CSC 252/452: Computer Organization
Fall 2025: Lecture 4

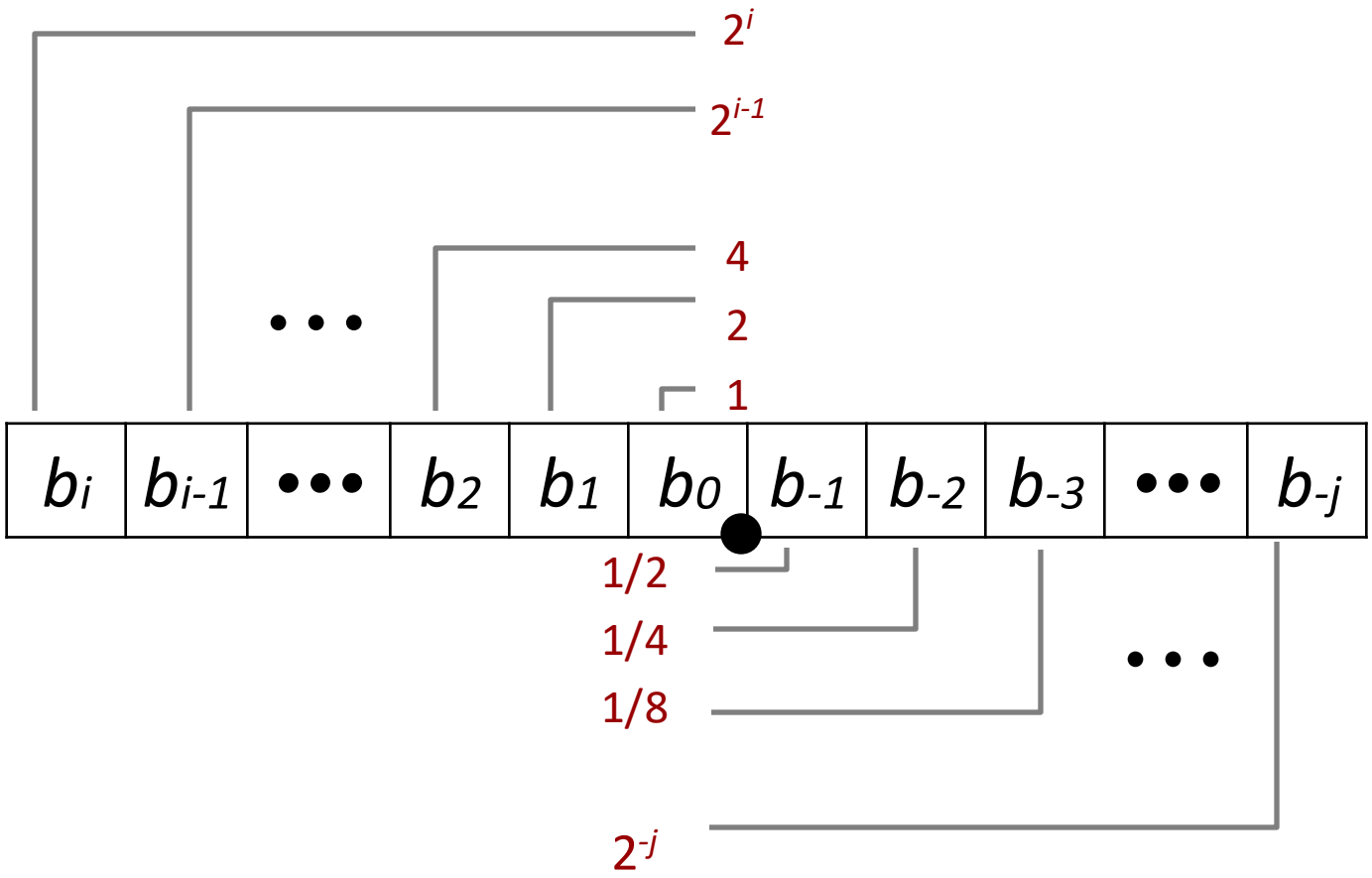
Instructor: Yanan Guo

Department of Computer Science
University of Rochester

Announcement

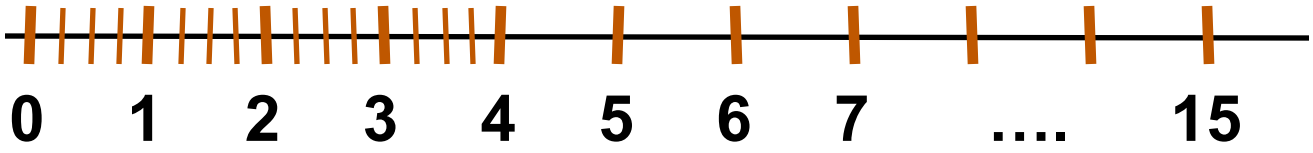
- Programming Assignment 1 is out
  - Details: https://www.cs.rochester.edu/courses/252/fall2025/labs/assignment1.html
  - Due on Sep. 15th, 11:59 PM
  - You have 3 slip days

Fractional Binary Numbers



Fixed-Point Representation

- Binary point stays fixed
- Fixed interval between representable numbers
  - The interval in this example is 0.25_{10}

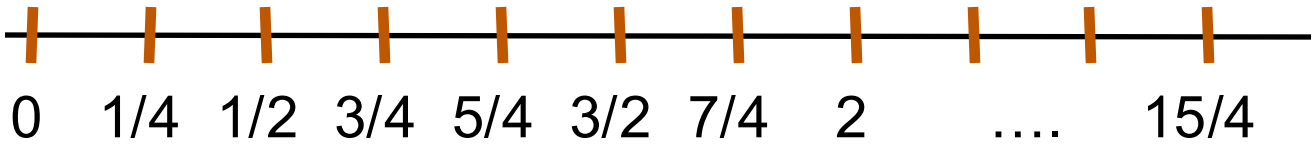


Decimal	Binary
0	00.00
0.25	00.01
0.5	00.10
0.75	00.11
1	01.00
1.25	01.01
1.5	01.10
1.75	01.11
2	10.00
2.25	10.01
2.5	10.10
2.75	10.11
3	11.00
3.25	11.01
3.5	11.10
3.75	11.11

Limitations of Fixed-Point (*#1*)

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

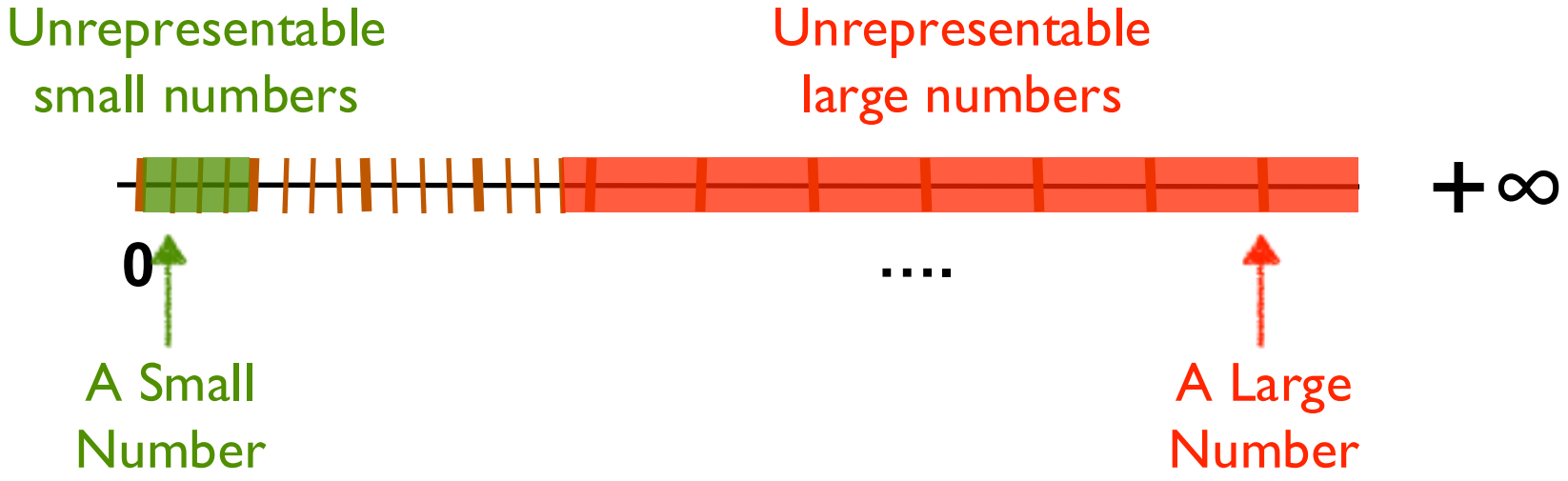
Decimal Value	Binary Representation
1/3	0.0101010101[01]…
1/5	0.001100110011[0011]…
1/10	0.0001100110011[0011]…



