Number Systems

- **Standard positional representation of numbers:**

  An unsigned number with whole and fraction portions is represented as:

$$a_n a_{n-1} a_{n-2} a_{n-3} \ldots a_1 a_0 . a_{-1} a_{-1} a_{-2} a_{-3} \ldots$$

The value of this number is given by:

$$N = a_n \times b^n + a_{n-1} \times b^{n-1} + \ldots a_0 \times b^0 + a_{-1} \times b^{-1} + \ldots$$

- **Where "b" is the base of the number system (e.g 2, 8, 10, 16) and "a" is a digit that range from 0 to b-1**
- **The "radix point" is used to separate the whole number from the fractional portion. In base 10, this is the decimal point; in base 2, the binary point.**

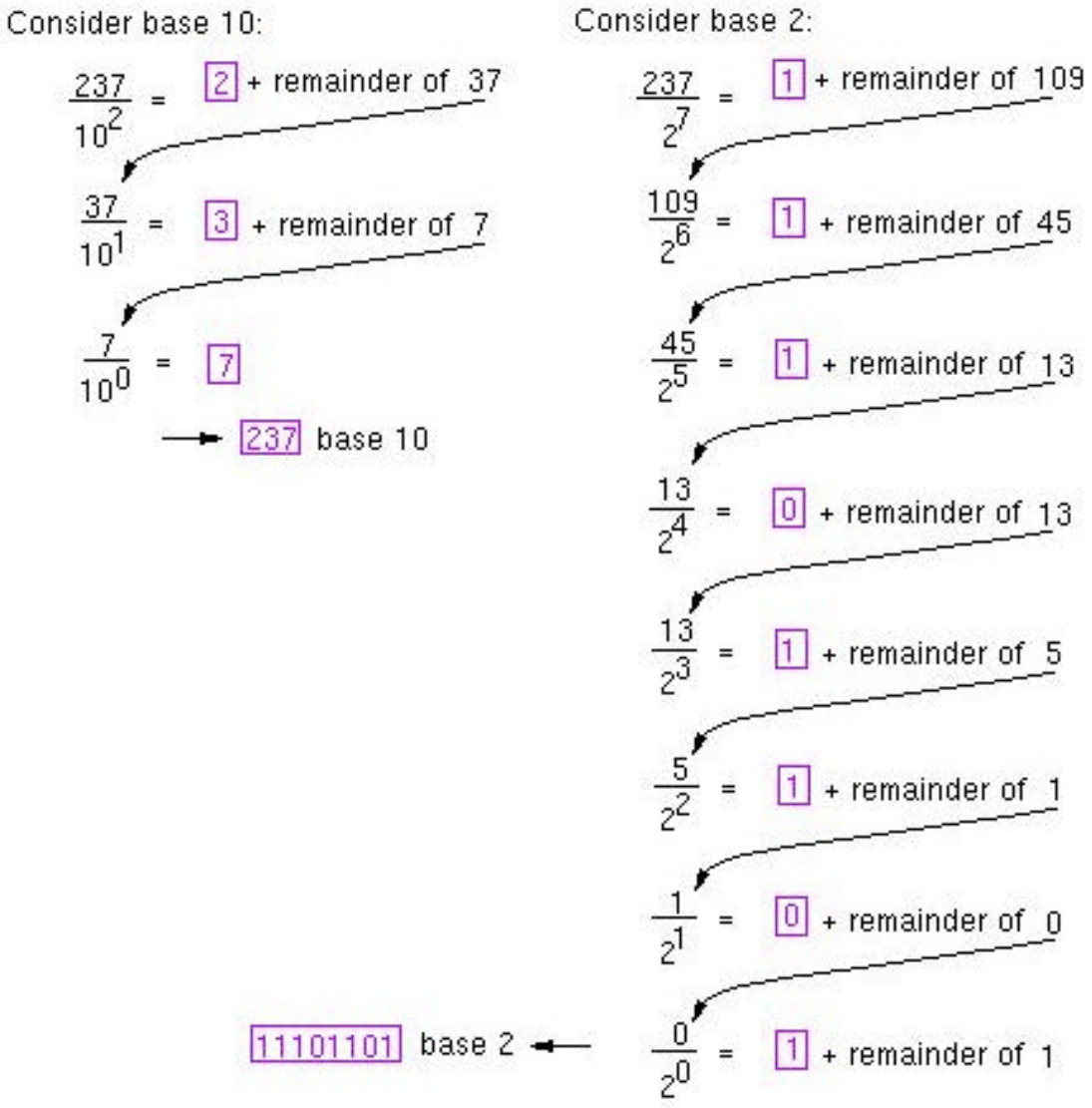
Number Systems Used in Computers

Name of Base	Base	Set of Digits	Example
Decimal	b=10	a= {0,1,2,3,4,5,6,7,8,9}	255_{10}
Binary	b=2	a= {0,1}	$\%11111111_{2}$
Octal	b=10	a= {0,1,2,3,4,5,6,7}	377_{8}
Hexadecimal	b=16	a= {0,1,2,3,4,5,6,7,8,9,A, B, C, D, E, F}	$\$FF_{16}$

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Hex	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Binary	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111

Converting from Decimal to Binary

An Example:



Algorithm for Converting from Decimal to Any Base

- **Separate the number into its whole number (wholeNum) and fractional (fracNum) portions.**
- **Convert the whole number portion by repeating the following until "wholeNum" is 0:**

```
digit = wholeNum % base;
wholeNum /= base;
```

The "digit" resulting from each step is the next digit under the new base starting with position 0 to the left of the decimal point.

- **Convert the fractional portion by repeating the following until "fracNum" is 0 or sufficient precision has been reached:**

```
fracNum *= base;
digit = (int)base;
fracNum -= base;
```

The "digit" resulting from each step is the next digit under the new base starting with position 0 to the right of the decimal point.

- **The final answer is the whole number and fractional computation result separated by a radix point.**

Algorithm for Converting from Any Base to Decimal

- **Separate the number into its whole number (wholeNum) and fractional (fracNum) portions.**
