# COMPUTER SYSTEMS AND ORGANIZATION

## Bitwise Operations

Daniel Graham

# BITWISE AND &

$$\begin{array}{r} 1100_2 \\ \& \; 0110_2 \\ \hline 0100_2 \end{array}$$

```
#python example
x = 12
y = 6
z = x & y
print(z)
#prints 4
```

Not different from a logical AND

# BITWISE OR |

$$\begin{array}{r} 1100_2 \\ \mid\ 0110_2 \\ \hline 1110_2 \end{array}$$

```
#python example
x = 12
y = 6
z = x | y
print(z)
#prints 14
```

# BITWISE OR XOR ^

$$
\begin{array}{rr}
 & 1100_2 \\
\hat{} & 0110_2 \\
\hline
 & 1010_2
\end{array}
$$

```
#python example
x = 12
y = 6
z = x ^ y
print(z)
#prints 10
```

# BITWISE RIGHT SHIFT

$1101001_2$

$>> 3$

---

$0001101_2$

```
#python example
x = 105
y = x >> 3
print(y)
#prints 13
```

# SIGN EXTENSIONS

$11000_2 >> 2 = 11110_2$ With Sign Extension. (The sign bit is copied)

$11000_2 >> 2 = 00110_2$ Without Sign Extension

# LEFT SHIFT

$1101_2$
$<< 3$
---
$1101000_2$

```
#python example
x = 13
y = x << 3
print(y)
#prints 104
```

# SHIFTING MULTIPLYING AND DIVIDING BY 2

A left shift is equivalent to multiplying by 2

0001 << 1 = 0010 (2).

0001 << 2 = 0100 (4)

0001 << 3 = 1000 (8)

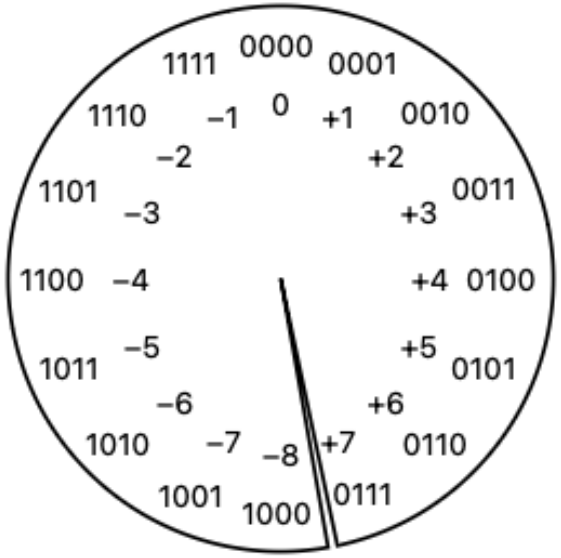
A right shift is equivalent to dividing by 2

01000 >> 1 = 0100 (4)

01000 >> 2 = 0010 (2)

01000 >> 3 = 0001 (1)

# BITWISE INVERT ~

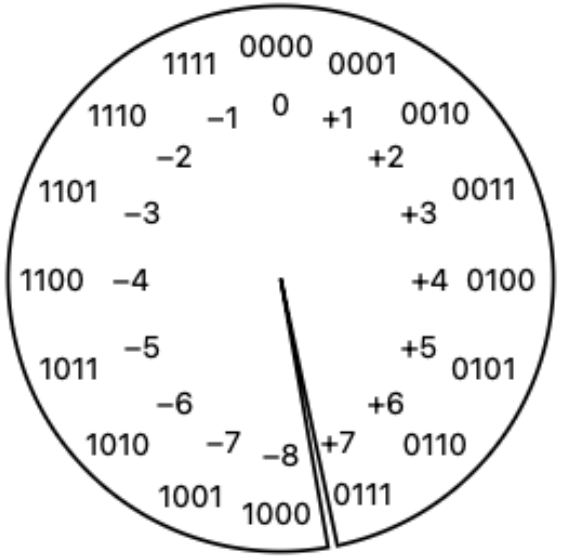



$$\frac{\sim\ 0000_2}{1111_2}$$

```
#python example
x = 0
z = ~x
print(z)
#prints -1
```

Note that this is different from logical not !

# BITWISE INVERT ~




$$\frac{\sim\ 0110_2}{1001_2}$$

```
#python example
x = 6
z = ~x
print(z)
#prints -7
```

# SETTING BITS TO 1

Set the last bit of this variable 1

$$\begin{array}{r} 0000_2 \\ \mid\ 0001_2 \\ \hline 0001_2 \end{array}$$

```
#python example
x = 0
x = x | 0x01
print(x)
#prints 1
```

# SETTING BITS TO 1

Set the third bit of x to 1

$$\begin{array}{r} 0000_2 \\ \mid\ 0100_2 \\ \hline 0100_2 \end{array}$$

```
#python example
x = 0
x = x | 0x04
print(x)
#prints 4
```

Question: What if it was already one?

# SETTING BITS TO 1

Set the n bit of x to 1

$$\begin{array}{r} 0000_2 \\ \mid\ 0001_2 \ \ll n\ (3) \\ \hline 1000_2 \end{array}$$

```
#python example
x = 0
n = 3
x = x | (0x01 << n)
print(x)
#prints 8
```

Question: What if it was already one?

# FLIPPING BITS

Flip the second bit of x. 1 => 0 and 0 => 1

$$\begin{array}{rr} & 1100_2 \\ \wedge & 0010_2 \\ \hline & 1110_2 \end{array}$$

What if the nth bit was 1 instead?