$b_3 b_2 . b_1 b_0$

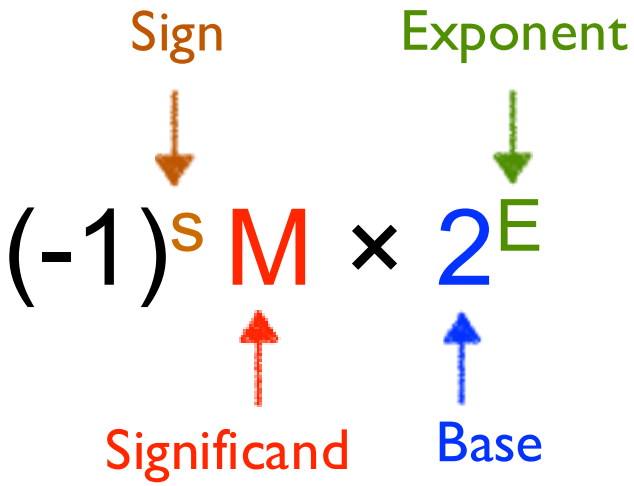
Limitations of Fixed-Point (#2)

- Can't represent very small and very large numbers at the same time
  - To represent very large numbers, the (fixed) interval needs to be large, making it hard to represent small numbers
  - To represent very small numbers, the (fixed) interval needs to be small, making it hard to represent large numbers



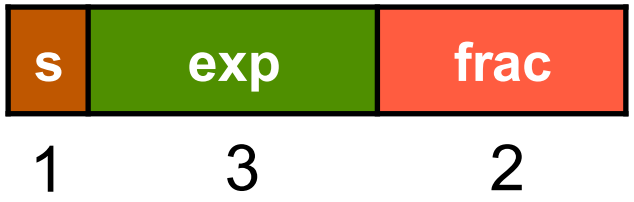
Primer: Floating Point Representation

- In binary: $(-1)^s\ M\ 2^E$
- Normalized form:
  - $1 <= M < 2$
  - $M = 1.b_0b_1b_2b_3\ldots$
    Fraction
- Encoding
  - MSB ***s*** is sign bit s
  - ***exp*** field encodes Exponent (but not exactly the same, more later)
  - ***frac*** field encodes Fraction (but not exactly the same, more later)



6-bit Floating Point Example

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



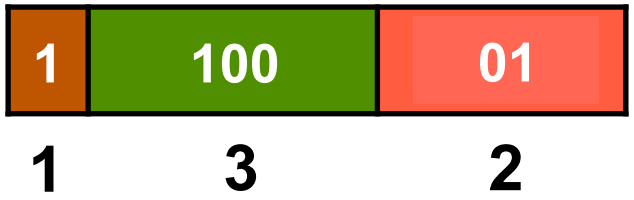
- *exp* has 3 bits, interpreted as an unsigned value
  - If *exp* were *E*, we could represent exponents from **0 to 7**
  - How about negative exponent?
  - Subtract a bias term: *E = exp - bias* (i.e., exp = E + bias)
  - bias is always $2^{k-1} - 1$, where k is number of exponent bits

E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

- Example when we use 3 bits for exp (i.e., *k* = 3):
  - bias = 3
  - If *E* = -2, ***exp*** is 1 (001_2)
  - Reserve 000 and 111 for other purposes (more on this later)
  - We can now represent exponents from **-2 (exp 001) to 3 (exp 110)**

6-bit Floating Point Example

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



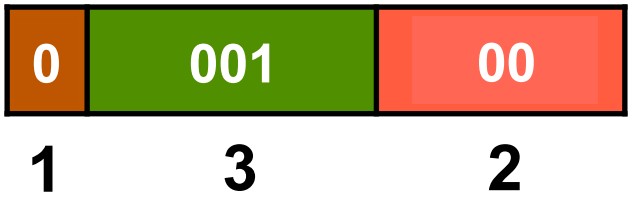
- *frac* has 2 bits, append them after "1." to form M
  - ***frac*** = 10 implies M = 1.10
- Putting it Together: An Example:

$$-10.1_2 = (-1)^1\ 1.01 \times 2^1$$

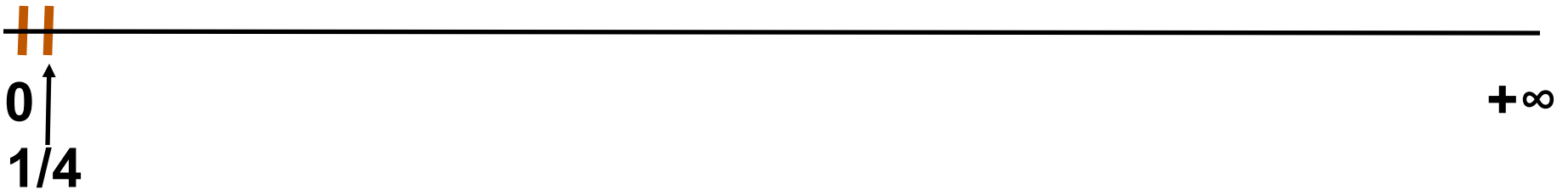
E	exp
~~-3~~	~~000~~
-2	001
-1	010
0	011
1	100
2	101
3	110
~~4~~	~~111~~

Representable Numbers (Positive Only)

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

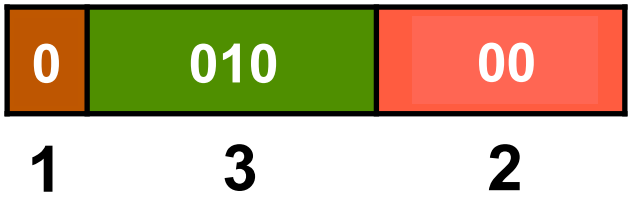


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

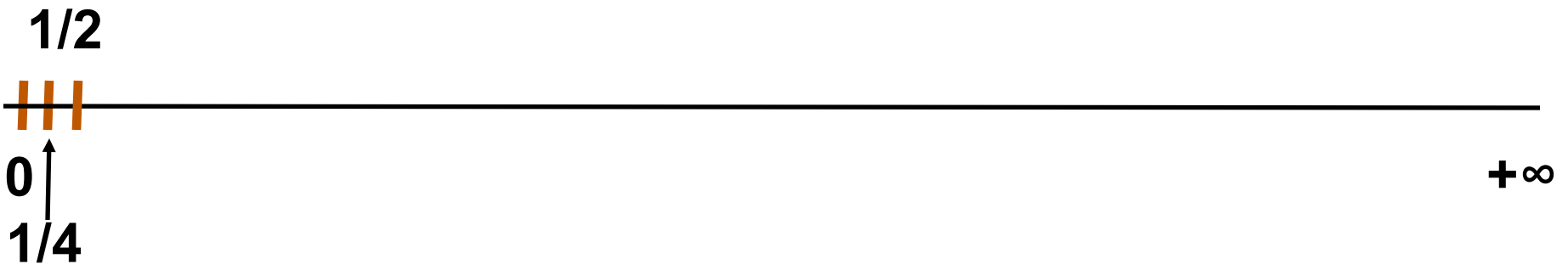


Representable Numbers (Positive Only)

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

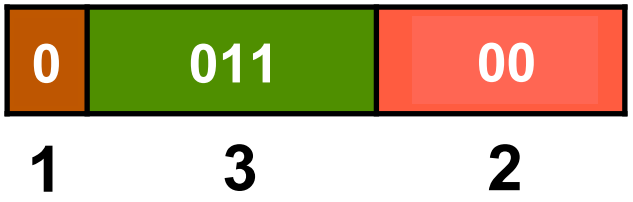


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

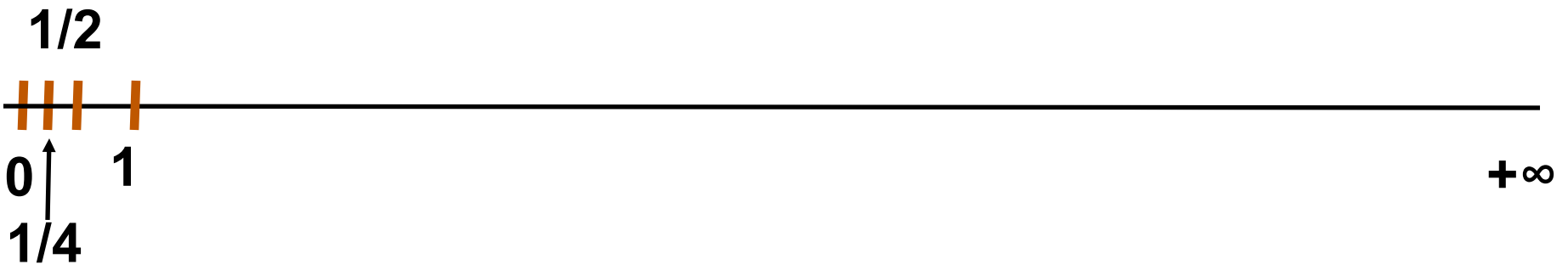


Representable Numbers (Positive Only)

