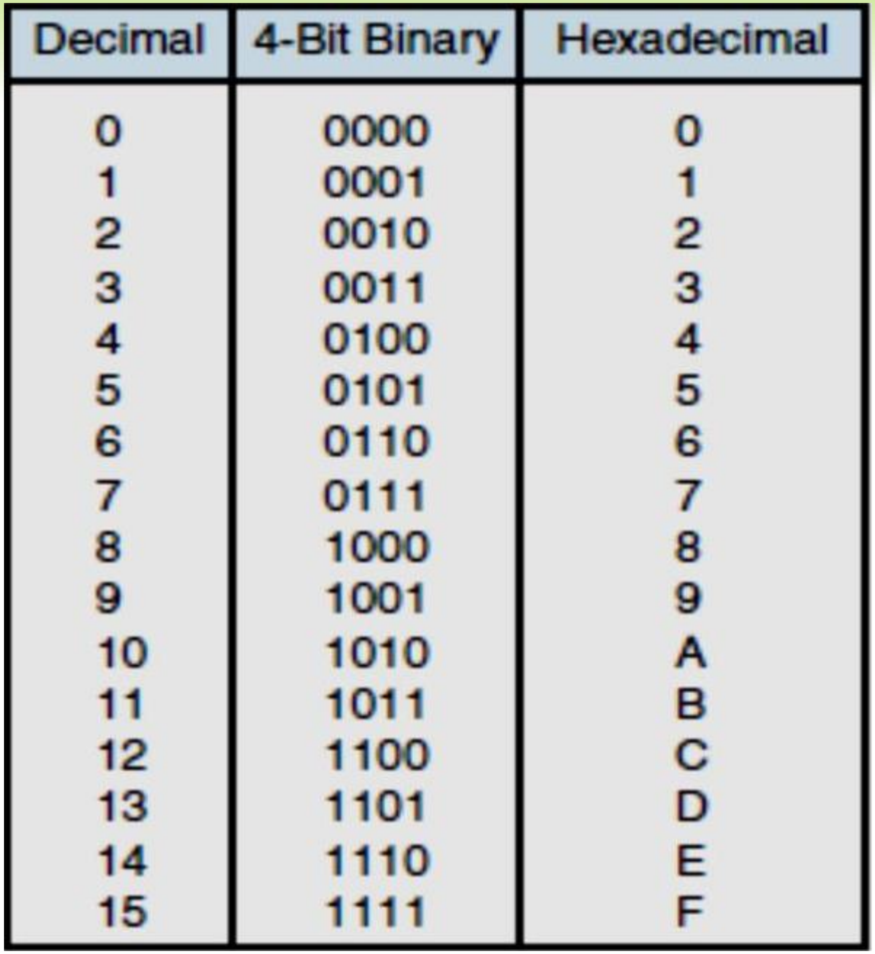
System Hardware

Tutorial 4 -Data Representation Part-2 Floating-Point

Objectives

- Understand the fundamental concepts of floating-point representation.
- Learn to apply the required steps to generate the intermediate forms of floating-point notations. Namely, Fixed-point and Blackboard notations.
- Addition & Multiplication of floating point numbers.

Converting Between Bases
Values of Number Bases



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

Floating-Point Representation

- Data representations that we have seen so far dealt with unsigned and signed integer values only.
- As is, these formats are not useful in scientific or business applications that deal with real number values.
- Floating-point representation solves this problem.

Floating-Point Representation

- Clever programmers can perform floating-point calculations using any integer format. ***Discuss How.***
- This is called ***floating-point emulation***, because floating point values aren’t stored as such; we can create programs that make it seem that floating-point values are being used.
- Most of today’s computers are equipped with specialized hardware that performs floating-point arithmetic with no special programming required.

