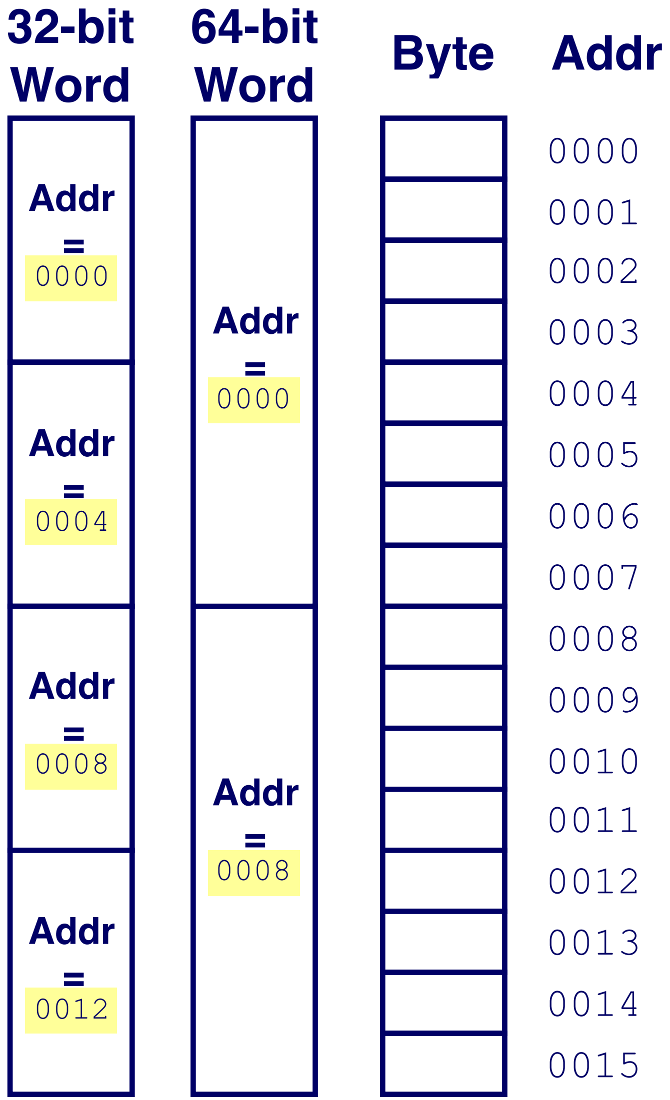
Word Size & Endianness

Word size

- Any given computer architecture has a "word size".
- Word size determines the number of bits used to store a memory address (a pointer in C).
- Therefore you can $2^{wordsize}$ number of memory addresses.
- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB of total RAM
- These days, machines have 64-bit word size, actually only uses 48 bits of it for addresses
  - Potentially, could have 2^{48} addresses, thats a lot of memory.
  - Theoretically up to 65,000 times amount of RAM of 32-bit systems. (~260TB)

Word-oriented memory organization

- Address of a word in memory is the address of the first byte in that word.
- Consecutive word addresses differ by 4 (32-bit) or 8 (64-bit).

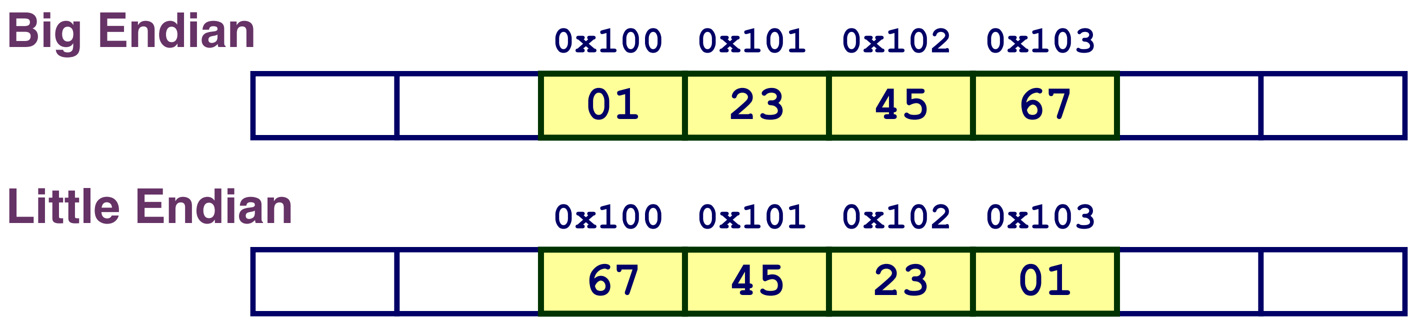


Byte ordering in a word

- There are two different conventions of byte ordering in a word
- **Big Endian**
  - Examples: Sun, PowerPC Mac, Internet
  - Most significant byte has lowest address
- **Little Endian**
  - Examples: x86, ARM processors running Android, iOS, Windows
  - Most significant byte has highest address
- In other words, if you have a multi-byte word, what order to the bytes appear? What “end” of the word does the MSB live at?

