Sewanee: The University of the South

CS 270: Computer Organization

Floating Point Representation

Instructor:
Professor Stephen P. Carl

Sewanee: The University of the South

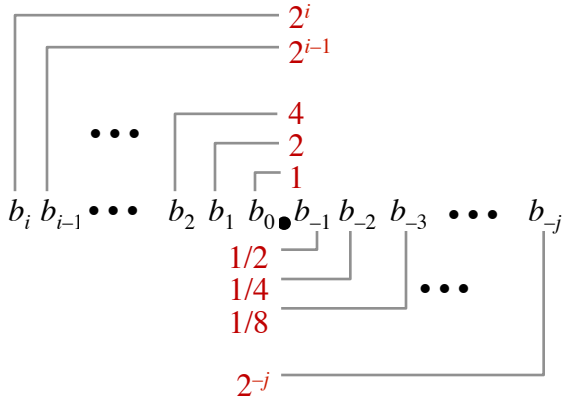
Summary: Integers

- **Representation: unsigned and signed**
- **Conversions and casting**
  Bit representation maintained but reinterpreted
- **Expanding, truncating**
  Truncating = modular arithmetic
- **Addition, negation, multiplication, shifting**
  Operations are mod 2^w
- **"Ring" properties hold**
  Associative, commutative, distributive, additive 0 and inverse
- **Ordering properties do not hold**
  $u > 0$ does not mean $u + v > v$
  $u, v > 0$ does not mean $u \cdot v > 0$

Fractional binary numbers

What is 1011.101?

Fractional Binary Numbers



Representation

Bits to right of "binary point" represent fractional powers of 2

This represents the rational number given by:

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

Sewanee: The University of the South

Fractional Binary Numbers: Examples

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
63/64	0.111111_2

Observations

To divide by 2, shift bits right

To multiply by 2, shift bits left

Numbers of the form $0.111111\ldots_2$ are just below 1.0

- $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
- Use notation $1.0 - \varepsilon$

Sewanee: The University of the South

Representable Numbers

Floating point representation

- Can only *exactly* represent numbers of the form $x/2^k$
- Other rational numbers have repeating bit representations:

Value	Representation
1/3	$0.0101010101[01]\ldots_2$
1/5	$0.001100110011[0011]\ldots_2$
1/10	$0.0001100110011[0011]\ldots_2$

IEEE Floating Point Standards

- **The IEEE Standard 754**
  - Was established in 1985 as uniform standard for floating point arithmetic
    - (Before that, many idiosyncratic formats)
  - Now supported by all major CPUs
- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, and underflow
  - But: hard to make fast in hardware - numerical analysts predominated over hardware designers in defining standard

Floating Point Representation

Numerical Form:

$$(-1)^s\, M\, 2^E$$

Sign bit ***s*** determines whether number is negative or positive

Significand ***M*** normally a fractional value in range [1.0,2.0).

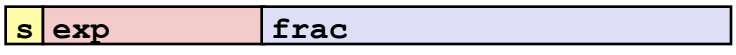
Exponent ***E*** weights value by power of two

Encoding:

MSB **`s`** is sign bit ***s***

`exp` field encodes ***E*** (but is not equal to E)

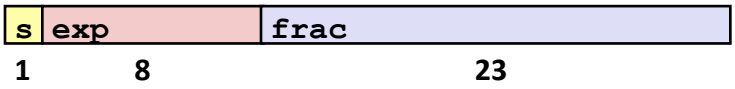
`frac` field encodes ***M*** (but is not equal to M)



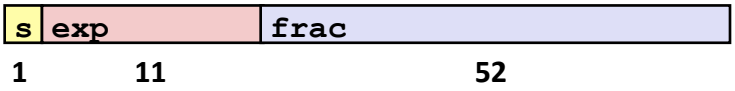
Sewanee: The University of the South

Precisions

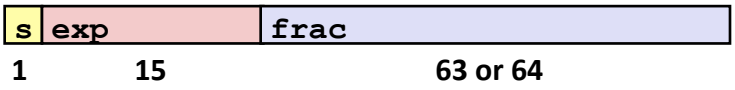
Single precision: 32 bits



Double precision: 64 bits



Extended precision: 80 bits (Intel only)



Sewanee: The University of the South

Normalized Values

- **Condition: `exp` ≠ `000…0` and `exp` ≠ `111…1`**
- **Exponent coded as *biased* value: *E* = *Exp* – *Bias***

