CMPT 295

Unit - Data Representation

Lecture 7 – Representing fractional numbers in memory
– IEEE floating point representation, their arithmetic operations and `float` in C

Last Lecture

- IEEE floating point representation
  1. **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 as part of the conversion process
       - Effect: "We get the leading bit for *free*" => leading bit is always assumed (never part of bit pattern)
  - Conversion: 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
  2. Denormalized
  3. Special cases

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 (single precision)

- How would **47.28** be encoded as IEEE floating point number?

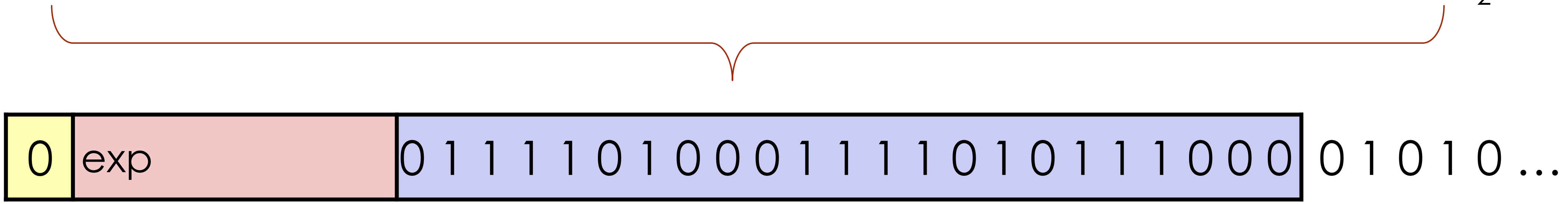
1. Convert 47.28 to binary (using the positional notation *R2B(X)*) =>
   - $47 = 101111_2$
   - $.28 = .01000111101011100001010001111010111000010100011110101110000101\ldots_2$

$.28 = .\overline{01000111101011100001}_2$

2. Normalize binary number:

$101111.01000111101011100001010001111010111000010100011110101110000101\ldots_2 (\times 2^0)$

$\Rightarrow 1.\underbrace{01111010001111010111000010100011110101110000101000111101011 1000\ldots}_{}{}_2 \times 2^5$



0	exp	0 1 1 1 1 0 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0

0 1 0 1 0 …

Rounding

This **selection** is done by looking at the bit pattern around the ***rounding position***.

- First, identify bit at *rounding position*
- Then select which kind of *rounding* we must perform:

1. **Round up**
    - When the value of the bits to the right of the bit at *rounding position* is > half the worth of the bit at *rounding position*
    - We "round" up by adding 1 to the bit at *rounding position*
2. **Round down**
    - When the value of the bits to the right of the bit at *rounding position* is < half the worth of the bit at *rounding position*
    - We "round" down by discarding the bits to the right of the bit at *rounding position*
3. **Round to "even number"**
    - When the value of the bits to the right of the bit at *rounding position* is exactly half the worth of bit at *rounding position*, i.e., when these bits are $100...0_2$
    - We "round" such that the bit at the *rounding position* becomes 0
        - If the bit at *rounding position* is 1 => then we "round to even number" by "rounding up" i.e., by adding 1
        - If the bit at *rounding position* is already 0 => then we "round to even number" by "rounding down" i.e., by discarding the bits to the right of the bit at *rounding position*

Rounding (and error)

- Example: *rounding position ->* round to nearest 1/4 (2 bits right of binary point)

Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up to even)	3
2 5/8	10.10100_2	10.10_2	(1/2—down to even)	2 ½

- Explain value of a bit versus worth of a bit
  - value of a bit
  - worth of a bit

Back to IEEE floating point representation

In the process of converting fractional decimal numbers to IEEE floating point numbers (i.e., bit patterns in fixed-size memory), we apply these same rounding rules …

Using the same numbers in our example:

Binary

$10.0\mathbf{0}011_2$

$10.0\mathbf{0}110_2$

$10.1\mathbf{1}100_2$

$10.1\mathbf{0}100_2$

and this is our 24th bit

Imagine this is our 23rd bit of the `frac`

=> *rounding position*

Homework

Let’s practice converting and rounding!

