Module 2:
Data Representation in Computer Systems

BOOK: COMPUTER ORGANIZATION AND DESIGN,3ED, DAVID L. PATTERSON AND JOHN L. HANNESSY, MORGAN KAUFMANN PUBLISHERS

Contents

Floating-Point Representation

Real Numbers

- Two's complement representation deal with signed integer values only.
- Without modification, 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

- Floating point aids in the representation of very big or very small fixed point numbers.

10000000000	Fixed point
1.0×10^{10}	Floating point

$976,000,000,000,000 \rightarrow 9.76 \times 10^{14}$

$0.000000000000976 \rightarrow 9.76 \times 10^{-14}$

Floating Point Numbers

Significand/**Fraction**/Mantissa

Exponent

$$736.637 \times 10^{68}$$

Base/**radix**

Fraction and **Exponent** can be +ve or –ve.

Decimal numbers use the radix 10, binary use 2

Normalized and Unnormalized

- In generalized normalization (in *general mathematics*), a floating point number is said to be normalized if the number after the radix point is a non-zero value.
- Un-normalized floating number is when the number after the radix point is '0'.
- Example:

number after the radix point is a non-zero value.

0.1234×10^{16}

➔ **normalized**

0.0011×10^{15}

➔ **unnormalized**

11.0123×10^{11}

➔ **unnormalized**

0.133×10^{5}

➔ **normalized**

Normalization Process

- Normalization is the process of deleting the zeroes until a non-zero value is detected.
- Example :

➲Move radix point to the right (in this case 2 points)

0.00234×10^4 ➔ **$0.234 \times 10^{4-2}$** ➔ **0.234×10^2**

➲ Move radix point to the left (in this case 2 points)

12.0024×10^4 ➔ **$0.120024 \times 10^{4+2}$** ➔ **0.120×10^6**

A rule of thumb:

moving the radix point **to the right** → **subtract exponent**

moving the radix point **to the left** → **add exponent**

Example

Decimal

24.89×10^{89} → $0.249 \times 10^{89+2}$ → 0.249×10^{91}

Binary

$11.00110 \times 2^{-110011}$

→ $0.1100110 \times 2^{(-110011)+(010)}$

→ $0.1100110 \times 2^{-110001}$

$0.00011011 \times 2^{1001}$

→ $0.11011 \times 2^{(1001)-(0011)}$

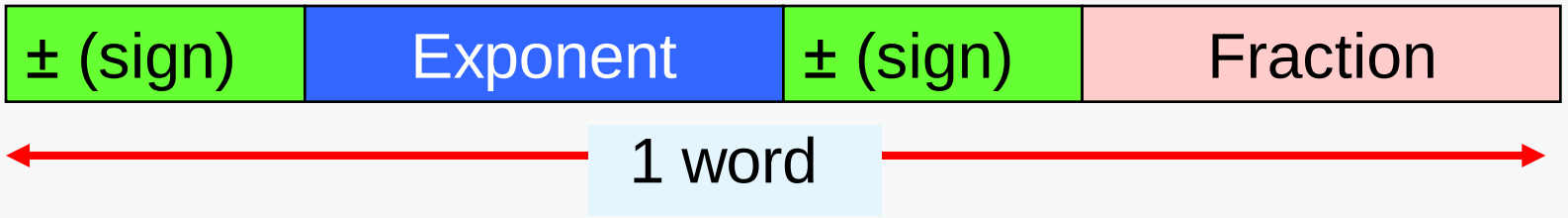
→ 0.11011×2^{0110}

Floating-Point Format for Binary Numbers

- The general form of floating-point is:

$$\pm\ 0.\text{Fraction} \times \text{Base}^{\pm\ \text{exponent, e'}}$$

- In binary:



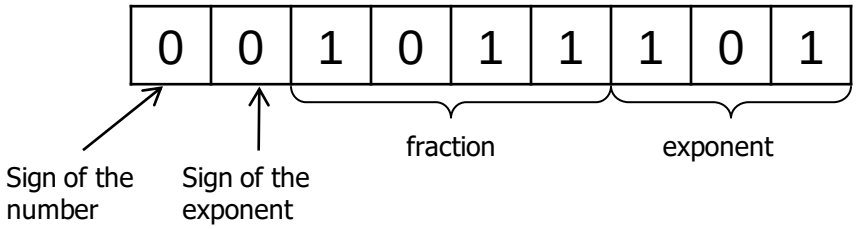
Example: Non-biased exponent

9 bit-hypothetical word

- the first bit is used for the sign of the number,
- the second bit for the sign of the exponent,
- the next four bits for the fraction, and
- the next three bits for the exponent

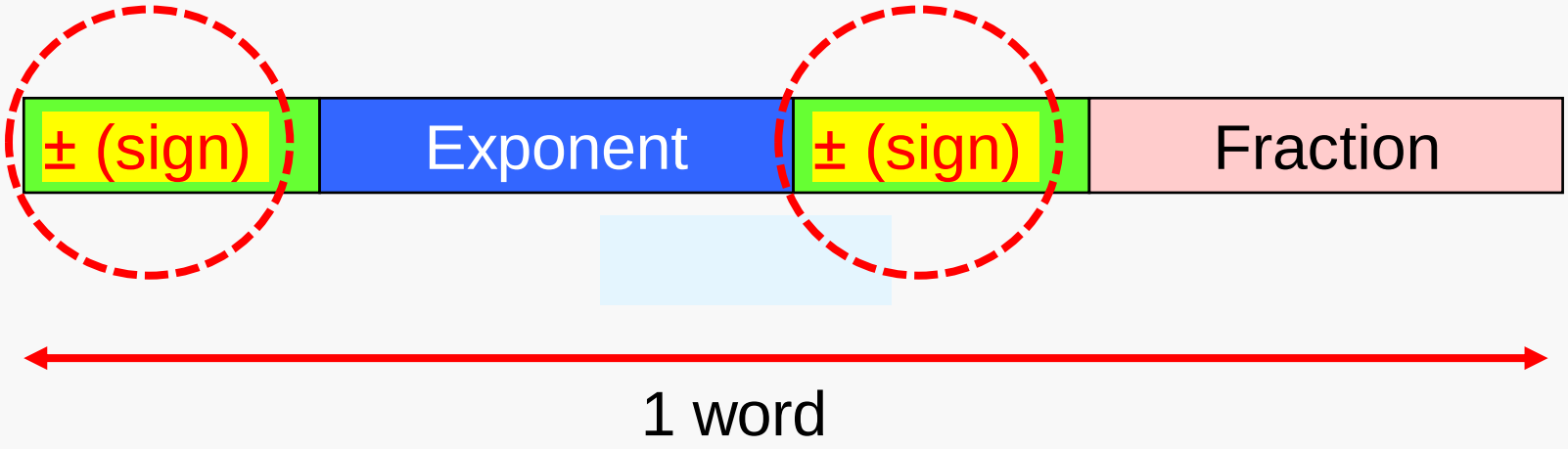
$$(54.75)_{10} = (110110\ .11)_2 = (1.1011011\ )_2 \times 2^5$$
$$\cong (1.1011)_2 \times (101)_2$$