# FLIPPING BITS

Flip the **nth** bit of x. 1 => 0 and 0 => 1

$$\begin{array}{rr} & 1100_2 \\ \wedge & 0010_2 \\ \hline & 1110_2 \end{array}$$

```
#python example
x = 12
n = 1
x = x ^ (0x01 << n)
print(x)
#prints 14
```

# MASKING (EXTRACTING BITS)

The idea of masking is that we can extract a certain section of bits by anding.

$$
\begin{array}{rl}
 & 11011100_2 \\
\& & 11110000_2 \\
\hline
 & 11010000_2
\end{array}
$$

Upper 4 bits extracted

# MASKING (EXTRACTING BITS)

The idea of masking is that we can extract a certain section of bits by anding.

$$\begin{array}{r} 11011100_2 \\ \& \; 00001111_2 \\ \hline 00001100_2 \end{array}$$

```
#python example
x = 220
mask = 0x0F
x = x & mask
print(x)
#prints 12
```

Lower 4 bits extracted

# MASKING (EXTRACTING BITS)

The idea of masking is that we can extract a certain section of bits by anding.

$$
\begin{array}{rr}
 & 11011100_2 \\
\& & 11110000_2 \\
\hline
 & 11010000_2
\end{array}
$$

```
#python example
x = 220
mask = ~0x0F
x = x & mask
print(x)
#prints 208
```

Upper 4 bits extracted

# COMBINING

We can also set multiple bits simultaneously

$$\begin{array}{r} 10100000_2 \\ \mid\ 00001111_2 \\ \hline 10101111_2 \end{array}$$

```
#python example
b = 0x0F
a = 0xA0
x = a | b
print(hex(x))
#prints 0xAF
```

# PARITY

Suppose you want to want to calculate the even parity of x.

If the total amount of "1" bits is odd, the parity value is 1, otherwise it is zero.

0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
    parity ^= (x&1)
     x >>= 1
```

# PARALLEL EVALUATION

Observe that xor is both transitive and associative; thus we can re-write

x0⊕x1⊕x2⊕x3⊕x4⊕x5⊕x6⊕x7

using transitivity as
x0⊕x4⊕x1⊕x5⊕x2⊕x6⊕x3⊕x7

and using associativity as
(x0⊕x4)⊕(x1⊕x5)⊕(x2⊕x6)⊕(x3⊕x7)

and then compute the contents of all the parentheses at once via
x ^ (x>>4).

# PARALLEL EVALUATION

x0⊕x1⊕x2⊕x3⊕x4⊕x5⊕x6⊕x7
using transitivity as

x0⊕x4⊕x1⊕x5⊕x2⊕x6⊕x3⊕x7

and using associativity as
(x0⊕x4)⊕(x1⊕x5)⊕(x2⊕x6)⊕(x3⊕x7)

and then compute all at once via
x ^ (x>>4).

```
x ^= (x>>16)
x ^= (x>>8)
x ^= (x>>4)
x ^= (x>>2)
x ^= (x>>1)
parity = (x & 1)
```

# PARITY

Suppose you want to want to calculate the even parity of x.

If the number of one's bit is number is odd the parity value is 1, otherwise it is zero

0010 parity bit is 1

0110 parity bit is 0

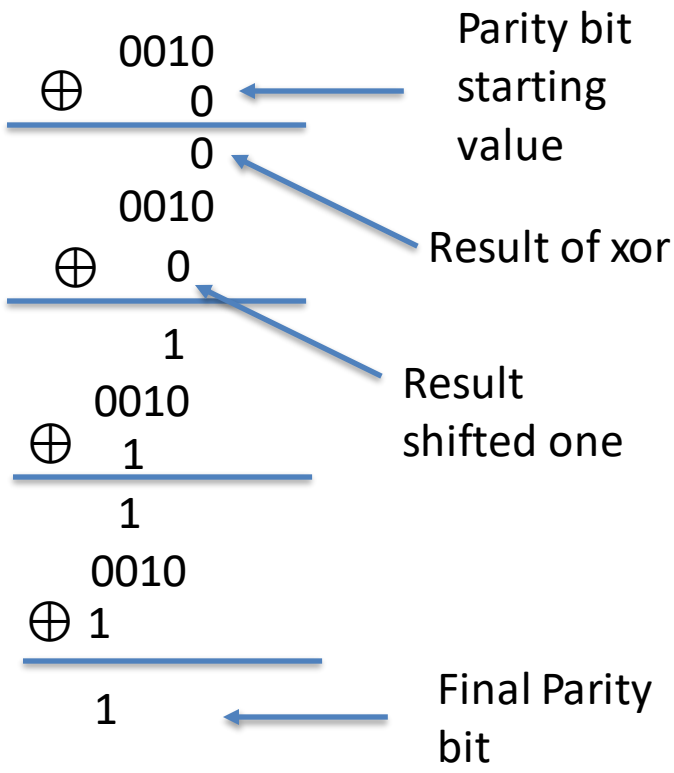
```
parity = 0
repeat 32 times:
    parity ^= (x&1)
     x >>= 1
```

# PARITY

0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
    parity ^= (x&1)
     x >>= 1
```



