What we Covered?

- Number Systems with Conversions
- Integers Representations
  - Unsigned
  - Signed
    - 3 Methods
      - Signed Magnitude
      - One’s complement
      - Two’s complement
  - Detection of Overflow condition (Addition Case)
    - Unsigned overflow detection
    - Signed overflow detection

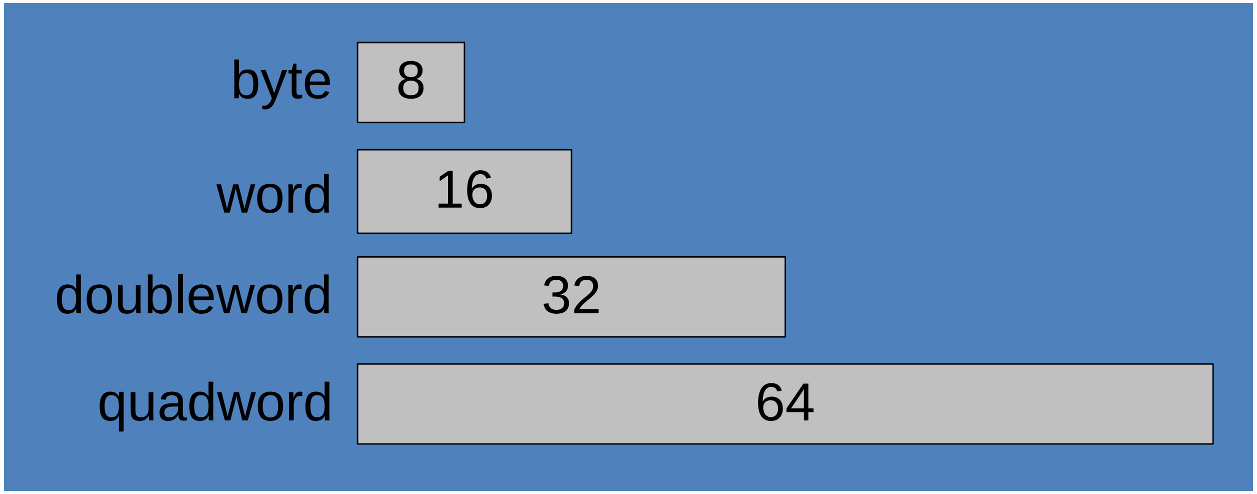
Lab Equivalence

- Lab # 1 → Conversions
  - Decimal → Binary & Hex
  - Binary/Hex → Decimal
  - Hex ←→ Binary
  - Decimal Fraction → Binary
  - Binary Fraction → Decimal

- Signed Numbers + Floating Point Numbers (Not done yet)

Integer Storage Sizes

Standard sizes:



Assembly Data Types

BYTES, WORD,DWORD,QWORD →Unsigned

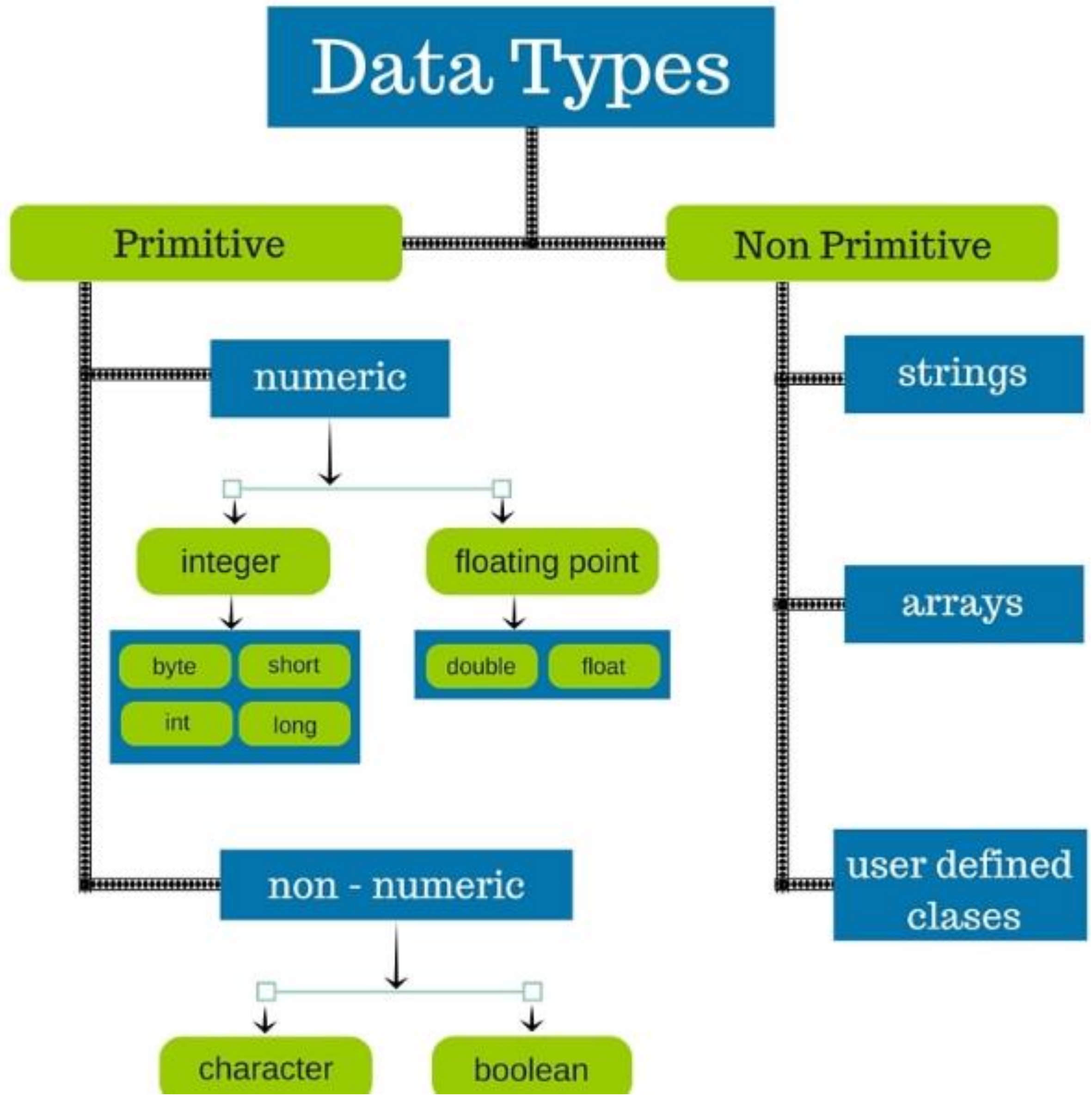
SBYTE, SWORD, SDWORD, SQWORD → Signed

Range of Unsigned Numbers

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

Ranges of Signed Integers

Storage Type	Range (low–high)	Powers of 2
Signed byte	−128 to +127	-2^7 to $(2^7 - 1)$
Signed word	−32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	−2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$



Data Types
Primitive
Non Primitive
numeric
integer
floating point
byte
short
int
long
double
float
non - numeric
character
boolean
strings
arrays
user defined clases

JAVA Data Types

TYPE		STORAGE	DEFAULT	EXAMPLES
Boolean		1 Bit	false	Boolean myBool=true;
Byte		1 Byte	0	
Char		2 Bytes	\u0000	Char myChar='a';
Short		2 Bytes	0	Short myShort=1000;
Int	Signed	4 Bytes	0	Int myInt=100000;
Long		8 Bytes	0	Long myLong=0;
Float		4 Bytes	0.0f	Float myFloat=10.0f;
Double		8 Bytes	0.0L	Double myDouble=20.0;

Floating-Point Representation

- Scientific and business applications deal with real numbers as well not just integers.

- Floating-point representation solves this problem.