Floating-Point Representation

- Decimal floating-point numbers allow an arbitrary number of decimal places to the right of the decimal point.
  - For example: $0.5 \times 0.25 = 0.125$
- They are often expressed in scientific notation.
  - For example:

    $0.125 = 1.25 \times 10^{-1}$

    $5{,}000{,}000 = 5.0 \times 10^{6}$

Floating-Point Representation Introduction

- Computers cannot actually represent real numbers.
- Computers can represent a finite subset of rational numbers.
- A rational number is expressed as a quotient or fraction p/q of 2 integers. q may be equal to 1 and so every integer is a rational number.
- Numbers represented inside computers are ***binary rationals.***
- The ***denominator of a binary rational is a power of 2*** (e.g. 1/2,1/4,1/8, 1/32, etc.) .
- Binary rationals are a reasonable estimation to real numbers where more bits give a better approximation.

Floating-Point Representation Introduction

- Mathematically, real numbers can be viewed as a convergent sequence of rationals.
- Example **5.75 = 4 + 1 + ½ + ¼ = $2^2 + 2^0 + 2^{-1} + 2^{-2}$**
- Schemes that use a limited number of bits to represent binary rationals as approximation to fractional numbers introduce a trade-off between ***precision*** and ***range***.
- **Get Precision** when the binary rationals are close together. More bits to represent the fractional part.
- **Get Range** when the difference between the smallest and largest binary rational is large. More bits for the whole part

Floating-Point Representation Introduction

- A number **n** has a finite binary expansion iff **n** is a binary rational.
- A reduced binary rational fraction is a number of the form **m/2^{n}**, where m < 2^{n} and ‘**2**’ is not a factor of '**m**'.
- The binary expansion of such a number has exactly '**n**' binary places to the right of the binary point.
- This curious fact is helpful in conversion exercises.
- 3/16 = 1/8 + 1/16 = .0011 in binary
- 7/32 = 1/8 + 1/16 + 1/32 = .00111 in binary

Floating-Point Representation

Converting from decimal real numbers to floating point notation involves 3 steps. The result of each step is considered as an intermediate form of floating-point representation:

- **Step 1 - Fixed-point Numbers**
- **Step 2 – Blackboard Notation**
- **Step 3 – Floating Point Representation**

Floating-Point Representation Fixed-Point Numbers

- A fixed-point number consists of an ***integral part*** and a ***fractional part*** separated by a ***radix point***.
- The value of a fixed-point number that has ***m integer*** digits and ***n fractional*** digits is:

x $= \sum_{i=-n}^{m-1} x_i * 2^i$ **=** $x_{m-1}x_{m-2}\ldots.\ x_0\ .\ x_{-1}x_{-2}\ \ldots.\ x_{-n}$

<u>Example</u> m=2, n=3

10.011 = $(1*2^1) + (0*2^0) + (0*2^{-1}) + (1*2^{-2}) + (1*2^{-3})$

- Note that the digits to the right of the radix point are given negative indices and are negative powers of 2.

Floating-Point Representation Fixed-Point Numbers

- In an (m+n)-digit binary fixed-point number system with **m** whole digits, numbers from 0 to $2^m - 2^{-n}$, in increments of 2^{-n}, can be represented.

- We call 2^{-n} the ***step size***, or ***resolution***. In this case, any two adjacent (m+n)-bit fixed-point binary rationals have a fixed distance 2^{-n} between them.

- Take m=3, n=4 as an example:

Floating-Point Representation
Fixed-Point Numbers

- In an (m+n)-digit binary fixed-point number system with **m** whole digits, numbers from 0 to $2^m - 2^{-n}$, in increments of 2^{-n}, can be represented.

- Take m=3, n=4 as an example: (increments are 2^{-4})

000.0000 ---- smallest fixed point number
000.0001
000.0010
000.0011
…
011.1111
111.1111 ---- largest fixed point number

Floating-Point Representation
Powers of 2 Table

When converting decimal integers to binary, we have used the powers of 2. To convert decimal real numbers, we will also use the negative powers of 2.