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

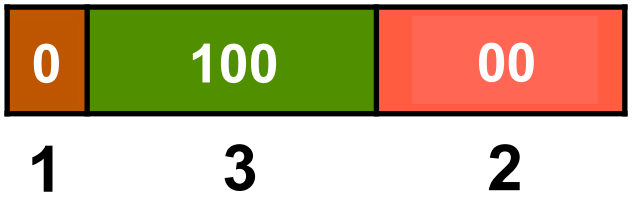


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

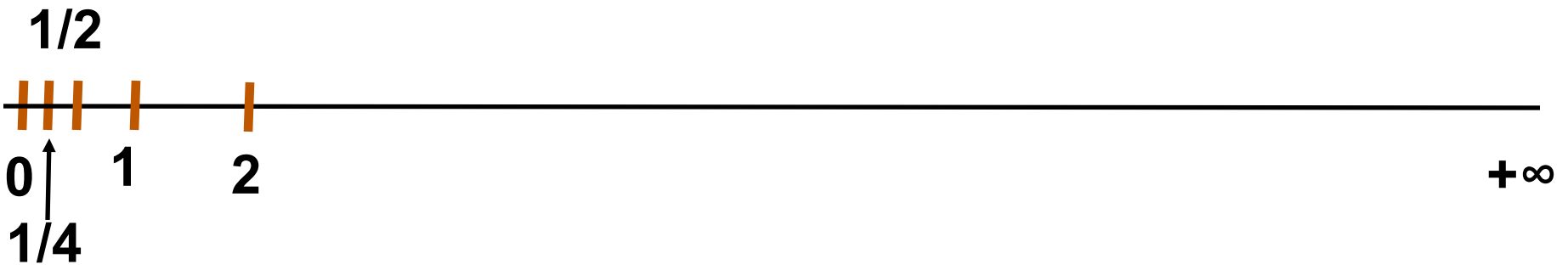


Representable Numbers (Positive Only)

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

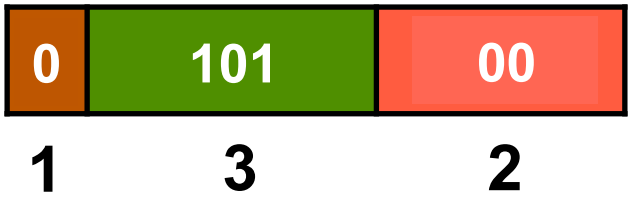


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

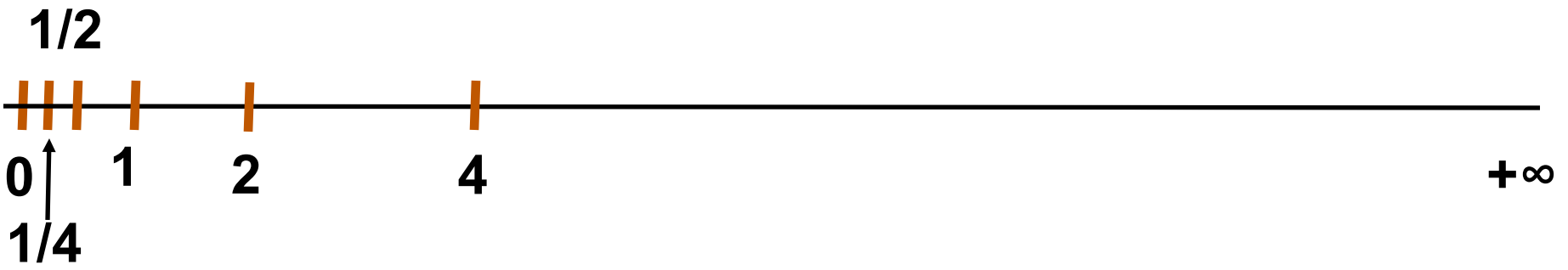


Representable Numbers (Positive Only)

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

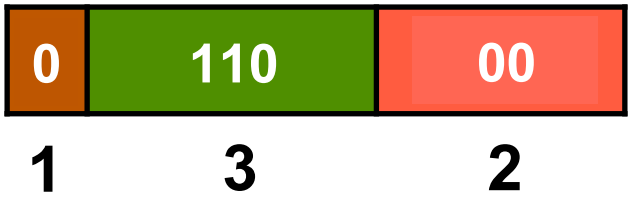


E	exp	E	exp
-3	000	1	100
-2	001	2	101
-1	010	3	110
0	011	4	111

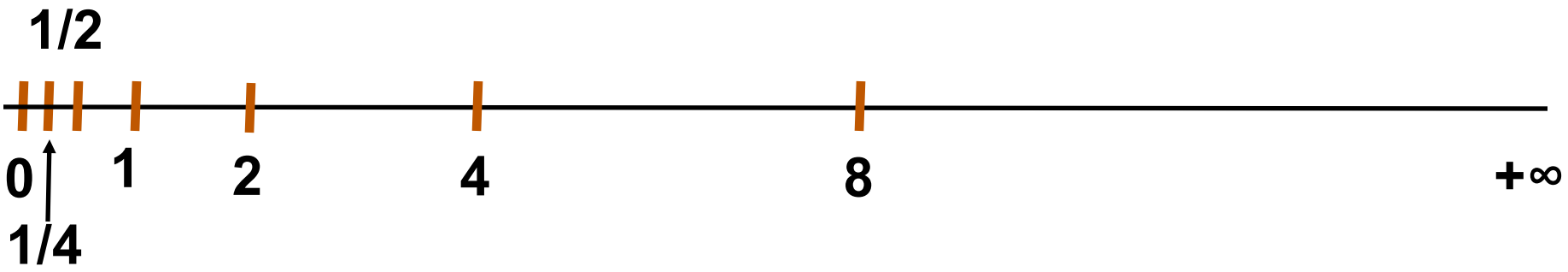


Representable Numbers (Positive Only)

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

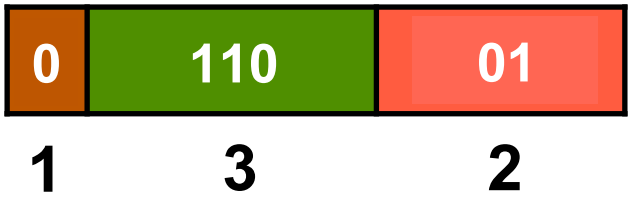


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

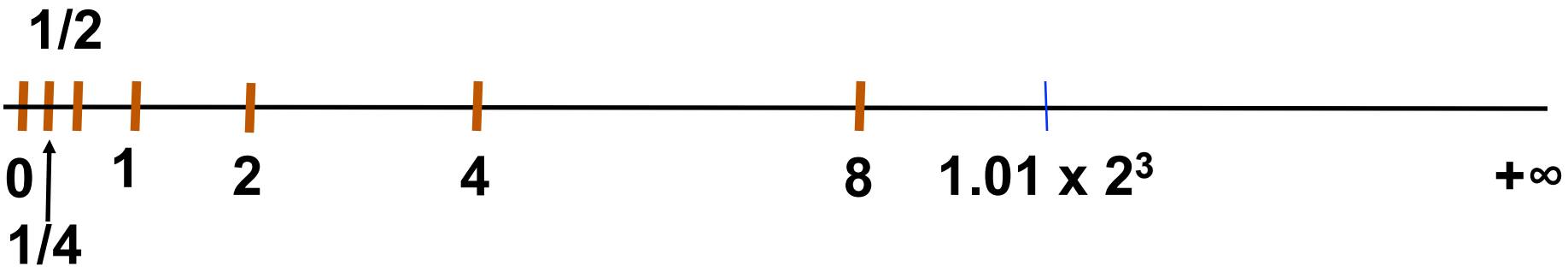


Representable Numbers (Positive Only)