We have the representation as



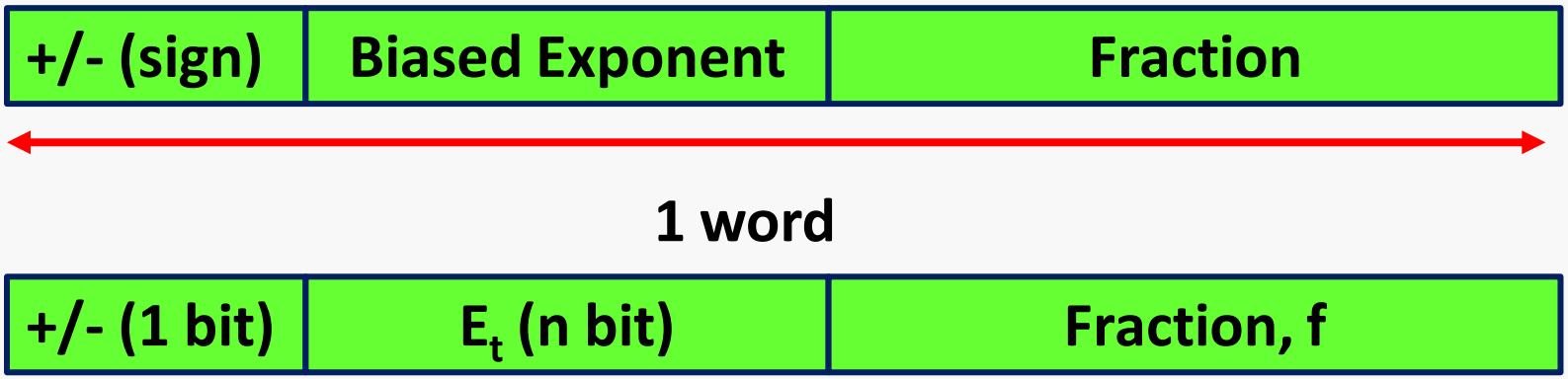
Biased Exponent

- The 2 signs bit are not good for design as it incurs extra cost. → need new representation

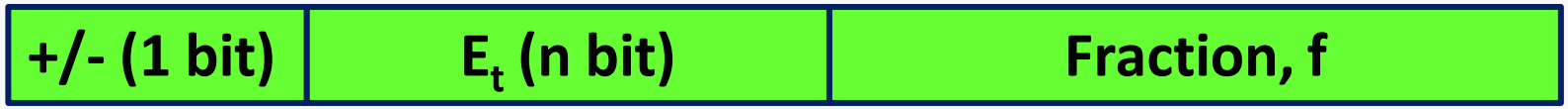


Biased Exponent

- A new value to represent the exponent without the sign bit is introduced
  - This will eliminate the sign for the exponent value that is the exponent will be positive. (indicative)



Biased Exponent



Biased value, $b = 2^{n-1}$

Normalized exponent, $e' = E_t - b$

Biased exponent, $E_t = e' + b$

Where,
E_t = biased exponent
n = bits of exponent format (i.e. the word format)

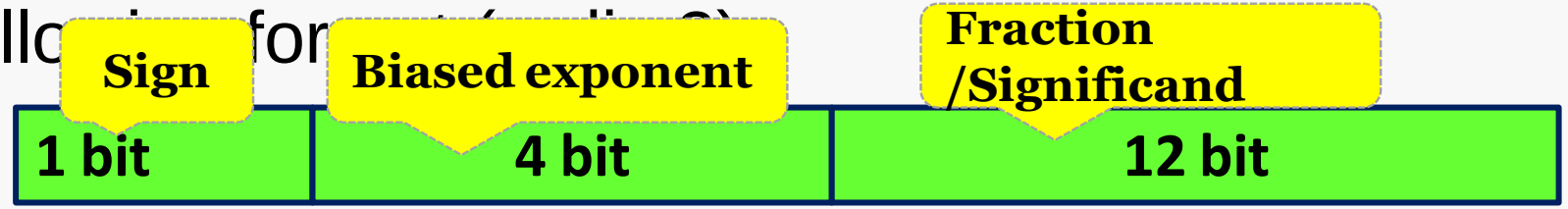
This is used unless the IEEE standard is mentioned – then is a different calculation

Conversion to Floating Point Number

- The steps:
  - Change to binary (if given decimal number)
  - Normalized the number
  - Change the number to biased exponent
  - Form the word (3 fields of a given format)

Example 3

- Transform -33.625 to floating point word using the follo[illegible] for[illegible]



- Step 1 : Change 33.625 to binary

This is the given word format

33 → 0100001

33.625 = 0100001.101

0.625 x 2 = 1.25 → 1
0.25 x 2 = 0.5 → 0
0.5 x 2 = 1.0 → 1
0.0

0.625 = .101

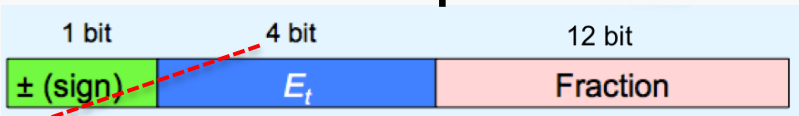
Example 3

- Step 2 : Normalized the number

Normalized exponent, e'

