# Signed Numbers

# Signed Numbers

- Until now we've been concentrating on **unsigned** numbers. In real life we also need to be able represent **signed** numbers ( like: -12, -45, +78).
- A signed number MUST have a sign (+/-). A method is needed to represent the sign as part of  the binary representation.
- Two signed number representation methods are:
  - **Sign/magnitude** representation
  - **Twos-complement** representation

# Sign/Magnitude Representation

In sign/magnitude (S/M) representation, the **leftmost** bit of a binary code represents the sign of the value:

- 0 for positive,
- 1 for negative;

The remaining bits represent the numeric value.

# Sign/Magnitude Representation

To compute negative values using Sign/Magnitude (S/M) representation:

1) Begin with the binary representation of the **positive** value

2) Then flip the leftmost zero bit.

# Sign/Magnitude Representation

**Ex 1.** Find the S/M representation of $-6_{10}$

**Step 1:** Find binary representation using 8 bits

$6_{10} = 00000110_2$

**Step 2:** If the number you want to represent is **negative,** flip **leftmost** bit

10000110

So: $\mathbf{-6_{10} = 10000110_2}$

(in 8-bit sign/magnitude form)

# Sign/Magnitude Representation

**Ex 2.** Find the S/M representation of $70_{10}$

**Step 1:** Find binary representation using 8 bits

$70_{10} = 01000110_2$

**Step 2:** If the number you want to represent is **negative**, flip left most bit

01000110 (positive -- no flipping)

So: $\mathbf{70_{10} = 01000110_2}$

(in 8-bit sign/magnitude form)

# Sign/Magnitude Representation

**Ex 3.** Find the S/M representation of $-36_{10}$

**Step 1:** Find binary representation using 8 bits

$-36_{10} = 00100100_2$

**Step 2:** If the number you want to represent is **negative**, flip left most bit

10100100

So: $\mathbf{-36_{10} = 10100100_2}$

(in 8-bit sign/magnitude form)

# Sign/Magnitude Representation

<u>32-bit example:</u>

0 000 0000 0000 0000 0000 0000 0000 1001 → +9

1 000 0000 0000 0000 0000 0000 0000 1001 → -9

Sign bit:

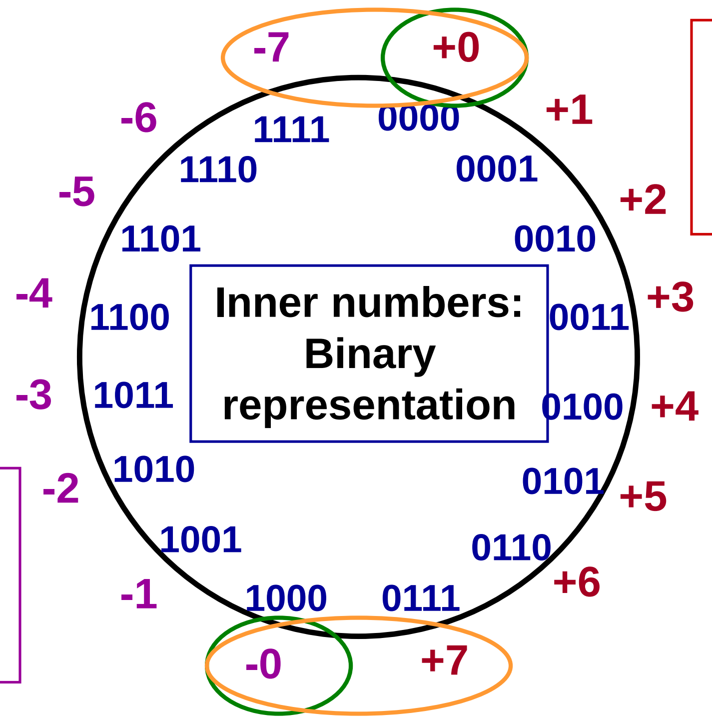
0 → positive

1 → negative

31 remaining bits for ***magnitude***
***(i.e. the value)***

# Problems with Sign/Magnitude

Seven Positive Numbers and “Positive” Zero




Seven Negative Numbers and “Negative” Zero

- **Two different representations for 0!**
- **Two discontinuities**

# Two’s Complement Representation

- Another method used to represent negative numbers (used by most modern computers) is two’s complement.