- How would **346.62** be encoded as IEEE floating point number (single precision) in memory?
  - Also, can you compute the minimum value of the error introduced by the rounding process since 346.62 can only be approximated when encoded as an IEEE floating point representation

2. Denormalized values

$V = (-1)^{s}\ M\ 2^{E}$
$E = 1 - bias$
$bias = 2^{k-1} - 1$
$M = frac$

- Condition: **`exp`** = **`00000000`** (single precision)
- Denormalized Values: $V = (-1)^{s}\ M\ 2^{E} = +/- 0.frac \times 2^{-126}$
  - Case 1: **`frac`** = **`000…0`** **`->`** +0 and –0
  - Case 2: **`frac`** ≠ **`000…0`** **`->`** numbers closest to 0.0 (equally spaced)
- Smallest:

0	00000000	00000000000000000000001

$V = (-1)^{s}\ M\ 2^{E} = 0.00000000000000000000001 \times 2^{-126} = \sim 1.4 \times 10^{-45}$

- Largest:

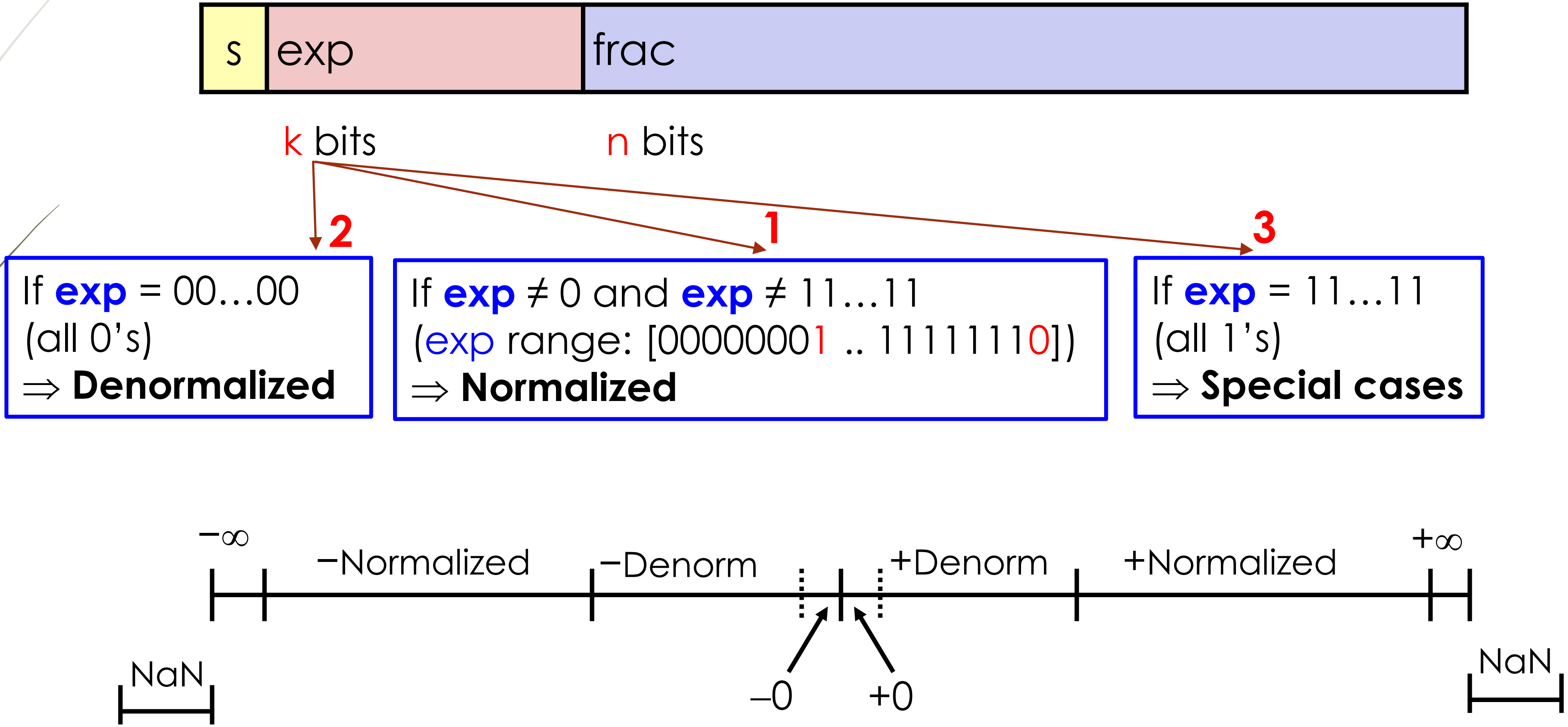
0	00000000	11111111111111111111111

$V = (-1)^{s}\ M\ 2^{E} = 0.11111111111111111111111 \times 2^{-126} = \sim 1.18 \times 10^{-38}$