0100001.101_2 → $0.100001101_2 \times 2^{0110}$

- Step 3: Change the number to biased exponent



Biased value, $b = 2^{n-1} = 2^{4-1} = 8_{10} = 1000_2$

Biased exponent, $E_t = e' + b = 0110 + 1000 = 1110$

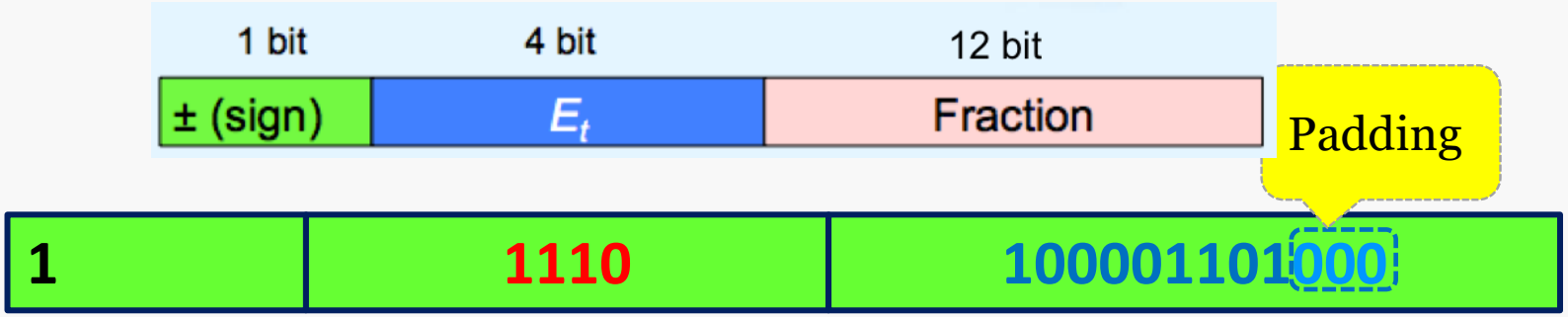
$0.100001101_2 \times 2^{0110}$ → $0.100001101_2 \times 2^{1110}$

Example 3

$0.100001101 \times 2^{1110}$

-33.625

- Step 4 : Form the word (3 fields)

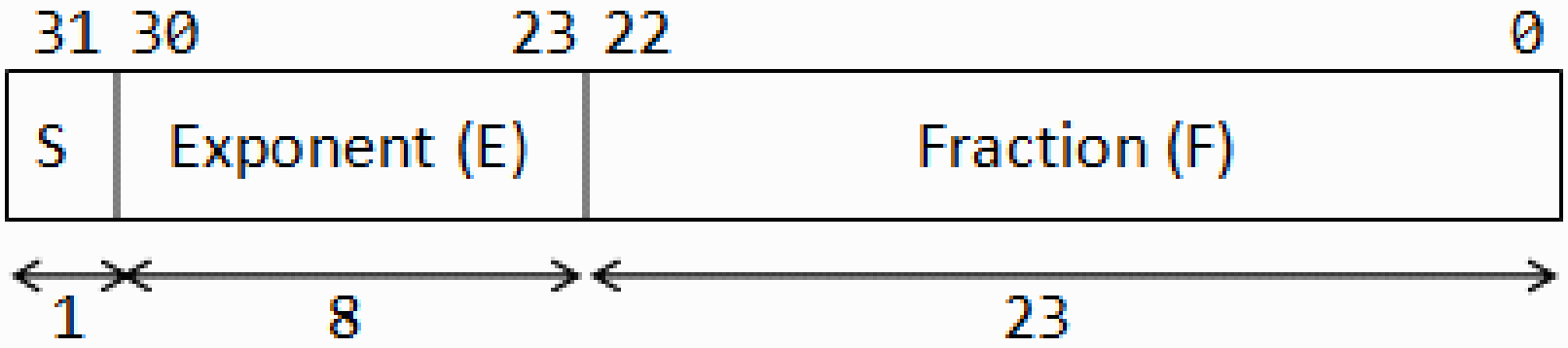


Rule of thumb:
-the biased exponent is always padded to the left
- the Fraction is always padded to the right

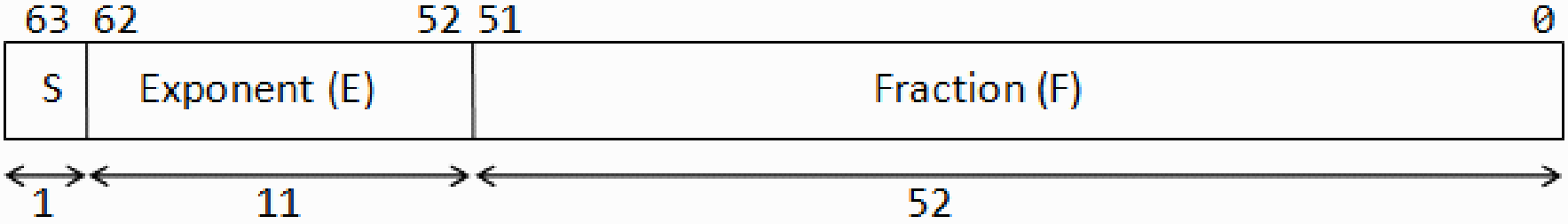
FLOATING POINT STANDARD

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE 754 Floating-Point Standard



32-bit Single-Precision Floating-point Number



64-bit Double-Precision Floating-point Number

IEEE 754 Floating-Point Standard

- Used in virtually every computer invented since 1980
- To pack more bits into the significand, the leading 1 bit of normalized binary number is made implicit
  - Original: $1.xxxxxxxx_{two} \times 2^{yyyy} \rightarrow (-1)^S \times F \times 2^E$
  - Modified: $(-1)^S \times (1+\text{Fraction}) \times 2^E$
  - Significand: 1 plus the fraction
    - Single precision: 24 bits
    - Double precision: 53 bits

The 1 in (1 + Fraction) is made implicit
➔ to pack more bits into the significand

Normalized Scientific Notation in IEEE 754

- In IEEE standard for normalization (used in computers), a floating point number is said to be normalized if there is only a single non-zero before the radix point.
- Example:

there is only a single non-zero before the radix point.

