Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

4.1: Floating Point Numbers

CSU11022 – Introduction to Computing II

D.A.Patterson, J.L.Hennessy, “Computer Organisation and Design: ARM Edition”, Morgan-Kaufmann, 2016.

(Section 3.5: Floating Point, available in the Library, doesn’t have to be the ARM Edition)

Dr Jonathan Dukes / jdukes@tcd.ie
School of Computer Science and Statistics

```java
// some really small numbers and one large number
float [] vals = {
    3.7e-5f, 4.8e-5f, 1.7e-5f, 2.4e-5f,
    3.7e-5f, 4.8e-5f, 1.7e-5f, 2.4e-5f,
    3.7e-5f, 4.8e-5f, 1.7e-5f, 2.4e-5f,
    3.7e-5f, 4.8e-5f, 1.7e-5f, 2.4e-5f,
    12345.0f
  };

  float result;

  // add the numbers first-to-last
  result = 0;
  for (int i = 0; i < vals.length; i++) {
    result += vals[i];
  }

  System.out.println("sum first-to-last: " + String.format("%.8f",result));
  // output: sum first-to-last: 12345.00097656

  // add the numbers last-to-first
  result = 0;
  for (int i = vals.length - 1; i >= 0; i--) {
    result += vals[i];
  }

  System.out.println("sum last-to-first: " + String.format("%.8f",result));
  // output: sum first-to-last: 12345.00000000
```

Binary number representation

32-bits … 2^{32} unique values that we can use to represent different things

e.g. unsigned integers

0 ... $2^{32}-1$ (or 0 ... 4,294,967,295)

e.g. signed integers using 2's complement

-2^{31} ... 0 ... $+2^{31}-1$ (or −2,147,483,648 ... 0 ... +2,147,483,647)

How do we represent **real** numbers like 2½ or 3.14159265… ?

Also, how do we represent values with really large or really small magnitudes?

e.g. 2.2×10^{11}

e.g. 1.3×10^{-8}

The values 2.2×10^{11} and 1.3×10^{-8} are examples of (normalized) scientific notation in decimal form

$$f \times 10^{e}$$

Values expressed in normalised scientific notation satisfy the condition:

$$1 \leq |f| < 10$$

Normalized scientific notation give us one *canonical form* in which to express a value using scientific notation and allows quick, visual comparison of magnitude

$$372.98$$

$$37.298 \times 10^{1}$$

$$3729.8 \times 10^{-1}$$

$$3.7298 \times 10^{2}$$

As computer scientists, we avoid expressing the same thing in different ways (a==b?)

Convert the following binary numbers to decimal numbers with fractions

10010101 **= $1 \times 2^7 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0$ = 149**

1.1 **= $1 \times 2^0 + 1 \times 2^{-1}$ = 1½ = 1.5**

101000.01 **= $1 \times 2^5 + 1 \times 2^3 + 1 \times 2^{-2}$ = 40¼ = 40.25**

Convert the following decimal numbers to binary floating point numbers

5¼ **= 101.01**

7.75 **= 111.11**

2.1 **= $10.0001100110\underline{0110 \ldots}$**

9.3125 **= 1001.0101**

0.75 x 2 = **1**.5
0.5 x 2 = **1**.0

0.3125 x 2 = **0**.625
0.625 x 2 = **1**.25
0.25 x 2 = **0**.5
0.5 x 2 = **1**.0

Like decimal values, we can express binary values using scientific notation (again, in normalized form)

e.g.

$1010.1 = 1.0101 \times 2^3$

$0.00101 = 1.01 \times 2^{-3}$

The general form is again:

$$f \times 2^{-3}$$

and in normalised form, f satisfies the following condition:

$$1_2 \leq |f| < 10_2$$

$$\begin{aligned} 5.75_{10} &= 101.11_2 \times 2^0 \\ &= 1.0111_2 \times 2^2 \end{aligned}$$

The normalized form of a binary number expressed using scientific notation forms the basis for its representation in a computer