- **Convert the whole number portion by starting at the left end advancing to the right where "wholeNum" is initially set to 0:**

  wholeNum = wholeNum * base + digit

  – **Example: 3107_{16} to decimal**

  0 +3 = 0 + 3 = 3

  (3*16) + 1 = 48 + 1 = 49

  (49*16) + 0 = 784 + 0 = 748

  (748*16) + 7 = 12544 +7 = 12551_{10}
- **Convert the fractional portion by starting at the right end and advancing to the left where "fracNum" is initially set to 0:**

  fracNum = (fracNum + digit) / base;
- **The final answer is the sum of the whole number and fractional parts.**

Algorithm for Converting Between Arbitrary Bases

- **If the old base is base 10, then use the approach already defined above for converting from base 10 to the new base.**
- **If the new base is base 10, then use the approach already defined for converting from any base to base 10.**
- **If neither the old or new base is base 10, then:**
  - **Convert from the current base to base 10.**
  - **Convert from base 10 to the new base.**

Binary to Hexadecimal Conversion

- **Separate the whole binary number portion into groups of 4 beginning at the decimal point and working to the left. Add leading zeroes as necessary.**
- **Separate the fraction binary number portion into groups of 4 beginning at the decimal (actually binary) point and working to the right. Add trailing zeroes as necessary.**
- **Convert each group of 4 to the equivalent hexadecimal digit.**
- **Example:**

$$3564.875_{10} = 1101\ 1110\ 1100.1110_2$$
$$= (D * 16^2) + (E*16^1) + (C*16^0)+(E*16^{-1})$$
$$= DEC.E_{16}$$

Binary to Octal Conversion

- **Separate the whole binary number portion into groups of 3 beginning at the decimal point and working to the left. Add leading zeroes as necessary.**
- **Separate the fraction binary number portion into groups of 3 beginning at the decimal (actually binary) point and working to the right. Add trailing zeroes as necessary.**
- **Convert each group of 3 to the equivalent octal digit.**
- **Example:**

$$3564.875_{10} = 110\ 111\ 101\ 100.111_2$$
$$= (6 * 8^3) + (7*8^2) + (5*8^1)+(4*8^0)+(7*8^{-1})$$
$$= 6754.7_8$$