123.456 → **normalized** **1.23456×10^2**

1010.1011_B → **normalized** **1.0101011×2^{011}**

Biased Notation in IEEE 754

- The desired notation must represent the most negative exponent as $00...00_{two}$ and the most positive as $11...11_{two}$
  - IEEE 754 uses a bias of 127 for single precision (and 1023 for double precision)
    - $-1 \rightarrow -1+127_{ten} = 0111\ 1110_{two}$
    - $+1 \rightarrow 1+127_{ten} = 1000\ 0000_{two}$

Bias
➔In single precision is **127**
➔In double precision **1023**

Biased Notation in IEEE 754

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

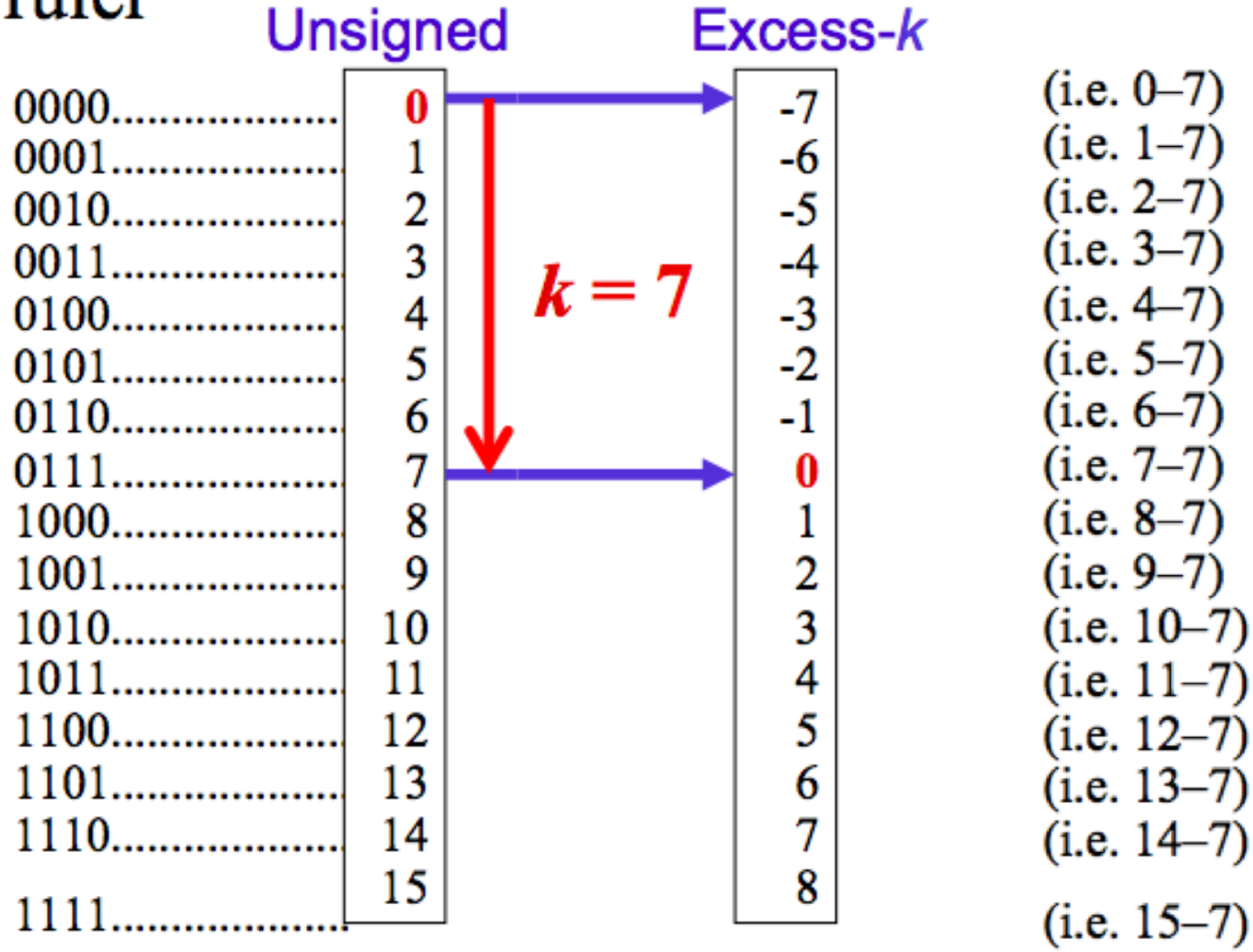
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single precision: Bias = 127;
  - Double precision: Bias =1023

Excess-*k* representation

- Yet another way to represent numbers in binary.
- For N bit numbers, *k* is $2^{N-1}-1$
- So for 4-bit integers, *k* is 7
- The value of each bit string is its *unsigned value* minus *k*. *(i.e. 'unsigned value' – k)*

Excess-*k* representation: 4-bit excess 7

"Sliding ruler"



Excess-*k* representation: Example

- Convert **`-14`** (decimal) to 8-bit excess-*k*
- What is ***k*** ?
  k = $2^{N-1}-1 = 2^{8-1}-1$ = **127**
- Find the number ***u*** such that ***u*** - ***k*** = -14.
  u - **127** = -14, implies ***u*** = **113**
- Convert ***u*** to unsigned binary:
  `001110001`

8-bit ; excess-127

8 bit excess-127

Binary value	Excess-127 interpretation	Unsigned interpretation
00000000	−127	0
00000001	−126	1
⋮	⋮	⋮
01111111	0	127
10000000	1	128
10000001	2	129
⋮	⋮	⋮
11111111	+128	255

8-bit ; excess-127

e		Meaning
0000 0000		Reserved
0000 0001		-126_{10}
0000 0010		-125_{10}
0111 1111		0_{10}
1111 1110		127_{10}
1111 1111		Reserved

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Significand is Fraction with the "1." restored
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)

Single-Precision Range

- Exponents 00000000 and 11111111 are reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110 $= 254_{10}$
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

DOUBLE-PRECISION RANGE

- Exponents 0000…00 and 1111…11 are reserved
- Smallest value
  - Exponent: 00000000001
    ⇒ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 ⇒ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    ⇒ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 ⇒ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