Floating-Point Representation

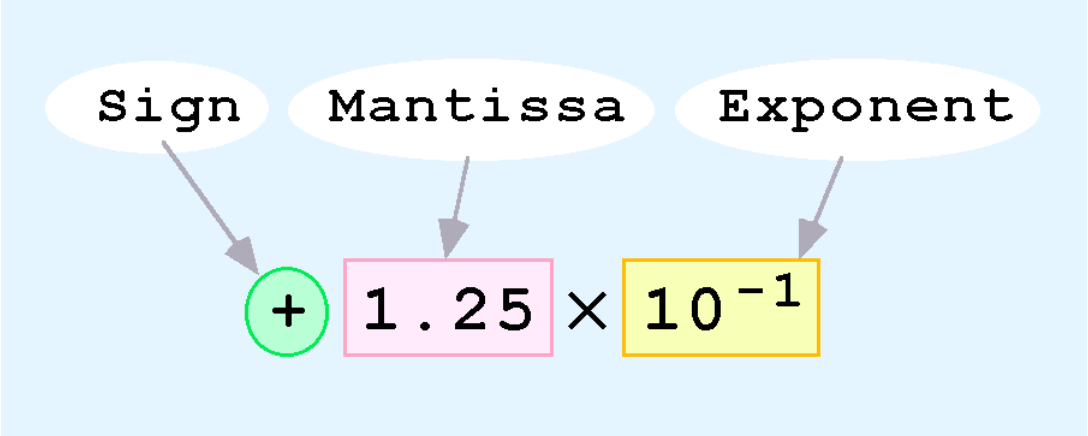
- we also need to be able represent numbers with fractional parts (like: -12.5 & 45.39).

- A range of very large and very small numbers
  - 976,000,000,000,000 $= 9.76 * 10^{14}$
  - 0.0000000000000976 $= 9.76 * 10^{-14h}$

Can <u>Integer</u> Data types store this number?

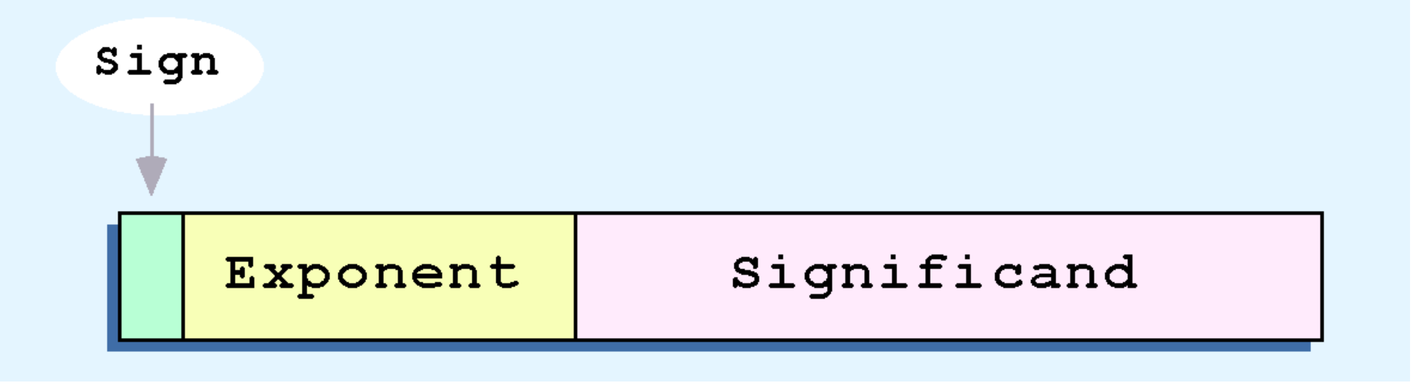
Floating-Point Representation

- Computers use a form of scientific notation for floating-point representation
- Numbers written in scientific notation have three components:



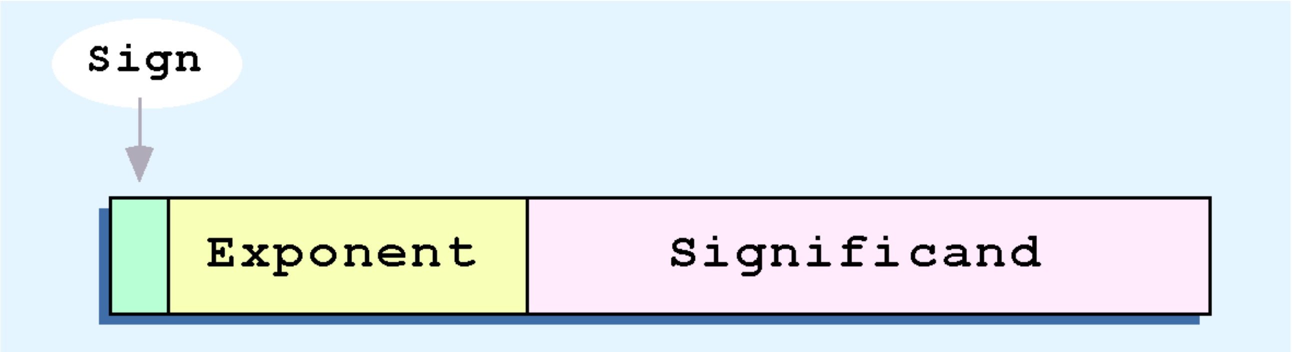
Floating-Point Representation

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



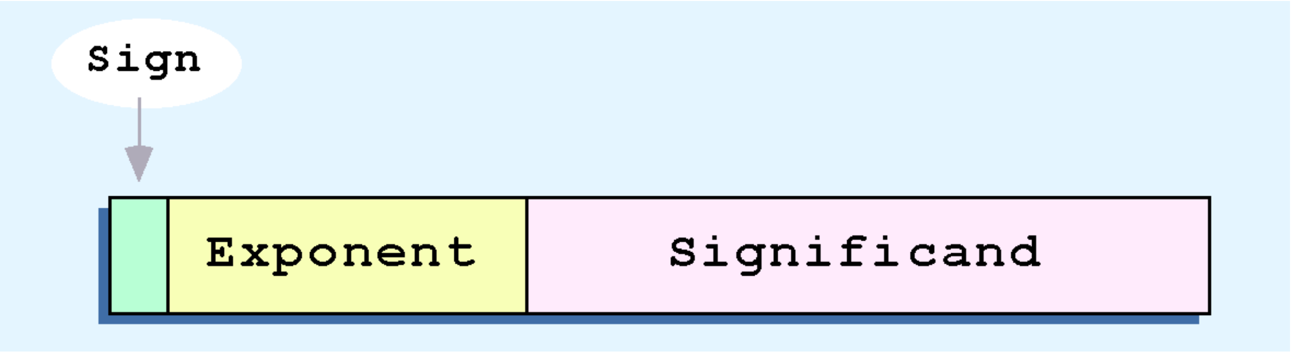
- This is the standard arrangement of these fields.

Floating-Point Representation



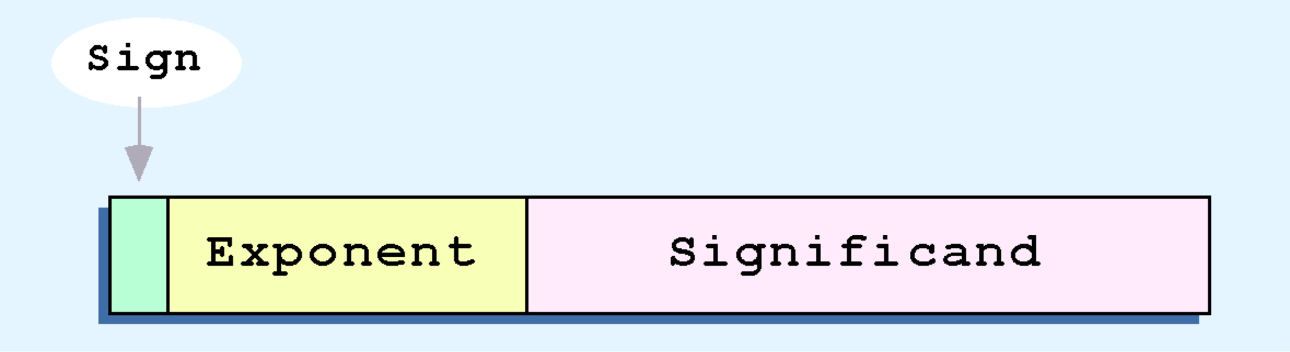
- Sign= 1 bit
- more exponent bits→ greater range
- more significand bits → greater accuracy