Binary Coded Decimal (BCD)

- **Binary Coded Decimal (BCD) is a way to store decimal numbers in binary. This technique uses 4 bits to store each digit from from 0 to 9. For example:**

$$98_{10} = 1001\ 1000 \quad \text{in BCD}$$

- **BCD wastes storage since 4 bits are used to store 10 combinations rather than the maximum possible 16.**
- **Arithmetic is more complex to implement in BCD.**
- **BCD is frequently used in calculators.**
- **Processors may have special instructions to aid BCD calculations.**

Signed Binary Numbers

- **Sign and Magnitude Representation:**
  - **For an *n-bit* binary number:**

    Use the first bit (most significant bit, MSB) position to represent the sign where 0 is positive and 1 is negative.

  - **Remaining n-1 bits represent the magnitude which may range from:**

    $$-2^{(n-1)} + 1 \quad \text{to} \quad 2^{(n-1)} - 1$$

  - **This scheme has two representations for 0; i.e., both positive and negative 0: for 8 bits: 00000000, 10000000**
  - **Arithmetic under this scheme uses the sign bit to indicate the nature of the operation and the sign of the result, but the sign bit is not used as part of the arithmetic.**

Signed Binary Numbers

Complementary Representation:

- **No sign digit or bit used**
- **The complement (negative representation) of an n-digit number is defined as:**

$$\text{base}^n - \text{number}$$

- **To obtain *two's Complement* representation of an n-bit binary number:**

$$2^n - \text{number} = 2^n - 1 + \text{number} + 1$$
$$= (1111\ldots111) + \text{number} + 1$$

or

- **Invert each bit**
- **Add 1 to the result**

Properties of Two's Complement Numbers

- **X plus the complement of X equals 0.**
- **There is one unique 0.**
- **Positive numbers have 0 as their leading bit (MSB); while negatives have 1 as their MSB.**
- **The range for an *n-bit* binary number in 2's complement representation is:**

  from $-2^{(n-1)}$ to $2^{(n-1)} - 1$

- **The complement of the complement of a number is the original number.**
- **Subtraction is done by *addition* to the complement of the number.**

Examples of Two's Complement Addition

For n = 8 bits

Decimal		Binary		Hex
11	=	00001011	=	000B
+ 21	=	+ 00010101	=	+ 0015
32	=	00100000	=	0020

Decimal		Binary		Hex
21	=	00010101	=	0015
- 11	=	+ 11110101	=	+ FFF5
10	=	00001010	=	000A

Decimal		Binary		Hex
11	=	00001011	=	000B
- 21	=	+ 11101011		+ FFEB
- 10	=	11110110	=	FFF6

Decimal		Binary		Hex
- 11	=	11110101	=	FFF5
- 21	=	+ 11101011	=	+ FFEB
- 32	=	1 11100000	=	FFE0

Two's Complement Arithmetic Overflow

- **Arithmetic overflow occurs with two's complement number addition if:**
  - **The operands have the same sign (either positive or negative).**
  - **The answer has a different sign.**
- **The overflow (V) may be represented algebraically by the following equation:**

$$V = a_{n-1}\, b_{n-1}\, \overline{s_{n-1}} + \overline{a_{n-1}}\, \overline{b_{n-1}}\, s_{n-1}$$

- **Consider 5-bit two's complement, the valid range of numbers is -16 to +15. Consider the following overflow situations:**

+12	01100	-12	10100
+13	01101	-13	10011
------	--------	-----	---------
	11001 = -7		00111 = +7
	Overflow		Overflow

What Values Can Be Represented in N Bits?