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

4.2: IEEE-754

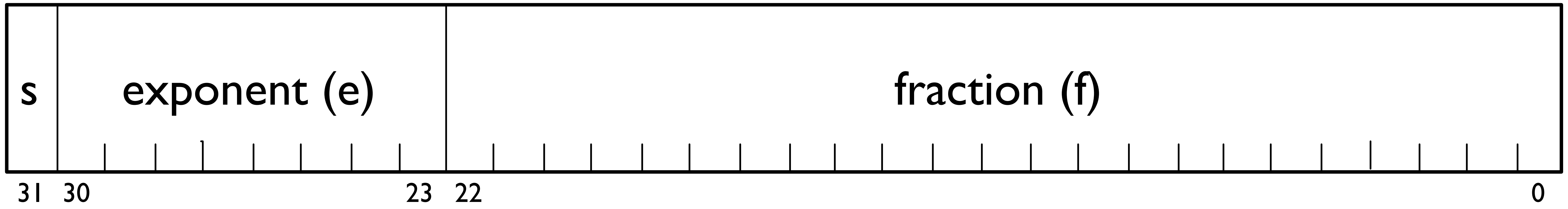
CSU11022 – Introduction to Computing II

Dr Jonathan Dukes / jdukes@tcd.ie
School of Computer Science and Statistics

https://www.h-schmidt.net/FloatConverter/IEEE754.html

IEEE 754 Floating-Point representation

Use a different interpretation of a 32-bit value to represent floating point numbers, e.g. IEEE 754



$$(-1)^s \times f \times 2^e$$

How can we represent …

… positive and negative values?

… values with positive and negative exponents?

Where is the binary (radix) point?

Sign of Exponent and Fraction

Sign bit?

0 ⇒ positive floating-point number

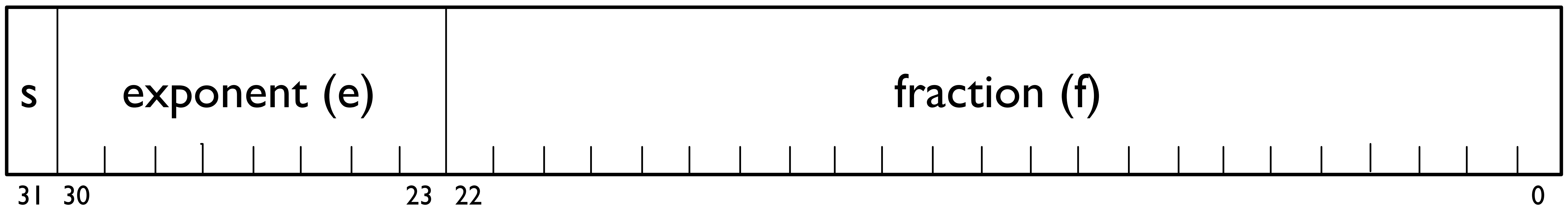
1 ⇒ negative floating-point number

Positive and negative exponents?

Option 1: 2’s Complement exponents

Option 2: Biased exponents

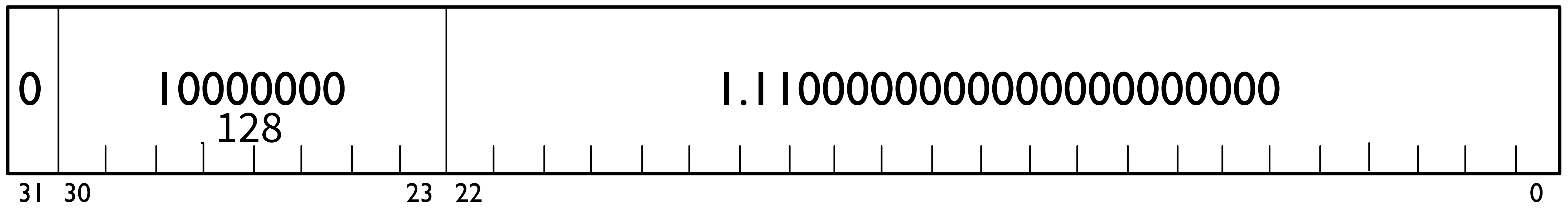
Subtract a constant bias (b = 127) from stored exponent to obtain signed exponent



$$(-1)^s \times f \times 2^{e-b}$$

Storing the fraction

The following two representations are of the same value (3.5_{10})



$$+1.11 \times 2^{(128 - 127)} = 11.1_2 = 3.5_{10}$$

$$+0.111 \times 2^{(129 - 127)} = 11.1_2 = 3.5_{10} \text{ (same value!)}$$

We don't want multiple representations of the same value!

if (a == b) ...