Power of 2	Decimal value	Powers of 2	Decimal value
2^0	**1**	2^{-1}	0.5
2^1	**2**	2^{-2}	0.250
2^2	**4**	2^{-3}	0.125
2^3	**8**	2^{-4}	0.0625
2^4	**16**	2^{-5}	0.03125
2^5	**32**	2^{-6}	0.015625
2^6	**64**	2^{-7}	0.0078125

Floating-Point Representation
Fixed-Point Numbers

Example

In a (2+3)-bit binary fixed-point number system, using the ***subtraction-remainder*** method (***convert the whole part and the fractional part separately***), the decimals

$2.375 = (1*2^{1}) + (0*2^{0}) + (0*2^{-1}) + (1*2^{-2}) + (1*2^{-3})=10.011$

$3.875 = (1*2^{1}) + (1*2^{0}) + (1*2-^{1}) + (1*2^{-2}) + (1*2^{-3}) + (0*2^{-4})=11.111$

In this number system, the values:

$(0 = 00.000)$ through $(2^{2} - 2^{-3} = 4 – 0.125 = 3.875 = 11.111)$

are represented in steps of $2^{-3} = 0.125$.

Note: When the total number of bits (**m+n**) used in the representation is fixed, there is a trade-off.

- A larger **m** leads to an enlarged ***range*** of numbers.
- A larger **n** leads to increased ***precision*** in specifying numbers, which leads to better approximations to real numbers.

Floating-Point Representation Fixed-Point Numbers

Another Example

Convert decimal 2.9 to (3+5)-binary fixed point

$2.9 =(1*2^{1}) + (1*2^{0}) + (1* 2^{-1}) + (1 *2^{-2}) + (1*2^{-3}) + (0*2^{-4}) + (1*2^{-5}) = 10.11101 =2.90625$

The above is a closer estimation than:

$2.9 =(1*2^{1}) + (1*2^{0}) + (1* 2^{-1}) + (1 *2^{-2}) + (1*2^{-3}) + (0*2^{-4}) + (0*2^{-5}) = 10.11100 = 2.875$

Floating-Point Representation Fixed-Point Numbers

Another Conversion Method

Integer 2 is handled separately: (010.?????).

Now we deal with fractional part:

.9 * 2 = .8 1

.8 * 2 = .6 1

.6 * 2 = .2 1

.2 * 2 = .4 0

.4 * 2 = .8 0

.8 * 2 = .6 1

Recipe: take the fractional part, multiply by 2, record the integer part as one digit of your answer, and repeat until the fractional part is 0, or you run out of binary digits.

Floating-Point Representation Fixed-Point Numbers

In the Previous Example (converting 2.9)

- Just taking the first five fractional digits gives (010.11100) = 2.875.
- But we can do something with the sixth digit: since it is 1, we can add 1 to the current approximation. This gives (010.11101) = 2.90625, which is closer to 2.9 than is (010.11100).
- This last refinement is called ***rounding***. For simplification, we ***will not use rounding*** even if it often improves accuracy.

Floating-Point Representation Fixed-Point Numbers

In the previous example, if we continue the process, we will observe a repeating pattern:

.9 * 2 = .8 1

.8 * 2 = .6 1

.6 * 2 = .2 1

.2 * 2 = .4 0

.4 * 2 = .8 0

.8 * 2 = .6 1

.6 * 2 = .2 1

.2 * 2 = .4 0

.4 * 2 = .8 0

2.9 can be represented as 010.11101

Floating-Point Representation Fixed-Point Numbers

Example

Convert (1101.0101) to decimal.

$(1 * 2^{0}) + (0 * 2^{1}) + (1 * 2^{2}) + (1 * 2^{3}) + (0 * 2^{-1}) + (1 * 2^{-2}) + (0 * 2-^{3}) + (1 * 2-^{4})$

= 13 + 0.25 + 0.0625 = 13.3125

Or 13 + 1/4 + 1/16 = 13 and 5/16

Floating-Point Representation Fixed-Point Numbers

Encoding Signed Fixed-Point Numbers

There is no special representation for signed fixed point numbers. Do conversion without the sign and just write the sign explicitly.