  Exp: unsigned value **`exp`**

  Bias = $2^{e-1} - 1$, where e is number of exponent bits
  - Single precision: 127 (*Exp*: 1…254, *E*: -126…127)
  - Double precision: 1023 (*Exp*: 1…2046, *E*: -1022…1023)

- **Significand coded with implied leading 1: *M* = `1.xxx…x`$_2$**

  `xxx…x`: bits of **`frac`**

  Minimum when **`000…0`** (*M* = 1.0)

  Maximum when **`111…1`** (*M* = 2.0 – ε)

  Get extra leading bit for "free"

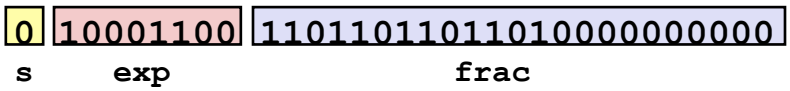
Normalized Encoding Example

- **Value:** `float F = 15213.0;`
  $15213_{10} = 11101101101101_2$
  $= 1.1101101101101_2 \times 2^{13}$

- **Significand**
  M = $1.\underline{1101101101101}_2$
  `frac` = $\underline{1101101101101}0000000000_2$

- **Exponent**
  E = 13
  Bias = 127
  Exp = 140 = 10001100_2

- **Result:**



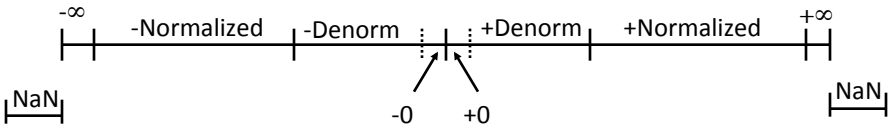
Denormalized Values

- **Condition: `exp` = 000…0**
- **Exponent value: *E* = –*Bias* + 1 (instead of E = 0 – Bias)**
- **Significand coded with implied leading 0: *M* = 0 . `xxx…x`$_2$**
  - **`xxx…x`**: bits of **`frac`**
- **Cases**
  - **`exp` = 000…0, `frac` = 000…0**
    - Represents value 0
    - Note distinct values: +0 and –0
  - **`exp` = 000…0, `frac` ≠ 000…0**
    - Numbers very close to 0.0
    - You lose precision as numbers get smaller
    - Equispaced on floating point "number line"

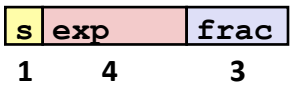
Special Values

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

Visualization: Floating Point Encodings



Tiny Floating Point Example



- **8-bit Floating Point Representation**
  - the sign bit is in the most significant bit.
  - the next four bits are the exponent, with a bias of 7.
  - the last three bits are the `frac`
- **Same general form as IEEE Format**
  - normalized, denormalized
  - representation of 0, NaN, infinity

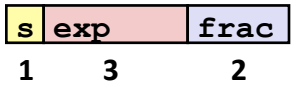
Dynamic Range (Positive Only)

	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	

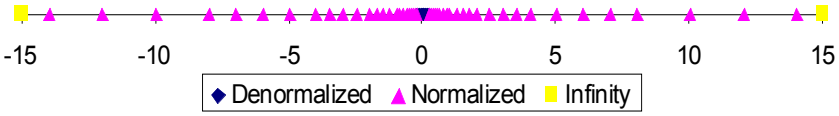
Sewanee: The University of the South

Distribution of Values

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is $2^{3-1}-1 = 3$



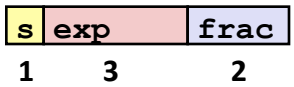
- **Notice how the distribution gets denser toward zero.**

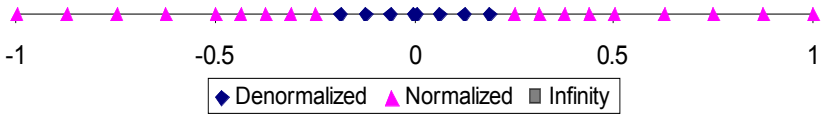


Sewanee: The University of the South

Distribution of Values (close-up view)

- **6-bit IEEE-like format**
  - e = 3 exponent bits
  - f = 2 fraction bits
  - Bias is 3