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

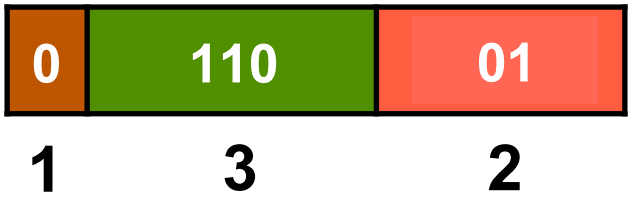


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

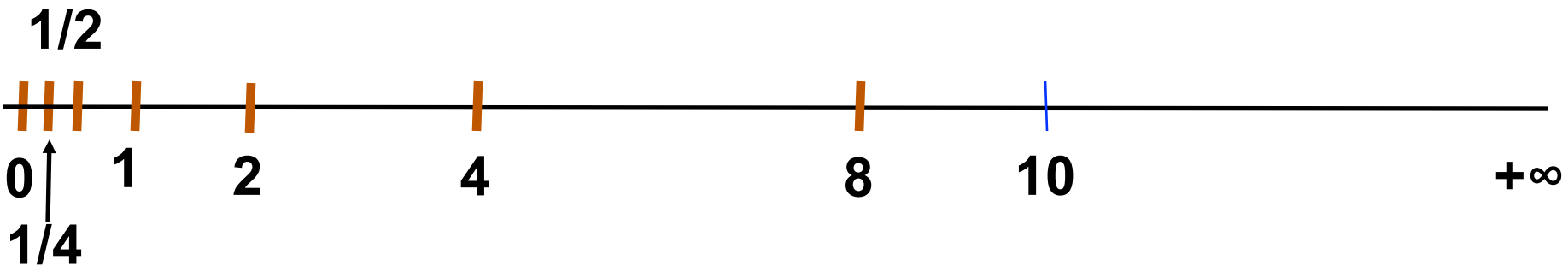


Representable Numbers (Positive Only)

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

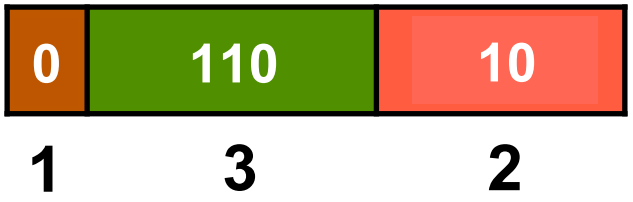


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

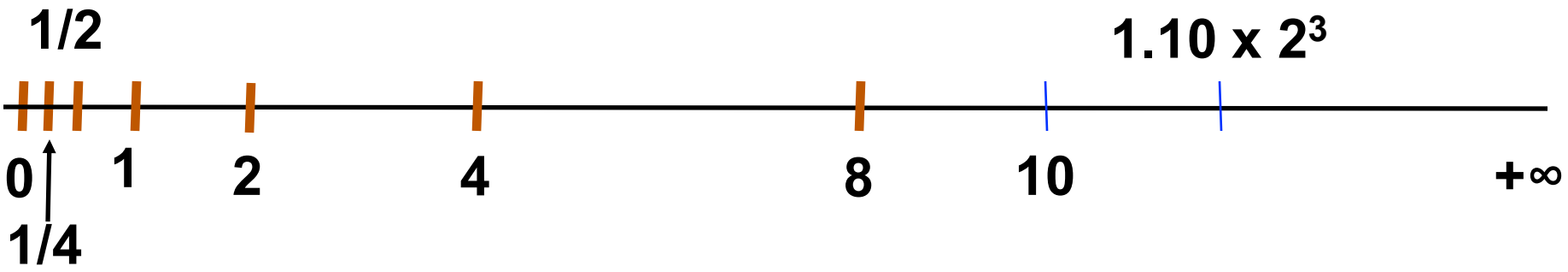


Representable Numbers (Positive Only)

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

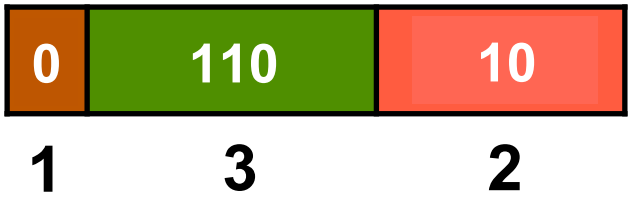


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

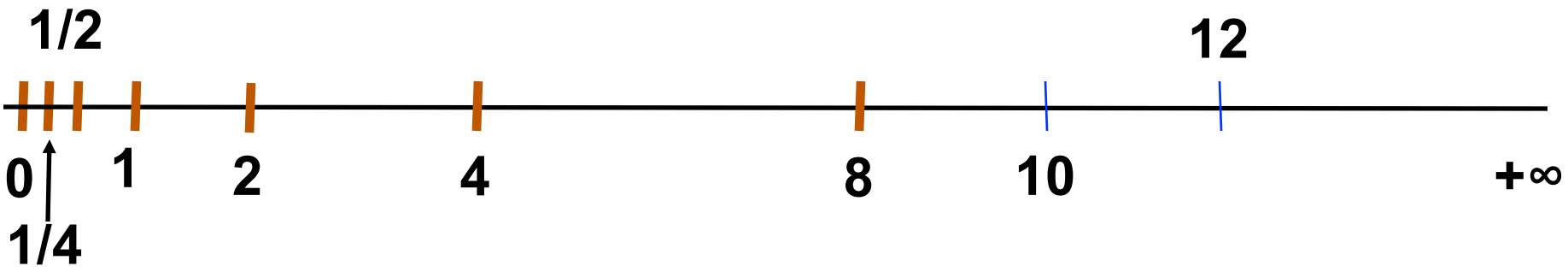


Representable Numbers (Positive Only)

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

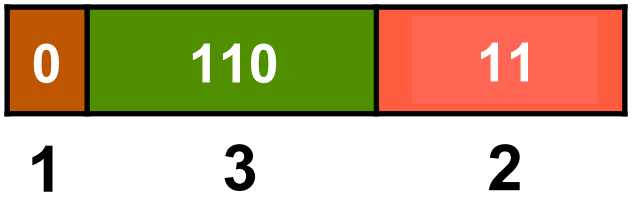


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

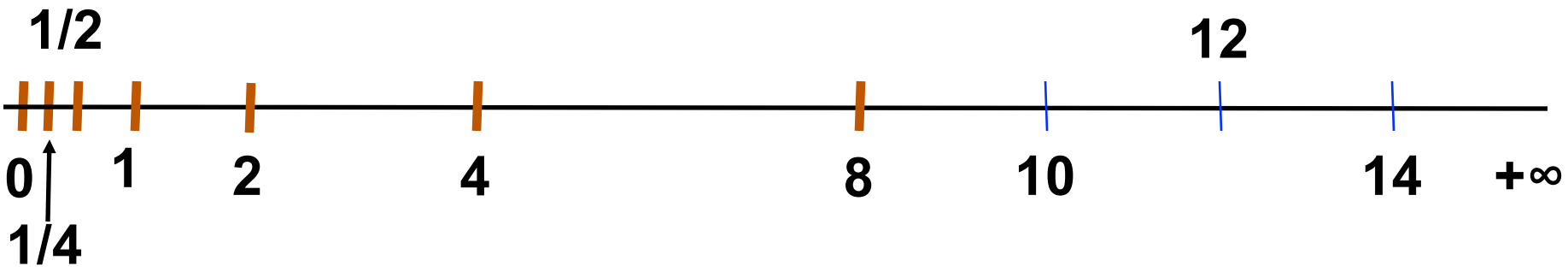


Representable Numbers (Positive Only)

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

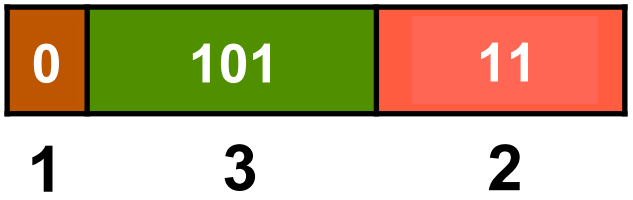


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

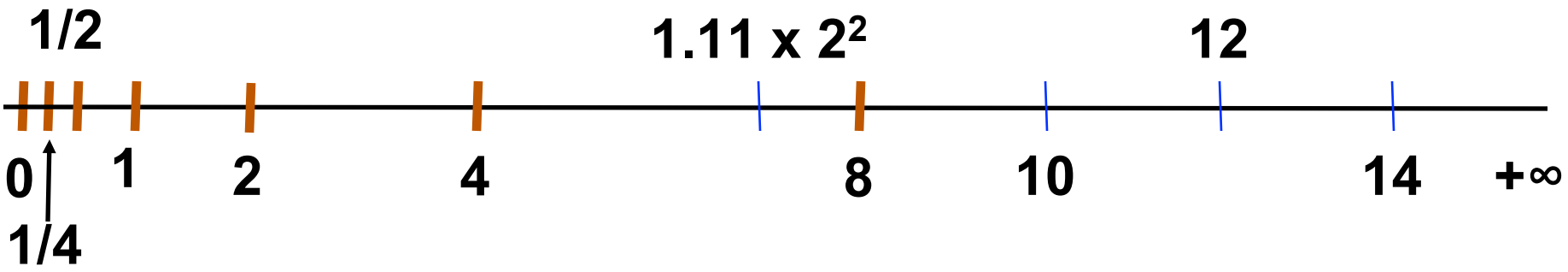


Representable Numbers (Positive Only)