IEEE 754 Conversion

- To convert a decimal number to single (or double) precision floating point:
  - Step 1: Normalize
  - Step 2: Determine Sign Bit
  - Step 3: Determine exponent
  - Step 4: Determine Significand

IEEE 754 Conversion : Example 1

- Convert 10.4_d to *single precision* floating point.
- **Step 1: Normalize**

10 ➔ 00001010

0.4 x 2 = 0.8 ➔ 0
0.8 x 2 = 1.6 ➔ 1
0.6 x 2 = 1.2 ➔ 1
0.2 x 2 = 0.4 ➔ 0
0.4 x 2 = 0.8 ➔ 0
0.8 x 2 = 1.6 ➔ 1

For continuous results, take the 1st pattern before it repeats itself

0.4 = .0110

10.4 = 1010.0110 x 2^0

➔ 1.0100110 x 2^3

IEEE 754 Conversion : Example 1

- **Step 2: Determine Sign Bit (S)**
  - Because (10.4) is positive, S = 0

➔ **1.0100110×2^3**

- **Step 3: Determine exponent**
  - Because its single precision ➔ bias = 127
  - Exponent = 3 + bias
    = 3 + 127
    = 130_d
    = **$1000\ 0010_b$**

IEEE 754 Conversion : Example 1

- **Step4: Determine Significand**
  - Drop the leading 1 of the significand

1.0100110×2^3 **➔ 0100110**

  - Then expand (padding) to 23 bits

01001100000000000000000

sign	Exponent	Significand
0	10000010	01001100000000000000000

Exponent = **$1000\ 0010_b$**

IEEE 754 Conversion : Example 2

- Convert -0.75_d to single precision floating point.
- **Step 1: Normalize**

0.75 x 2 = 1.5 → 1
0.5 x 2 = 1.0 → 1
0.0 x 2 = 0 → 0

- 0.75 = - 0.11

→ -0.11×2^0

→ -1.1×2^{-1}

IEEE 754 Conversion : Example 2

- **Step 2: Determine Sign Bit (S)**
  - Because (-0.75) is negative, S = 1

- **Step 3: Determine exponent**
  - Because its single precision ➔ bias = 127
  - Exponent = -1 + bias
    = -1 + 127
    = 126_d
    = 01111110_b

IEEE 754 Conversion : Example 2

- **Step4: Determine Significand**
  - Drop the leading 1 of the significand

-1.1×2^{-1} → 0.1

  - Then expand (padding) to 23 bits

10000000000000000000000

sign	Exponent	Significand
1	01111110	10000000000000000000000

IEEE 754 Conversion : Example 3

- Convert -0.75_d to *double precision* floating point.
- **Step 1: Normalize**

0.75 x 2 = 1.5 → 1
0.5 x 2 = 1.0 → 1
0.0 x 2 = 0 → 0

- 0.75 = - 0.11

→ -0.11×2^0

→ -1.1×2^{-1}

IEEE 754 Conversion : Example 3

- **Step 2: Determine Sign Bit (S)**
  - Because (-0.75) is negative, S = 1

- **Step 3: Determine exponent**
  - Because its double precision ➔ bias = 1023
  - Exponent = -1 + bias

    = -1 + 1023

    = 1022_d

    = 01111111110_b

IEEE 754 Conversion : Example 3

- **Step4: Determine Significand**
  - Drop the leading 1 of the significand

-1.1×2^{-1} ➔ 0.1

  - Then expand (padding) to 52 bits

10000000000000000000.....00

-0.75

sign	Exponent (11)	Significand (52)
1	01111111110	10000000000000000000...00

Converting Binary to Decimal Floating-Point

Remember: Biased notation ➔ (-1)sign x (1 + Fraction) x 2 (exponent-bias)

- What decimal number is represented by this single precision float?

Sign (1 bit)	Exponent(8 bit)	Significand(23 bit)
1	10000001	01000000000000000000000

- Extract the values:

Sign = 1
Exponent = 10000001b = 129d

Significand
= **$(0 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3})$**
= **¼ = 0.25**

The Fraction = **-(1 + 0.25)**

The number
= - $(1.25 \times 2^{(\text{exponent-bias})})$
= - $(1.25 \times 2^{(129 - 127)})$
= - (1.25×2^{2})
= - (1.25 x 4) = -5.0

Math basic … fraction number!

