition:
  - Unsigned/signed:
    - Behave the same way at the bit level
    - Interpretation of resulting bit vector (sum) may differ
  - Unsigned addition -> true sum may overflow its w bits in memory



    - If so, then actual sum = (x + y) mod 2^w (equivalent to subtracting 2^w from true sum (x + y))
  - Signed addition -> true sum may overflow its w bits in memory



    - If so then …
      - actual sum = $U2T_w$ [(x + y) mod 2^w]
      - true sum may be too +ve -> positive overflow OR too –ve -> negative overflow

Conclusion: the same bit pattern is interpreted differently.

- Subtraction
  - Becomes an addition where the 2nd operand is transformed into its additive inverse in two's complement
- Multiplication:
  - Unsigned: actual product = (x * y) mod 2^w
  - Signed: actual product = $U2T_w$ [(x * y) mod 2^w]
  - Can be replaced by additions and shifts

Conclusion: the same bit pattern is interpreted differently.

Questions

- Why are we learning this?
- What can we do in our program when we suspect that overflow may occur?

Demo – Looking at integer additions in C

- What does the demo illustrate?
  - Unsigned addition
    - Without overflow
    - With overflow
    - Can overflow be predicted?
  - Signed addition
    - Without overflow
    - With positive overflow and negative overflow
    - Can overflow be predicted?
- This demo (code and results) posted on our course web site

Today's Menu

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …

We'll illustrate what we covered today by having a demo!

R2B(X)

Converting a fractional decimal number into a binary number (bit vector)

- How would 346.625 (= 346 5/8) be represented as a binary number?
- Expanding the subtraction method we have already seen:

346.625 -> 346 - 256 = 90 -> 1 x 2^8 MSb

90 - 128 -> ☹ -> 0 x 2^7

90 - 64 = 26 -> 1 x 2^6

26 - 32 -> ☹ -> 0 x 2^5

26 - 16 = 10 -> 1 x 2^4

10 - 8 = 2 -> 1 x 2^3

2 - 4 -> ☹ -> 0 x 2^2

2 - 2 = 0 -> 1 x 2^1

0 - 1 -> ☹ -> 0 x 2^0 LSb

.625 - 0.5 = 0.125 -> 1 x 2^{-1} MSb

.125 - 0.25 -> ☹ -> 0 x 2^{-2}

.125 - 0.125 = 0 -> 1 x 2^{-3} LSb

Negative Powers of 2

$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$
$2^{-5} = 0.03125$

Binary representation is: (MSb) 1 0 1 0 1 1 0 1 0 (LSb) . (MSb) 1 0 1 (LSb) $_2$

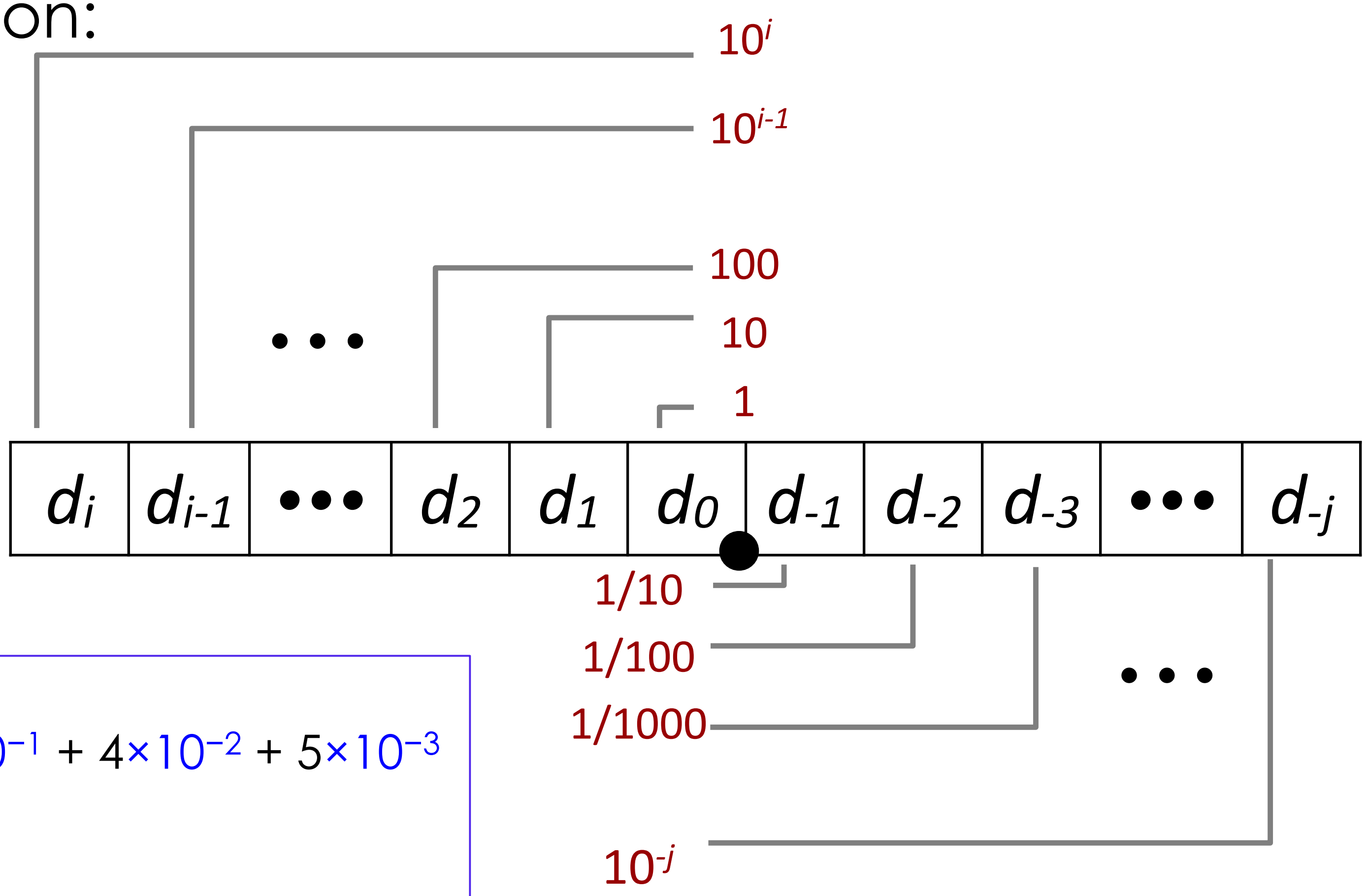
B2R(X)

Converting a binary number into a fractional decimal number

- How would 1011.101_2 be represented as a fractional decimal number?

Review: Fractional decimal numbers

- Positional notation:



<u>Example</u>:

$$2.345 = 2\times10^{0} + 3\times10^{-1} + 4\times10^{-2} + 5\times10^{-3}$$

10^{0}

10^{-1}

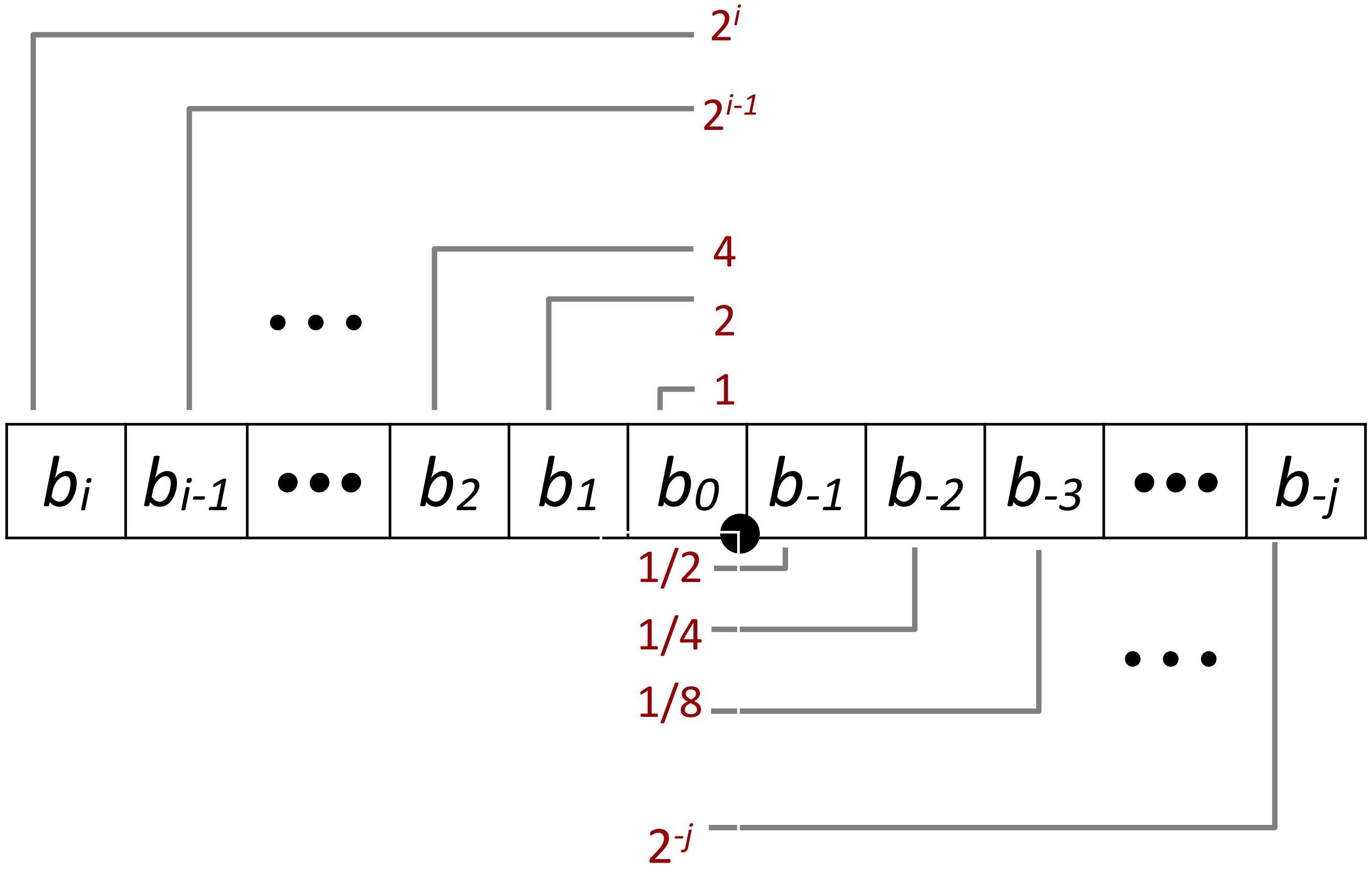
10^{-2}

10^{-3}

B2R(X)

Converting a binary number into a fractional decimal number

- Positional notation: can this be a possible encoding scheme?



B2R(X)

Converting a binary number into a fractional decimal number

- How would 1011.101_2 be represented as a fractional decimal number?
- Using the positional encoding scheme:

1011.101_2 =>

1011_2 -> $1 \times 2^3 + 1 \times 2^1 + 1 \times 2^0 = 11_{10}$

$.101_2$ -> $1 \times 2^{-1} + 1 \times 2^{-3} = 0.5 + 0.125 = 0.625_{10}$

Result:

Negative Powers of 2
$2^{-1} = 0.5$
$2^{-2} = 0.25$
$2^{-3} = 0.125$
$2^{-4} = 0.0625$
$2^{-5} = 0.03125$
$2^{-6} = 0.015625$
$2^{-7} = 0.0078125$
$2^{-8} = 0.00390625$

Positional notation as encoding scheme?

