System Program :

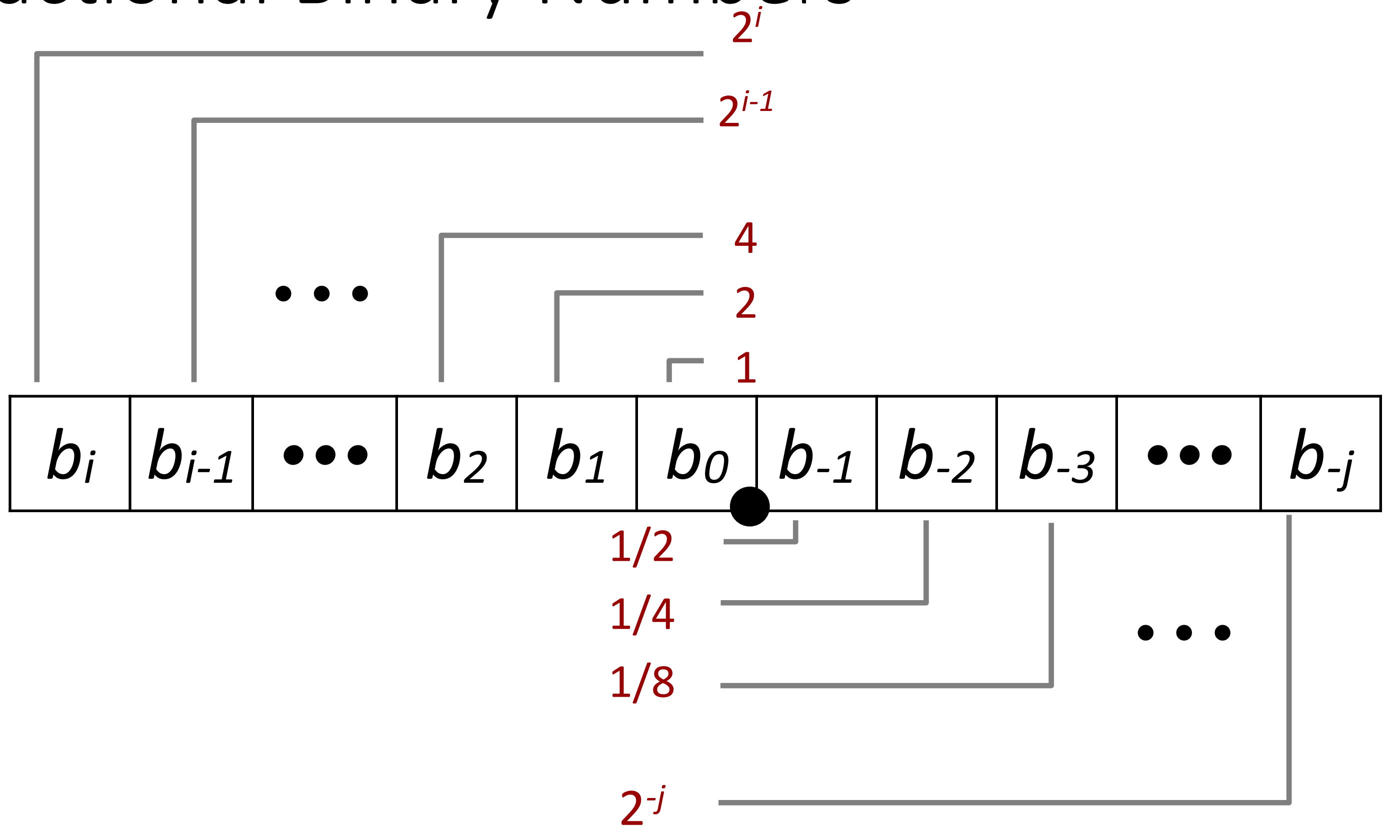
float

Honguk Woo

Fractional binary numbers

- What is 1011.101_2?

Fractional Binary Numbers



Representation

- Bits to right of “binary point” represent fractional powers of 2
- Represents rational number:

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

Fractional Binary Numbers: Examples

- Value Representation

Value	Representation
5 3/4	**101.11_2**
2 7/8	**10.111_2**
1 7/16	**1.0111_2**
63/64	**0.111111_2**

- Observations
  - Divide by 2 by shifting right
  - Multiply by 2 by shifting left
  - Numbers of form $0.111111\ldots_2$ just below 1.0
    - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i + \ldots \rightarrow 1.0$
    - Use notation $1.0 - \varepsilon$

Representable Numbers

- Limitation #1
  - 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]…$_2$
1/5	0.001100110011[0011]…$_2$
1/10	0.0001100110011[0011]…$_2$

파티션 위치에 따라 정밀도가 달라진다.