Sewanee: The University of the South

Interesting Numbers {single precision}

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm. Single ≈ 1.4×10^{-45}	00…00	00…01	$2^{-23} \times 2^{-126}$
Largest Denormalized Single ≈ 1.18×10^{-38}	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-126}$
Smallest Pos. Normalized Just larger than largest denormalized	00…01	00…00	1.0×2^{-126}
One	01…11	00…00	1.0
Largest Normalized Single ≈ 3.4×10^{38}	11…10	11…11	$(2.0 - \varepsilon) \times 2^{127}$

Sewanee: The University of the South

Interesting Numbers {double precision}

Description	exp	frac	Numeric Value
Zero	00…00	00…00	0.0
Smallest Pos. Denorm. Double ≈ 4.9×10^{-324}	00…00	00…01	$2^{-52} \times 2^{-126}$
Largest Denormalized Double ≈ 2.2×10^{-308}	00…00	11…11	$(1.0 - \varepsilon) \times 2^{-1022}$
Smallest Pos. Normalized ▪ Just larger than largest denormalized	00…01	00…00	1.0×2^{-1022}
One	01…11	00…00	1.0
Largest Normalized ▪ Double ≈ 1.8×10^{308}	11…10	11…11	$(2.0 - \varepsilon) \times 2^{1023}$

Special Properties of FP Encoding

- **FP Zero Same as Integer Zero**
  - All bits = 0
- **Can (Almost) Use Unsigned Integer Comparison**
  - Must first compare sign bits
  - Must consider -0 = 0
  - NaNs problematic
    - Will be greater than any other values
    - What should comparison yield?
  - Otherwise OK
    - Denorm vs. normalized
    - Normalized vs. infinity

Floating Point Operations: Basic Idea

$x +_f y = \text{Round}(x + y)$

$x \times_f y = \text{Round}(x \times y)$

Basic idea

First, compute exact result

Then, *make it fit* into desired precision

- Might overflow if exponent too large
- Might need to round to fit into `frac`

Rounding

Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
Towards zero	$1	$1	$1	$2	–$1
Round down (-∞)	$1	$1	$1	$2	–$2
Round up (+∞)	$2	$2	$2	$3	–$1
Nearest Even (default)	$1	$2	$2	$2	–$2

What are the advantages of the modes?

Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased: the sum of the set of positive numbers will consistently be over- or under-estimated

- **Applying to Other Decimal Places / Bit Positions**
  - When exactly halfway between two possible values, round so that *least significant digit is even*
  - E.g., round to nearest hundredth

1.2349999	1.23	(Less than half way)
1.2350001	1.24	(Greater than half way)
1.2350000	1.24	(Half way—round up)
1.2450000	1.24	(Half way—round down)

Rounding Binary Numbers

- **Binary Fractional Numbers**
  "Even" when least significant bit is 0
  "Half way" when bits to right of rounding position = $100\ldots_2$

- **Examples**
  Round to nearest 1/4 - 2 bits to 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)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

FP Multiplication

$(-1)^{s1}\ M1\ \ 2^{E1}\ \ \mathbf{x}\ \ (-1)^{s2}\ M2\ \ 2^{E2}$

- **Exact Result: $(-1)^s\ M\ \ 2^E$**
  Sign *s*: *s1* ^ *s2*
  Significand *M*: *M1* * *M2*
  Exponent *E*: *E1* + *E2*

- **Fixing**
  If *M* ≥ 2, shift *M* right, increment *E*
  If *E* out of range, set overflow
  Round *M* to fit `frac` precision

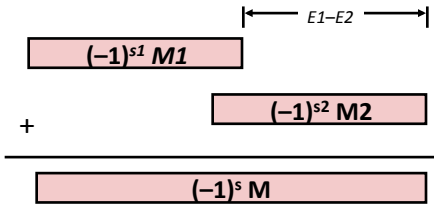
- **Implementation**
  Biggest chore is multiplying significands

Floating Point Addition

$(-1)^{s1}$ M1 2^{E1} + $(-1)^{s2}$ M2 2^{E2}

Assume *E1* > *E2*

- **Exact Result:** $(-1)^s$ *M* 2^E
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent *E*: *E1*



- **Fixing**
  - If *M* ≥ 2, shift *M* right, increment *E*
  - if *M* < 1, shift *M* left *k* positions, decrement *E* by *k*
  - Overflow if *E* out of range
  - Round *M* to fit `frac` precision