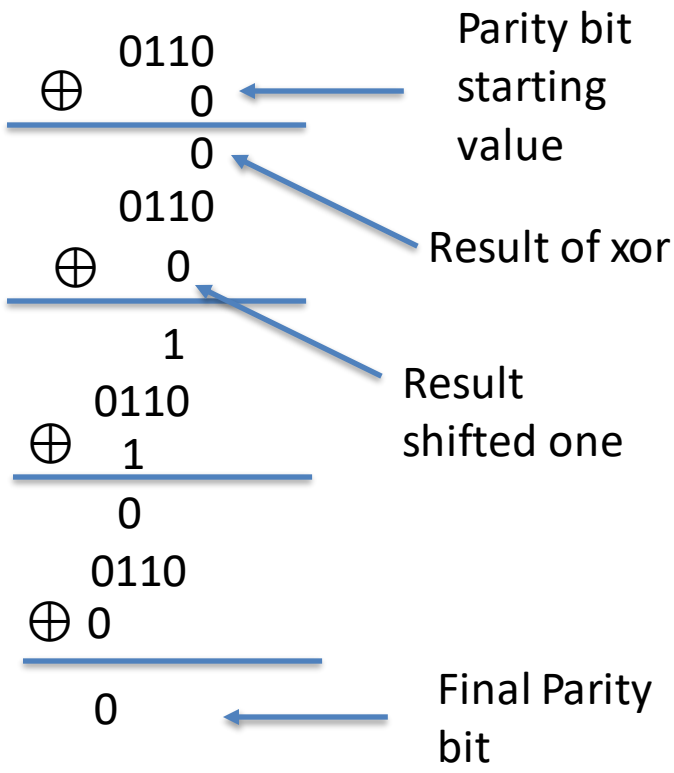

Same as just xoring each bit 0 ⊕ 0 ⊕ 1 ⊕ 0 = 1

# PARITY

0010 parity bit is 1

0110 parity bit is 0

```
parity = 0
repeat 32 times:
    parity ^= (x&1)
     x >>= 1
```




Same as just xoring each bit 0 ⊕ 1 ⊕ 1 ⊕ 0 = 0

# PARALLEL EVALUATION

Observe that xor is both transitive and associative; thus we can re-write

x0⊕x1⊕x2⊕x3⊕x4⊕x5⊕x6⊕x7

using transitivity as
x0⊕x4⊕x1⊕x5⊕x2⊕x6⊕x3⊕x7

and using associativity as
(x0⊕x4)⊕(x1⊕x5)⊕(x2⊕x6)⊕(x3⊕x7)

and then compute the contents of all the parentheses at once via
x ^ (x>>4).

# PARALLEL EVALUATION

x0⊕x1⊕x2⊕x3⊕x4⊕x5⊕x6⊕x7
using transitivity as

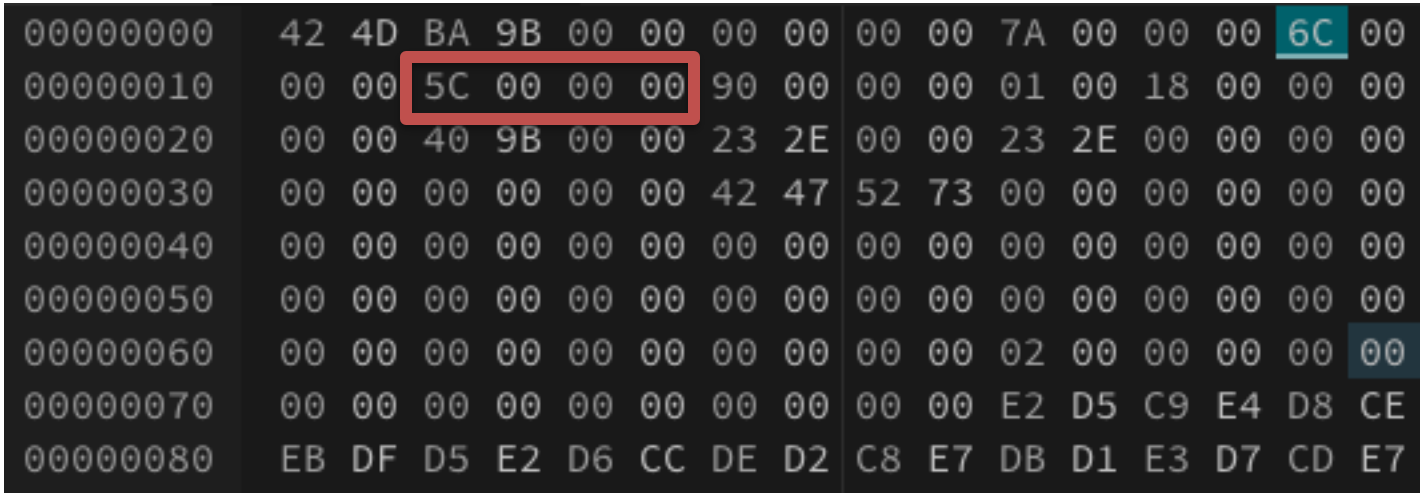
x0⊕x4⊕x1⊕x5⊕x2⊕x6⊕x3⊕x7

and using associativity as
(x0⊕x4)⊕(x1⊕x5)⊕(x2⊕x6)⊕(x3⊕x7)

and then compute all at once via
x ^ (x>>4).

```
x ^= (x>>16)
x ^= (x>>8)
x ^= (x>>4)
x ^= (x>>2)
x ^= (x>>1)
parity = (x & 1)
```

# ENDIANNESS



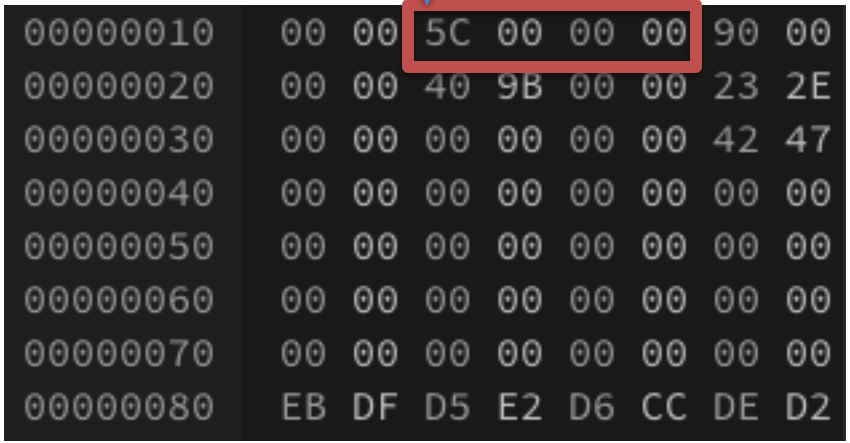
| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00000000 | 42 | 4D | BA | 9B | 00 | 00 | 00 | 00 | 00 | 00 | 7A | 00 | 00 | 00 | 6C | 00 |
| 00000010 | 00 | 00 | 5C | 00 | 00 | 00 | 90 | 00 | 00 | 00 | 01 | 00 | 18 | 00 | 00 | 00 |
| 00000020 | 00 | 00 | 40 | 9B | 00 | 00 | 23 | 2E | 00 | 00 | 23 | 2E | 00 | 00 | 00 | 00 |
| 00000030 | 00 | 00 | 00 | 00 | 00 | 00 | 42 | 47 | 52 | 73 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000040 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000050 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 |
| 00000060 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 02 | 00 | 00 | 00 | 00 | 00 |
| 00000070 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | 00 | E2 | D5 | C9 | E4 | D8 | CE |
| 00000080 | EB | DF | D5 | E2 | D6 | CC | DE | D2 | C8 | E7 | DB | D1 | E3 | D7 | CD | E7 |