- Limitation #2
  - Just one setting of binary point within the w bits
    파티션(.)
    - Limited range of numbers (very small values? very large ones?)

IEEE Floating Point

- IEEE Standard 754 (flogting-point arithmetic)
  - Established in 1985 as uniform standard for floating point arithmetic
    - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- Driven by numerical concerns
  - Nice standards for rounding, overflow, underflow
  - Hard to make fast in hardware
    - Numerical analysts predominated over hardware engineers in defining standard.

Floating Point Representation

- Numerical Form:

  $(-1)^s * M * 2^E$

  - **Sign** bit s determines whether number is negative or positive
  - **Significand (fractional value, mantissa)** M normally a fractional value in range [1,2)
  - **Exponent** E weights value by power of two

- Encoding

s	**exp**	**frac**

  - MSB is sign bit s
  - **`exp`** field encodes E (Exponent, but is not equal to E)
  - **`frac`** field encodes M (Mantissa, but is not equal to M)

Precision Options

- Single precision: 32 bits (4byte) float

s	exp	frac
1	8-bits	23-bits

+3

- Double precision: 64 bits (8byte) double.

s	exp	frac
1	11-bits	52-bits

+4

- Extended precision: 80 bits (Intel only)

s	exp	frac
1	15-bits	63 or 64-bits

Three cases of floating-point numbers

- Normalized
- Denormalized
- Special

- These cases depend on the bit pattern of exp

Normalized Values (1)

$v = (-1)^s \, \boldsymbol{M} \, 2^{\boldsymbol{E}}$

s	exp	frac
1	8-bits	23-bits

- Condition: `exp` $\neq$ 000…0 and `exp` $\neq$ 111…1
- Exponent coded as biased value
  - E = *Exp* – *Bias*
  - *Exp*: **unsigned** value denoted by exp
  - *Bias*: $\mathbf{2^{k-1} - 1}$ where k is the number of exponent bits
    - Single precision (8bit): 127 (*Exp* : **1..254**, E: -126..127) = [음, 양]
    - Double precision (11bit): 1023 (*Exp* : **1..2046**, E: -1022..1023)
- Significand (fractional value) coded with implied leading 1
  - $M = 1.xxx…x_2$
    - Minimum when frac = 000…0 (M = 1.0)
    - Maximum when frac = 111…1 (M = 2.0 - ε)
  - Get extra leading bit for free

앞에 '1'이 이미 있다고 가정
=> 하나의 bit를 더 사용가능

$$v = (-1)^s \, \boldsymbol{M} \, 2^{\boldsymbol{E}}$$

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

s	exp	frac
1	8-bits	23-bits

Normalized Encoding Example (1)

$$v = (-1)^s \, M \, 2^E$$
$$E = Exp - Bias$$

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

X = 0xa1 Y = 199 E = 43

- Significand (single precision, 23bit)

$$M = 1.\underline{1101101101101}_2$$
$$\mathtt{frac} = \underline{1101101101101}0000000000_2$$

(뒤를 채운다.)

- Exponent (single precision, 8bit)

$$E = 13$$
$$Bias = 127$$
$$Exp = 140 = 10001100_2$$

E = EXP - Bias
13 = EXP - 127
∴ EXP = 140

- Result:

s	exp	frac
0	10001100	11011011011010000000000

Normalized Encoding Example (2)

$v = (-1)^s M 2^E$
$E = Exp - Bias$

- Value: `float F = 15213.0;`
  - Numerical form $15213_{10} = 11101101101101_2$
    $= \mathbf{1.1101101101101_2 \times 2^{13}}$
- Significand (single precision, 23bit)

  M = $\mathtt{1.}\underline{\mathtt{1101101101101}}_2$

  frac= $\underline{\mathtt{1101101101101}}\mathtt{0000000000}_2$
- Exponent (single precision, 8bit)