Example: -5.75 in (3+5)-bit binary fixed point.

- 5.75 = (101.11000)
- - 5.75 = - (101.11000)

Floating-Point Representation Blackboard Notation

- **Recall**: In scientific notation, we write a single digit to the left of the decimal point, e.g., 3.26 * 10^{5}.
- We adopt the same convention for ***binary rationals.***
- **Example**: 0.3125 = 0.0101 = $1.01*2^{-2}$
- Every floating-point number is a scaled version of a (1+f)-bit binary fixed-point number, where **'f'** is the number of bits set aside for the fractional part.

That is, we write a positive binary rational with a single nonzero digit to the left of the radix point, and continue with the fractional part (all this times some a positive or negative power of two).

Floating-Point Representation Blackboard Notation

Example: Convert (1101.1010) to blackboard floating point notation.

1101.1010 $= 1.1011010 * 2^3$ ***Normalized Notation***

= 1.1011010 [3]

Example: Convert (0.000111) to blackboard floating point.

0.000111 $= 1.11 * 2^{-4}$ = 1.11 [-4]

Floating-Point Representation Blackboard Notation

In a previous example converting decimal 0.9, if we continue the process, we will observe a repeating pattern:

.9 * 2 = .8 1

.8 * 2 = .6 1

.6 * 2 = .2 1

.2 * 2 = .4 0

.4 * 2 = .8 0

.8 * 2 = .6 1

.6 * 2 = .2 1

.2 * 2 = .4 0

.4 * 2 = .8 0

0.9 can be represented as 0.1110011001100 or 0.1<1100>*

In **Blackboard Notation**: 0.9 = 1.<1100>* [-1]

Similarly, 0.2 = 0.001100110011 = (0.<0011>*) = (1.<1001>*) [-3]

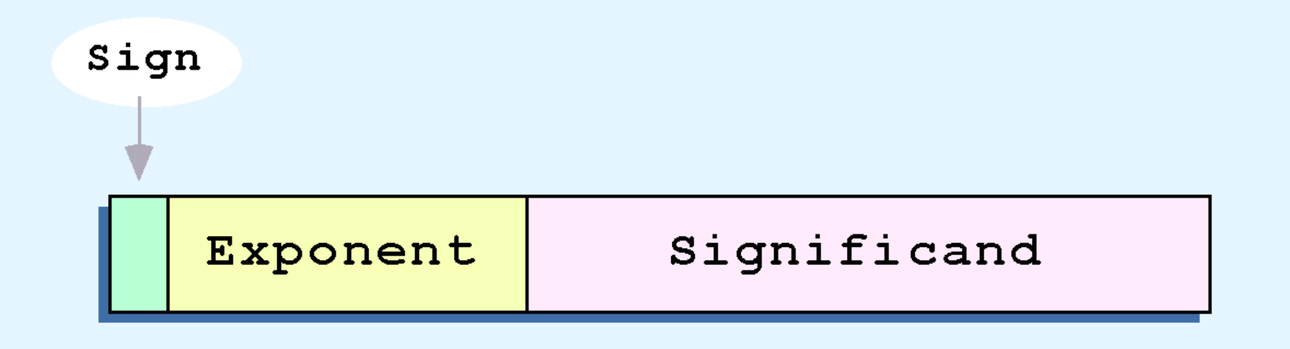
Floating-Point Representation From Blackboard To Floating-Point

In the general case, a floating point representation includes 3 parts in the order: **sign-bit (0 for positive & 1 for negative number), signed exponent, fractional part**.

Consider a 16-bit register. One bit is used to represent the sign, leaving us with 15 bits. Use ***4 bits (one hex digit) to represent the signed exponent*** in two's complement semantics. That leaves only 11 bits to store the ***fractional part*** of the significand.

Note: The above floating point configuration is arbitrary and may change from one example to another.