What 32 –bit number is stored at location 0x12

0x5C000000 ($1543503872_{10}$) or 0x0000005C ($92_{10}$)

Answer: it depends

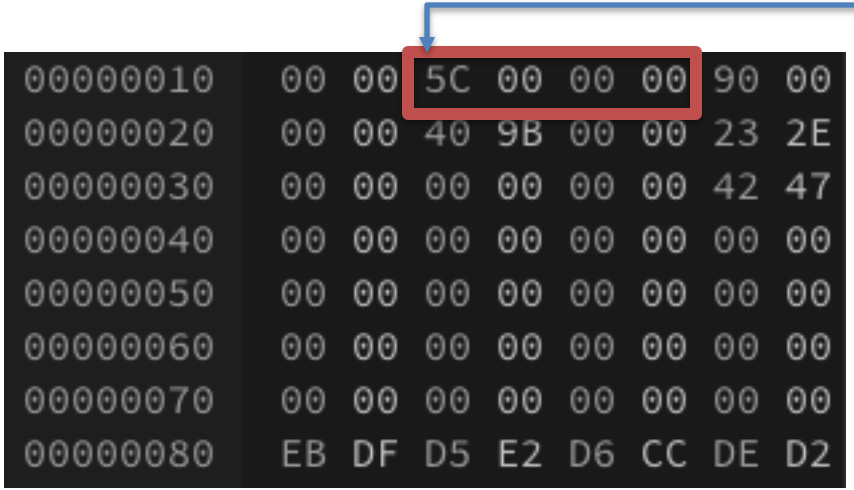
# ENDIANNESS




Little ENDIAN
0x0000005C ($92_{10}$)

Least significant byte at lowest address

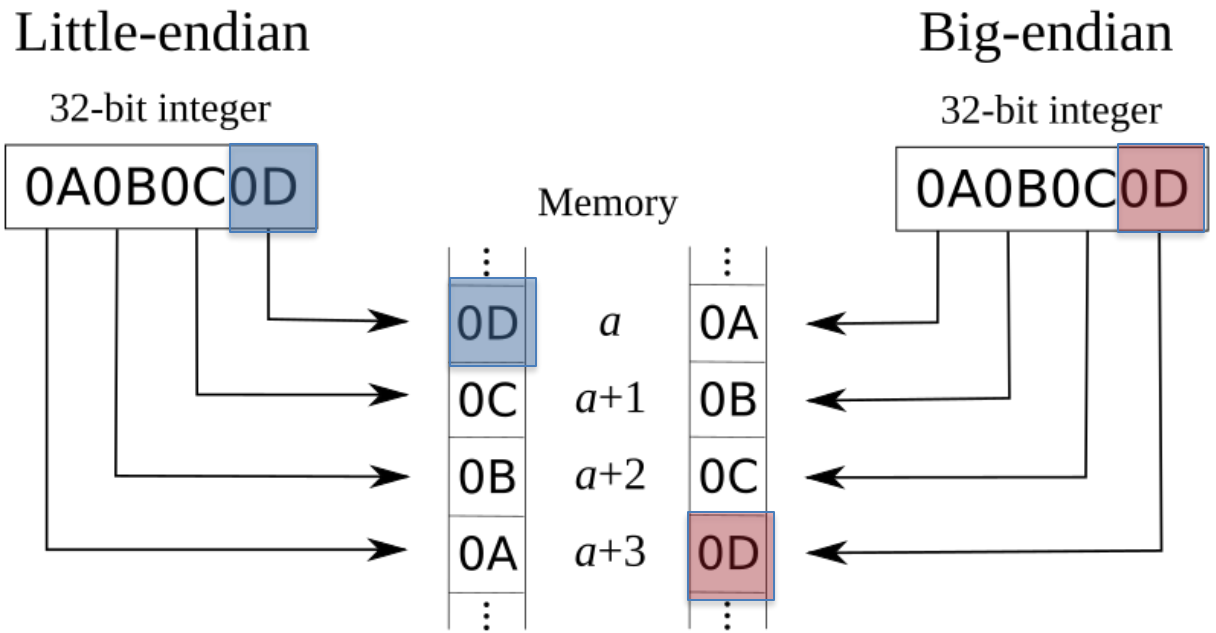
# ENDIANNESS



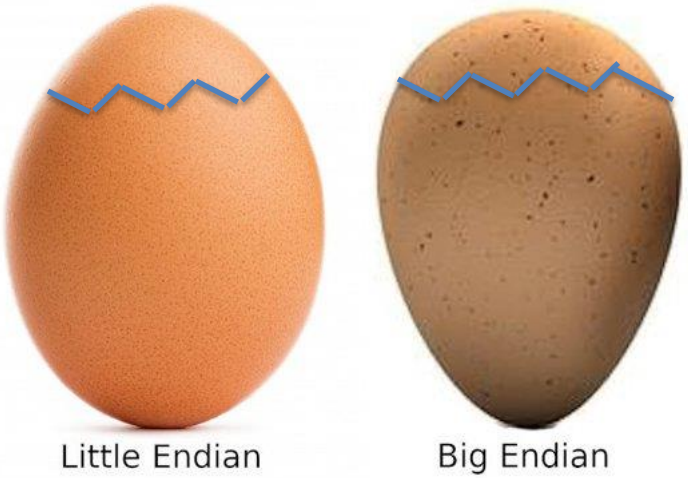

Big ENDIAN
0x5C000000 ($1543503872_{10}$)

Most significant Byte at lowest address

# ENDIANNESS



# WHICH END SHOULD YOU CRACK YOUR EGGS AT




A term borrowed from Gulliver's travels: The Big-Endians, who broke their boiled eggs at the big end, rebelled against the king, who demanded that his subjects break their eggs at the little end.