  E = 13

  $Bias$ = 127

  Exp = 140 = $\mathbf{10001100_2}$

s	exp	frac
0	10001100	11011011011010000000000

```
Binary:   0100 0110 0110 1101 1011 0100 0000 0000
140:       100 0110 0
15213:              1110 1101 1011 01
```

normalized

$v = (-1)^s M 2^E$
$E = Exp - Bias$

Denormalized Values

denormalized

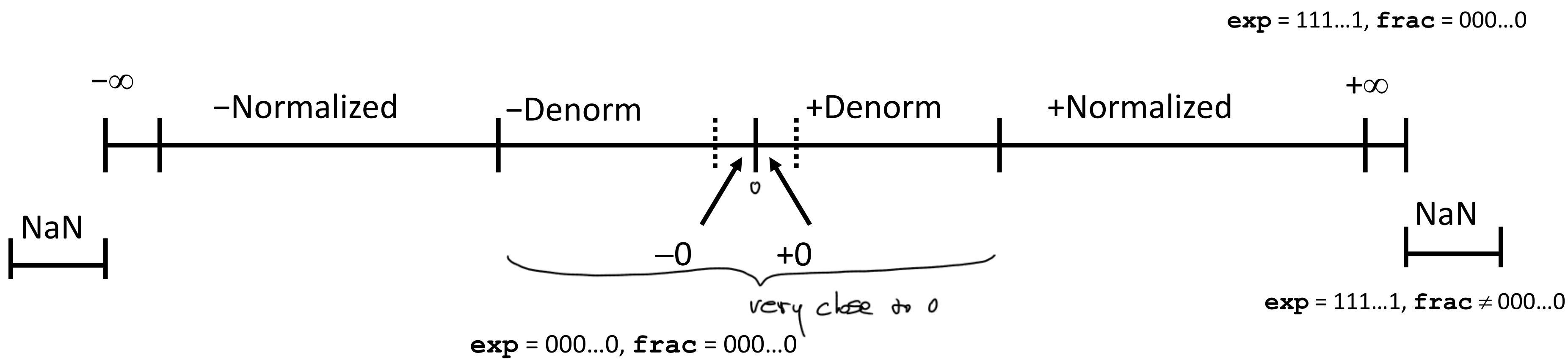
$v = (-1)^s M 2^E$
$E = 1 - Bias$

- Condition: `exp` = 000…0
- Value
  - Exponent value E = 1 – Bias (e.g. single precision: bias 127, so **-126**)
  - Significand value $M = 0.xxx…x_2$ (leading 0, no implied leading 1)
- Cases
  - **`exp`** = 000…0, **`frac`** = 000…0
    - Represents value 0
    - Note that have distinct values +0 and -0 (depending on sign bit)
  - **`exp`** = 000…0, **`frac`** ≠ 000…0
    - **Numbers very close to 0.0**
    - Equispaced : Possible numeric values are spaced evenly near 0.0

Special Values

- Condition: `exp` = 111…1
- Cases
  - **`exp`** = 111…1, **`frac`** = 000…0
    - Represents value ∞ (**infinity**)
    - Infinity can represent results of overflows
    - Both positive and negative
    - e.g. 1.0/0.0 = -1.0/-0.0 = $+\infty$, 1.0/-0.0 = $-\infty$
  - **`exp`** = 111…1, **`frac`** $\neq$ 000…0
    - Not-a-Number (NaN)
    - Represents case when no numeric value can be determined
    - e.g., sqrt(-1), $\infty - \infty$

Visualization: Floating Point Encodings



$0.xxx\ldots x_2 \ * \ 2^{-126}$

$1.xxx\ldots x_2 \ * \ 2^{-126}$ $\quad E \in [-126, 127]$

$1.xxx\ldots x_2 \ * \ 2^{127}$

Quiz: FP

Encoding this real number into a single precison floating-point number ;
What’s the bit pattern in 32bits?

43.25

43.25

normalized

$v = (-1)^s\, M\, 2^E$
$E = Exp - Bias$

$v = 101011.01 = 1.\boxed{0101101} \times 2^{5}$ M E

S=0

E = exp − Bias
5 = exp − 127
∴ exp = 132

M = 1.0101101

E = 5

S | EXP | frag

2	132	
2	66	0
2	33	0
2	16	1
2	8	0
2	4	0
2	2	0
	1	0

43.25

normalized

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

$E = \mathit{Exp} - \mathit{Bias}$

$v = 101011.01 = 1.0101101 \times 2^5$

$M = 1.0101101$

$E = 5$

$Exp = 5+127 = 132 = 10000100$

frac = 0101101 (normalized)

$S = 0$

s	exp	frac
1	8-bits	23-bits

Tiny Floating Point Example

- 8-bit Floating Point Representation
  - the sign bit is in the most significant bit
  - the next 4 bits are the **exp**, with a bias of 7 (**2^{k-1} – 1**)
  - the last 3 bits are the **`frac`**