Storing floating-point numbers in normalized form avoids this problem:

$$1_2 \leq |f| < 10_2, \text{ so f is in the form } 1\,.\,ddddd\ldots$$

Normalization and the *Hidden Bit*

With normalisation

$$\mathbf{0.0101 \times 2^{-4}}$$

... becomes ...

$$\mathbf{1.0100 \times 2^{-6}}$$

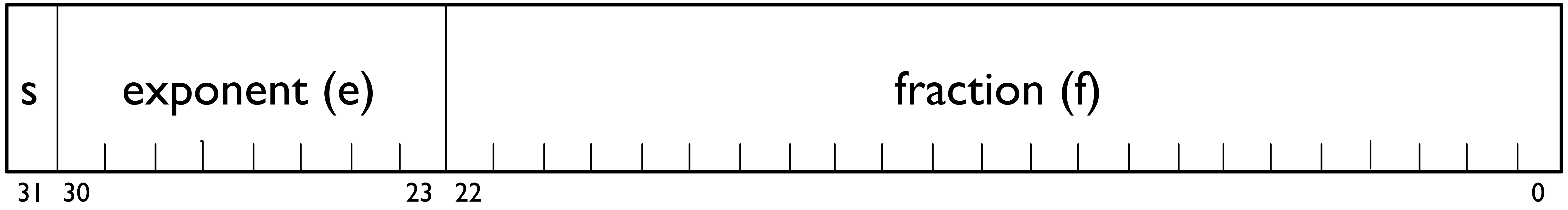
adjust fraction so there is a single 1 to left of radix point

compensate by adjusting exponent accordingly

If there is always going to be a 1 to the left of the radix point, we don’t need to store it!

Increases precision by one bit!

Final IEEE 754 Floating-Point Representation



$$(-1)^s \times 1.f \times 2^{(e-b)}$$

https://www.h-schmidt.net/FloatConverter/IEEE754.html

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

4.3: IEEE-754 Examples

CSU11022 – Introduction to Computing II

Dr Jonathan Dukes / jdukes@tcd.ie
School of Computer Science and Statistics

https://www.h-schmidt.net/FloatConverter/IEEE754.html

1.25

$1.25 = 1.01_2$ (already normalised)

$s = 0$

$e = 0 + 127 = 127 = 01111111_2$

$f = 1.01_2$ or $.01_2$ after removing the hidden bit

s	e	f
0	01111111	01000000000000000000000

0011	1111	1010	0000	0000	0000	0000	0000
3	F	A	0	0	0	0	0

10.75

$10.75 = 1010.11_2 \times 2^0 = 1.01011_2 \times 2^3$

$s = 0$

$e = 3 + 127 = 130 = 10000010_2$

$f = 1.01011_2$ or $.01011_2$ after removing the hidden bit

s	e	f
0	10000010	01011000000000000000000

0100 0001 0010 1100 0000 0000 0000 0000

4 1 2 C 0 0 0 0

-0.125

$-0.125 = -0.001_2 \times 2^0 = 1.0_2 \times 2^{-3}$

$s = 1$

$e = -3 + 127 = 124 = 01111100_2$

$f = 1.0_2$ or $.0_2$ after removing the hidden bit

s	e	f
1	01111100	00000000000000000000000

```
1011 1110 0000 0000 0000 0000 0000 0000
 B    E    0    0    0    0    0    0
```

decode 0x414a0000


```
s        e                      f
0   10000010   10010100000000000000000
```

s = 0 (positive)

e = 130 ($2^{130-127} = 2^3$)

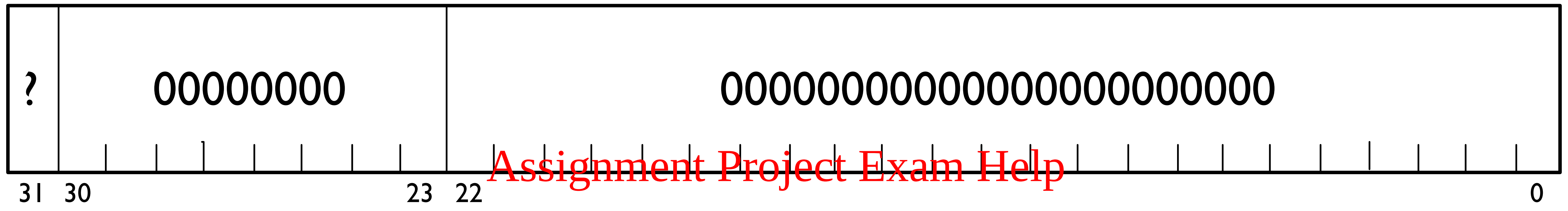
f = **1.**100101 (after adding the **hidden bit**)