# FLOATING POINT

# FLOATING POINT

- How can we represent decimal values in binary?
- Why do errors like these occur?

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
>>> (0.1 + 0.1 + 0.1) == 0.3
False
>>> x = 0.1 + 0.1 + 0.1
>>> x
0.30000000000000004
```

Floating point rounding error →

```
>>> 0.3 + 0.3 + 0.3
>>> 0.8999999999999999
```

# FLOATING POINT

Base Ten fraction representation

0.125 has value 1/10 + 2/100 + 5/1000

Base 2 fraction representation

0.001 has value 0/2 + 0/4 + 1/8.

# FLOATING POINT

Some fractions can only be approximated when written in a base. For example, 1/3 can only be approximated with written in base 10

0.3 == 1.0/3.0. (False in python)

0.33333 == 1.0/3.0 (False in python)

0.3333333 == 1.0/3.0 (False in python)

0.3333333333333333 == 1.0/3.0 (True in Python) But not really, because no matter how many digits we write it will not be equal to 1/3 (Because this is a repeating fraction, like PI)

# FLOATING POINT

Similarly, 0.1 is an infinitely repeating fraction in base 2.

0.0001100110011001100110011001100110011001100110011…

So, we have a precision problem. How can we represent floating point numbers?

# FLOATING POINT

$7.4 * 10^3$

Floating point is scientific notation in base 2

Notice that all the values are the same the point just floats

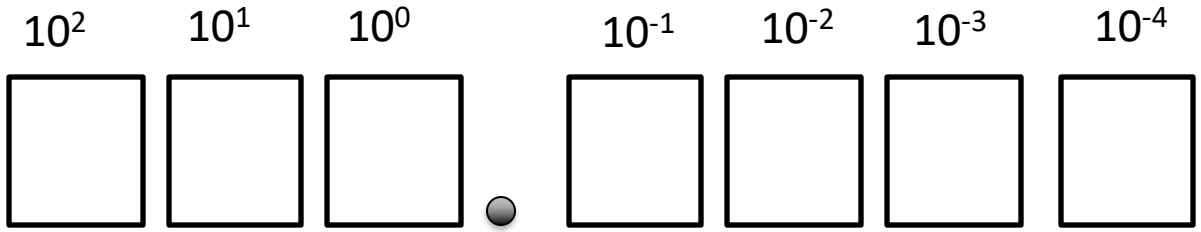
$74.0 * 10^2$

$740.0 * 10^1$

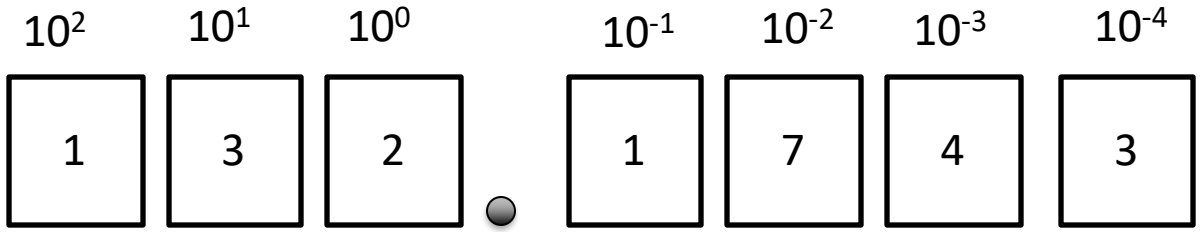
$7400.0 * 10^0$