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

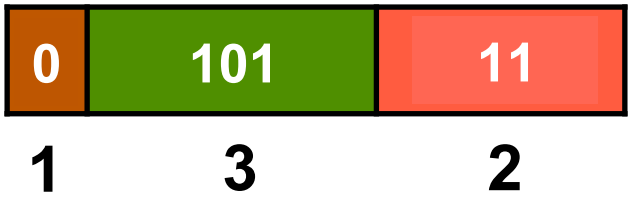


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

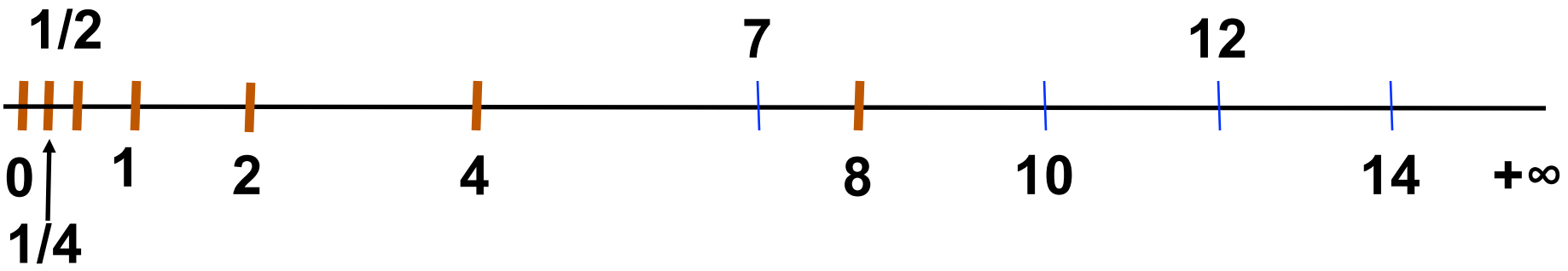


Representable Numbers (Positive Only)

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

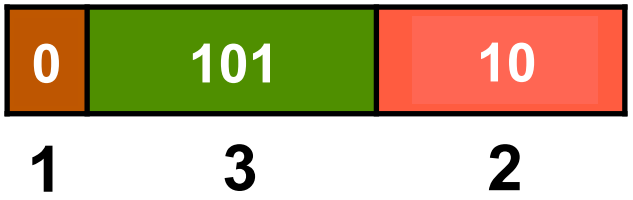


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

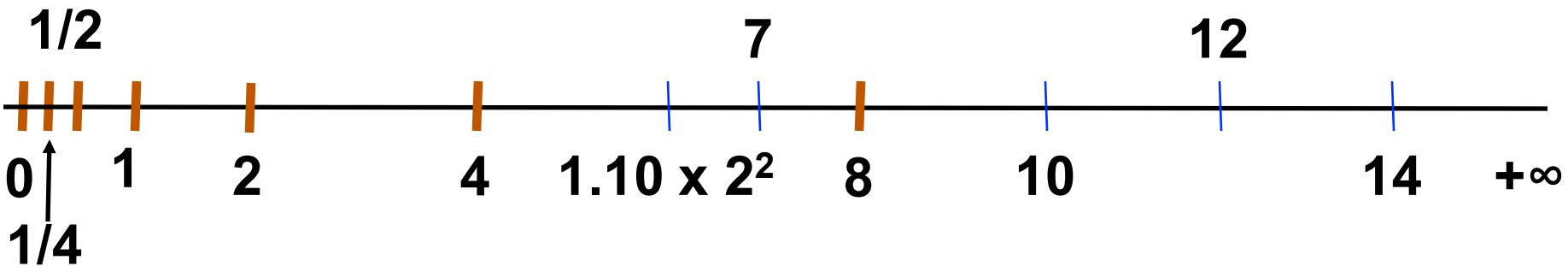


Representable Numbers (Positive Only)

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

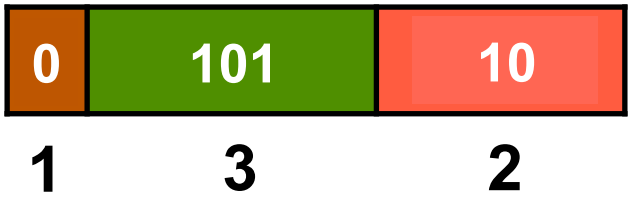


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

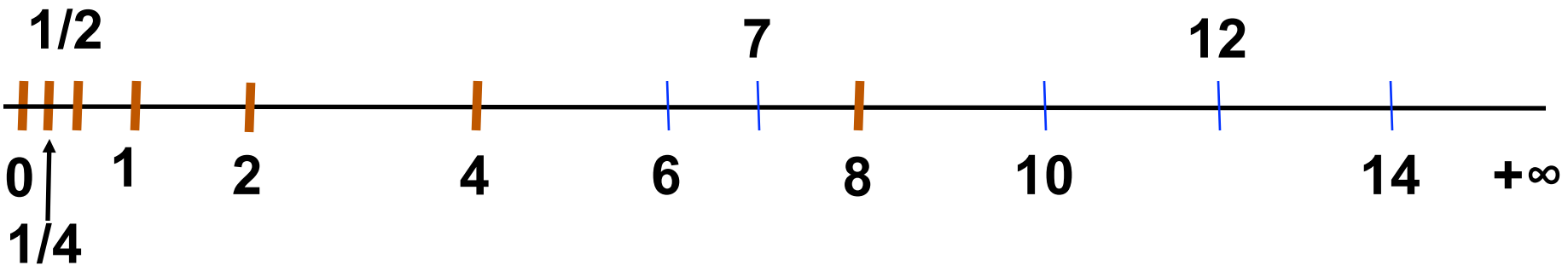


Representable Numbers (Positive Only)

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

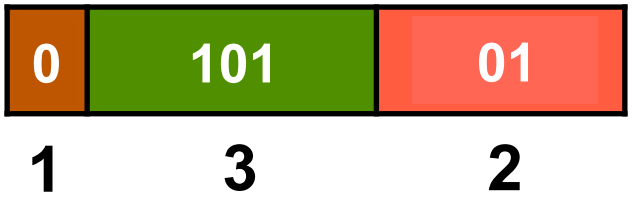


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

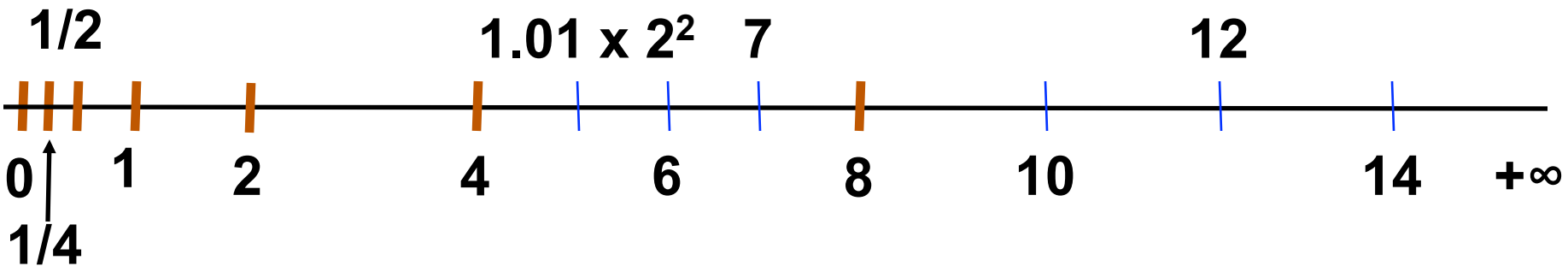


Representable Numbers (Positive Only)