- The leftmost bit STILL serves as a sign bit:
  - 0 for positive numbers,
  - 1 for negative numbers.

# Two’s Complement Representation

To compute **negative** values using Two’s Complement representation:

1) Begin with the binary representation of the positive value
2) Complement (flip each bit -- if it is 0 make it 1 and visa versa) the entire positive number
3) Then add one.

# Two's Complement Representation

**Ex 1**. Find the 8-bit two's complement representation of $-6_{10}$

**Step 1:** Find binary representation of the positive value in 8 bits

$6_{10} = 00000110_2$

# Two's Complement Representation

Ex 1 continued

**Step 2:** Complement the entire positive value

Positive Value: 00000110

Complemented: 11111001

# Two’s Complement Representation

Ex 1, **Step 3**: Add one to complemented value

```
(complemented)    ->    11111001
(add one)         ->  +        1
                       --------
                        11111010
```

So: $\mathbf{-6_{10} = 11111010_2}$

(in 8-bit 2's complement form)

# Two's Complement Representation

**Ex 2**. Find the 8-bit two's complement representation of $20_{10}$

**Step 1:** Find binary representation of the positive value in 8 bits

$$20_{10} = 00010100_2$$

20 is positive, so STOP after step 1!

So: $\mathbf{20_{10} = 00010100_2}$

(in 8-bit 2's complement form)

# Two's Complement Representation

**Ex 3**. Find the 8-bit two's complement representation of $-80_{10}$

**Step 1:** Find binary representation of the positive value in 8 bits

$80_{10} = 01010000_2$

-80 is negative, so continue…

# Two’s Complement Representation

Ex 3

**Step 2:** Complement the entire positive value

Positive Value: **01010000**

Complemented: **10101111**

# Two's Complement Representation

Ex 3, **Step 3**: Add one to complemented value

```
(complemented) ->    10101111
(add one)      ->  +        1
                   ----------
                     10110000
```

So: $\mathbf{-80_{10} = 10110000_2}$

(in 8-bit 2's complement form)

# Two's Complement Representation

**Alternate method -- replaces previous steps 2-3**

**Step 2**: Scanning the positive binary representation from right to left, ⟵

find first **one** bit, from low-order (right) end

**Step 3**: Complement (flip) the remaining bits to the **left**.

00000110

(left complemented) --> 11111010

# Two's Complement Representation

**Ex 1:** Find the Two's Complement of $-76_{10}$

**Step 1:** Find the 8-bit binary representation of the positive value.

$$76_{10} = 01001100_2$$

# Two's Complement Representation

**Step 2:** Find first one bit, from low-order (right) end, and complement the pattern to the left.

`01001100`

(left complemented) -> `10110100`

So: **$-76_{10} = 10110100_2$**

(in 8-bit 2's complement form)

# Two's Complement Representation

**Ex 2:** Find the Two's Complement of $72_{10}$

**Step 1:** Find the 8 bit binary representation of the positive value.

$$72_{10} = 01001000_2$$

**Steps 2-3:** 72 is positive, so STOP after step 1!

*So:* $\mathbf{72_{10} = 01001000_2}$

(in 8-bit 2's complement form)

# Two's Complement Representation

**Ex 3:** Find the Two's Complement of $-26_{10}$

**Step 1:** Find the 8-bit binary representation of the positive value.

$26_{10} = 00011010_2$

# Two's Complement Representation

Ex 3, **Step 2:** Find first one bit, from low-order (right) end, and complement the pattern to the left.

00011010

(left complemented) -> 11100110

So: $\mathbf{-26_{10} = 11100110_2}$

(in 8-bit 2's complement form)

# Two's Complement Representation

32-bit example:

0 000 0000 0000 0000 0000 0000 0000 1001 → +9

1 111 1111 1111 1111 1111 1111 1111 0111 → -9

Sign bit:
0 --> positive
1 --> negative

31 remaining bits for *magnitude*
*(i.e. value stored in two's complement form)*

# Two's Complement to Decimal

**Ex 1:** Find the decimal equivalent of the 8-bit 2's complement value $11101100_2$

**Step 1:** Determine if number is positive or negative:

Leftmost bit is 1, so number is negative.