```
>>> x = 1.7E308
>>> x
1.7e+308
>>> z = x + 0.1e308
>>> z
inf
>>>
```

# FLOATING POINT BASE 10

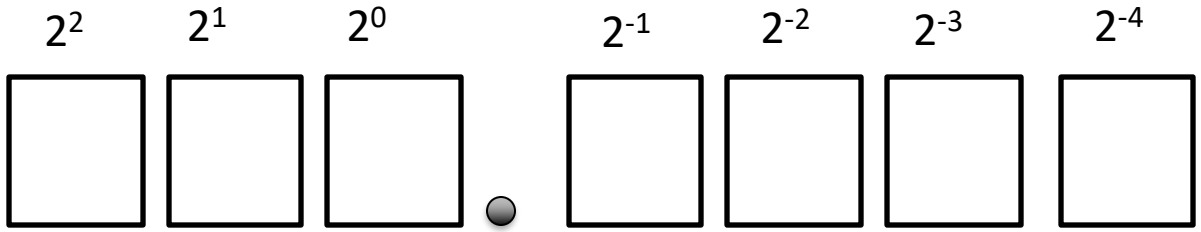


# FLOATING POINT BASE 10

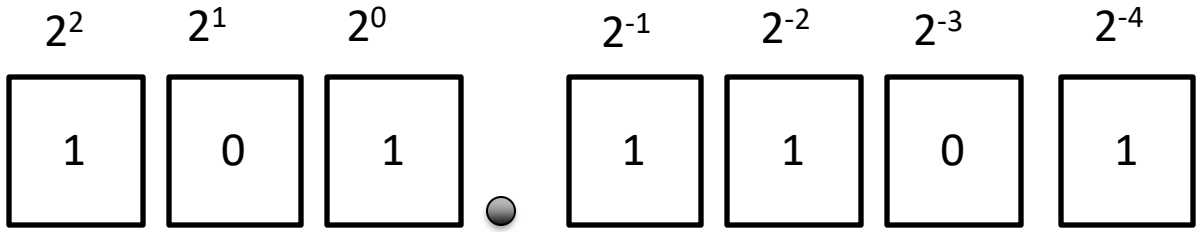



$1 \times 10^2 + 3 \times 10^1 + 2 \times 10^0 + 1 \times 1/10 + 7 \times 1/100 + 4 \times 1/1000 + 3 \times 1/10000 = 132.1743$

# FLOATING POINT BASE 2



# FLOATING POINT BASE 2




$1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5$

$1 \times 1/2 + 1 \times 1/4 + 0 \times 1/8 + 1 \times 1/16 = 13/16$

$5\ 13/16 = 5+0.8125 = 5.8125$

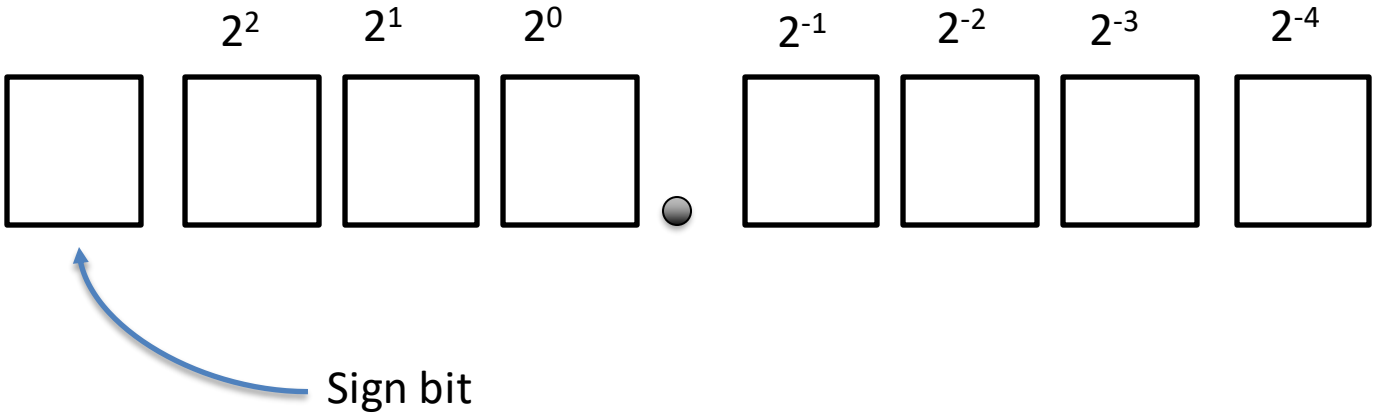
# NOW WE JUST NEED THE EXPONENT


Now we need the exponent so we can float the point.

$74.0 * 10^2$
$740.0 * 10^1$
$7400.0 * 10^1$

$-74.0 * 10^2$
$- 740.0 * 10^1$
$- 7400.0 * 10^1$

# FLOATING POINT BASE 2 (NEGATIVE NUMBERS)

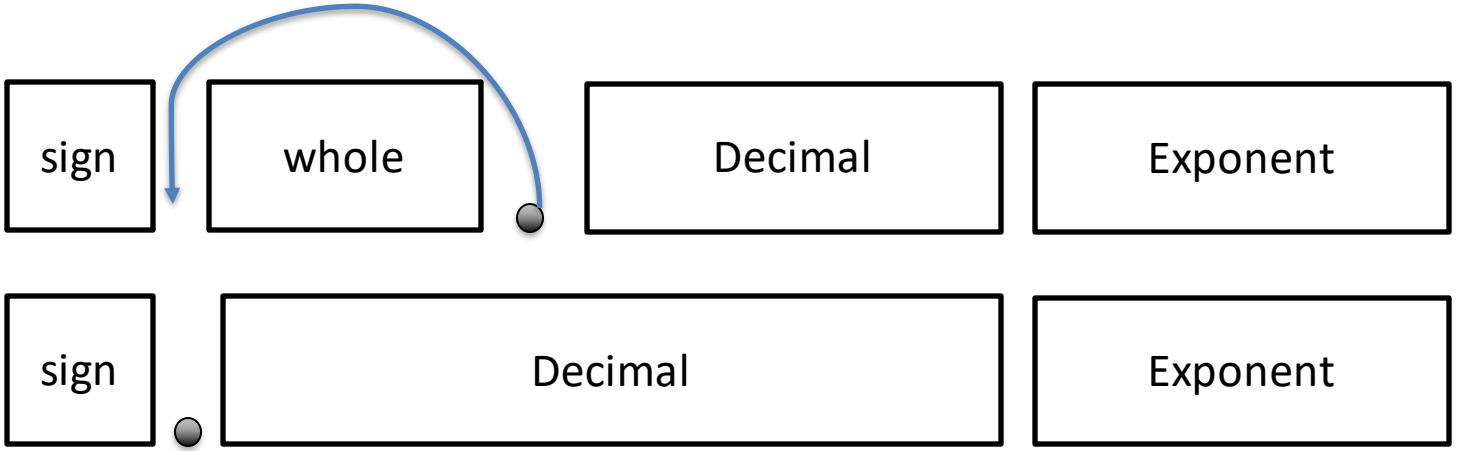


# NOW WE JUST NEED THE EXPONENT


number = sign( whole + decimal) x $2^{exponent}$

1 11.111 x $2^{E}$

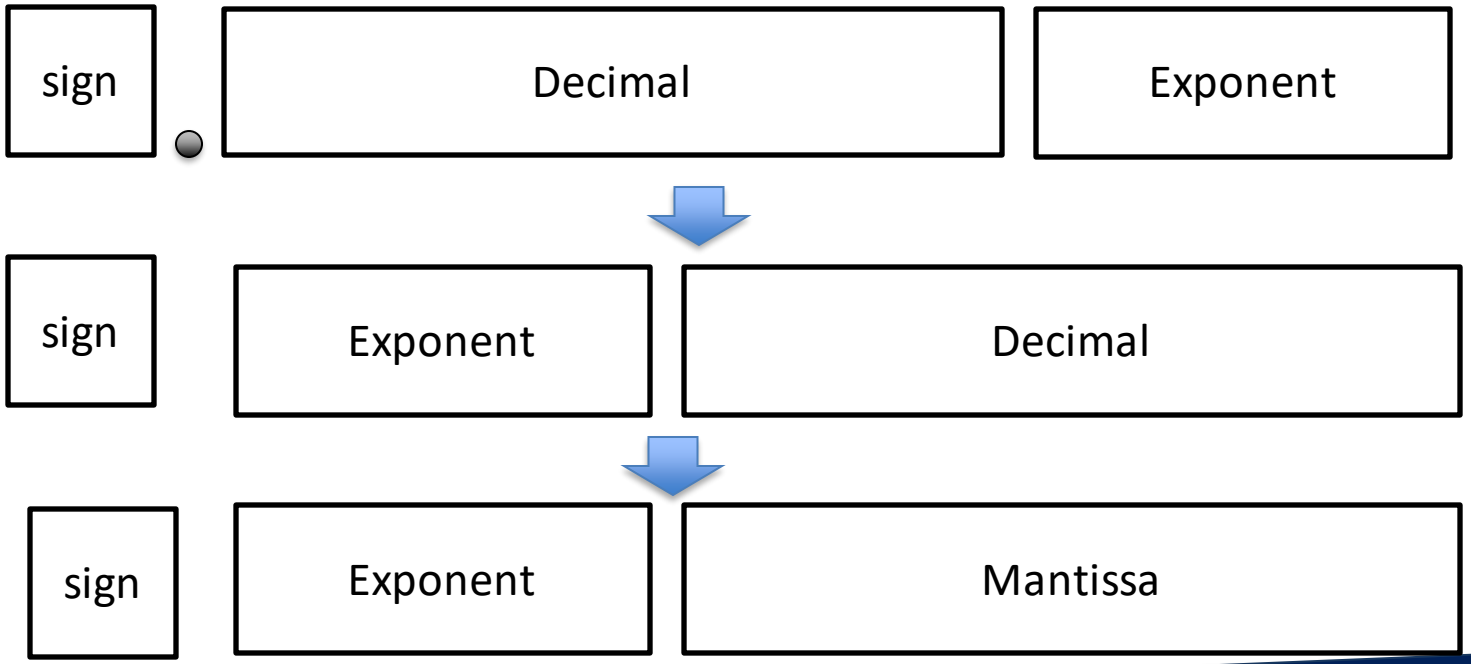
# ADDING THE EXPONENT DELETING THE WHOLE NUMBER SECTION



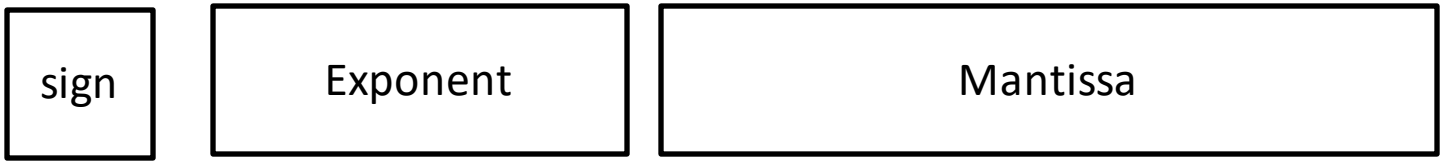

$-74.01 * 10^{2}$

$-0.7401 * 10^{4}$