Floating-Point Representation



- We introduce a hypothetical "Simple Model" to explain the concepts
- In this model:
  - A floating-point number is 14 bits in length
  - The exponent field is 5 bits
  - The significand field is 8 bits

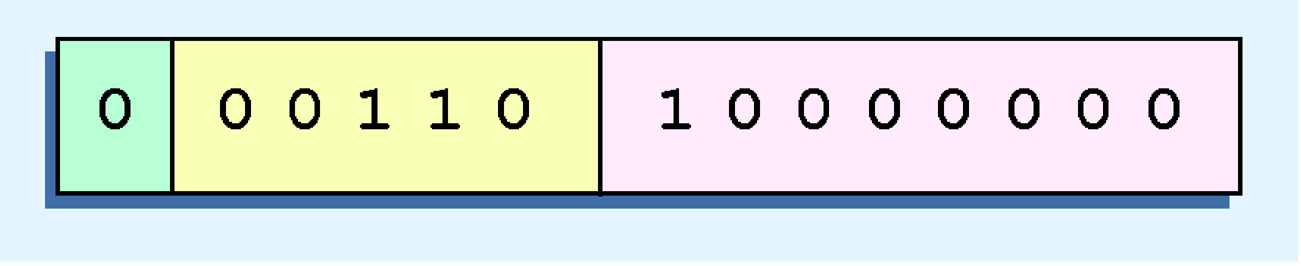
Floating-Point Representation



- 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.

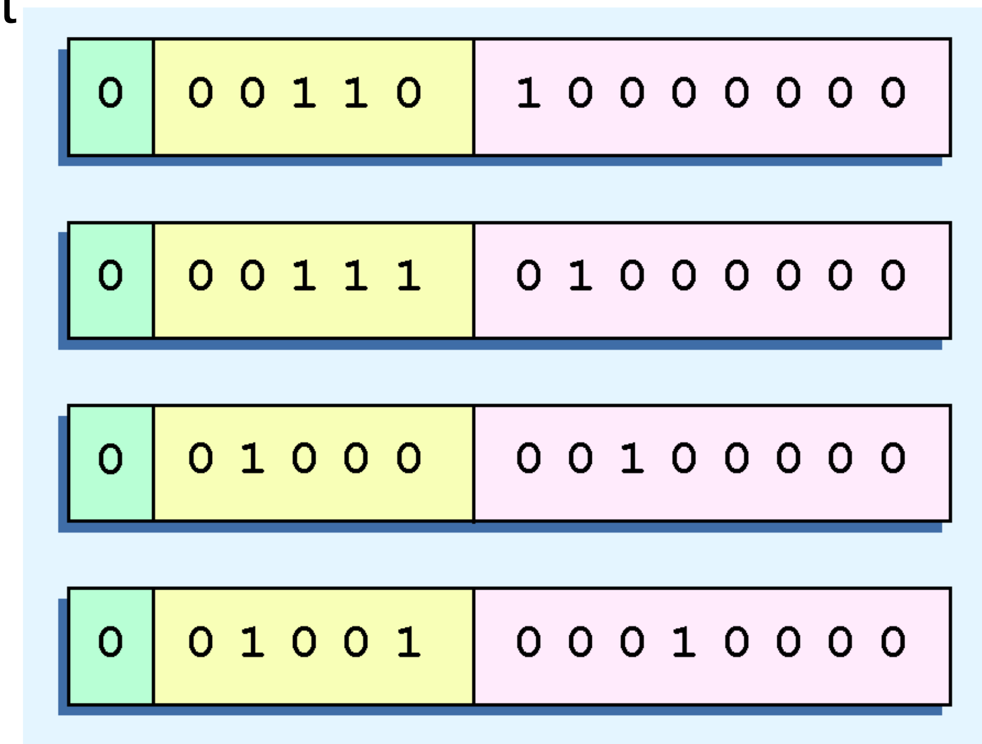
Floating-Point Representation

- Example:
  - Express 32_{10} in the simplified 14-bit floating-point model.
- We know that 32 is 2^5. So in (binary) scientific notation $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
  - In a moment, we'll explain why we prefer the second notation versus the first.
- Using this information, we put 110 ($= 6_{10}$) in the exponent field and 1 in the significand as shown.