# Two’s Complement to Decimal

Ex 1, **Step 2:** Find first one bit, from low-order (right) end, and complement the pattern to the left.

11101100

(left complemented)→ 00010100

# Two's Complement to Decimal

Ex 1, **Step 3**: Determine the numeric value:

$$00010100_2 = 16 + 4 = 20_{10}$$

So: $11101100_2 = -20_{10}$

(8-bit 2's complement form)

# Two's Complement to Decimal

**Ex 2:** Find the decimal equivalent of the 8-bit 2's complement value $01001000_2$

**Step 1:** Determine if number is positive or negative:

Leftmost bit is 0, so number is positive.

Skip to step 3.

# Two's Complement to Decimal

**Ex2, Step 3**: Determine the numeric value:

$$01001000_2 = 64 + 8 = 72_{10}$$

So: $01001000_2 = 72_{10}$
(8-bit 2's complement form)

# Two's Complement to Decimal

**Ex 3:** Find the decimal equivalent of the 8-bit 2's complement value $11001000_2$

**Step 1:** Determine if number is positive or negative:

Leftmost bit is 1, so number is negative.

# Two’s Complement to Decimal

**Ex 3, Step 2:** Find first one bit, from low-order (right) end, and complement the pattern to the left.

11001000

(left complemented)→ 00111000

# Two's Complement to Decimal

**Ex 3, Step 3**: Determine the numeric value:

$$00111000_2 = 32 + 16 + 8 = 56_{10}$$

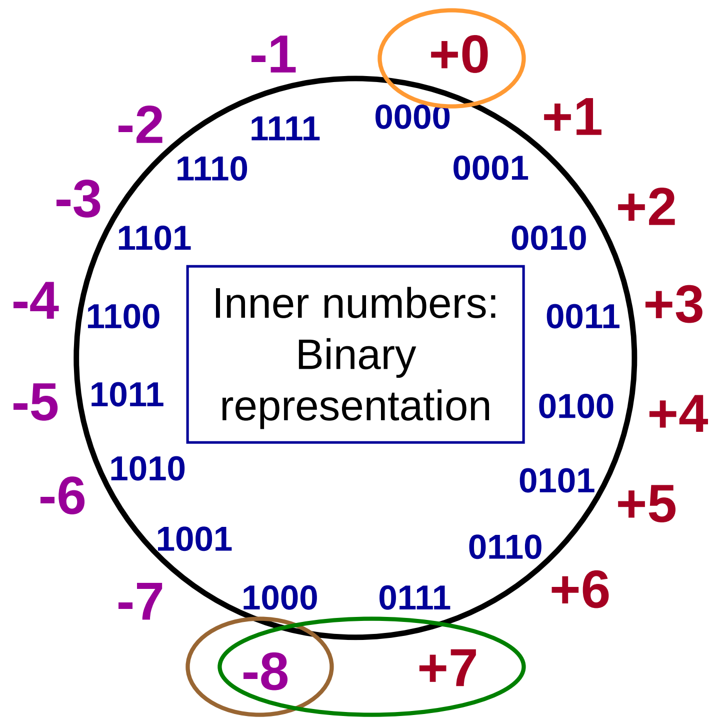
So: $11001000_2 = -56_{10}$

(8-bit 2's complement form)

# S/M problems solved with 2s complement (negative no. used to be presented in this form )

Re-order Negative numbers to eliminate one Discontinuity

Note:
Negative Numbers **still have** 1 for the most significant bit (MSB)




Eight Positive Numbers

- **Only one discontinuity now**
- **Only one zero**
- **One extra negative number**

# Two’s Complement Representation

Biggest reason two’s complement used in most systems today?

The binary codes can be added and subtracted as if they were unsigned binary numbers, without regard to the signs of the numbers they actually represent.