+1.100101 x 2^3 = +1100.101 = **+12.625**

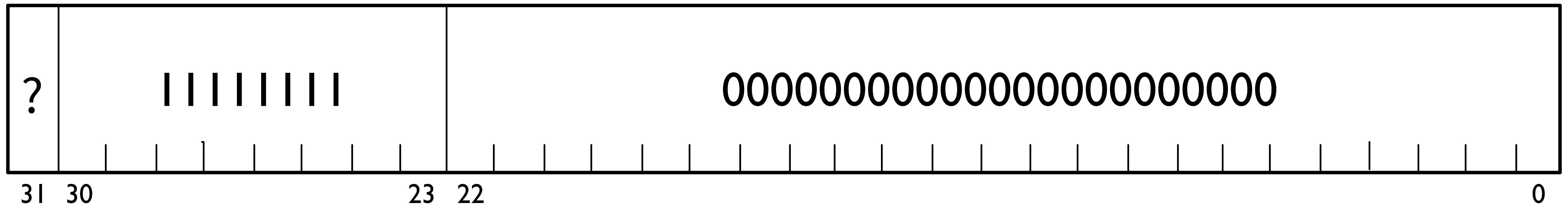
Special Values

Special bit patterns, e.g.

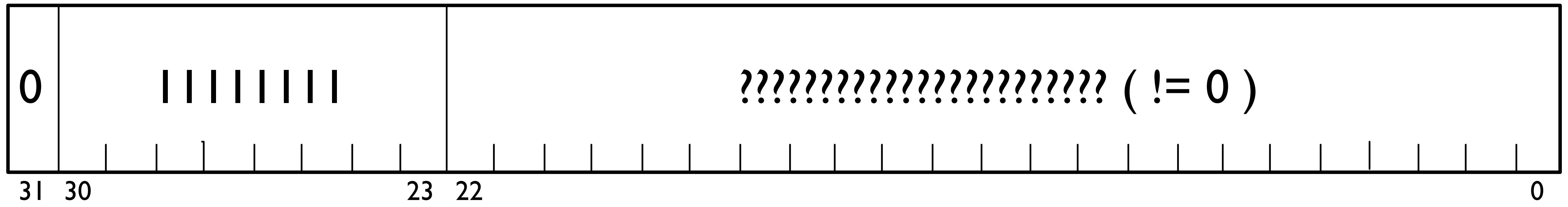
Zero (±)



Infinity (±)

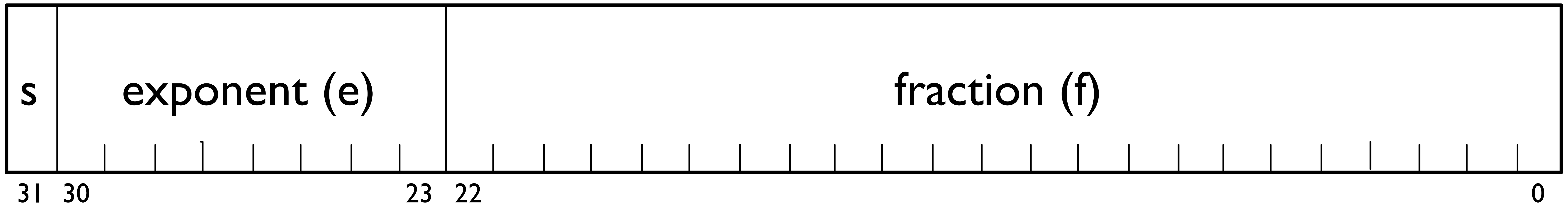


Not a Number (NaN)