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

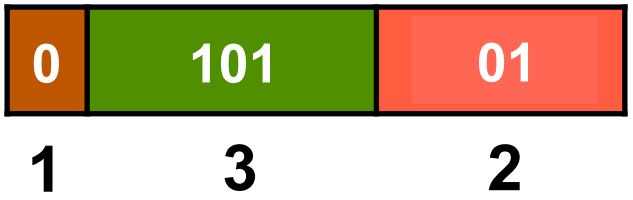


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

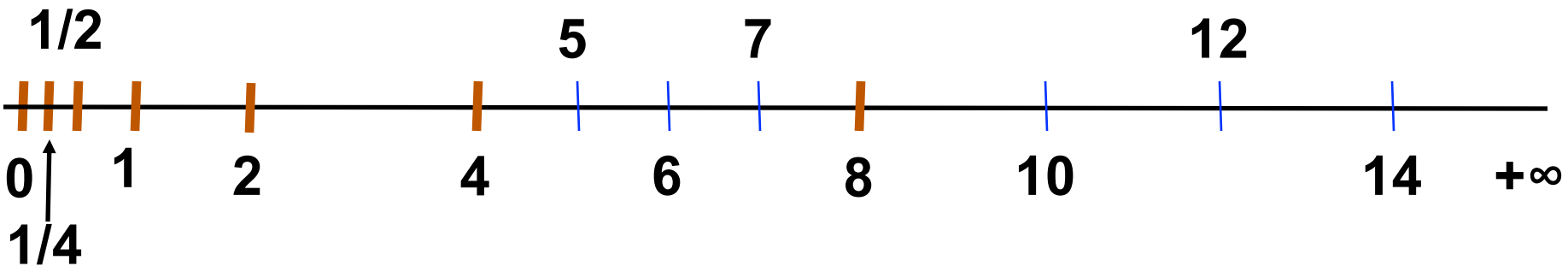


Representable Numbers (Positive Only)

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

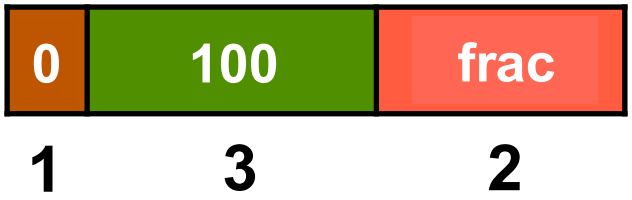


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

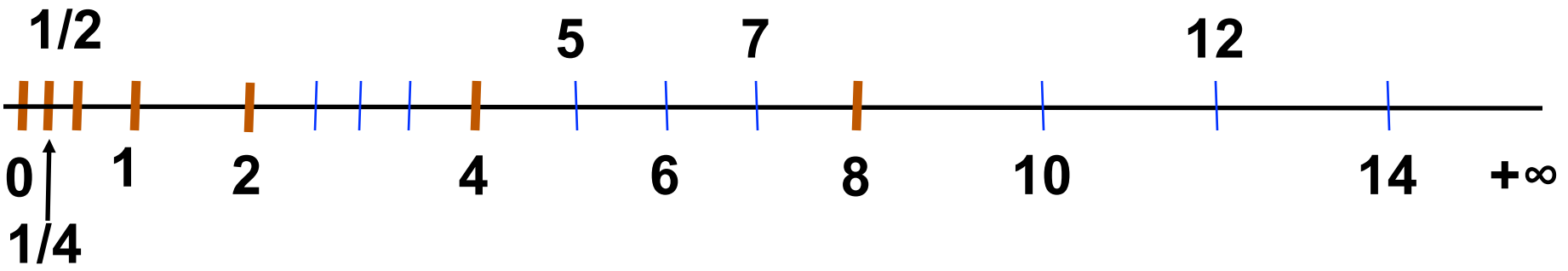


Representable Numbers (Positive Only)

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

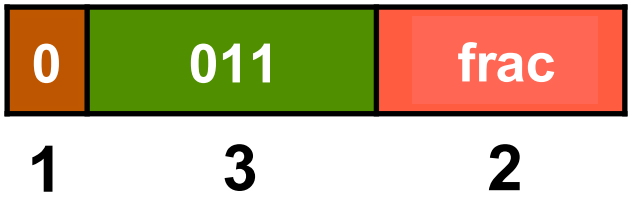


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

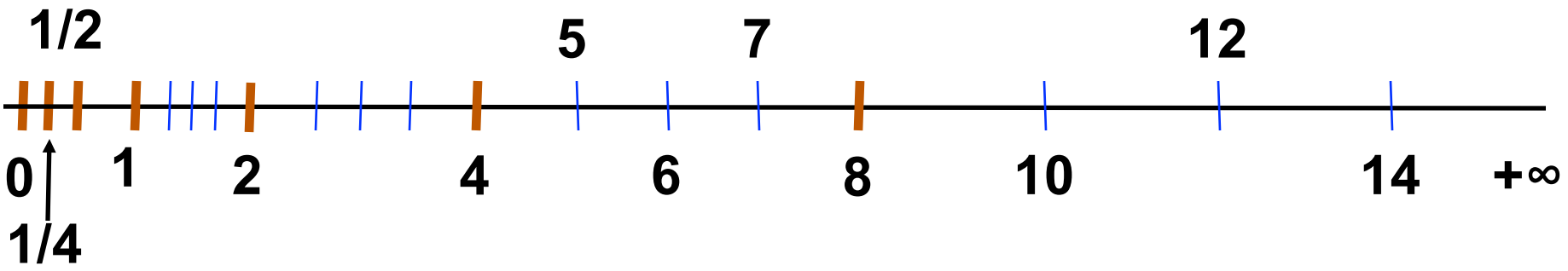


Representable Numbers (Positive Only)

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

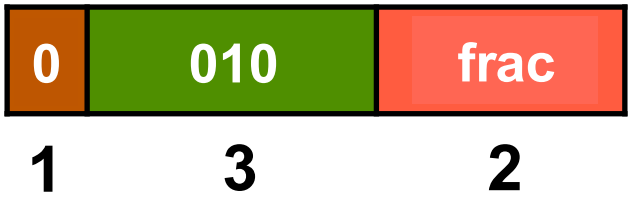


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

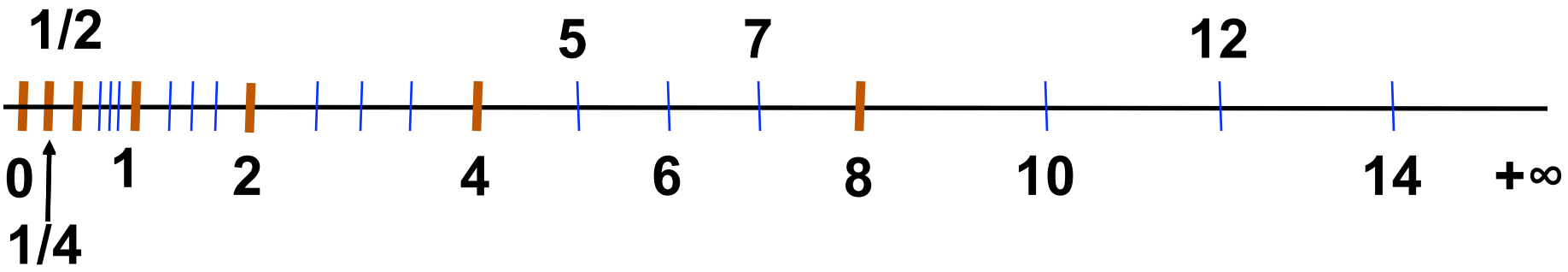


Representable Numbers (Positive Only)

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

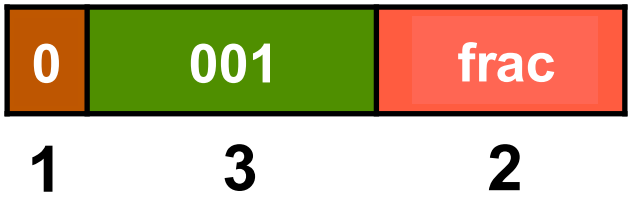


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

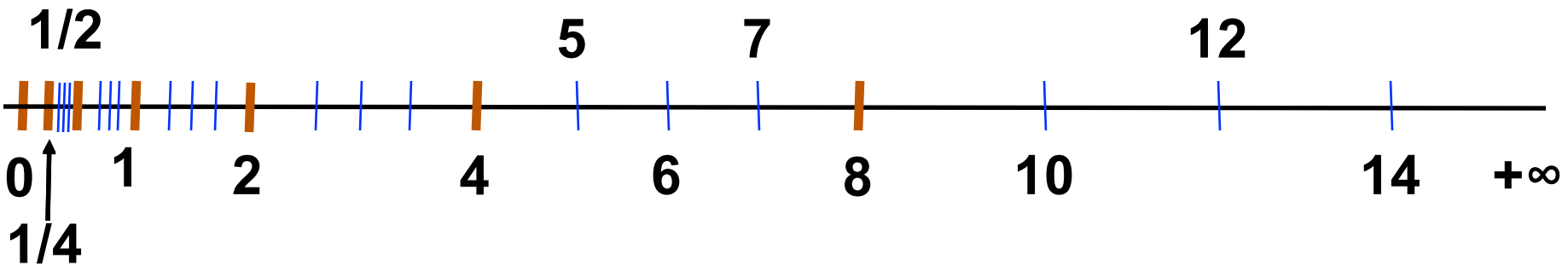


Representable Numbers (Positive Only)