- One way to answer this question is to investigate whether the encoding scheme allows for **arithmetic operations**
- Let's see: Using the positional notation as an encoding scheme produces fractional binary numbers that can be
  - added
  - multiplied by 2 by shifting left
  - divided by 2 by shifting right (unsigned)

- Example: $1011.101_2 = 11\ 5/8 \Rightarrow 8 + 2 + 1 + 1/2 + 1/8$

Divide by 2: >> $101.1101_2 = 5\ 13/16 \Rightarrow 4 + 1 + 1/2 + 1/4 + 1/16$

Divide by 2: >> $10.11101_2 = 2\ 29/32 \Rightarrow 2 + 1/2 + 1/4 + 1/8 + 1/32$

$1011.101_2 = 11\ 5/8 \Rightarrow 8 + 2 + 1 + 1/2 + 1/8$

Multiply by 2: << $10111.01_2 = 23\ 1/4 \Rightarrow 16 + 4 + 2 + 1 + 1/4$

So far so good! ☺

Positional notation as encoding scheme?

- Advantage (so far):
  - Straightforward arithmetic: can shift to multiply and divide, convert
- Disadvantage:
  - Cannot encode all fractional numbers:
    - Can only represent numbers of the form $x/2^k$ (what about 1/5 or -34.8)
  - Only one setting of binary point within the w bits -> this limits the range of possible values
    - What is this range?

      Example -> w = 32 bits and binary point located at 16th bit :

      1111111111111111.1111111111111111

      [0 .. 131071] [0 .. 1 - ε]
  - Range: [0.0 .. 131071.99999….]

Not so good anymore! ☹

Representing fractional numbers in memory

- Here is another possible encoding scheme:
  IEEE floating point representation (IEEE Standard 754)
- Overview:
  - Binary Numerical Form: $V = (-1)^s \, M \, 2^E$
    - **s** – Sign bit -> determines whether number is negative or positive
    - **M** – Significand (or Mantissa) -> fractional part of number
    - **E** – Exponent
  - Form of bit pattern:

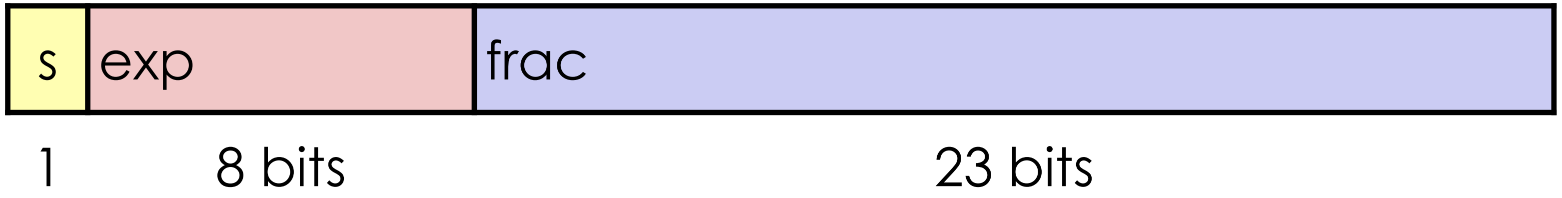
s	exp	frac

    - Most significant bit (MSb) **s** (similar to sign-magnitude encoding)
    - exp field encodes **E** (but is not equal to E)
    - frac field encodes **M** (but is not equal to M)

IEEE Floating Point Representation Precision options

- Single precision: 32 bits ≈ 7 decimal digits, range:$10^{\pm 38}$

In C:



- Double precision: 64 bits ≈ 16 decimal digits, range:$10^{\pm 308}$

IEEE Floating Point Representation
Three "kinds" of values

Numerical Form: $V = (-1)^s\, M\, 2^E$

s	exp	frac
	k bits	n bits

11…11 (all 1's)
special cases

00…00 (all 0's)
denormalized

exp ≠ 0 and exp ≠ 11…11
normalized

E = exp – bias
and bias = $2^{k-1} - 1$

Why is **E** biased? Using single precision as an example:

- exp range: [00000001 .. 11111110] and bias = $2^{8-1} - 1$
- **E** range: [-126 .. 127]
- **If no bias: E** range: [1 .. 254] **=> 2^1 to 2^{254}**

so cannot express numbers < 2 ☹

M = 1 + frac

Why adding 1 to frac?
Because number V is first normalized before it is converted.

Review: Scientific Notation and normalization

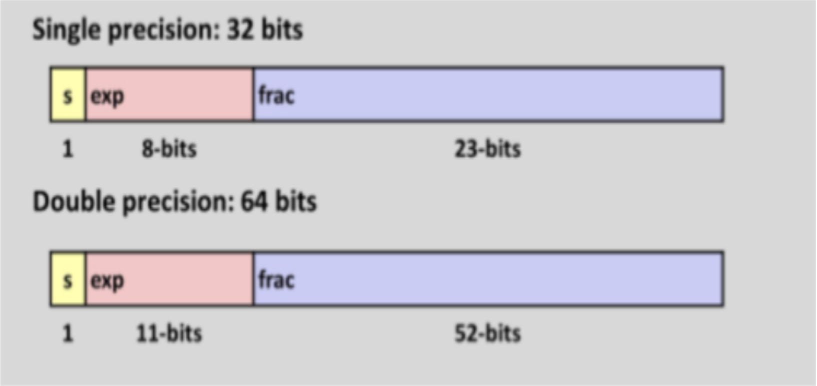
- From Wikipedia:
  - **Scientific notation** is a way of expressing numbers that are too large or too small (usually would result a long string of digits) to be conveniently written in decimal form.
  - In scientific notation, nonzero numbers are written in the form $m \times 10^{n}$
  - In **normalized notation**, the exponent **n** is chosen so that the absolute value of the significand **m** is at least 1 but less than 10.
- Examples:
  - A proton's mass is 0.0000000000000000000000000016726 kg -> 1.6726×10^{-27} kg
  - Speed of light is 299,792,458 m/s -> $2.99792{,}458 \times 10^{8}$ m/s

Syntax	$+/-$	$d_0 . d_{-1}\, d_{-2}\, d_{-3} \ldots d_{-n}$	$\times\ b$	^{exp}
	sign	significand	base	exponent

- Let's try: $1\,0\,1\,0\,1\,1\,0\,1\,0\,.\,1\,0\,1_2$ ->

Summary

- Representing integral numbers (signed/unsigned) in memory:
  - Encode schemes allow for small range of values exactly
- Representing fractional numbers in memory:
  1. Positional notation (advantages and disadvantages)
  2. IEEE floating point representation: wider range, mostly approximately
- Overview of IEEE Floating Point representation
  - V = $(-1)^s \times M \times 2^E$
  - Precision options
  - 3 kinds: **normalized,** denormalized and special values



Today's Menu

- Representing data in memory – Most of this is review
  - "Under the Hood" - Von Neumann architecture
  - Bits and bytes in memory
    - How to diagram memory -> Used in this course and other references
    - How to represent series of bits -> In binary, in hexadecimal (conversion)
    - What kind of information (data) do series of bits represent -> Encoding scheme
    - Order of bytes in memory -> Endian
  - Bit manipulation – bitwise operations
    - Boolean algebra + Shifting
- Representing integral numbers in memory
  - Unsigned and signed
  - Converting, expanding and truncating
  - Arithmetic operations
- Representing real numbers in memory
  - IEEE floating point representation
  - Floating point in C – casting, rounding, addition, …