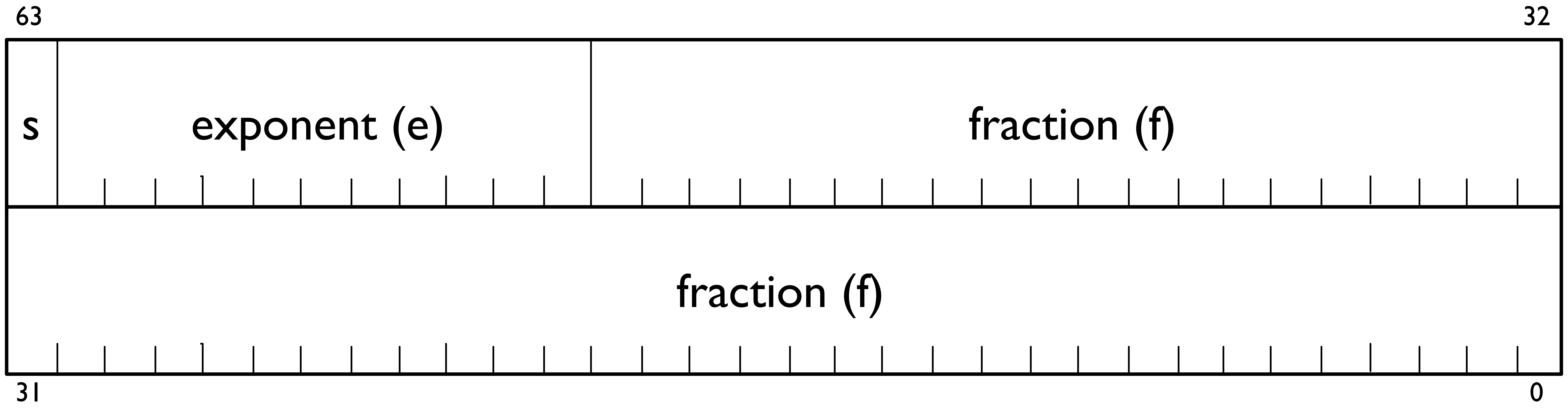


Single and Double Precision

32-Bit Single Precision (bias = 127)



64-Bit Double Precision (bias = 1023)



Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

4.4: Floating Point addition

CSU11022 – Introduction to Computing II

Dr Jonathan Dukes / jdukes@tcd.ie
School of Computer Science and Statistics

https://www.h-schmidt.net/FloatConverter/IEEE754.html

Floating Point Addition

We can add the fractions of two floating point values if their exponents are the same

If their exponents are not the same to begin, shift the fraction of the value with the smaller exponent to compensate