Mathematical Properties of FP Add

Algebraic properties

Closed under addition?	***Yes***
▪ But may generate infinity or NaN	
Commutative?	***Yes***
Associative?	***No***
▪ Overflow and inexactness of rounding	
0 is additive identity?	***Yes***
Every element has additive inverse	***Almost***
▪ Except for infinities & NaNs	

Monotonicity

$a \geq b \Rightarrow a+c \geq b+c$?	***Almost***
▪ Except for infinities & NaNs	

Mathematical Properties of FP Mult

Algebraic Properties

Closed under multiplication? ***Yes***
- But may generate infinity or NaN

Multiplication Commutative? ***Yes***

Multiplication is Associative? ***No***
- Possibility of overflow, inexactness of rounding

1 is multiplicative identity? ***Yes***

Multiplication distributes over addition? ***No***
- Possibility of overflow, inexactness of rounding

Monotonicity

$a \geq b \ \& c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***
- Except for infinities & NaNs

Floating Point in C

- **C Guarantees Two Levels**

  `float` single precision

  `double` double precision

- **Casting between `int`, `float`, and `double` changes bit representation:**

  `Double/float` → `int`
  - Truncate fractional part, like rounding toward zero
  - Not defined when out of range or NaN: Generally sets to TMin

  `int` → `double`
  - Exact conversion, as long as int has ≤ 53 bit word size

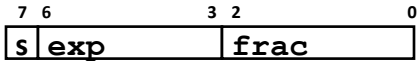
  `int` → `float`
  - Will round according to rounding mode

Sewanee: The University of the South

Creating Floating Point Numbers

Steps

1) Normalize to have leading 1
2) Round to fit within fraction
3) Postnormalize to deal with effects of rounding



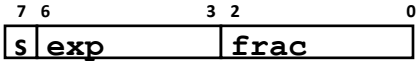
Example: Convert 8-bit unsigned numbers to tiny floating point format

Example Numbers

128	10000000
15	00001101
33	00010001
35	00010011
138	10001010
63	00111111

Sewanee: The University of the South

1) Normalize



To set binary point to get numbers of the form 1.xxxxx,

- adjust all to have leading one
- decrement exponent as shift left

Value	*Binary*	*Fraction*	*Exponent*
128	10000000	1.0000000	7
15	00001101	1.1010000	3
17	00010001	1.0001000	5
19	00010011	1.0011000	5
138	10001010	1.0001010	7
63	00111111	1.1111100	5

Sewanee: The University of the South

2) Rounding

Guard bit: LSB of result

Round bit: 1st bit removed **Sticky bit: OR of remaining bits**

Round up conditions

If Round = 1, Sticky = 1 ➔ > 0.5

If Guard = 1, Round = 1, Sticky = 0 ➔ Round to even

Value	*Fraction*	*GRS*	*Incr?*	*Rounded*
128	1.0000000	000	N	1.000
15	1.1010000	100	N	1.101
17	1.0001000	010	N	1.000
19	1.0011000	110	Y	1.010
138	1.0001010	011	Y	1.001
63	1.1111100	111	Y	10.000

Sewanee: The University of the South

3) Postnormalize

Note:

- Rounding may have caused overflow
- Handle by shifting right once & incrementing exponent

Value	*Rounded*	*Exp*	*Adjusted*	*Result*
128	1.000	7		128
15	1.101	3		15
17	1.000	4		16
19	1.010	4		20
138	1.001	7		134
63	10.000	5	1.000/6	64

Sewanee: The University of the South

Floating Point Puzzles

For each of the following C expressions, either:

- Argue that it is true for all argument values
- Explain why not true

```
int x = …;
float f = …;
double d = …;
```

Assume neither `d` nor `f` is NaN

```
x == (int)(float) x
x == (int)(double) x
f == (float)(double) f
d == (float) d
f == -(-f);
2/3 == 2/3.0
d < 0.0       ⇒   ((d*2) < 0.0)
d > f         ⇒   -f > -d
d * d >= 0.0
(d+f)-d == f
```

Sewanee: The University of the South

Summary

- **IEEE Floating Point represents numbers of form $M \times 2^E$**
- **One can reason about operations independent of implementation**
  - As if computed with perfect precision and then rounded
- **Not the same as real arithmetic**
  - Violates associativity/distributivity
  - Makes life difficult for compilers & programming industrial-strength numerical applications