# Two's Complement Representation

For example, to add +4 and -3, we simply add the corresponding binary codes, 0100 and 1101:

```
  0100    (+4)
 +1101    (-3)
 -----    ----
  0001    (+1)
```

NOTE: A carry to the leftmost column has been ignored.

The result, 0001, is the code for +1, which IS the sum of +4 and -3.

# Twos Complement Representation

Likewise, to subtract +7 from +3:

```
     0011   (+3)
   - 0111   (+7)
     ----   ----
     1100   (-4)
```

NOTE: A “phantom” 1 was borrowed from beyond the leftmost position.

The result, 1100, is the code for -4, the result of subtracting +7 from +3.

# Two’s Complement Representation

Summary - Benefits of Twos Complements:

- Addition and subtraction are simplified in the two’s-complement system,
- -0 has been eliminated, replaced by one extra negative value, for which there is no corresponding positive number.

# Valid Ranges

- For any integer data representation, there is a LIMIT to the **size** of number that can be stored.

- The limit depends upon **number of bits** available for data storage.

# **Unsigned** Integer Ranges

**Range = 0 to $(2^n – 1)$**

where **n** is the number of bits used to store the unsigned integer.

Numbers with values GREATER than **$(2^n – 1)$** would require more bits. If you try to store too large a value without using more bits, OVERFLOW will occur.

# **Unsigned** Integer Ranges

Example: On a system that stores unsigned integers in 16-bit words:

**Range = 0 to $(2^{16} - 1)$**

**= 0 to 65535**

Therefore, you cannot store numbers larger than 65535 in 16 bits.

# **Signed** S/M Integer Ranges

**Range = $-(2^{(n-1)} - 1)$ to $+(2^{(n-1)} - 1)$**

where **n** is the number of bits used to store the sign/magnitude integer.

Numbers with values GREATER than $+(2^{(n-1)} - 1)$ and values LESS than $-(2^{(n-1)} - 1)$ would require more bits. If you try to store too large/too small a value without using more bits, OVERFLOW will occur.

# **S/M** Integer Ranges

Example: On a system that stores unsigned integers in 16-bit words:

**Range = $-(2^{15} – 1)$ to $+(2^{15} – 1)$**

**= -32767 to +32767**

Therefore, you cannot store numbers larger than 32767 or smaller than -32767 in 16 bits.

# **Two's Complement** Ranges

**Range = $-2^{(n-1)}$ to $+(2^{(n-1)} - 1)$**

where **n** is the number of bits used to store the two-s complement signed integer.

Numbers with values GREATER than +**$(2^{(n-1)} - 1)$** and values LESS than **$-2^{(n-1)}$** would require more bits. If you try to store too large/too small a value without using more bits, OVERFLOW will occur.

# Two's Complement Ranges

Example: On a system that stores unsigned integers in 16-bit words:

**Range = $-2^{15}$ to $+(2^{15} - 1)$**

**= -32768 to +32767**

Therefore, you cannot store numbers larger than 32767 or smaller than -32768 in 16 bits.

# Using Ranges for Validity Checking

- Once you know how small/large a value can be stored in n bits, you can use this knowledge to check whether you answers are valid, or cause overflow.
- Overflow can only occur if you are adding **two positive** numbers or **two negative** numbers

# Using Ranges for Validity Checking

Ex 1:

Given the following 2's complement equations in 5 bits, is the answer valid?

```
  11111   (-1)
+11101   (-3)
  11100   (-4)
```

**Range =**

**-16 to +15**

**→ VALID**

# Using Ranges for Validity Checking

Ex 2:

Given the following 2's complement equations in 5 bits, is the answer valid?

```
 10111   (-9)
+10101  (-11)
 01100  (-20)
```

**Range =**

**-16 to +15**

**→ INVALID**

# Floating Point Numbers

# Floating Point Numbers

- Now you've seen **unsigned** and **signed** integers. In real life we also need to be able represent numbers with fractional parts (like: -12.5 & 45.39).
  - Called **Floating Point** numbers.
  - You will learn the IEEE 32-bit floating point representation.