Byte ordering in a word *con't*

- Variable *x* has 4-byte *value* of *0x01234567*
- Address given by dereferencing *x* is *0x100*



- We can test this programmatically. See *memory/endian.c*

Byte ordering representation in C

- Casting any pointer to unsigned char* allows is to treat the memory as a byte array.
- Using printf format specifiers
  - %p - print pointer
  - %x - print value in hexadecimal
- See *memory/byte_ordering.c*

Floating Point

Fractional binary numbers

- How can we represent fractional binary numbers?
- One idea: use same approach as with decimal numbers, except use powers of 2 (as opposed to 10).
- So what is $\mathbf{1011.101_2}$?

Fractional binary numbers

- How can we represent fractional binary numbers?
- One idea: use same approach as with decimal numbers, except use powers of 2 (as opposed to 10).
- So what is $\mathbf{1011.101_2}$?

  $(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) + (1 * 2^{-1}) + (0 * 2^{-2}) + (1 * 2^{-3})$

  $8 + 2 + 1 + \frac{1}{2} + \frac{1}{8}$

  11.625_{10}

Fractional binary numbers

- How can we represent fractional binary numbers?
- One idea: use same approach as with decimal numbers, except use powers of 2 (as opposed to 10).
- So what is $\mathbf{1011.101_2}$?

  $(1 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) + (1 * 2^{-1}) + (0 * 2^{-2}) + (1 * 2^{-3})$

  $8 + 2 + 1 + ½ + ⅛$

  11.625_{10}

- Going the other direction
  - 5 3/4 —> 101.11_2
  - 2 7/8 —> 10.111_2

Insufficient representation

- That way of representing floating point numbers is simple, but has two significant limitations.
  - Only numbers that can be written as the sum of powers of 2 can be represented exactly.
    - Example
      - **1/3** $0.0101010101[01]\ldots_2$
      - **1/5** $0.001100110011[0011]\ldots_2$
      - **1/10** $0.0001100110011[0011]\ldots_2$
  - Just one possible location for the binary point.
    - This limits how many bits can be used for the fractional part and the whole number part.
    - We can either represent very large numbers well or very small numbers well, but not both.

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
- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Numerical analysts predominated over hardware designers in defining standard
  - Therefore, hard to make fast in hardware (i.e. its slow!)

+ Floating Point Representation

- Numerical form

$$(-1)^s * M_2 * 2^E$$

  - Sign bit *s* determines whether number is negative or positive
  - Mantissa **M** *normally* a fractional value, range [1.0, 2.0)
  - Exponent **E** weights value by power of two

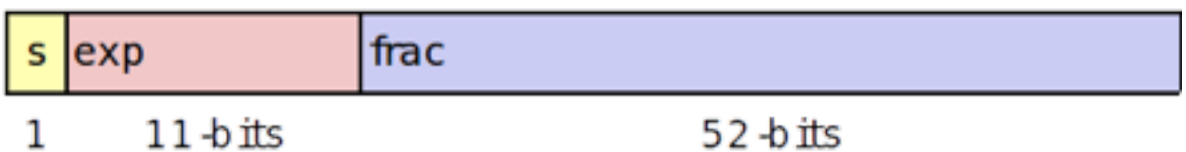
- Encoding
  - Most significant bit is sign bit **s**
  - **exp** field *encodes* **E** *(but is not equal to* **E***)*
  - **frac** field *encodes* **M** *(but is not equal to* **M***)*

s	exp	frac

IEEE precision options

Single precision: 32 bits

Double precision: 64 bits



Interpreting IEEE Values

- Three possible methods by which we evaluate a given bit vector representing a floating point type.
  - ‘Normalized’ values
  - ‘Denormalized’ values
  - ‘Special’ values
- Normalized is the most common case.
- Denormalized is for representing numbers very close to zero.
- Special, is well, special.
- The value of **exp** determines what kind of value it is, and therefore how it is encoded and interpreted.