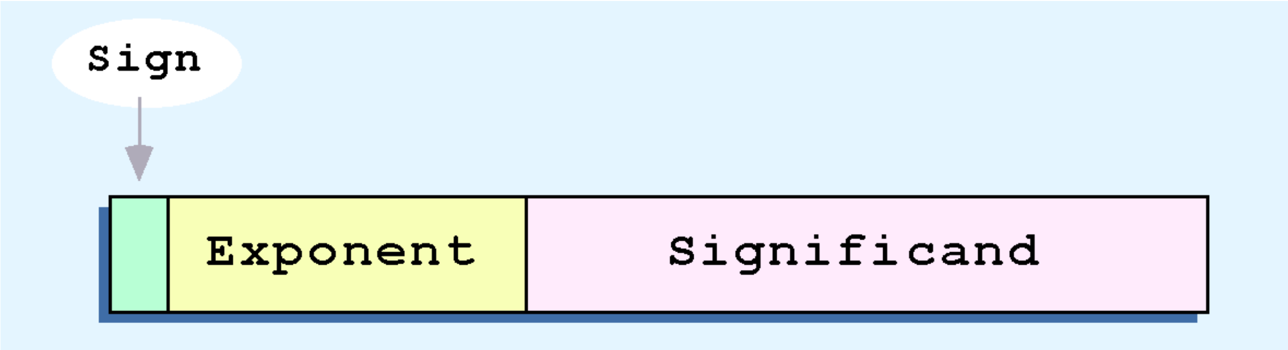


Floating-Point Representation

- The illustrations shown at the right are *all* equivalent representations for 32 using our simplified model.
- Not only do these synonymous representations waste space, but they can also cause confusion.



Floating-Point Representation



- Another problem with our system is that we have made no allowances for negative exponents. We have no way to express 0.5 (=2 $^{-1}$)! (Notice that there is no sign in the exponent field.)

All of these problems can be fixed with no changes to our basic model.

Floating-Point Representation

- To resolve the problem of synonymous forms, we establish a rule that the first digit of the significand must be 1, with no ones to the left of the radix point.
- This process, called *normalization*, results in a unique pattern for each floating-point number.
  - In our simple model, all significands must have the form 0.1xxxxxxxx
  - For example, $4.5 = 100.1 \times 2^0 = 1.001 \times 2^2 = 0.1001 \times 2^3$. The last expression is correctly normalized.

In our simple instructional model, we use no implied bits.

Floating-Point Representation

- To provide for negative exponents, we will use a *biased exponent*.
- A bias is a number that is approximately midway in the range of values expressible by the exponent. We subtract the bias from the value in the exponent to determine its true value.
  - In our case, we have a 5-bit exponent. We will use 16 for our bias. This is called *excess-16* representation.
- In our model, exponent values less than 16 are negative, representing fractional numbers.

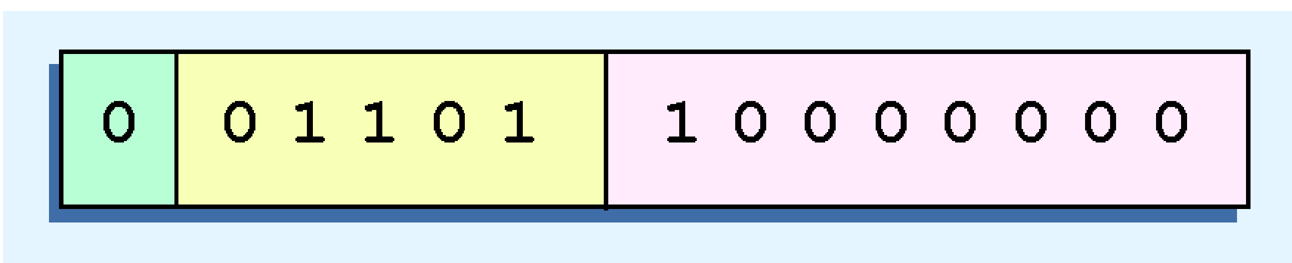
Floating-Point Representation

- Example:
  - Express 32_{10} in the revised 14-bit floating-point model.
- We know that $32 = 1.0 \times 2^5 = 0.1 \times 2^6$.
- To use our excess 16 biased exponent, we add 16 to 6, giving 22_{10} ($=10110_2$).
- So we have:

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

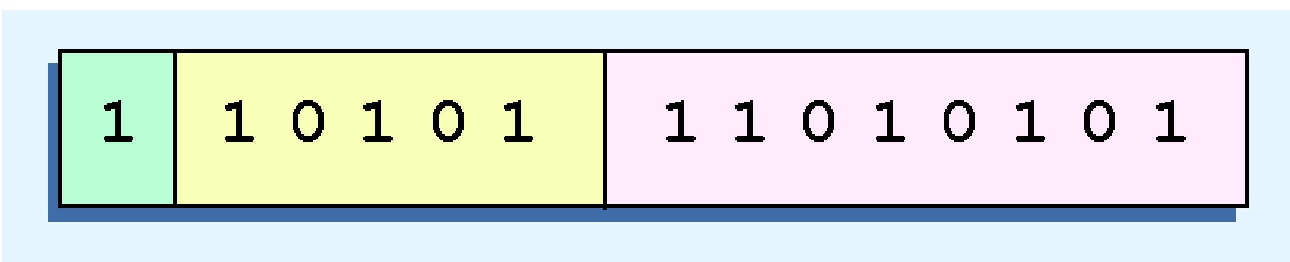
Floating-Point Representation

- Example:
  - Express 0.0625_{10} in the revised 14-bit floating-point model.
- We know that 0.0625 is 2^{-4}. So in (binary) scientific notation $0.0625 = 1.0 \times 2^{-4} = 0.1 \times 2^{-3}$.
- To use our excess 16 biased exponent, we add 16 to -3, giving 13_{10} ($=01101_2$).



Floating-Point Representation

- Example:
  - Express -26.625_{10} in the revised 14-bit floating-point model.
- We find $26.625_{10} = 11010.101_2$. Normalizing, we have: $26.625_{10} = 0.11010101 \times 2^{5}$.
- To use our excess 16 biased exponent, we add 16 to 5, giving 21_{10} $(=10101_2)$. We also need a 1 in the sign bit.

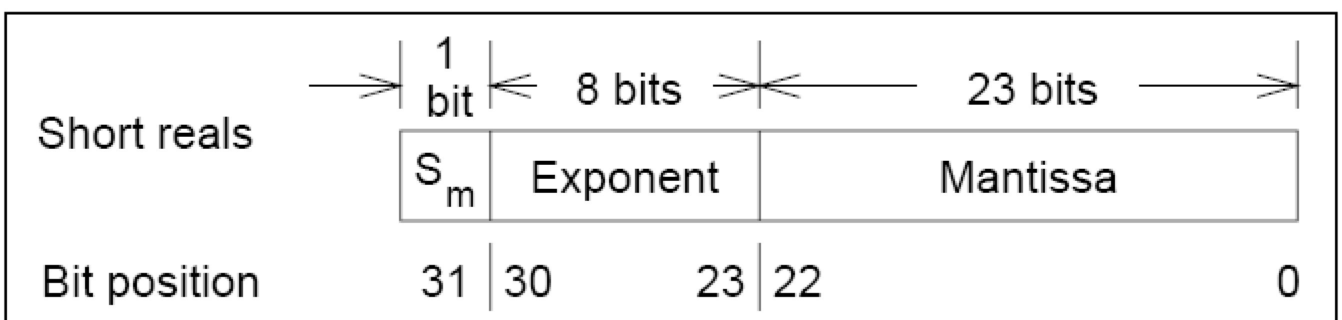


- Float x= -26.625;

- 14 bit
- Excess 16 method for expnent

Floating-Point Representation

- The IEEE has established a standard for floating-point numbers
- The IEEE-754 *single precision* floating point standard uses an 8-bit exponent (with a bias of 127) and a 23-bit significand.
- The IEEE-754 *double precision* standard uses an 11-bit exponent (with a bias of 1023) and a 52-bit significand.