- **Unsigned:** 0 to $2^N - 1$
- **2s Complement:** -2^{N-1} to $2^{N-1} - 1$
- **1s Complement:** $-2^{N-1} + 1$ to $2^{N-1} - 1$
- **BCD** 0 to $10^{N/4} - 1$
- **For 32 bits:**

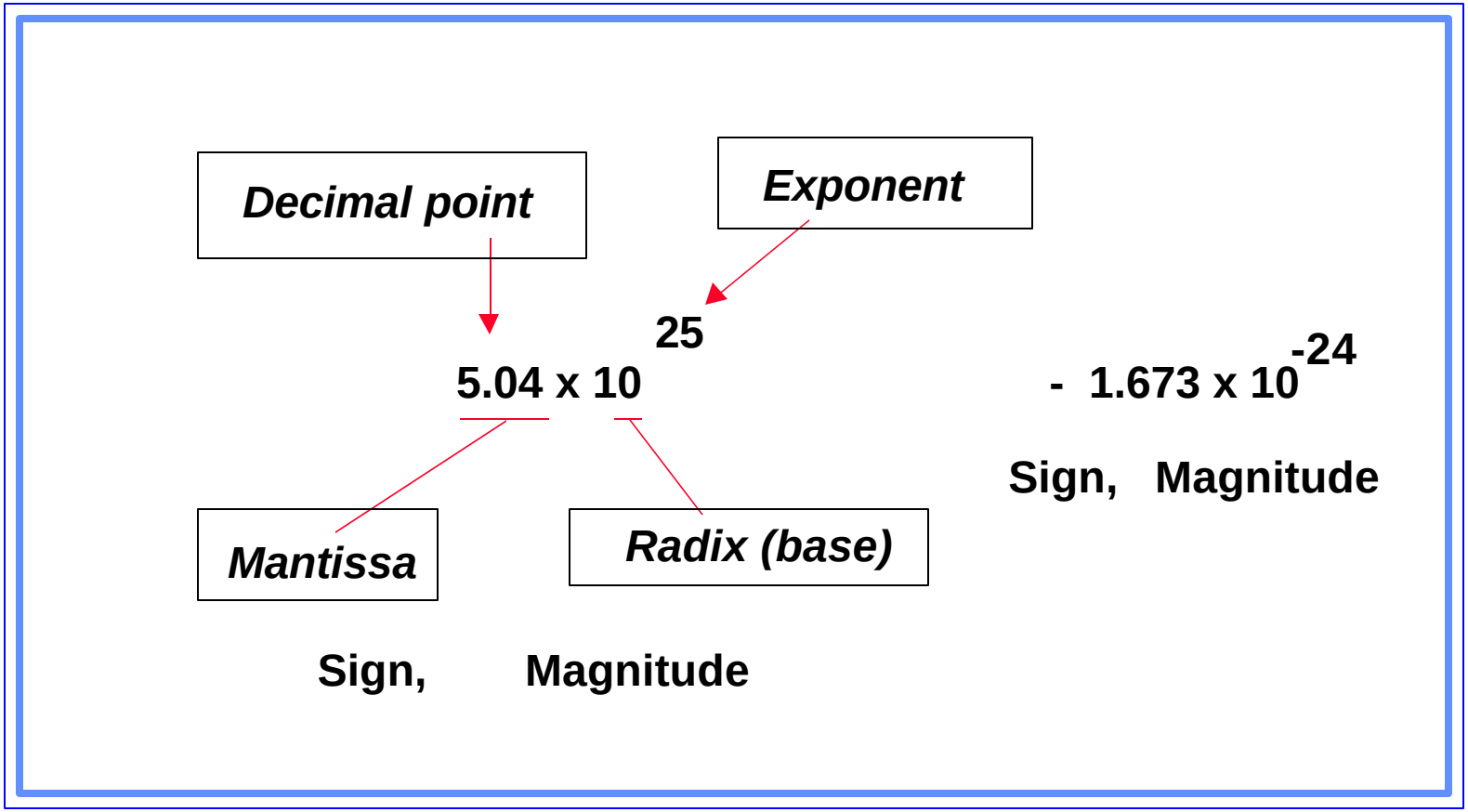
Unsigned:	**0**	**to**	**4,294,967,295**
2s Complement	**- 2,147,483,648**	**to**	**2,147,483,647**
2s Complement	**- 2,147,483,647**	**to**	**2,147,483,647**
BCD	**0**	**to**	**99,999,999**