- .154 = 1/10 + 5/100 + 4/1000

Decimal = base 10

$$1.\frac{1}{10^1} + 5.\frac{1}{10^2} + 4.\frac{1}{10^3}$$

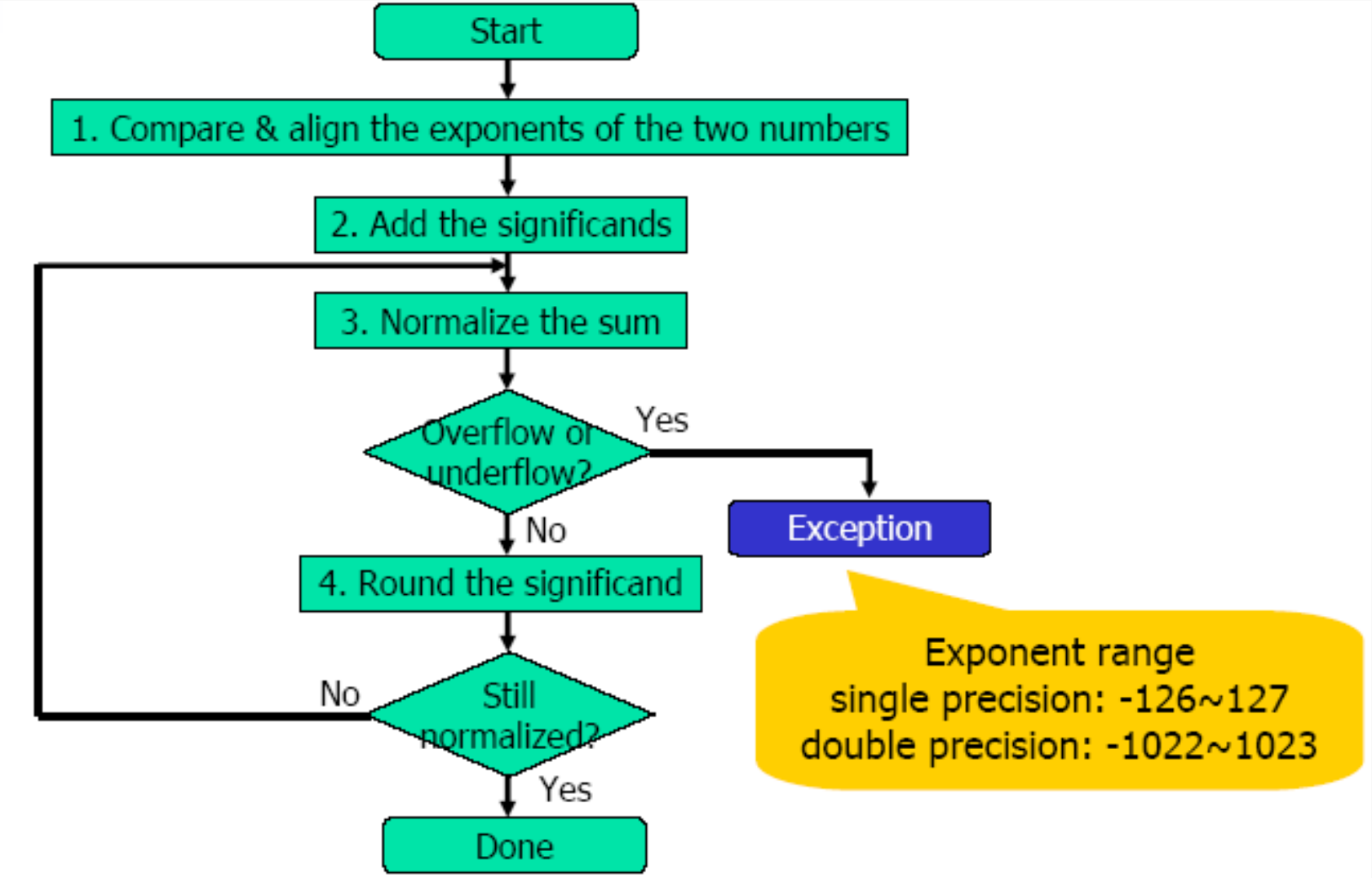
- .1011 = 1/2 + 0/4 + 1/8 + 1/16

Binary = base 2

$$1.\frac{1}{2^1} + 0.\frac{1}{2^2} + 1.\frac{1}{2^3} + 1.\frac{1}{2^4}$$

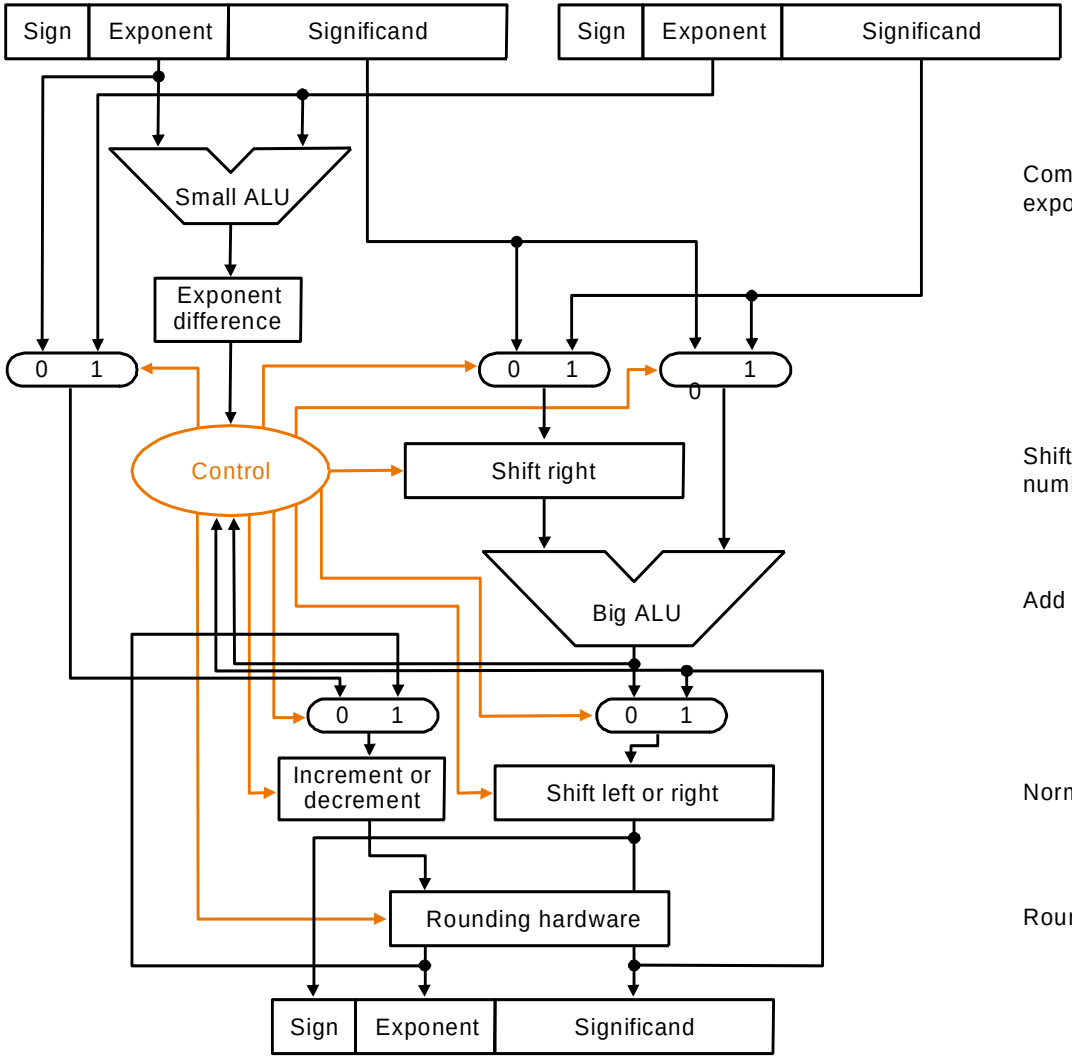
Floating-Point Arithmetic

Floating-Point Addition Flows



Floating-Point ALU

Block diagram of an arithmetic unit dedicated to floating-point addition.