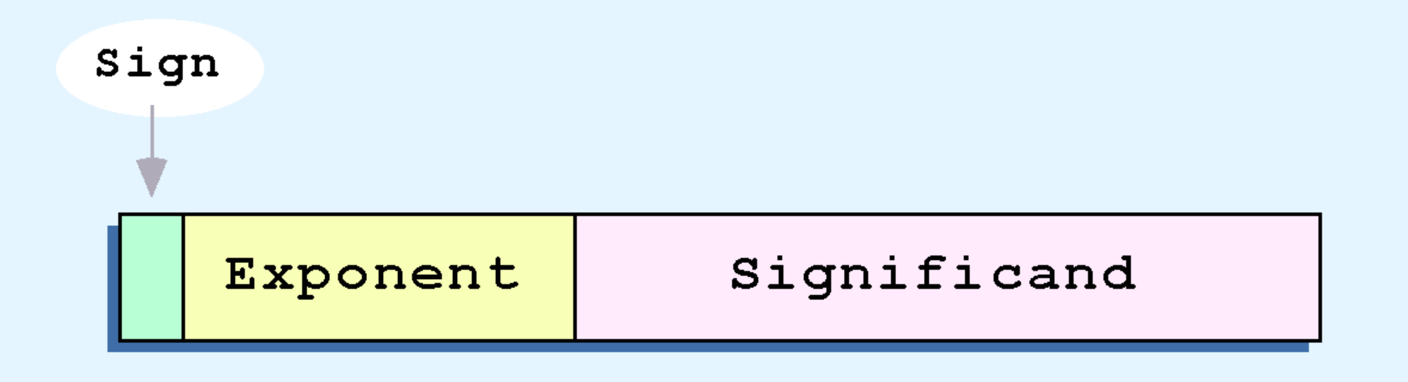
Floating-Point Representation



- This is a hypothetical "Simple Model" to explain the concepts
- In this model:
  - A floating-point number is 16 bits in length
  - The exponent field is 4 bits
  - The significand field is 11 bits. Also called fraction

Floating-Point Representation

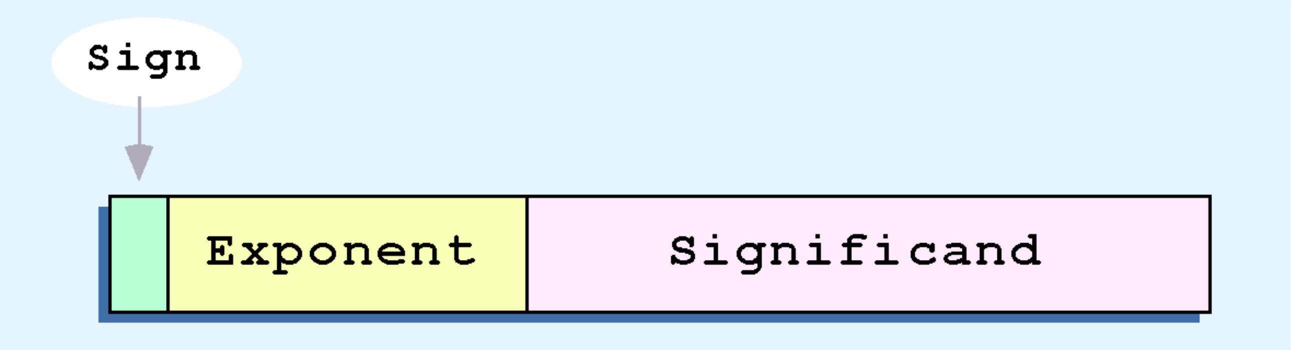
- Computer representation of a floating-point number consists of three fields:



- The Significand is also called ***Fraction***
- This is the standard arrangement of these fields.
- The width of the Exponent and Significand fields varies.