IEEE-754 fp numbers - 1

32 bits:	1	8 bits	23 bits
	s	biased exp.	fraction

$$N = (-1)^{s} \times 1.\text{fraction} \times 2^{(\text{biased exp.} - 127)}$$

- Sign: 1 bit
- Mantissa: 23 bits
  - We "normalize" the mantissa by dropping the leading 1 and recording only its fractional part (why?)
- Exponent: 8 bits
  - In order to handle both +ve and -ve exponents, we add 127 to the actual exponent to create a "biased exponent":

Floating-Point Representation

- 2^{-127} => biased exponent = 0000 0000 (= 0)
- 2^{0} => biased exponent = 0111 1111 (= 127)
- 2^{+127} => biased exponent = 1111 1110 (= 254)

Example

Convert 22.625 to IEEE FPS format (single precision)

1. In scientific notation: $10110.101*2^{0}$

   Normalized form: $1.0110101*2^{4}$

2. Bias the exponent: 4 + 127 = 131

   $131_{10} = 10000011_{2}$

3. Place into the correct fields.

   S = 0

   E = 10000011

   F = 011 0101 0000 0000 0000 0000

0	10000011	01101010000000000000000
S	E	F

Example

Convert -83.7 to IEEE FPS format (single precision)

2*.7 = 1 + .4

2*.4 = 0 + .8

2*.8 = 1 + .6

2*.6 = 1 + .2

2*.2 = 0 + .4

2*.4 = 0 + .8

2*.8 = 1 + .6

2*.6 = 1 + .2

2*.2 = 0 + .4

. . .

Note: this is in single-precision floating point representation format but not FPS format

$-83.7_{10} = -1010011.101100110$

1. In scientific notation:

   $-1010011.101100110 * 10^0$

   Normalized form: $-1.01001110110\overline{0110} * 2^6$

2. Bias the exponent: 6 + 127 = 133

   $133_{10} = 10000101_2$

3. Place into the correct fields.

   S = 1

   E = 10000101

   F = 01001110110011001100110

1	10000101	01001110110011001100110
S	E	F

FP → Decimal.

Convert the following 32-bit binary number to its decimal floating point equivalent:

Sign	Exponent	Mantissa
1	01111101	010..0

FP→Decimal …

Step 1: Extract the biased exponent and unbias it

Biased exponent = 01111101_2 = 125_{10}

Unbiased Exponent: 125 – 127 = -2

FP → Decimal

Step 2: Write Normalized number in the form:

$$\mathbf{1.}\ \underline{\quad\text{Mantissa}\quad\quad\quad} \times 2^{\overset{\text{Exponent}}{----}}$$

For our number:

$$-1.01 \times 2^{-2}$$

FP → Decimal

Step 3: Denormalize the binary number from step 2 (i.e. move the decimal and get rid of (x 2^n) part):

-0.0101_2 (negative exponent – move left)

Step 4: Convert binary number to the FP equivalent (i.e. Add all column values with 1s in them)

$$-0.0101_2 = -(0.25 + 0.0625)$$

$$= -0.3125_{10}$$

Program I Expect?

- Decimal number → Floating Point Representation

Step 1: *Convert the real number to binary.*

1a: Convert the integer part to binary

1b: Convert the fractional part to binary

1c: Put them together with a binary point.

Step 2: *Normalize the binary number.*

Move the binary point left or right until there is only a single 1 to the left of the binary point while adjusting the exponent appropriately. You should increase the exponent value by 1 if the binary point is moved to the left by one bit position; decrement by 1 if moving to the right.

Step 3: *Convert the exponent to excess or biased form.*

For short reals, use 127 as the bias;

For long reals, use 1023 as the bias.

Step 4: *Separate the three components.*

Separate mantissa, exponent, and sign to store in the desired format.

Apply on 78.8125 D

Floating-Point Representation

- Both the 14-bit model that we have presented and the IEEE-754 floating point standard allow two representations for zero.
  - Zero is indicated by all zeros in the exponent and the significand, but the sign bit can be either 0 or 1.
- This is why programmers should avoid testing a floating-point value for equality to zero.
  - Negative zero does not equal positive zero.

Thanks!