s	exp	frac

Normalized values

- Precondition: ***exp*** $\neq$ 000…0 and ***exp*** $\neq$ 111…1
- For some bit pattern: $\mathbf{value_{10} = (-1)^{s} * M_{2} * 2^{E}}$
- **s** = sign bit ***s***
- **E** = [***exp***] - **bias**
  - **bias** = $2^{k\text{-}1}$ - 1
  - **k** = number of bits in ***exp***
- **M** = 1.[***frac***]
  - By assuming the leading bit is 1, we get an extra bit for "free"
  - Smallest value when all bits are zero: 000…0, M = 1.0
  - Largest value when all bits are one: 111…1, M = 2.0- ε

s	exp	frac

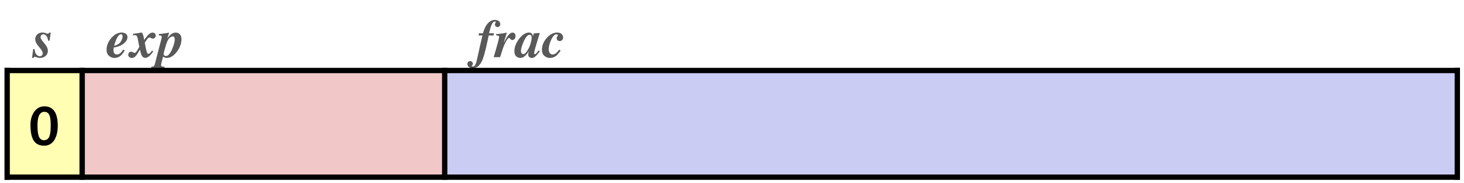
Normalized encoding example

- float f = 15213.0;

  $15213_{10} = 11101101101101_2$

  $= 1.1101101101101_2 \times 2^{13}$

$value_{10} = (-1)^s * M_2 * 2^E$



Normalized encoding example, *con't*

$$\text{value}_{10} = (-1)^s * M_2 * 2^E$$

- float f = 15213.0;

  $15213_{10} = 11101101101101_2$

  $= 1.1101101101101_2 \times 2^{13}$

- Mantissa

  M $= 1.1101101101101_2 \times 2^{13}$

  frac $= 11011011011010000000000_2$

s	*exp*	*frac*
0		11011011011010000000000

+ Normalized encoding example, *con't*

$$\textbf{value}_{10} = (-1)^s * M_2 * 2^E$$

- float f = 15213.0;

  $15213_{10} = 11101101101101_2$

  $= 1.1101101101101_2 \text{ x } 2^{13}$

- Mantissa

  $\mathbf{M} = 1.1101101101101_2 \text{ x } 2^{13}$

  $\textit{frac} = 11011011011010000000000_2$

- Exponent

  $\mathbf{E} = 13$

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

  $\textit{exp} = 140 = \mathbf{10001100_2}$

s	*exp*	*frac*
0	10001100	11011011011010000000000

Why?

- For normalized 32-bit single precision…
  - The value of ***exp*** is in the range $0 < exp < 255$
  - The value of **E** is in the range $-127 < E <= 127$
  - Fairly large numbers; $\mathbf{<= 2^{127}}$
  - Fairly small numbers; $\mathbf{>=2^{-126}}$
- For normalized 64-bit double precision, obviously this range is greater.
- Note that there is always a leading 1 in the value of mantissa **M** for 'normalized values', so we cannot represent numbers that are *very* small.
- Next, we will observe what happens when ***exp*** is either 00…0 or 11…1

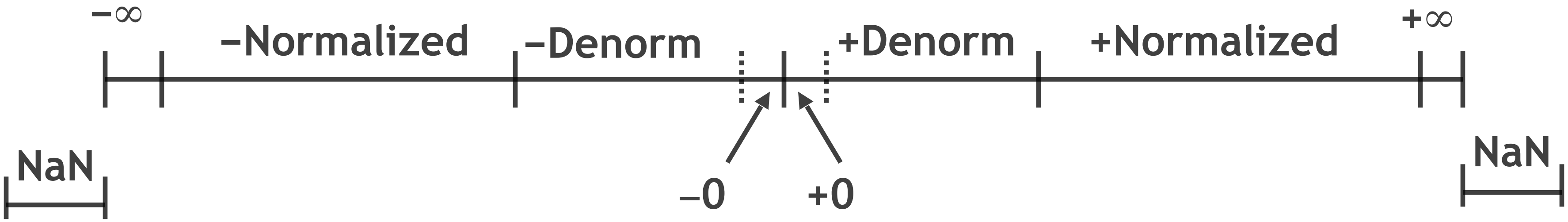
Denormalized values

- Precondition: ***exp*** = 000…0
- For some bit pattern: $\mathbf{value_{10} = (-1)^s * M_2 * 2^E}$
- **M** = 0.[***frac***]
  - No implicit 1 prefix.
  - Allows for representation of numbers much closer to 0
- **E = 1 - bias**
  - **bias** = $2^{k\text{-}1}$ - 1
  - **k** = number of bits in ***exp***
  - Differs from 'normalized', as ***exp*** obviously 0
- If ***exp*** = 000…0, ***frac*** = 000…0 represents 0.0
- If ***exp*** = 000…0, ***frac*** ≠ 000…0 represent numbers *very close* to 0.0

