CMPT 295

Unit - Data Representation

Lecture 6 – Representing fractional numbers in memory
– IEEE floating point representation – cont'd

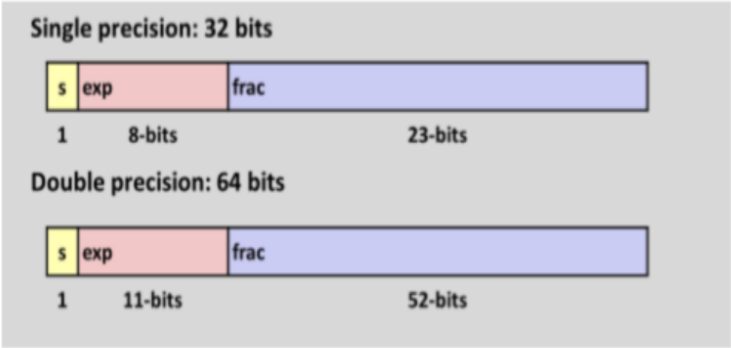
Have you heard of that new band "1023 Megabytes"?

They're pretty good,
but they don't have a gig just yet.

Last Lecture

- Representing integral numbers in memory
  - Can encode a small range of values exactly (in 1, 2, 4, 8 bytes)
    - For example: We can represent the values -128 to 127 exactly in 1 byte using a `signed char` in C
- Representing fractional numbers in memory
  1. Positional notation has some advantages, but also disadvantages -> so not used!
  2. IEEE floating point representation: can encode a much larger range of values approximately (in 4 or 8 bytes)

     e.g., single precision: $[10^{-38}..10^{38}]$
- Overview of IEEE floating point representation
  - Precision options
  - $V = (-1)^s \times M \times 2^E$
  - **s** –> sign bit
  - exp encodes **E** (but != **E**)
  - frac encodes **M** (but != **M**)



We interpret the bit vector (expressed in IEEE floating point encoding) stored in memory using this equation

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, …

IEEE Floating Point Representation
Three "kinds" of values

We interpret the bit vector (expressed in IEEE floating point encoding) stored in memory using this equation

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

exp and frac interpreted as unsigned

s	exp	frac
	k bits	n bits

2

If **exp** = 00…00
(all 0's)
⇒ **Denormalized**

Equations:
E = 1 – bias and bias = $2^{k-1} - 1$
M = frac

1

If **exp** ≠ 0 and **exp** ≠ 11…11
(exp range: [00000001 .. 11111110])
⇒ **Normalized**

Equations:
E = exp – bias and bias = $2^{k-1} - 1$
M = 1 + frac

3

If **exp** = 11…11
(all 1's)
⇒ **Special cases**

Case 1: frac = **000…0**
Case 2: frac ≠ **000…0**

IEEE floating point representation - **normalized**

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

s	exp	frac
	k bits	n bits

If **exp** ≠ 0 and **exp** ≠ 11…11
(exp range: [00000001 .. 11111110])
⇒ **Normalized**

Equations:
$E = exp - bias$ and $bias = 2^{k-1} - 1$
$M = 1 + frac$

If we set s=0 and M=1.0

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

s	exp	frac
1	8 bits	23 bits

- **exp** range: [00000001 .. 11111110] => $[1_{10} .. 254_{10}]$
- **If E is not biased** (i.e., **E = exp**), then **E** range: $[1_{10} .. 254_{10}]$
- V range: $[2^1 .. 2^{254}] = [2^0 .. \sim2.89 \times 10^{76}]$ — so cannot express numbers < 2 ☹
- **By biasing E** (i.e., **E = exp – bias**), then **E** range: [1 – 127 .. 254 - 127]
  = [-126 .. 127] (since k = 8, $bias = 2^{8-1} - 1 = 127$)
- V range: $[2^{-126} .. 2^{127}] = [\sim1.18 \times 10^{-38} .. \sim1.7 \times 10^{38}]$ — so can now express very small (and large) numbers ☺

~ means "approx."