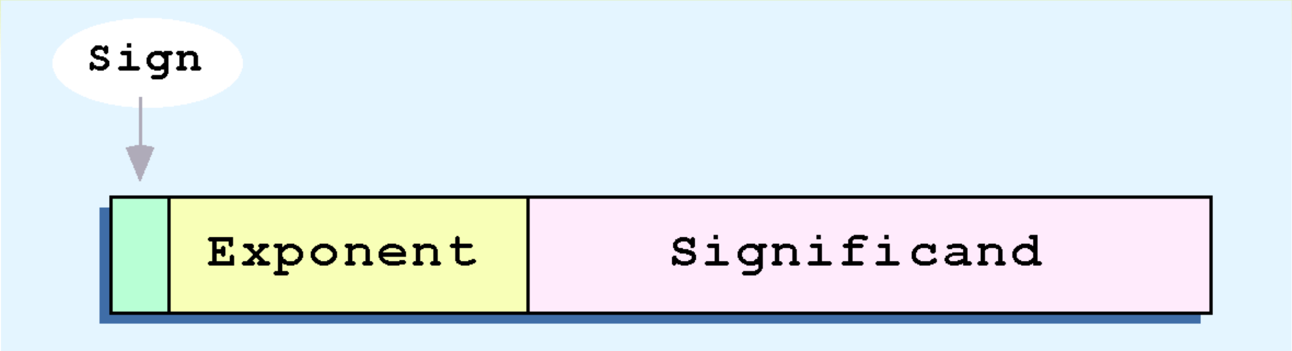
Note: Although "Significand", "Mantissa" and "Fraction" do not technically mean the same thing, many people use these terms interchangeably. We use the term "Fraction" to refer to the fractional part of a floating point number.

Floating-Point Representation



- The one-bit sign field is the sign of the whole floating-point number stored.
- The size of the exponent field determines the range of values that can be represented.
- The size of the Significand determines the precision of the representation.

Floating-Point Representation



- The significand is always preceded by an implied binary point.
- Thus, the significand always contains a fractional binary value.
- The exponent indicates the power of 2 by which the significand is multiplied.

Floating-Point Representation From Blackboard To Floating-Point

Example:

Represent 5/16 as a floating-point number in a 16-bit register.

5/16 = 5 * 2^{-4} = 101 * 2^{-4} = 1.01 * 2^{-2} ***Normalized Notation***

The 4-bit exponent (-2) in two's complement is 1110.

The explicit "1" to the left of the radix point in Normalized Notation is dropped and the fractional part (becomes 01) . We pad with 9 zeros to the right (total of 11-digits for fractional part). **The next slide shows why??**

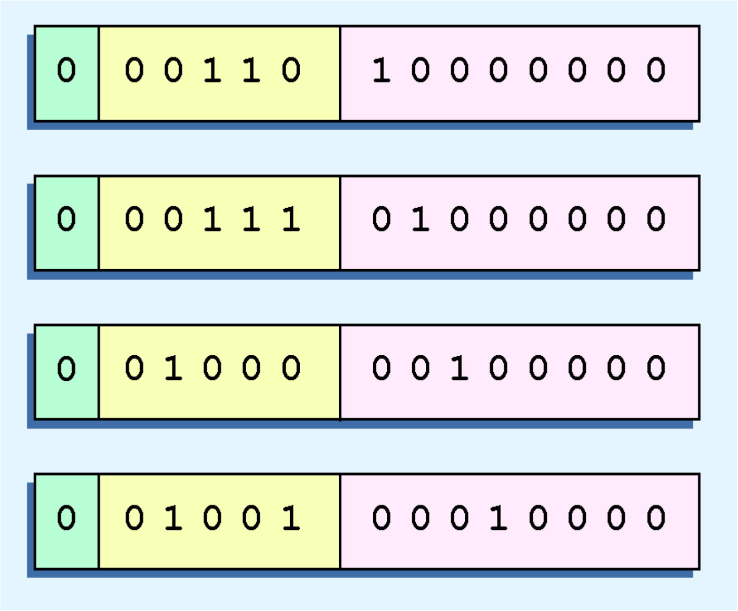
5/16 = 0 |1110 | 01000000000 which is 7200 in hex

Another Example: Put (1.1010101) [-3] in a 16-bit register as above

This is: 0 | 1101 | 10101010000, which is 6d50 in hex.