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

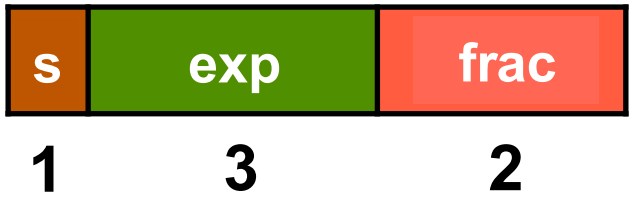


E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~



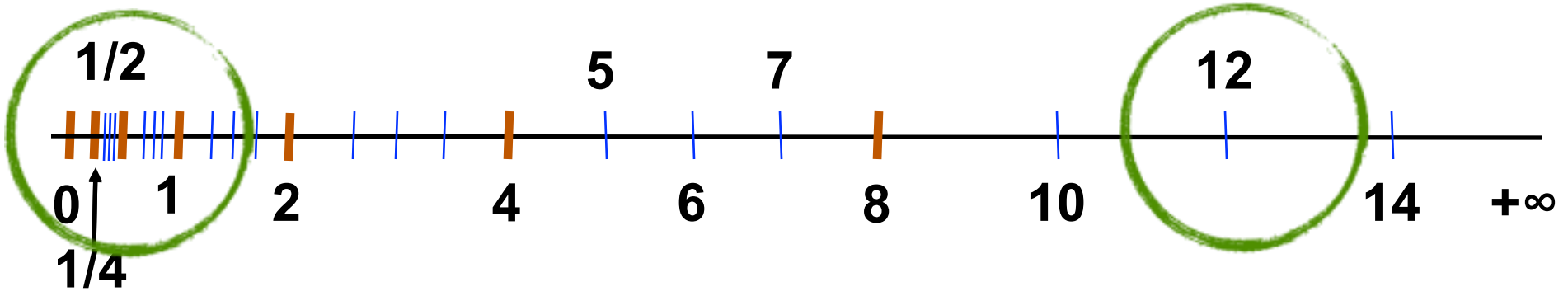
Representable Numbers (Positive Only)

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



E	exp	E	exp
~~-3~~	~~000~~	1	100
-2	001	2	101
-1	010	3	110
0	011	~~4~~	~~111~~

- **Uneven interval (c.f., fixed interval in fixed-point)**
  - More dense toward 0, sparser toward infinite
  - Allow encoding small and large numbers at the same time

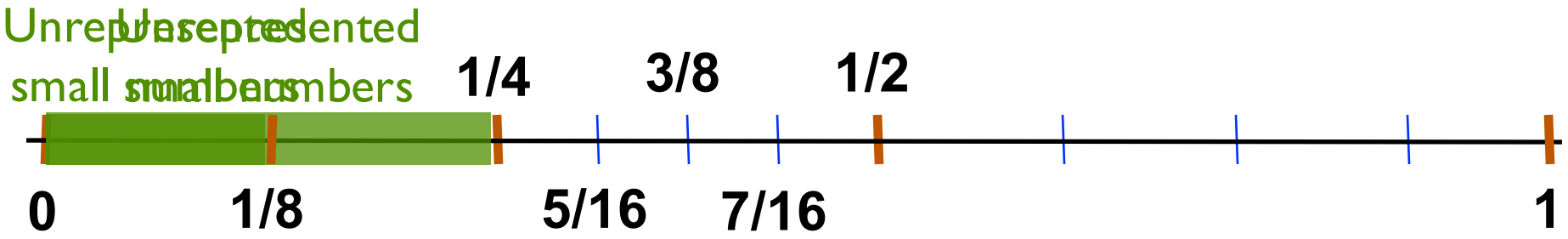


Representable Numbers (Positive Only)

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

E	exp	E	exp
-3	000	1	100
-2	001	2	101
-1	010	3	110
0	011	4	111

- Always round to 0 is inelegant
- Using 000 for *exp* would only "delay" the problem rather than solving it

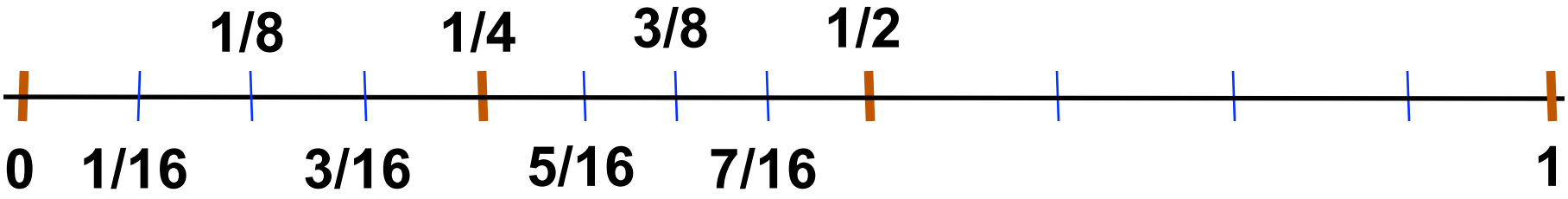


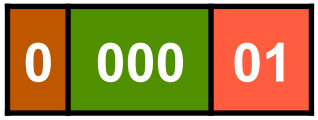
Subnormal (De-normalized) Numbers

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

E	exp	E	exp
-3	000	1	100
-2	001	2	101
-1	010	3	110
0	011	4	111

- Idea: Evenly divide between 0 and 1/4 rather than exponentially decreasing **when *exp* = 0** (subnormal/denormalized numbers)
- *E* = (*exp* + 1) – bias (instead of *exp* - *bias*)
- *M* = 0.*frac* (instead of 1.frac)
- Subnormal numbers allow graceful underflow





$= (-1)^0\ 0.01 \times 2^{(0+1-3)} = 1/16$

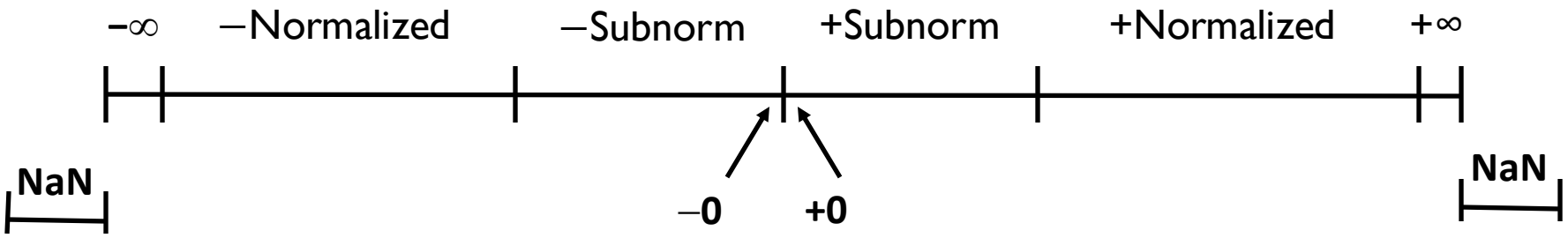
Special Values

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

E	exp	E	exp
-2	000	1	100
-2	001	2	101
-1	010	3	110
0	011		111

- There are many special values in scientific computing
  - +/- ∞, Not-a-Numbers (NaNs) (e.g., 0 / 0, 0 / ∞, ∞ / ∞, sqrt(–1), ∞ - ∞, ∞ x 0, etc.)
- exp = 111 is reserved to represent these numbers
- exp = 111, frac = 00
  - +/- ∞ (depending on the *s* bit). Overflow results.
  - Arithmetic on ∞ is exact: 1.0/0.0 = –1.0/–0.0 = + ∞, 1.0/–0.0 = -∞
- exp = 111, frac != 00
  - Represent NaNs

Visualization: Floating Point Encodings



Infinite Amount of Real Numbers

Sparse Dense Sparse

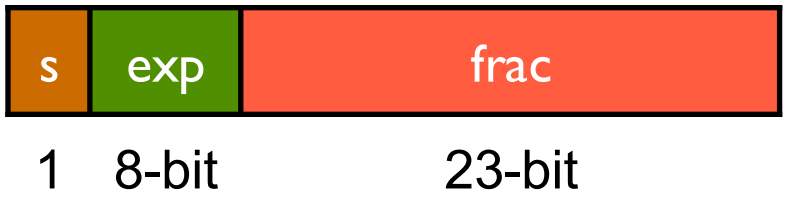
Finite Amount of Floating Point Numbers

Today: Floating Point

- Background: Fractional binary numbers and fixed-point
- Floating point representation
- IEEE 754 standard
- Rounding, addition, multiplication
- Floating point in C
- Summary