Why adding 1 to frac?
Because the number (or value) 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 to be conveniently written in decimal form (as they are long strings of digits).
  - In scientific notation, nonzero numbers are written in the form $+/- M \times 10^{n}$
  - In **normalized notation**, the exponent **n** is chosen such that the absolute value of the significand **M** is at least 1 (**M** = 1.0) but less than **the base**

M range for **base 10** => [1.0 .. **10.0** – ε]
M range for **base 2** => [1.0 .. **2.0** – ε]

- Examples:
  - A proton's mass is 0.0000000000000000000000000016726 kg -> 1.6726×10^{-27} kg
    - $\times 2^0$ 16.726×10^{-28}
  - Speed of light is 299,792,458 m/s -> $2.99792{,}458\times10^{8}$ m/s

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

- Let's try: 1010110110.101_2 -> $\times 2^0$ $1.010110110101_2 \times 2^8$

moving the binary point makes the number smaller ∴ makes the exponent larger

Let's try normalizing these fractional binary numbers!

always

1. $101011010.101_2 \times 2^0 = 1.01011010101_2 \times 2^8$

   M↓ E↑

2. $0.000000001101_2 \times 2^0 = 1.101_2 \times 2^{-9}$

   M↑ E↓

3. $11000000111001_2 \times 2^0 = 1.1000000111001_2 \times 2^{13}$

   M↓ E↑

IEEE floating point representation (single precision => k = 8 bits, n => 23 bits)

- Once V is normalized, we apply the equations
  - $V = (-1)^s M 2^E = 1.01011010101_2 \times 2^8$
  - s = 0 → +ve
  - E = exp – bias where bias = $2^{k-1} - 1 = 2^7 - 1 = 128 - 1 = 127$

    exp = E + bias = 8 + 127 = 135_{10} → U2B(135_{10}) = 10000111_2
  - M = 1 + frac ⇒ frac = M – 1 ⇒ $1.01011010101_2 - 1_2 = .01011010101_2$

    ↑ not part of bit pattern

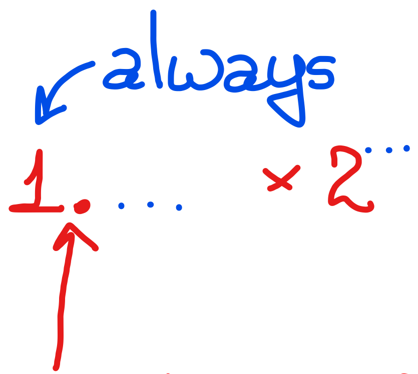
s	exp	frac
0	10000111	01011010101000000000000
	k bits => 8 bits	n bits => 23 bits

must pad until we have 23 bits

- bit vector in memory: 0100|0011|1010|1101|0101|0000|0000|0000

  in hex: 0x43AD5000

Why adding 1 to **frac** (or subtracting 1 from **M**)?



- Because the number (or value) V is first normalized before it is converted.
  - As part of this normalization process, we transform our binary number such that its significand **M** is within the range [1.0 .. **2.0** – ε]
  - Remember: **M** range for **base 2** => [1.0 .. **2.0** – ε]
  - This implies that **M** is always at least 1.0, so its integral part always has the value 1
  - So since this bit is always part of **M**, IEEE 754 does not explicitly save it in its bit pattern (i.e., in memory)
  - Instead, this bit is implied!

Why adding 1 to frac (or subtracting 1 from M)?

We get the leading bit for *free*!