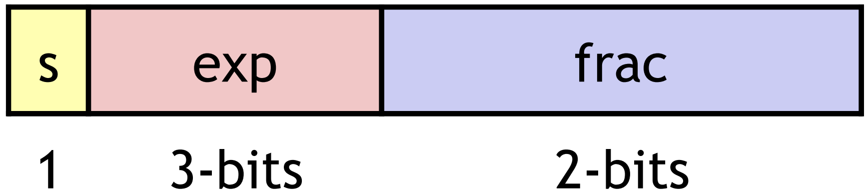
Special values

- Precondition: $\boldsymbol{exp} = 111\ldots1$
- If $\boldsymbol{exp} = 111\ldots1$, $\boldsymbol{frac} = 000\ldots0$
  - Represents positive or negative infinity, a result of overflow
  - Examples:
    - $-1.0/-0.0 = +\infty$
    - $1.0/-0.0 = -\infty$
- If $\boldsymbol{exp} = 111\ldots1$, $\boldsymbol{frac} \neq 000\ldots0$
  - Not-a-Number (NaN)
  - A case when no numeric value can be determined
  - Examples:
    - sqrt(-1)
    - ∞-∞
    - ∞*0

Floating point encoding number line



+ Tiny Floating Point



- 6-bit Floating Point Representation
  - The sign bit ***s*** is in the most significant bit
  - The next three bits are the ***exp***, with a **bias** of 3
    - Note that the **bias** is the same for all 6-bit precision numbers!
  - The last two bits are the ***frac***

- IEEE Format
  - normalized, denormalized and special values

Tiny Normalized Example 1

$$\text{value}_{10} = (-1)^s * M_2 * 2^E$$
$$E = exp - \text{bias}$$

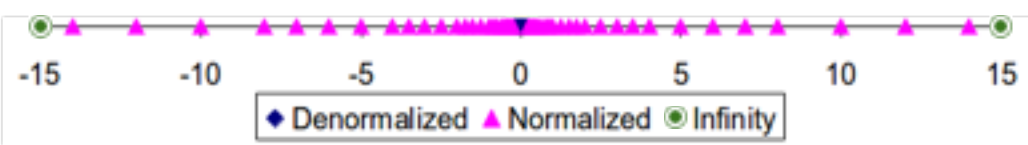
- **000100_2** (smallest positive value)
  - **$s = (-1)^0 = 1$**
  - **$M = 1.00_2$**
  - **bias** $= 2^{3-1} - 1 = 3_{10}$
  - **E** $= 001_2 - 3_{10} = 1_{10} - 3_{10} =$ **-2_{10}**
  - **$1 * 1.00_2 * 2^{-2} = .01_2 = 0.25_{10}$**

s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

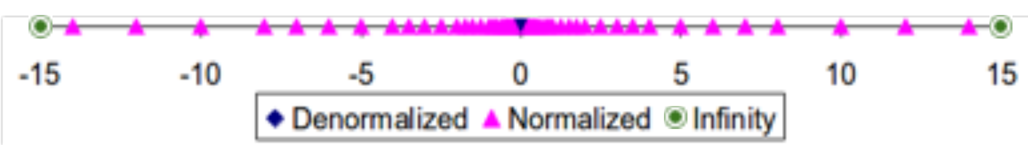
Normalized Denormalized Special



Tiny Normalized Example 2

$$value_{10} = (-1)^s * M_2 * 2^E$$
$$E = exp - bias$$

- **011011_2** (largest positive value)
  - **$s = (-1)^0 = 1$**
  - **$M = 1.11_2$**
  - **bias =** $2^{3-1} - 1 = 3_{10}$
  - **E** $= 110_2 - 3_{10} = 6_{10} - 3_{10} = 3_{10}$
  - **$1 * 1.11_2 * 2^3 = 1110_2 = 14.0_{10}$**