# IEEE 754 FLOATING POINT STANDARD



# IEEE 754

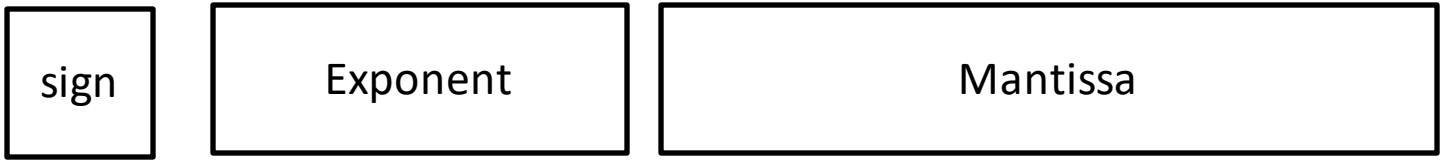



$$number = sign(1 + Mantissa) \times 2^{exponent + bias}$$

On 32-bit machines bias is normally 127 (Yes this is bias representation we talked about earlier)
0b01111111 (8 digits) is what we use when adding to the exponent

For 64-bit machines, the bias would be 1023, and this format allows for handling of negative exponents as the biased exponent is always stored as a positive number

# IEEE 754




$$number = sign(1 + Mantissa) \times 2^{exponent + bias}$$

Remember this is a base 2 binary string

$1.ffff \times 2^{exponent + bias}$

Remember $2^0$ is one

# BINARY STRING

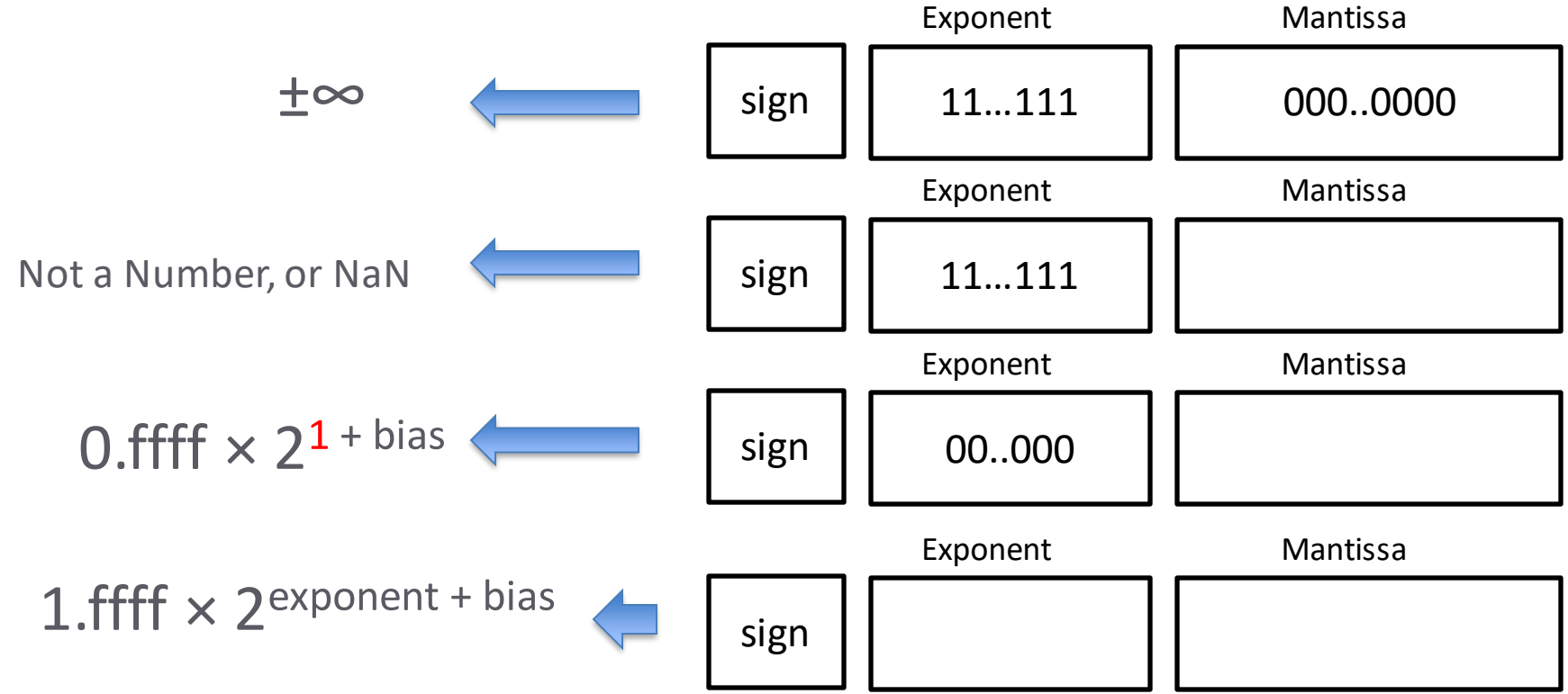
$0.1101 = 1.101 \times 2^{-1}$

$0.01101 = 1.101 \times 2^{-2}$

$0.001101 = 1.101 \times 2^{-3}$

Keep going until you get to your first 1. (one 1 left to the decimal point)

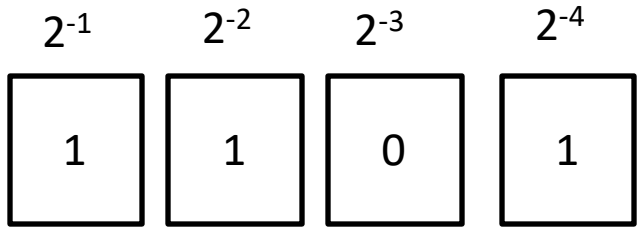
# IEEE 754 FOUR CASES



# CONVERSION EXAMPLE

Let's convert 0.8125 to floating-point representation

0.8125 x 2=1.6250 **1**
0.6250 x 2=1.2500 **1**
0.2500 x 2=0.5000 **0**
0.5000 x 2=1.0000 **1**




1 x 1/2 + 1x 1/4 + 0 x 1/8 + 1 x 1/16 = 13/16

0.1101
= 1.101 x $2^{-1}$

# CONVERSION EXAMPLE PART 2

$0.1101 = 1.101 \times 2^{-1}$