$2^{4-1} - 1 = 7$

- Same general form as IEEE Format
  - normalized, denormalized
  - representation of 0, NaN, infinity

s	exp	frac
1	4-bits	3-bits

- Denormalized
  - `exp` = 000…0
- Normalized
  - `exp` ≠ 000…0 and `exp` ≠ 111…1
- Special
  - `exp` = 111…1

4 bits exp, bias 7

Dynamic Range (Positive Only)

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

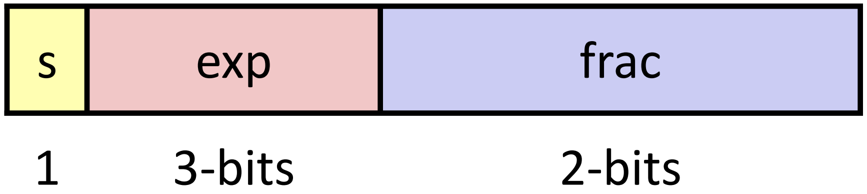
n: E = Exp – Bias

d: E = 1 – Bias

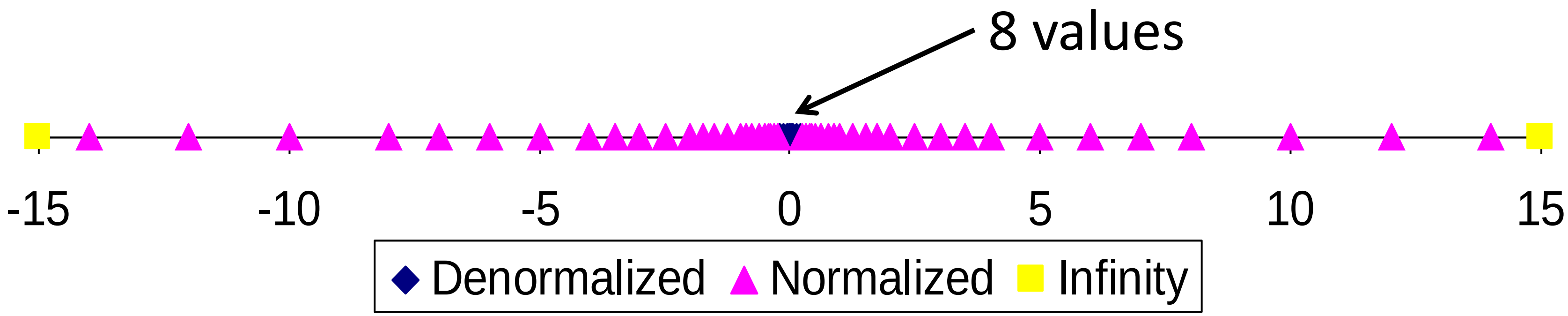
	s	exp	frac	E	Value	
	0	0000	000	-6	0	
	0	0000	001	-6	1/8*1/64 = 1/512	------ closest to zero
Denormalized numbers	0	0000	010	-6	2/8*1/64 = 2/512	
	…					
	0	0000	110	-6	6/8*1/64 = 6/512	
굉장히 소무성하게 넘어간다.	0	0000	111	-6	7/8*1/64 = 7/512	------ largest denorm
	0	0001	000	-6	8/8*1/64 = 8/512	------ smallest norm (Leading 1)
	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
Normalized numbers	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	
굉장히 조우성하게 넘어간다.	0	1110	111	7	15/8*128 = 240	------ largest norm
	0	1111	000	n/a	inf	