# Floating Point Numbers

- In the decimal system, a decimal point (**radix point**) separates the whole numbers from the fractional part
- Examples:

  37.25 ( whole = 37, fraction = 25/100)

  123.567

  10.12345678

# Floating Point Numbers

For example, 37.25 can be analyzed as:

| | | | |
|---|---|---|---|
| $10^{1}$ | $10^{0}$ | $10^{-1}$ | $10^{-2}$ |
| Tens | Units | Tenths | Hundredths |
| **3** | **7** | **2** | **5** |

**37.25 = (3 x 10) + (7 x 1) + (2 x 1/10) + (5 x 1/100)**

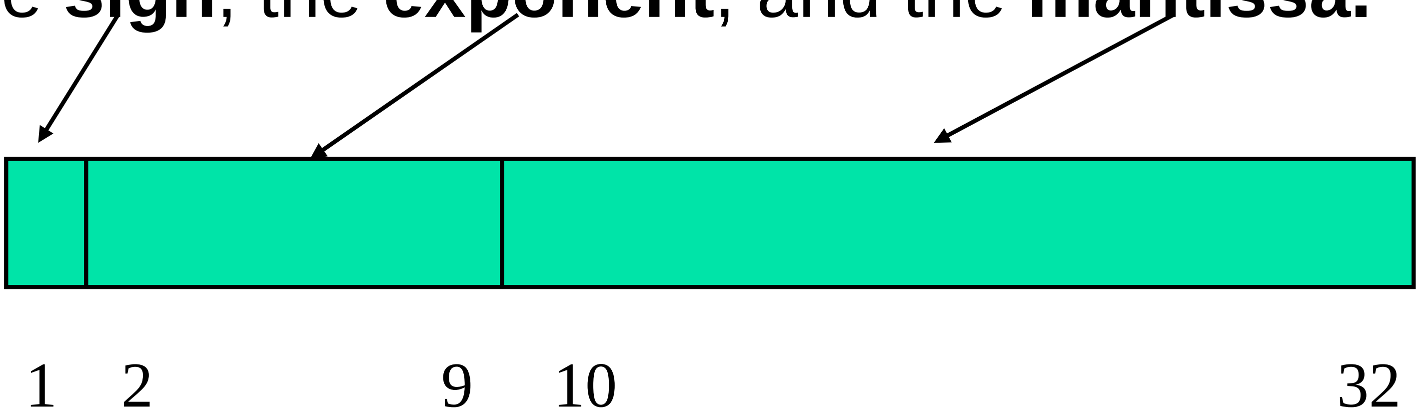
# Problem storing binary form

- We have no way to store the radix point!
- Standards committee came up with a way to store floating point numbers (that have a decimal point)

# IEEE Floating Point Representation

- Floating point numbers can be stored into 32-bits, by dividing the bits into three parts:

the **sign**, the **exponent**, and the **mantissa.**



# IEEE Floating Point Representation

- The first (leftmost) field of our floating point representation will STILL be the sign bit:
  - 0 for a positive number,
  - 1 for a negative number.

# Storing the Binary Form

How do we store a radix point?

- All we have are zeros and ones…

Make sure that the radix point is ALWAYS in the same position within the number.

Use the IEEE 32-bit standard

→ the **leftmost** digit must be a 1

# Solution is Normalization

Every binary number, **except the one corresponding to the number zero**, can be normalized by choosing the exponent so that the radix point falls to the right of the leftmost 1 bit.

$37.25_{10} = 100101.01_2 = 1.0010101 \times \mathbf{2}^{\mathbf{5}}$

$7.625_{10} = 111.101_2 = 1.11101 \times \mathbf{2}^{\mathbf{2}}$

$0.3125_{10} = 0.0101_2 = 1.01 \times \mathbf{2}^{\mathbf{-2}}$

# IEEE Floating Point Representation

- The second field of the floating point number will be the **exponent**.
- The exponent is stored as an unsigned 8-bit number, RELATIVE to a **bias of 127.**
  - Exponent 5 is stored as (127 + 5) or 132
    - 132 = 10000100
  - Exponent -5 is stored as (127 + (-5)) or 122
    - 122 = 01111010