3. Special values

- Condition: **`exp`** = `111…1`
- Case 1: **`frac`** = **`000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., 1.0/0.0 = −1.0/−0.0 = +∞, 1.0/−0.0 = −∞
- Case 2: **`frac`** ≠ **`000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - e.g., sqrt(–1), ∞ − ∞, ∞ × 0
  - NaN propagates other NaN: e.g., NaN + x = NaN

Axis of all floating point values



s | exp | frac

k bits

n bits

2

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

1

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

3

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

−∞

−Normalized

−Denorm

+Denorm

+Normalized

+∞

NaN

–0

+0

NaN

What if floating point represented with 8 bits

To get a feel for all possible values expressible using IEEE like conversion, we use a small w. Here, instead of w = 32, we use w = 8. This way, we can enumerate all values.

	s	exp	frac	E	Value	
Denormalized numbers	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	**closest to zero**
	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
	0	0000	111	-6	7/8*1/64 = 7/512	**largest denorm**
Normalized numbers	0	0001	000	-6	8/8*1/64 = 8/512	**smallest norm**
	0	0001	001	-6	9/8*1/64 = 9/512	
	…					
	0	0110	110	-1	14/8*1/2 = 14/16	
	0	0110	111	-1	15/8*1/2 = 15/16	**closest to 1 below**
	0	0111	000	0	8/8*1 = 1	
	0	0111	001	0	9/8*1 = 9/8	**closest to 1 above**
	0	0111	010	0	10/8*1 = 10/8	
	…					
	0	1110	110	7	14/8*128 = 224	
	0	1110	111	7	15/8*128 = 240	**largest norm**
	0	1111	000	n/a	inf	

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

Denormalized:

$E = 1 - bias$

$bias = 2^{k-1} - 1$

$M = frac$

Normalized:

$E = exp - bias$

$bias = 2^{k-1} - 1$

$M = 1 + frac$

Conversion in C

- Casting between **`int`**, **`float`**, and **`double`** changes bit pattern
  - **`double`/`float`** → **`int`**
    - Truncates fractional part
  - **`int`** → **`float`**
    - Exact conversion, as long as `frac` (obtained when the **`int`** is normalized) fits in 23 bits
    - Will round according to rounding rules
  - **`int`** → **`double`**
    - Exact conversion, as long as `frac` (obtained when the **`int`** is normalized) fits in 52 bits
    - Will round according to rounding rules

Demo - C code

- **Conversion** – Observe the change in bit pattern
  - `int` → `float`
  - `float` → `int`
- **Addition**
- **Associativity** – For floating point numbers f1, f2 and f3:
  - Is it always true that $(f1 + f2) + f3 = f1 + (f2 + f3)$?
  - Is it always true that $(f1 * f2) * f3 = f1 * (f2 * f3)$?
- **Rounding** – Effect of errors caused by rounding

Floating point arithmetic

- $\mathbf{x} +_{\mathbf{f}} \mathbf{y} = \mathbf{Round}(\mathbf{x} + \mathbf{y})$
- $\mathbf{x} \times_{\mathbf{f}} \mathbf{y} = \mathbf{Round}(\mathbf{x} \times \mathbf{y})$
- Basic idea:
  - First compute true result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into **`frac`**

Summary

- Most fractional decimal numbers cannot be exactly encoded using IEEE floating point representation -> rounding
- Denormalized values
  - Condition: **`exp = 0000…0`**
  - 0 <= denormalized values < 1, equidistant because all have same 2^E
- Special values
  - Condition: **`exp`** = **`1111…1`**
    - Case 1: **`frac`** = **`000…0 ->`** ∞ (infinity)
    - Case 2: **`frac`** ≠ **`000…0 ->`** NaN
- Impact on C
  - Conversion/casting, rounding
  - Arithmetic operators:
    - Behaviour not the same as for *real* arithmetic => violates associativity

Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point data & operations