IEEE 754 Floating Point Standard

- Single precision: 32 bits



- Double precision: 64 bits

IEEE Floating Point

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

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

Single Precision (32-bit) Example

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

bias = $2^{(8-1)}-1 = 127$

1 8-bit 23-bit

$15213_{10} = 11101101101101_2$

$= (-1)^0\ 1.11011011011012 \times 2^{13}$

exp = E + bias = 140_{10}

Today: Floating Point

- Background: Fractional binary numbers and fixed-point
- Floating point representation
- IEEE 754 standard
- Rounding, addition, multiplication
- Floating point in C
- Summary

Floating Point Computations

- The problem: Computing on floating point numbers might produce a result that can't be precisely represented
- Basic idea
  - We perform the operation & produce the infinitely precise result
  - Make it fit into desired precision
    - Possibly overflow if exponent too large
    - Possibly round to fit into frac

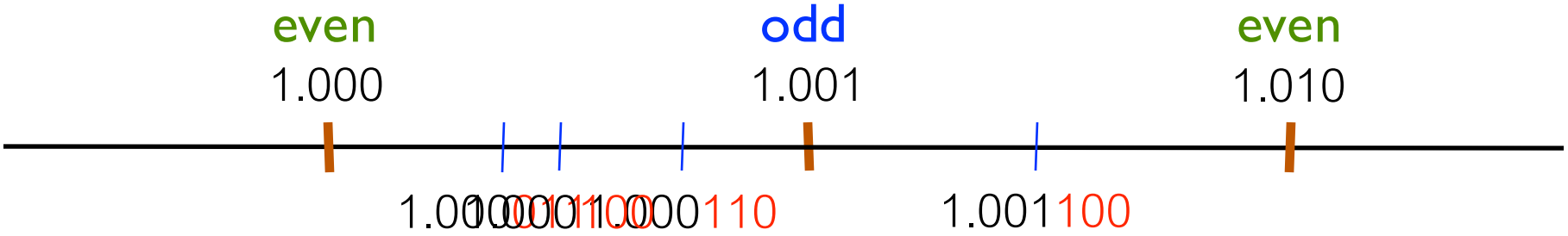
Rounding Modes (Decimal)

- Common ones:
  - Towards zero (chop)
  - Round down (-∞)
  - Round up (+∞)
- Nearest Even: Round to nearest; if equally near, then to the one having an even least significant digit (bit)

Rounding Mode	**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

Rounding Modes (Binary Example)

- Nearest Even; if equally near, then to the one having an even least significant digit (bit)
- Assuming 3 bits for *frac*



Precise Value	Rounded Value	Notes
1.000011	1.000	1.000 is the nearest (down)
1.000110	1.001	1.001 is the nearest (up)
1.000100	1.000	1.000 is the nearest even (down)
1.001100	1.010	1.010 is the nearest even (up)

Floating Point Addition

- $(-1)^{s1}\ M1\ \ 2^{E1}\ \ +\ \ (-1)^{s2}\ M2\ \ 2^{E2}$
- Exact Result: $(-1)^{s}\ M\ \ 2^{E}$
  - Sign *s*, significand *M*:
    - Result of signed align & add
  - Exponent E: E1
    - Assume E1 > E2
- 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

$1.000 \times 2^{-1} + 1.101 \times 2^{-2}$

align $1.000 \times 2^{-1} + 0.1101 \times 2^{-1}$

add 1.1101×2^{-1}

1.110×2^{-1}

Mathematical Properties of FP Add

- Commutative? a+b =b+a — ***Yes***
- Associative? a+b+c = a+(b+c) — ***No***
  - Overflow and inexactness of rounding
  - (3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14

$$1.000 \times 2^5 + 1.101 \times 2^{-3}$$

$$1.000 \times 2^5 + 0.0000000\,1101 \times 2^5$$

$$1.\ 0000000\,1101 \times 2^5$$

$$1.000 \times 2^5$$

Mathematical Properties of FP Add

- Commutative? a+b =b+a — ***Yes***
- Associative? a+b+c = a+(b+c) — ***No***
  - Overflow and inexactness of rounding
  - (3.14+1e10)-1e10 = 0, 3.14+(1e10-1e10) = 3.14
- 0 is additive identity? — ***Yes***
- Every element has additive inverse (negation)? — ***Almost***
  - Except for infinities & NaNs
- Monotonicity: a ≥ b ⇒ a+c ≥ b+c? — ***Almost***
  - Except for infinities & NaNs

Floating Point Multiplication

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

- Multiplication Commutative? ***Yes***
- Multiplication is Associative? ***No***
  - Possibility of overflow, inexactness of rounding
  - Ex: `(1e20*1e20)*1e-20= inf, 1e20*(1e20*1e-20)= 1e20`
- 1 is multiplicative identity? ***Yes***
- Multiplication distributes over addition? ***No***
  - a*(b+c) = a*b+a*c?
  - Possibility of overflow, inexactness of rounding
  - `1e20*(1e20-1e20)= 0.0,  1e20*1e20 - 1e20*1e20 = NaN`
- Monotonicity: $a \geq b \ \& \ c \geq 0 \Rightarrow a * c \geq b * c$? ***Almost***
  - Except for infinities & NaNs

Floating Point in C

Fixed point (implicit binary point)

SP floating point

DP floating point

C Data Type	Bits	Max Value	Max Value (Decimal)
`char`	8	$2^7 - 1$	127
`short`	16	$2^{15} - 1$	32767
`int`	32	$2^{31} - 1$	2147483647
`long`	64	$2^{63} - 1$	$\sim 9.2 \times 10^{18}$
`float`	32	$(2 - 2^{-23}) \times 2^{127}$	$\sim 3.4 \times 10^{38}$
`double`	64	$(2 - 2^{-52}) \times 2^{1023}$	$\sim 1.8 \times 10^{308}$

- To represent 2^{31} in fixed-point, you need at least 32 bits
  - Because fixed-point is a weighted positional representation
- In floating-point, we directly encode the exponent
  - Floating point is based on scientific notation
  - Encoding 31 only needs 6 bits in the *exp* field

Floating Point in C

- **`double/float` → `int`**
  - Truncates fractional part
  - Like rounding toward zero
  - Not defined when out of range or NaN
- **`int` → `float`**
  - Can't guarantee exact casting. Will round according to rounding mode

- **`int` → `double`**

Floating Point Review

$$v = (-1)^s \times 1.frac \times 2^E$$

- Denormalized (exp == 000)
  - E = (exp + 1) – bias
  - M = 0.frac
- Normalized (exp != 000)
  - E = exp – bias
  - M = 1.frac

	s	*exp*	*frac*	Value	Value
Denormalized	0	000	00	0.00 x 2^{-2}	0
	0	000	11	0.11 x 2^{-2}	3/16
Normalized	0	001	00	1.00 x 2^{-2}	1/4
	0	001	11	1.11 x 2^{-2}	7/16
	0	010	00	1.00 x 2^{-1}	1/2
	0	010	11	1.11 x 2^{-1}	7/8
	0	100	00	1.00 x 2^{0}	1
	0	100	11	1.11 x 2^{0}	1 3/4
	0	101	00	1.00 x 2^{1}	2
	0	101	11	1.11 x 2^{1}	3 1/2
	0	110	00	1.00 x 2^{2}	4
	0	110	11	1.11 x 2^{2}	7
Special Value	0	111	00	infinite	infinite
	0	111	11	NaN	NaN

Floating Point Review

- Bit patterns representing non-negative numbers are ordered the same way as integers, so could use regular integer comparison.
- You don't get this property if:
  - *exp* is interpreted as signed
  - *exp* and *frac* are swapped

	s	*exp*	*frac*	Value	Value
Denormalized	0	000	00	0.00×2^{-2}	0
	0	000	11	0.11×2^{-2}	3/16
Normalized	0	001	00	1.00×2^{-2}	1/4
	0	001	11	1.11×2^{-2}	7/16
	0	010	00	1.00×2^{-1}	1/2
	0	010	11	1.11×2^{-1}	7/8
	0	100	00	1.00×2^{0}	1
	0	100	11	1.11×2^{0}	1 3/4
	0	101	00	1.00×2^{1}	2
	0	101	11	1.11×2^{1}	3 1/2
	0	110	00	1.00×2^{2}	4
	0	110	11	1.11×2^{2}	7
Special Value	0	111	00	infinite	infinite
	0	111	11	NaN	NaN