# Try It Yourself

How would the following exponents be stored (8-bits, 127-biased):

$$2^{-10}$$

$$2^{8}$$

(Answers on next slide)

# Answers

$2^{-10}$

| | | | |
|---|---|---|---|
| exponent | -10 | | 8-bit |
| bias | +127 | | value |
| | 117 | → | 01110101 |

---

$2^{8}$

| | | | |
|---|---|---|---|
| exponent | 8 | | 8-bit |
| bias | +127 | | value |
| | 135 | → | 10000111 |

# IEEE Floating Point Representation

- The **mantissa** is the set of 0's and 1's to the right of the radix point of the **normalized** (when the digit to the left of the radix point is 1) binary number.

  Ex: $1.\mathbf{00101} \times 2^3$

  (The mantissa is 00101)

- The mantissa is stored in a 23 bit field, so we add zeros to the right side and store:

  **00101**000000000000000000

# Decimal Floating Point to IEEE standard Conversion

**Ex 1**: Find the IEEE FP representation of 40.15625

**Step 1**.

Compute the binary equivalent of the whole part and the fractional part. (i.e. convert 40 and .15625 to their binary equivalents)

# Decimal Floating Point to IEEE standard Conversion

| | | | |
|---|---|---|---|
| 40 | | .15625 | |
| − 32 | **Result:** | −.12500 | **Result:** |
| 8 | 101000 | .03125 | .00101 |
| − 8 | | −.03125 | |
| 0 | | .0 | |

So: $40.15625_{10} = 101000.00101_2$

# Decimal Floating Point to IEEE standard Conversion

**Step 2**. Normalize the number by moving the decimal point to the right of the leftmost one.

$$101000.00101 \;=\; 1.0100000101 \times \mathbf{2}^{5}$$

# Decimal Floating Point to IEEE standard Conversion

**Step 3**. Convert the exponent to a biased exponent

$$127 + 5 = 132$$

And convert biased exponent to 8-bit unsigned binary:

$$132_{10} = 10000100_2$$

# Decimal Floating Point to IEEE standard Conversion

**Step 4**. Store the results from steps 1-3:

| Sign | Exponent (from step 3) | Mantissa (from step 2) |
| --- | --- | --- |
| **0** | **10000100** | **01000001010000000000000** |

# Decimal Floating Point to IEEE standard Conversion

**Ex 2**: Find the IEEE FP representation of **–24.75**

**Step 1**. Compute the binary equivalent of the whole part and the fractional part.

| | | | |
|---|---|---|---|
| 24 | | .75 | |
| – 16 | **Result:** | – .50 | **Result:** |
| 8 | 11000 | .25 | .11 |
| – 8 | | – .25 | |
| 0 | | .0 | |

So: $-24.75_{10} = -11000.11_2$

# Decimal Floating Point to IEEE standard Conversion

**Step 2**.

Normalize the number by moving the decimal point to the right of the leftmost one.

$$-11000.11 = -1.100011 \times 2^4$$

# Decimal Floating Point to IEEE standard Conversion.

**Step 3**. Convert the exponent to a biased exponent

$$127 + 4 = 131$$

$$\Longrightarrow \quad 131_{10} = 10000011_{2}$$

**Step 4**. Store the results from steps 1-3

| Sign | Exponent | mantissa |
|---|---|---|
| **1** | **10000011** | **1000110..0** |

# IEEE standard to Decimal Floating Point Conversion.

- Do the steps in reverse order
- In reversing the normalization step move the radix point the number of digits equal to the exponent:
  - If exponent is positive, move to the right
  - If exponent is negative, move to the left

# IEEE standard to Decimal Floating Point Conversion.

**Ex 1**: Convert the following 32-bit binary number to its decimal floating point equivalent:

| Sign | Exponent | Mantissa |
| --- | --- | --- |
| 1 | 01111101 | 010..0 |

# IEEE standard to Decimal Floating Point Conversion..

**Step 1**: Extract the biased exponent and unbias it

Biased exponent = $01111101_2$ = $125_{10}$

Unbiased Exponent: 125 – 127 = -2

# IEEE standard to Decimal Floating Point Conversion..

**Step 2:** Write Normalized number in the form:

**1 .** ___Mantissa___ x 2 $^{\text{Exponent}}$

For our number:

$-1.01 \times 2^{-2}$

# IEEE standard to Decimal Floating Point Conversion.

**Step 3:** Denormalize the binary number from step 2 (i.e. move the decimal and get rid of ($x\ 2^n$) part):

$-0.0101_2$ (negative exponent – move left)

**Step 4**: Convert binary number to the FP equivalent (i.e. Add all column values with 1s in them)

$$-0.0101_2 = -(0.25 + 0.0625)$$

$$= -0.3125_{10}$$

# IEEE standard to Decimal Floating Point Conversion.

**Ex 2**: Convert the following 32 bit binary number to its decimal floating point equivalent:

| Sign | Exponent | Mantissa |
|---|---|---|
| 0 | 10000011 | 10011000..0 |

# IEEE standard to Decimal Floating Point Conversion..

**Step 1**: Extract the biased exponent and unbias it

Biased exponent = $1000011_2$ = $131_{10}$

Unbiased Exponent: 131 – 127 = 4

# IEEE standard to Decimal Floating Point Conversion..

**Step 2:** Write Normalized number in the form:

$$\mathbf{1}\,.\,\underline{\quad\text{Mantissa}\quad} \times 2^{\text{Exponent}}$$

For our number:

$$1.10011 \times 2^{4}$$

# IEEE standard to Decimal Floating Point Conversion.

**Step 3**: Denormalize the binary number from step 2 (i.e. move the decimal and get rid of (x $2^n$) part:

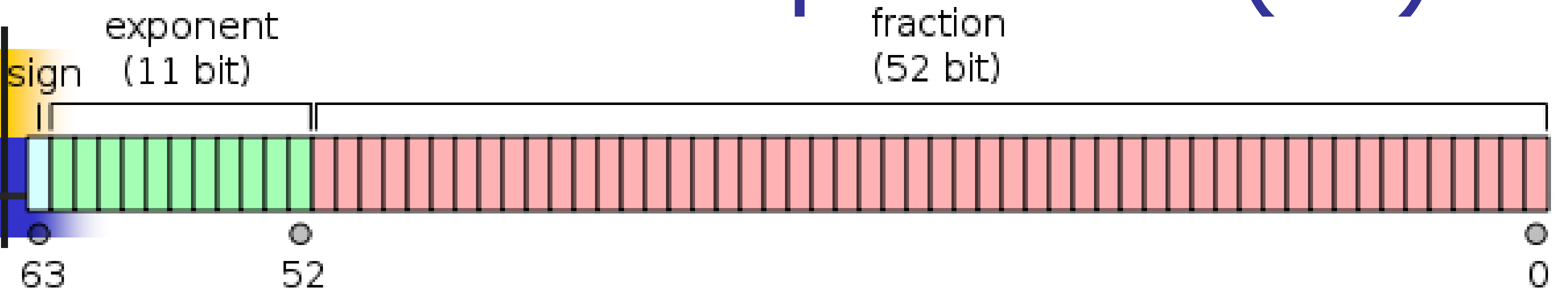
$11001.1_2$ (positive exponent – move right)

**Step 4**: Convert binary number to the FP equivalent (i.e. Add all column values with 1s in them)

11001.1 = 16 + 8 + 1 +.5

**= $25.5_{10}$**

# IEEE-754 double precision (64)




Sign bit: 1 bit
Exponent width: 11 bits
Precision: 52 bits
$E_{min} = -1022$
$E_{max} = 1023$
Exponent bias = 1023

$0010\ 0000\ 0000\ 0000_{16} = 2^{-1022} \approx 2.2250738585072014 \times 10^{-308}$ (Min normal positive double)

$7fef\ ffff\ ffff\ ffff_{16} = (1 + (1 - 2^{-52})) \times 2^{1023} \approx 1.7976931348623157 \times 10^{308}$ (Max Double)