Condition 1

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"

Denormalized Values

Condition 2

- Condition: `exp` = `000…0`

- Exponent value: $E = -Bias + 1$ (instead of E = 0 – Bias)
- Significand coded with implied leading 0: $M = 0 . \texttt{xxx…x}_2$
  - `xxx…x`: bits of `frac`
- Cases
  - `exp` = `000…0`, `frac` = `000…0`
    - Represents value 0
    - Note distinct values: +0 and –0 (why?)
  - `exp` = `000…0`, `frac` ≠ `000…0`
    - Numbers very close to 0.0
    - Lose precision as get smaller
    - Equispaced

Special Values

Condition 3

- **Condition: `exp` = `111…1`**

- **Case: `exp` = `111…1`, `frac` = `000…0`**
  - Represents value ∞ (infinity)
  - Operation that overflows
  - Both positive and negative
  - E.g., $1.0/0.0 = -1.0/-0.0 = +\infty$, $1.0/-0.0 = -\infty$

- **Case: `exp` = `111…1`, `frac` ≠ `000…0`**
  - Not-a-Number (NaN)
  - Represents case when no numeric value can be determined
  - E.g., sqrt(–1), $\infty - \infty$, $\infty * 0$

Closer Look at Round-To-Even

- **Default Rounding Mode**
  - Hard to get any other kind without dropping into assembly
  - All others are statistically biased
    - Sum of 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

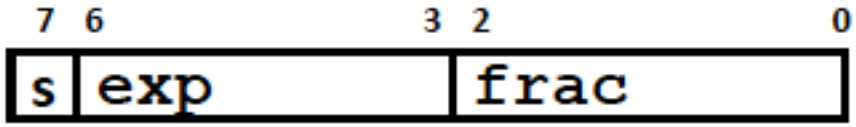
- **Binary Fractional Numbers**
  - "Even" when least significant bit is 0
  - "Half way" when bits to right of rounding position = $100\ldots_2$

- **Examples**
  - Round to nearest 1/4 (2 bits 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

Creating Floating Point Number

- **Steps**
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding

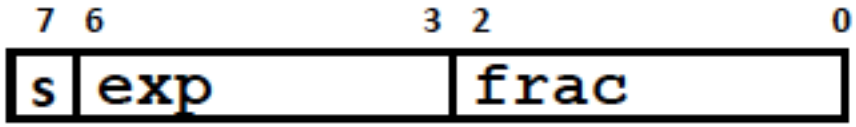


- **Case Study**
  - Convert 8-bit unsigned numbers to tiny floating point format
  - Example Numbers

10000000	**0 1110 000**
00001101	**0 1010 101**
00010001	**0 1011 000**
00010011	
10001010	
00111111	

Creating Floating Point Number

- **Steps**
  - Normalize to have leading 1
  - Round to fit within fraction
  - Postnormalize to deal with effects of rounding



- **Case Study**
  - Convert 8-bit unsigned numbers to tiny floating point format
  - Example Numbers

128	10000000	**0 1110 000**
15	00001101	**0 1010 101**
33	00010001	**0 1011 000**
35	00010011	**0 1011 010**
138	10001010	**0 1111 000**
63	00111111	**0 1100 11111 → 0 1101 000**

FP Multiplication

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

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

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

- **Implementation**
  - Biggest chore is multiplying significands

Mathematical Properties of FP Mult

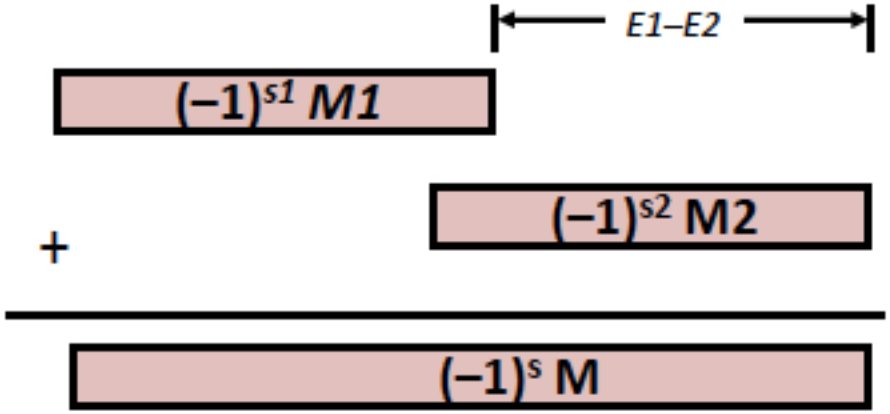
- **Compare to Commutative Ring**
  - Closed under multiplication? *Yes*
    - But may generate infinity or NaN
  - Multiplication Commutative? *Yes*
  - Multiplication is Associative? *No*
    - Possibility of overflow, inexactness of rounding
  - 1 is multiplicative identity? *Yes*
  - Multiplication 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 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 $M \geq 2$, 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

- **Compare to those of Abelian Group**
  - 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

Floating Point in C

- **C Guarantees Two Levels**

  `float` single precision

  `double` double precision

- **Conversions/Casting**
  - Casting between `int`, `float`, and `double` changes bit representation
  - `Double/float` → `int`
    - Truncates 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

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

```
1.  x  ==  (int)(double) x
2.  x  ==  (int)(float) x
3.  f  ==  (float)(double) f
4.  d  ==  (double)(float) d
5.  f  ==  -(-f);
6.  1.0/2  ==  1/2.0
7.  d×d  >=  0.0
8.  (f+d)-f  ==  d
```

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

1.	`x == (int)(double) x`	**T**
2.	`x == (int)(float) x`	**F**
3.	`f == (float)(double) f`	**T**
4.	`d == (double)(float) d`	**F**
5.	`f == -(-f);`	**T**
6.	`1.0/2 == 1/2.0`	**T**
7.	`d×d >= 0.0`	**T**
8.	`(f+d)-f == d`	**F**