Decimal Floating-Point Addition

- Assume 4 decimal digits for significand and 2 decimal digits for exponent
  - Step 1: Align the decimal point of the number that has the smaller exponent
  - Step 2: Add the significand
  - Step 3: Normalize the sum
    - check for overflow/underflow of the exponent after normalisation
  - Step 4: Round the significand
    - If the significand does not fit in the space reserved for it, it has to be rounded off
  - Step 5: Normalize it (if need be)

A) Decimal Floating-Point Addition

Example: $9.999_d \times 10^1 + 1.610_d \times 10^{-1}$

- **Step 1**: Align the decimal point of the number that has the smaller exponent

  Make $1.610_d \times 10^{-1}$ to 10^1
  ➔ $-1 + x = 1$ ➔ $x = 2$ → move 2 to left
  ➔ $0.0161_d \times 10^1$

- **Step 2**: add the significand

$$
\begin{array}{rl}
 & 9.9990 \times 10^1 \\
+ & 0.0161_d \times 10^1 \\
\hline
 & 10.0151 \times 10^1
\end{array}
$$

A) Decimal Floating-Point Addition

Example: $9.999_d \times 10^1 + 1.610_d \times 10^{-1}$

- **Step 3**: Normalize the sum

  $10.0151 \times 10^1 \rightarrow 1.00151 \times 10^2$

- **Step 4**: Round the significand (to 4 decimal digits for significand)

  $1.00151 \times 10^2 \rightarrow 1.0015 \times 10^2$

- **Step 5**: Normalize it (if need be)

  No need as its normalized

B) Binary Floating-Point Addition

Example: $0.5_d + (-0.4375_d)$

- Convert the numbers to binary

$0.5 \rightarrow 0.10_b \times 2^0 \rightarrow 1.0_b \times 2^{-1}$

$-0.4375 \rightarrow -0.0111_b \times 2^0 \rightarrow -1.11_b \times 2^{-2}$

- **Step 1**: Align the decimal point of the number that has the smaller exponent

Make $1.11_b \times 2^{-2}$ to 2^{-1}

$\rightarrow -2 + x = -1 \rightarrow x = 1 \rightarrow$ move 1 to left

$\rightarrow - 0.111_b \times 2^{-1}$

B) Binary Floating-Point Addition

Example: $0.5_d + (-0.4375_d)$

- **Step 2:** add the significand

$$\begin{array}{r} 1.000 \times 2^{-1} \\ + \ -0.111 \times 2^{-1} \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{r} 1.000 \times 2^{-1} \\ - \ 0.111 \times 2^{-1} \\ \hline 0.001 \times 2^{-1} \end{array}$$

- **Step 3**: Normalize the sum

$0.001 \times 2^{-1} \rightarrow 1.0000 \times 2^{-4}$

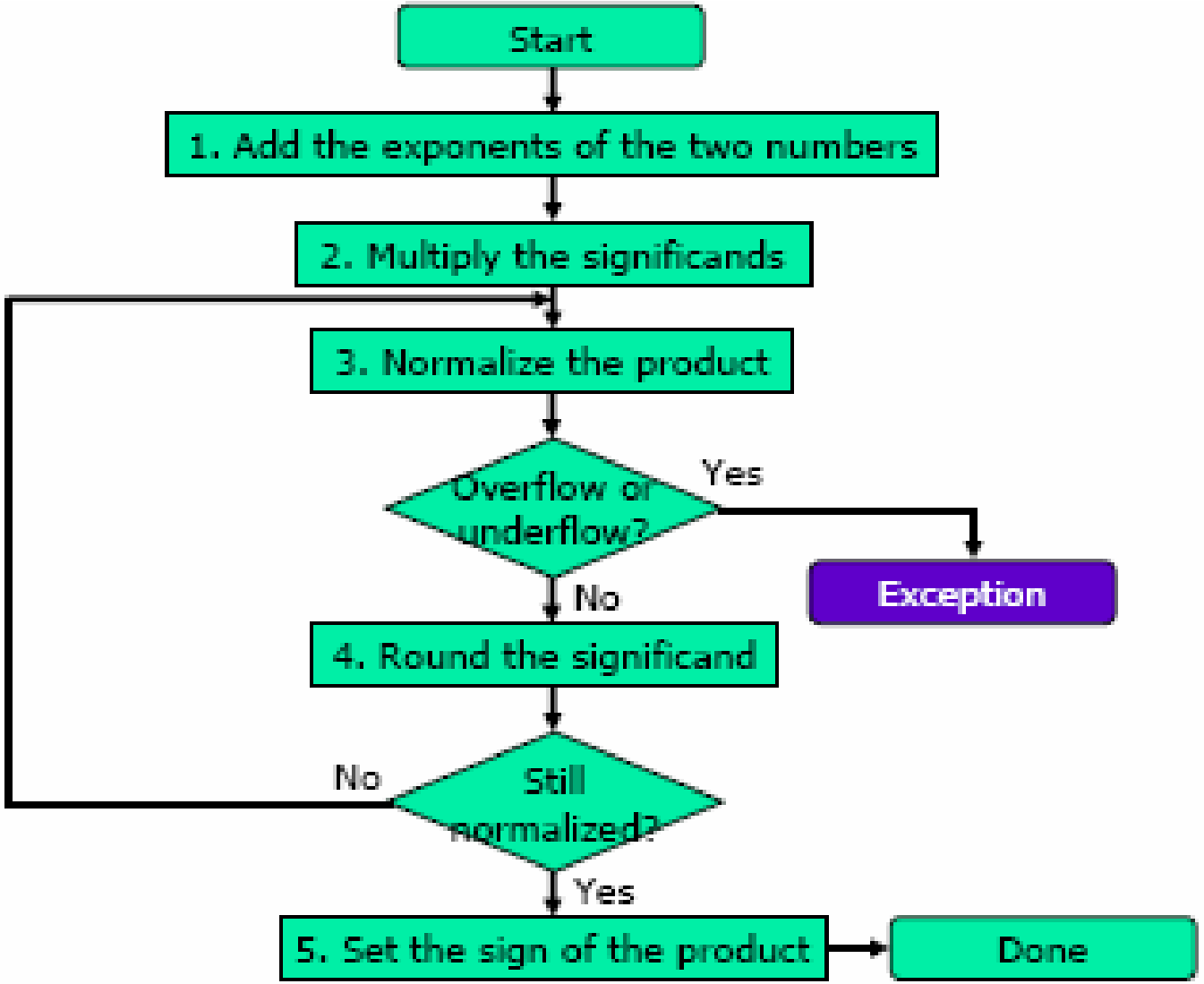
- **Step 4:** Round the significand (to 4 decimal digits for significand)