Distribution of Values

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

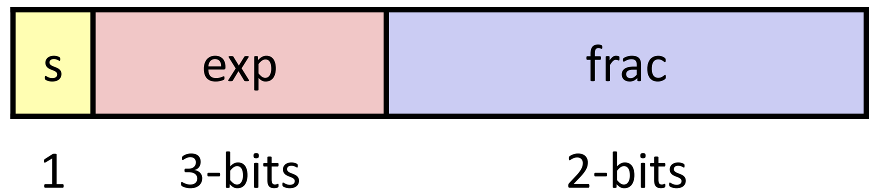


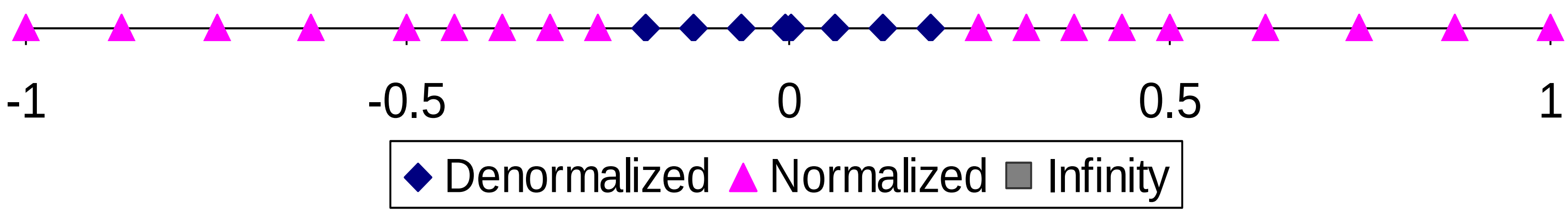
- Notice how the distribution gets denser toward zero.



Distribution of Values (close-up view)

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





Special Properties of IEEE Encoding

floating point

- "FP zero" is the 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

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

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

- Rounding : for a real-number, approximation in floating-point format
- Basic idea
  - First compute exact result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into **`frac`**

Huristics : 발견법.

Rounding

- Rounding Modes (illustrate with $ rounding)

	$1.40	$1.60	$1.50	$2.50	–$1.50
Toward-zero	$1	$1	$1	$2	–$1
Round-down (−∞)	$1	$1	$1	$2	–$2
Round-up (+∞)	$2	$2	$2	$3	–$1
Round-to-even (Round-to-nearest, default)	$1	$2	$2	$2	–$2

ㄴ짝수

중간은 even 쪽으로 근사.

Closer Look at Round-To-Even

- Default Rounding Mode
  - All others are statistically biased
    - Sum of a set of positive numbers will consistently be over- or under- estimated
- Round-to-even: 50% round-up, 50% round-down ⇒ 균일하게 정리됨. (확률 반반)

- 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

7.8949999	7.89	(Less than half way)
7.8950001	7.90	(Greater than half way)
7.8950000	7.90	(Half way—round up) : round to even
7.8850000	7.88	(Half way—round down) : round to even

Rounding Binary Numbers

- Binary Fractional Numbers
  - “Even” when least significant bit is `0`
  - “Half way” when bits to right of rounding position = `100`…$_2$

- Examples
  - Round to nearest 1/4 (2 bits right of binary point)

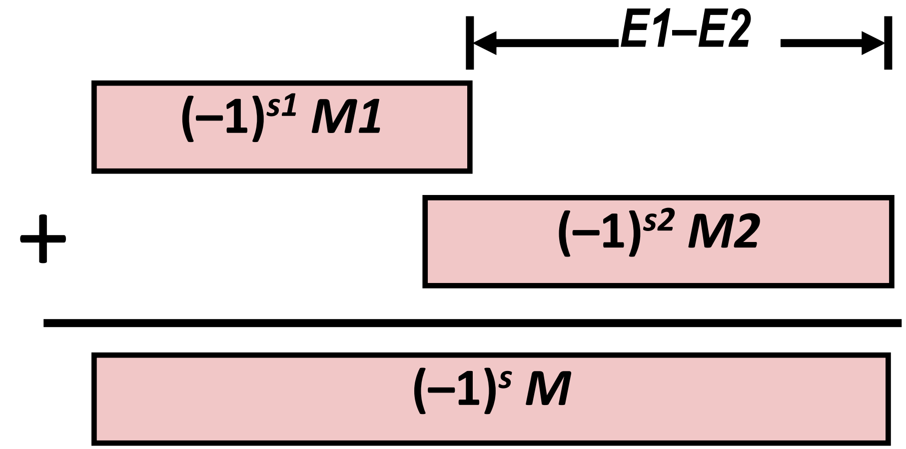
Value	Binary	Rounded	Action	Rounded Value
2 3/32	10.00011_2	10.00_2	(<1/2—down)	2
2 3/16	10.00110_2	10.01_2	(>1/2—up)	2 1/4
2 7/8	10.11100_2	11.00_2	(1/2—up)	3
2 5/8	10.10100_2	10.10_2	(1/2—down)	2 1/2

개념만 알아두자

Floating Point Addition

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

Align binary points:
Shift-right M2 by (E1-E2)



- Exact Result: $(-1)^{s}\ \boldsymbol{M}\ \ 2^{E}$
  - 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