- ***But, what about?***
  - **Very large numbers?**

    9,369,396,989,487,762,367,254,859,087,678
  - **. . . or very small number?**

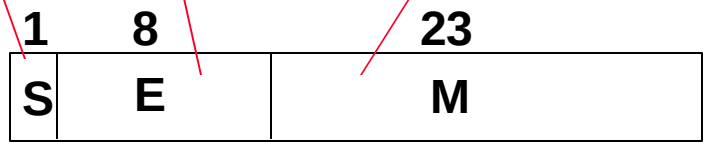
    0.00000000000000000000000318579157

Scientific Notation



Representation of Floating Point Numbers in Single Precision *IEEE 754 Standard*

$$\text{Value} = N = (-1)^S \times 2^{E-127} \times (1.M)$$



$0 < E < 255$
Actual exponent is:
$e = E - 127$

exponent:
excess 127 binary integer added

mantissa:
sign + magnitude, normalized binary significand with a hidden integer bit: 1.M

Example: 0 = 0 00000000 0 . . . 0 -1.5 = 1 01111111 10 . . . 0

Magnitude of numbers that can be represented is in the range:

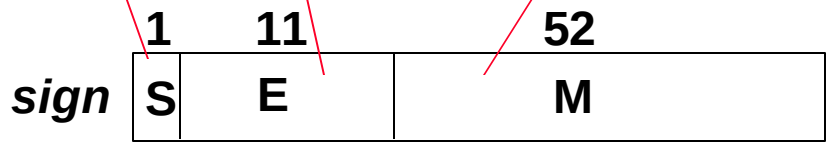
$2^{-126}(1.0)$ to $2^{127}(2 - 2^{-23})$

Which is approximately:

1.8×10^{-38} to 3.40×10^{38}

Representation of Floating Point Numbers in Double Precision *IEEE 754 Standard*

$$\text{Value} = N = (-1)^S \times 2^{E-1023} \times (1.M)$$



$0 < E < 2047$
Actual exponent is:
$e = E - 1023$

exponent:
excess 1023 binary integer added

mantissa:
sign + magnitude, normalized binary significand with a hidden integer bit: 1.M

Example: 0 = 0 00000000000 0 . . . 0 -1.5 = 1 01111111111 10 . . . 0

Magnitude of numbers that can be represented is in the range:

$$2^{-1022}(1.0) \quad \text{to} \quad 2^{1023}(2 - 2^{-52})$$

Which is approximately:

$$2.23 \times 10^{-308} \quad \text{to} \quad 1.8 \times 10^{308}$$

IEEE 754 Special Number Representation

Single Precision		Double Precision		Number Represented
Exponent	Significand	Exponent	Significand	
0	0	0	0	0
0	nonzero	0	nonzero	Denormalized number
1 to 254	anything	1 to 2046	anything	Floating Point Number
255	0	2047	0	Infinity
255	nonzero	2047	nonzero	NaN (Not A Number)

Basic Floating Point Addition Algorithm

Addition (or subtraction) involves the following steps:

(1) Compute exponent difference: Ye - Xe

(2) Align binary point: right shift Xm that many positions to form Xm 2^{Xe-Ye}

(3) Compute sum of aligned *mantissas*: Xm2 $^{Xe-Ye}$ + Ym

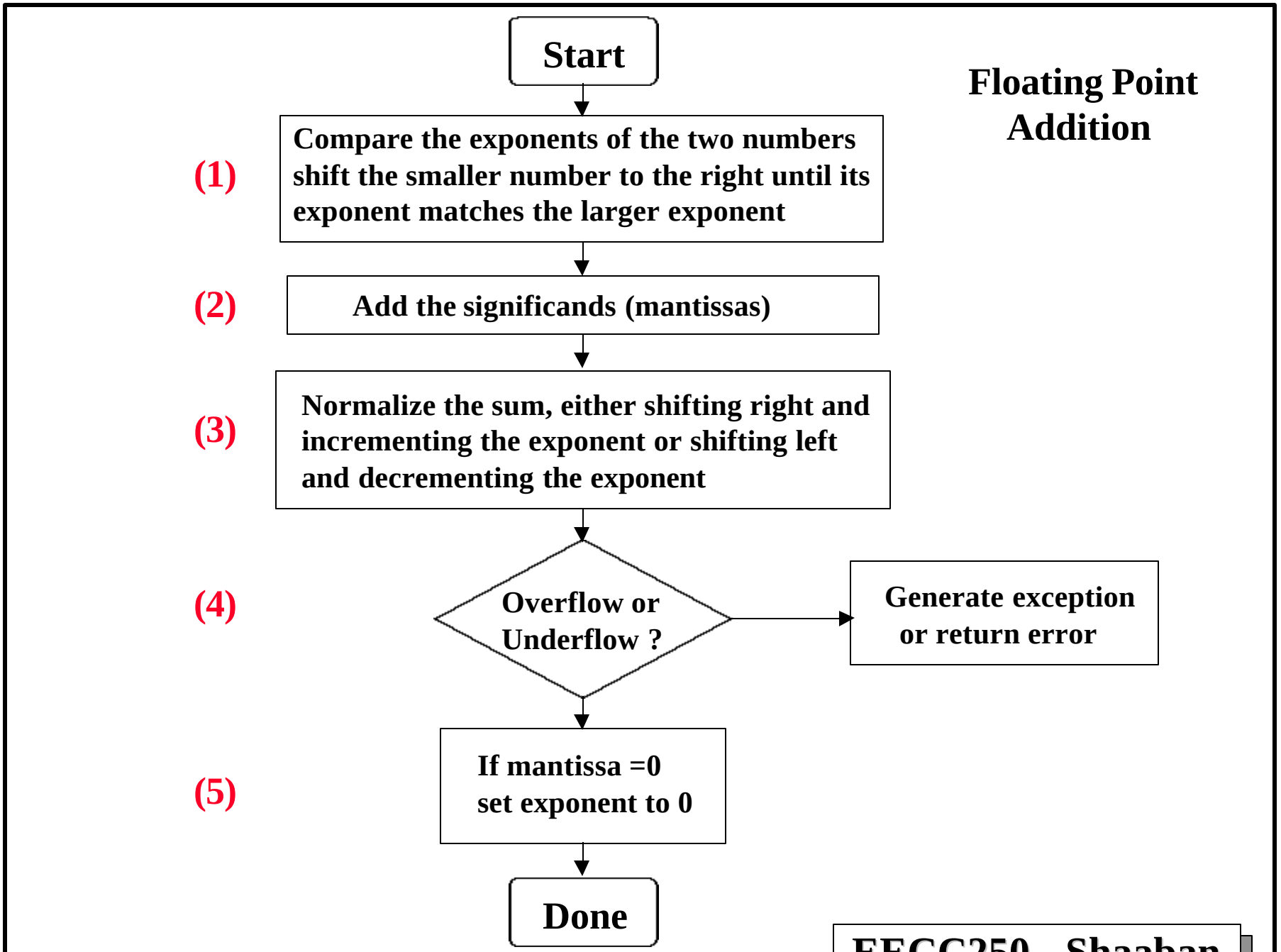
(4) If normalization of result is needed, then a normalization step follows:

- **Left shift result, decrement result exponent (e.g., 0.001xx…) or**
- **Right shift result, increment result exponent (e.g., 101.1xx…)**

Continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(5) If result mantissa is 0, may need to set the exponent to zero by a special step.

Floating Point Addition



Logical Operations on Binary Values: AND, OR

AND, *, $\wedge$

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise AND operation of two numbers

Example: A AND B

```
A   11011010   AND
B    01001101
 =   01001000
```

OR , +, $\vee$

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR operation of two numbers

Example: A OR B

```
A   11011010   OR
B    01001101
 =   11011111
```

Logical Operations on Binary Values: XOR, NOT

Exclusive OR, EOR, XOR, ⊗

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise XOR operation of two numbers

Example: A XOR B

A 1101101 0 XOR
B 01001101
= 10010111

NOT, ~, $\overline{X}$

Inverts or complements a bit

Example:

A = 1101100

$\overline{A}$ = 0010011