Implying this bit has the following effects:

1. We save 1 bit when we convert (represent) a fractional decimal number into a bit pattern using IEEE 754 floating point representation this bit is not in (subtracted from M)
2. We have to add this 1 bit back when we convert from a bit pattern (IEEE 754 floating point representation) back to a fractional decimal

Example: $V = (-1)^{s} M\ 2^{E} = 1.01011010101 \times 2^{8}$

M = 1. 01011010101 => **M** = 1 + frac

This bit is implied hence not stored in the bit pattern produced by the IEEE 754 floating point representation, and what we store in the frac part of the IEEE 754 bit pattern is 01011010101

IEEE floating point representation (single precision)

- What if the 4 bytes starting at M[0x0000] represented a fractional decimal number (encoded as an IEEE floating point number) -> value?

single precision

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

	Interpreted as unsigned	Interpreted as unsigned
1	10000111	01011010101000000000000
	k = 8 bits	n = 23 bits

Address	M[ ]
size-1	
⋮	
0x0003	11000011
0x0002	10101101
0x0001	01010000
0x0000	00000000

Little endian

- exp $\neq$ 0 and exp $\neq 11111111_2$ -> **normalized**
- **s** =
- **E** = exp – bias where bias = $2^{k-1} - 1 = 2^{7} - 1 = 128 - 1 = 127$
- **E** = ______________ - 127 =
- **M** = 1 + frac = 1 + ________________________________
- V = __________________________

Let's give it a go!

- What if the 4 bytes starting at M[`0x0000`] represented a fractional decimal number (encoded as an IEEE floating point number) -> value?

single precision

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

0	10001100	11011011011000000000000
	Interpreted as unsigned	Interpreted as unsigned
	k = 8 bits	n = 23 bits

Address	M[ ]
`size-1`	
⋮	
`0x0003`	01000110
`0x0002`	01101101
`0x0001`	10110000
`0x0000`	00000000

Little endian

- exp ≠ 0 and exp ≠ 11111111_2 -> **normalized**
- **s** =
- **E** = exp – bias where bias = $2^{k-1} - 1 = 2^{7} - 1 = 128 - 1 = 127$
- **E** = ______________ - 127 =
- **M** = 1 + frac = 1 + ________________________________
- V = __________________________

IEEE floating point representation (single precision)

- How would **47.21875** be encoded as IEEE floating point number?

1. Convert 47.28 to binary (using the positional notation *R2B(X)*) =>
   - $47 = 101111_2$
   - $.21875 = .00111_2$
2. Normalize binary number:

   $101111.00111 \Rightarrow 1.0111100111_2 \times 2^5$

   $V = (-1)^s\, M\, 2^E$
3. Determine …

   s = 0

   E = exp – bias where bias = $2^{k-1} - 1 = 2^7 - 1 = 128 - 1 = 127$

   exp = **E** + bias = 5 + 127 = 132 => *U2B(132)* => 10000100

   M = 1 + frac -> frac = **M** - 1 => $1.0111100111_2 - 1 = .0111100111_2$

4.

0	10000100	01111001110000000000000

5. 0x423CE000

IEEE floating point representation (single precision)

- How would **12345.75** be encoded as IEEE floating point number?

single precision

1. Convert 12345.75 to binary
   - 12345 => .75 =>
2. Normalize binary number:
3. Determine …

$V = (-1)^{s} \mathbf{M}\, 2^{\mathbf{E}}$

s =

E = exp – bias where bias = $2^{k-1} - 1 = 2^{7} - 1 = 128 - 1 = 127$

exp = **E** + bias =

M = 1 + frac -> frac = **M** - 1

4. 

5. Express in hex:

Summary

- IEEE Floating Point Representation
  1. Denormalized
  2. Special cases
  3. Normalized => **exp** ≠ 000…0 and **exp** ≠ 111…1
     - Single precision: **bias** = 127, **exp**: [1..254], **E**: [-126..127] => [10^{-38} … 10^{38}]
     - Called "normalized" because binary numbers are normalized
       - Effect: "We get the leading bit for *free*"
         - Leading bit is always assumed (never part of bit pattern)
- IEEE floating point number as encoding scheme
  - Fractional decimal number ⇔ IEEE 754 (bit pattern)
  - $V = (-1)^{s} M\, 2^{E}$
    - **s** is sign bit, **M** = 1 + frac, **E** = exp – bias, bias = $2^{k-1} - 1$ and k is width of exp

Next Lecture

- 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, …