s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

Normalized Denormalized Special

+ Tiny Denormalized Example 1

$$\text{value}_{10} = (-1)^s * M_2 * 2^E$$
$$E = 1 - \text{bias}$$

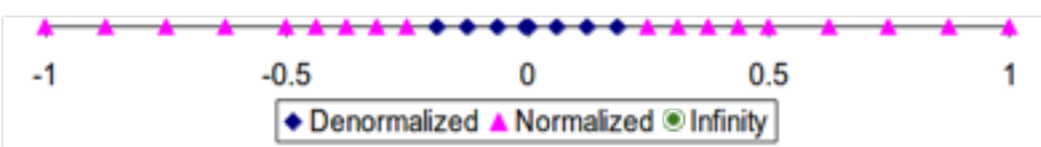
- 100011_2 (smallest negative value)
  - $s = (-1)^1 = -1$
  - $M = 0.11_2$
  - $\text{bias} = 2^{3-1} - 1 = 3_{10}$
  - $E = 1_{10} - 3_{10} = -2_{10}$
  - $-1 * 0.11_2 * 2^{-2} = -0.0011_2 = -0.1875_{10}$

s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

Normalized Denormalized Special



Tiny Denormalized Example 2

$$\text{value}_{10} = (-1)^s * M_2 * 2^E$$
$$E = 1 - \text{bias}$$

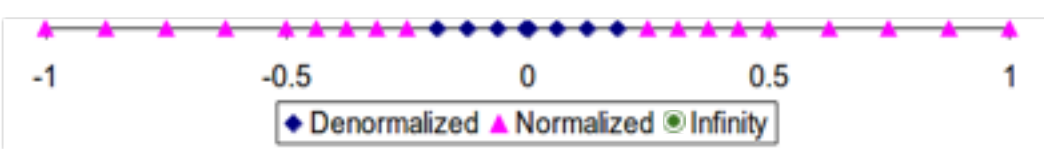
- 000001_2 (smallest positive less than 1)
  - $s = (-1)^0 = 1$
  - $M = 0.01_2$
  - bias $= 2^{3-1} - 1 = 3_{10}$
  - $E = 1_{10} - 3_{10} = -2_{10}$
  - $1 * 0.01_2 * 2^{-2} = 0.0001_2 = 0.0625_{10}$

s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

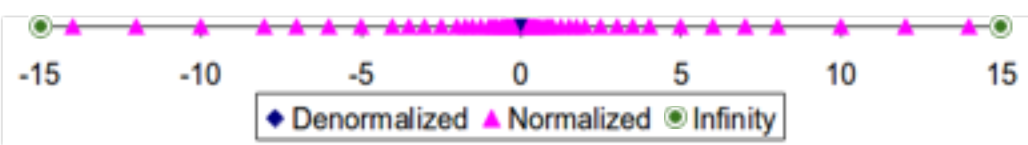
Normalized Denormalized Special



Tiny special values

Result of overflow or infeasibility

- *exp* = 111, *frac* = 00
  - 011100, 111100
  - Positive or negative infinity
- *exp* = 111, *frac* ≠ 00
  - 011101, 011110, 011111, 111101, 111110, 111111
  - Not-a-Number (NaN)



s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

Normalized Denormalized Special

Exercises

Exercise 1

$$\text{value}_{10} = (-1)^s * M_2 * 2^E$$
$$E = ? - \text{bias}$$

- 100111_2
  - $s = (-1)^s = ?$
  - $M = \ ?_2$
  - $\text{bias} = \ ?_{10}$
  - $E = ?_{10}$
  - $\text{value}_{10} = ? = -0.4375_{10}$

s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

Normalized · Denormalized · Special

+ Exercise 2

$$value_{10} = (-1)^s * M_2 * 2^E$$
$$E = ? - bias$$

- 100001_2
  - $s = (-1)^s = ?$
  - $M = \ ?_2$
  - $bias = \ ?_{10}$
  - $E = ?_{10}$
  - $value_{10} = ? = -0.0625_{10}$

s	exp	frac
1	3-bits	2-bits

All possible 6-bit sequences

000000	010000	100000	110000
000001	010001	100001	110001
000010	010010	100010	110010
000011	010011	100011	110011
000100	010100	100100	110100
000101	010101	100101	110101
000110	010110	100110	110110
000111	010111	100111	110111
001000	011000	101000	111000
001001	011001	101001	111001
001010	011010	101010	111010
001011	011011	101011	111011
001100	011100	101100	111100
001101	011101	101101	111101
001110	011110	101110	111110
001111	011111	101111	111111

◉ Normalized ◉ Denormalized ◉ Special