Floating-Point Representation
Taking a consistent Approach

- Without a consistent scientific blackboard notation, all the illustrations shown at the right are equivalent representations for 32 using a 14-bit model.
- We know that 32 is 2^5. So in (binary) scientific notation:
- 32 = 1.0×2^5 or 0.1×2^6 or 0.01×2^7 or 0.001×2^8 etc.
- Not only do these synonymous representations waste space, but they can also cause confusion.
- Using normalized blackboard notation and dropping the explicit "1" to the left of the radix point results a unique pattern for each floating-point number.



0	0 0 1 1 0	1 0 0 0 0 0 0 0
0	0 0 1 1 1	0 1 0 0 0 0 0 0
0	0 1 0 0 0	0 0 1 0 0 0 0 0
0	0 1 0 0 1	0 0 0 1 0 0 0 0

Floating-Point Representation Addition & Subtraction

- Floating-point addition and subtraction are done using methods analogous to how we perform calculations using pencil and paper. The addition is carried out on both operands in blackboard notation.
- The first thing that we do is express both operands in the same exponential power, then add the numbers, preserving the exponent in the sum.
- If the exponent requires adjustment, we do so at the end of the calculation.

Floating-Point Representation Addition & Subtraction

Example: Find the sum of 12_{10} and 1.25_{10} using the 16-bit floating-point model.

$12_{10} = 1100 = 1.100 \times 2^{3}$

$1.25_{10} = 1.01 \times 2^{0} = 0.00101 \times 2^{3}$

$12_{10} + 1.25_{10} = 1.10101000000[3]$

$= 1101.01 = 13.25$

```
 1.10000000000 [3]
+0.00101000000 [3]
 1.10101000000 [3]
```

Example: $11.75_{10} + 1.25_{10}$

$11.75_{10} = 1011.11 = 1.01111 \times 2^{3}$

$1.25_{10} = 1.01 \times 2^{0} = 0.00101 \times 2^{3}$

$11.75_{10} + 1.25_{10} =$ **1.101**$\times 2^{3} = 15$

```
    1111
 1.01111000000 [3]
+0.00101000000 [3]
 1.10100000000 [3]
```

Example: $11.75_{10} + 15.5625_{10}$

```
 1 111
 1.01111000000 [3]
+1.11110010000 [3]
11.01101010000 [3]
```

Floating-Point Representation Multiplication

- Floating-point multiplication is also carried out in a manner akin to how we perform multiplication using pencil and paper. Just like the addition, floating point multiplication is carried out on both operands in blackboard notation.
- Note that we do not need to express both operands in the same exponential power.
- We multiply the two operands, determine the position of the radix point in the result and add the two exponents.
- If the exponent requires adjustment, we do so at the end of the calculation.

Floating-Point Representation Multiplication

Example: Find the product of 11.75_{10} and 1.25_{10} using the 16-bit floating-point model.

11.75_{10} = 1011.11=1.01111 x 2^{3}

1.25_{10} = 1.01 x 2^{0}

11.75_{10} X 1.25_{10}=**1.110111[3]** =14.6875

The multiplication method used here is called Fused-Multiply-Add. Each bit of the multiplier is multiplied be the whole multiplicand and the result is incrementally added to the accumulator (ACC). The accumulator is initialized to zero at the beginning.

```
  1.01111  [3] multiplicand
X 1.01     [0] multiplier
_______________
  000000000 Acc=0
     101111
___________
     101111 Acc=2f
   10111100
   ________
   11101011 Acc=EB
```

Floating-Point Representation Conclusion

- We have seen different models to represent floating point numbers.
- No matter how many bits we use in a floating-point representation, the model must be finite.
- The real number system is, of course, infinite, so our models give an approximation of a real value.
- No model is large enough and adequate for all calculations. IEEE Double Precision models have been introduced to offer higher precision and range.
- By using a greater number of bits in our model, we can reduce these errors, but we can never totally eliminate them.