Fits in the 4 decimal digits

- **Step 5**: Normalize it (if need be)

No need as its normalized

Floating-Point Multiplication Flows



A) Floating-Point Multiplication (decimal)

Example: $(1.110_d \times 10^{10}) \times (9.200_d \times 10^{-5})$

- Assume 4 decimal digits for significand and 2 decimal digits for exponent
- Step 1: Add the exponent of the 2 numbers

10 + (-5) = 5

If biased is considered ➔ 10 + (-5) + 127 = 132

- Step 2: Multiply the significands

```
    9.200
  x 1.110
-------------
   92000
  9200
 9200
-------------
10.212000
```

➔ 10.212000 ➔ 10.2120×10^5

A) Floating-Point Multiplication (decimal)

Example: $(1.110_d \times 10^{10}) \times (9.200_d \times 10^{-5})$

- Step 3: Normalize the product

  $10.2120 \times 10^5 \rightarrow 1.02120 \times 10^6$

- Step 4: Round the significand (4 decimal digits for significand)

  1.0212×10^6

- Step 5: Normalize it (if need be)

  Still normalized

- Step 6: Set the sign of the product

  $+1.0212 \times 10^6$

B) Floating-Point Multiplication (binary)

Example: $(1.000_b \times 2^{-1}) \times (-1.110_b \times 2^{-2})$

- Assume 4 binary digits for significand and 2 binary digits for exponent
- Step 1: Add the exponent of the 2 numbers

  -1 + (-2) = -3

  If biased is considered ➔ -1 + (-2) + 127 = 124
- Step 2: Multiply the significands

```
    1.110
x   1.000
----------
 1110000

-> 1.110000
```

➔ 1.110000 ➔ 1.110000×2^{-3}

B) Floating-Point Multiplication (binary)

Example: $(1.000_b \times 2^{-1}) \times (-1.110_b \times 2^{-2})$

- Step 3: Normalize the product

  1.110000×2^{-3} ➔ already normalized

- Step 4: Round the significand (4 binary digits for significand)

  1.1100×2^{-3}

- Step 5: Normalize it (if need be)

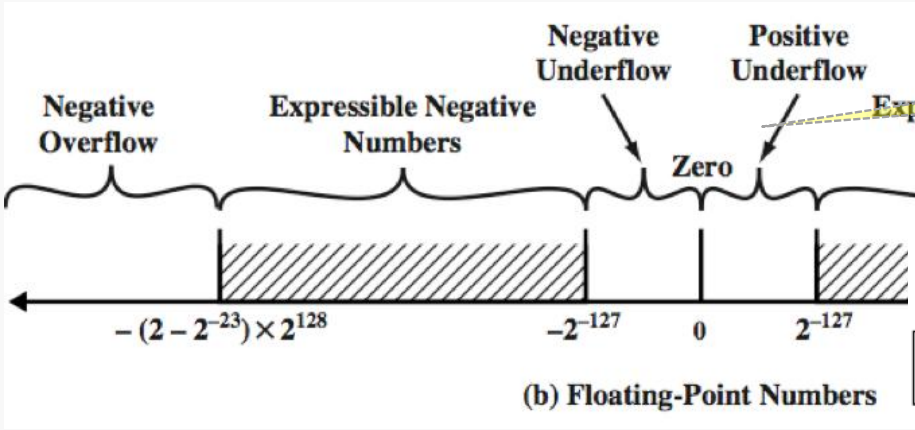
  Still normalized

- Step 6: Set the sign of the product

  $-1.1100_b \times 2^{-3}$ ➔ $-7/32_d$

Accurate Arithmetic: Overflow & Underflow

- If a calculation exceeds the limits of the floating point scheme ➔ then CPU will flag this error.



If number is **too tiny** to be represented

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent	Fraction

- i.e. a +ve OR –ve exponent becomes too large to fit in the exponent field:
  - **+ve** exponent TOO LARGE ⇒ '**overflow**'
  - **-ve** exponent TOO LARGE ⇒ '**underflow**'

Accurate Arithmetic : Truncation & Rounding

- Some number have infinite decimal points (the irrational numbers) ➔ $1/3_d = 0.3333333333$
- Truncation is done to fit the decimal points into *manageable* units.
  - Truncation is where decimal values beyond the truncation point are simply discarded and it can cause error in floating point calculations.
- Rounding :If you have a number such as 3.456 then if you have to round this to 3 significant digits, the number becomes 3.46
  - A small error called the rounding error has occurred
- ***Note : the CPU will not flag any error when truncation and rounding occurs, as it is acting within its limits. ➔ programmers must assess if this will lead to significant errors

Summary

- Over the decades, computer arithmetic has become largely standardized, greatly enhancing the portability of programs.
- **Two's complement binary intege**r arithmetic is found in every modern computer.
- And **binary floating-point** arithmetic is supported by the **IEEE 754** standard.
- Computer arithmetic is **distinguished** from ***paper-and-pencil arithmetic*** by the constraints of ***limited precision***.
- ... may result in invalid operations through calculating numbers larger or smaller than the predefined limits, known as"***overflow***" or "***underflow***"

Module 2: Outline

- Introduction
- Fixed-Number Representation
  - Unsigned Numbers
  - Signed Numbers
- Fixed-Number Arithmetic
  - Addition and Subtraction
  - Multiplication
  - Division
- Floating-Point Representation
  - IEEE-754 Floating-Point Standard
- Floating-Point Arithmetic
  - Addition
  - Multiplication

End of Module 2