Sign: 0

Mantissa: 101

Exponent: -1 + 127 = 126(d)
= 01111110(b)

0 01111110 1010000 00000000 00000000

# CONVERSION PART 3

0.1 x 2 = 0.2 **0**
0.2 x 2 = 0.4 **0**
0.4 x 2 = 0.8 **0**
0.8 x 2 = 1.6 **1**
0.6 x 2 = 1. 2 **1**
0.1 x 2 = 0.2 **0**

**…… repeats …**

Just like the 1/3, 0.1 keeps repeating but in binary

**0 01111011 1001100 11001100 1100110**

123 ½ + $1/2^4$ + $1/2^5$ + $1/2^8$ + $1/2^9$ + $1/2^{12}$ + $1/2^{13}$ + $1/2^{16}$ + $1/2^{17}$ + $1/2^{20}$ + $1/2^{21}$

$(-1)^s \times (1 + m) \times 2^{\text{exponent + bias}}$

0.099999994039535522460937 5 = $(-1)^0 \times \left(1 + \frac{1258291}{2097152}\right) \times 2^{123 - 127}$

No quite 0.1

# REMEMBER TO ALWAYS BE CAREFUL WITH FLOATING POINT

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Note this is on a 64-bit machine. Even less precise on a 32-bit machine

Numbers are not always what they seem.

Some guidance

1. Don't test for equality with floating point numbers
2. Worry more about addition and subtraction than Multiplication and Division
3. Numeric Operations don't always return numbers

Ref
https://www.codeproject.com/Articles/29637/Five-Tips-for-Floating-Point-Programming