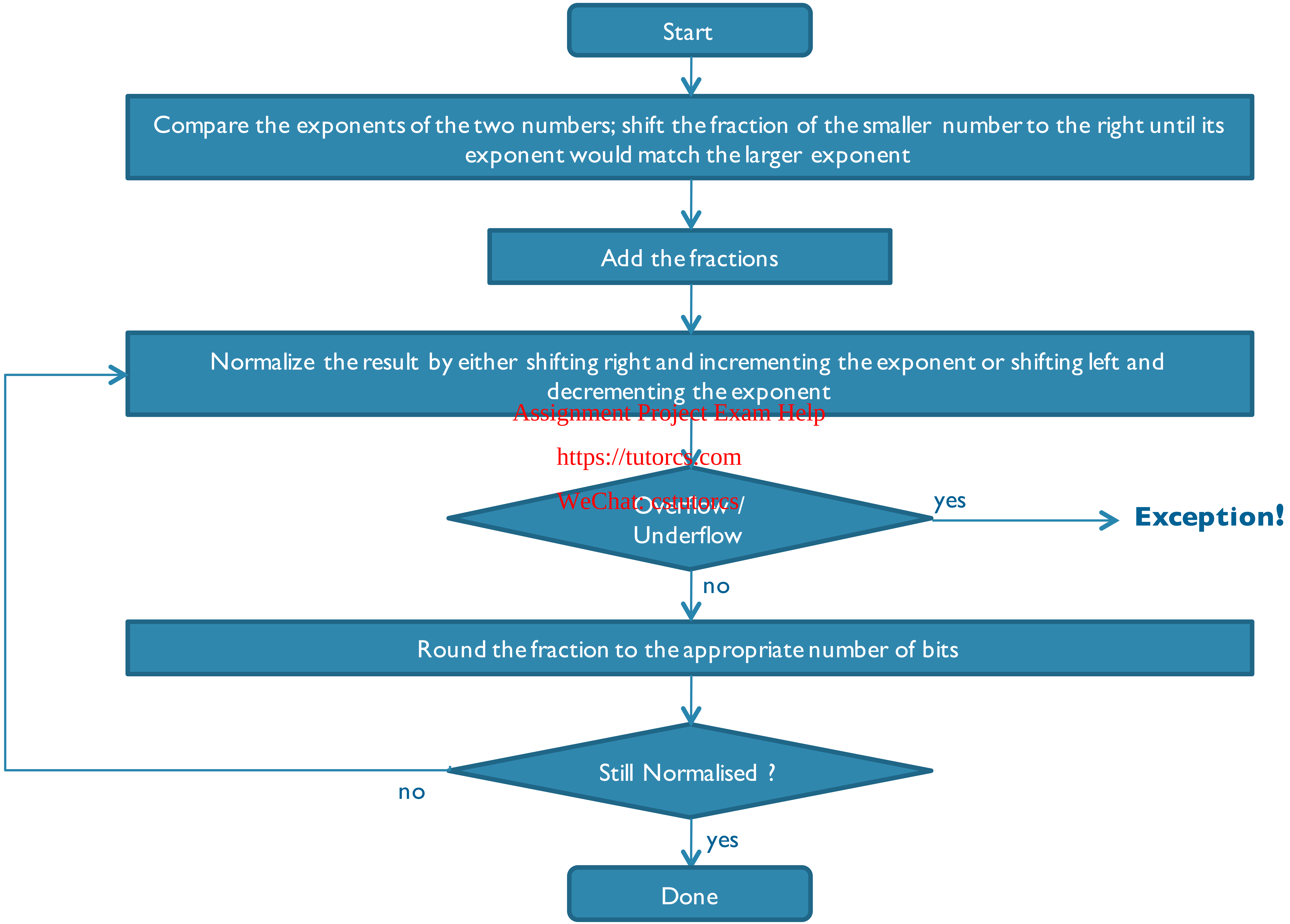
e.g.

$$1.01101 \times 2^{3} + 1.00110 \times 2^{-2}$$

$$= 1.01101 \times 2^{3} + 0.0000100110 \times 2^{3}$$

$$= 1.0111000110 \times 2^{3}$$

```
  1.0110100000
+ 0.0000100110
  ------------
  1.0111000110
```


Start
Compare the exponents of the two numbers; shift the fraction of the smaller number to the right until its exponent would match the larger exponent
Add the fractions
Normalize the result by either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow / Underflow
yes
Exception!
no
Round the fraction to the appropriate number of bits
Still Normalised ?
no
yes
Done

1.5 + 1.75 = 3.25


```
A = 0x3fc00000 (1.5)
0 01111111 10000000000000000000000
s    e              f

s 0 (+)
e 01111111 (127-127=0, 2^0)
f 1.10000 (remember hidden bit!)
```

```
B = 0x3fe00000 (1.75)
0 01111111 11000000000000000000000
s    e              f

s 0 (+)
e 01111111 (127-127=0, 2^0)
f 1.110000 (remember hidden bit!)
```



```
 1.100000 x 2^0 A
 1.110000 x 2^0 B
-----------------
11.010000 x 2^0 Result (not normalised)
1.1010000 x 2^1 Result (normalised)
0 10000000 10100000000000000000000 (encoding s e f)
0x40500000 (3.25)
```

1.5 + 10.5 = 12.0


```
A = 0x3fc00000 (1.5)
0 01111111 10000000000000000000000
s    e                f

s 0 (+)
e 01111111 (127-127=0, 2^0)
f 1.10000 (remember hidden bit!)
```

```
B = 0x41280000 (10.5)
0 10000010 01010000000000000000000
s    e                f

s 0 (+)
e 10000010 (130-127=3, 2^3)
f 1.0101000 (remember hidden bit!)
```



```
0.0011000 x 2^3 A (adjust fraction so exponents are equal)
1.0101000 x 2^3 B
1.1000000 x 2^3 Result (already normalised)
0 10000010 10000000000000000000000 (encoding s e f)
0x41400000 (12.0)
```

Floating Point Addition

What about adding negative values (S==1)?

Proceed as before but before adding, the fractions of values with S==1 should be converted to their 2's Compliment