Data Representation II

Sabah Sayed

Topics

1. Fractions Conversion
2. Signed Integer
3. Arithmetic Operations on Signed Integers
4. Storing Integers
5. Storing Fractions

Fractions Conversion

- Decimal to decimal (just for fun)

$$
\begin{array}{lrl}
3.14 \Rightarrow & 4 \times 10^{-2} & = 0.04 \\
 & 1 \times 10^{-1} & = 0.1 \\
 & 3 \times 10^{0} & = \underline{3} \\
 & & 3.14
\end{array}
$$

Fractions Conversion

- Binary to decimal

  Multiply each bit (in the fraction) by 2^{-n}, where *n* is the "weight" of the bit

  The weight is the position of the bit in the Fraction, starting from -1 on the left

  Add the results

Fractions Conversion

- Binary to decimal

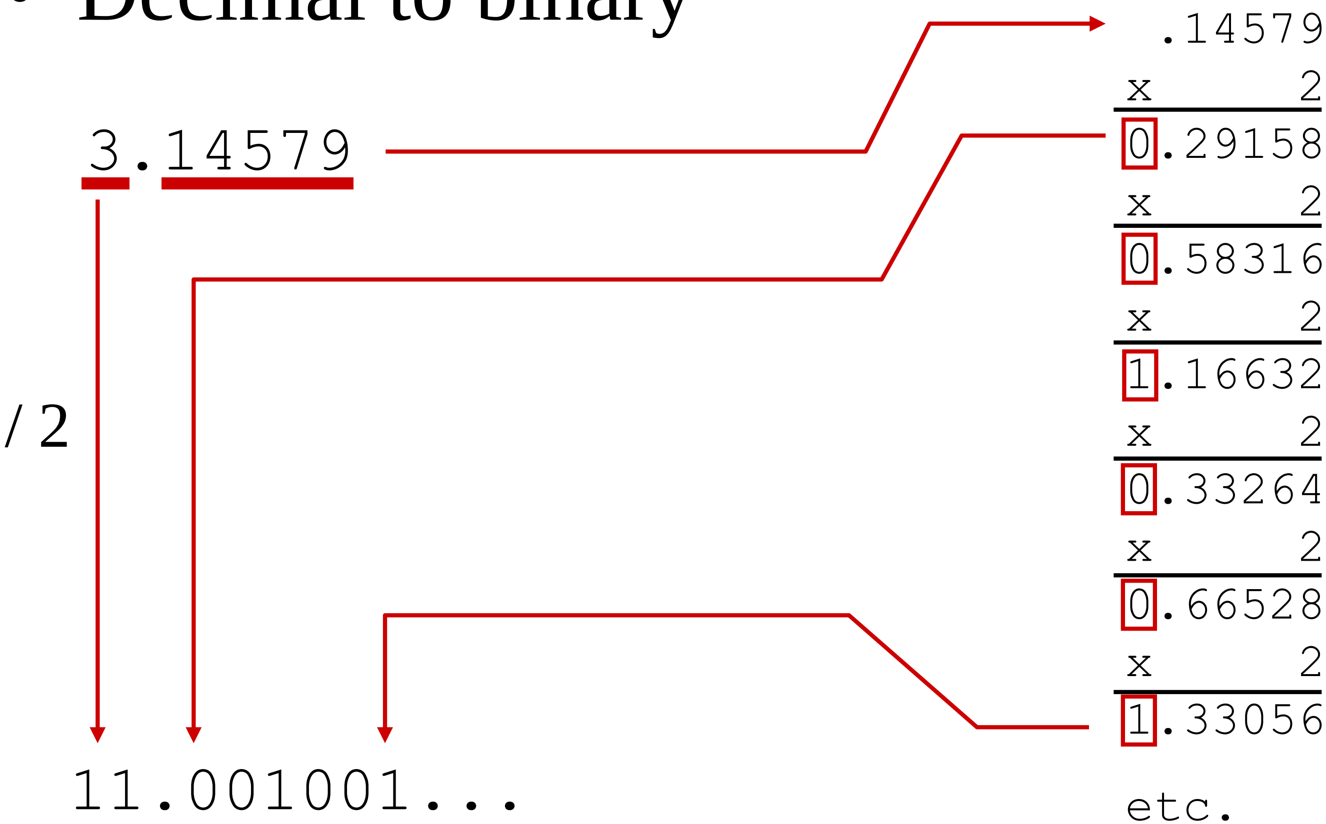
$$
10.1011 \Rightarrow \quad
\begin{array}{l}
1 \times 2^{-4} = 0.0625 \\
1 \times 2^{-3} = 0.125 \\
0 \times 2^{-2} = 0.0 \\
1 \times 2^{-1} = 0.5 \\
0 \times 2^{0} = 0.0 \\
1 \times 2^{1} = \underline{2.0} \\
\qquad\qquad 2.6875
\end{array}
$$

Fractions Conversion

- Value of Base B to decimal (general rule)
  - Multiply each digit (in the fraction) by B^{-n}, where *n* is the “weight” of the digit
  - The weight is the position of the digit in the Fraction, starting from -1 on the left
  - Add the results

Fractions Conversion

- Decimal to binary



Fractions Conversion

- Decimal Value to Base B (general rule)
  - Multiply the fraction by B, keep track of the remainder
  - First remainder is the leftmost digit in the fraction, … etc.

Fractions Conversion

- Can you guess the rule for fraction conversion between:
  - Binary and Octal ?
  - Binary and Hexadecimal ?
- Same as explained in previous lecture, but maintain the position of the fraction

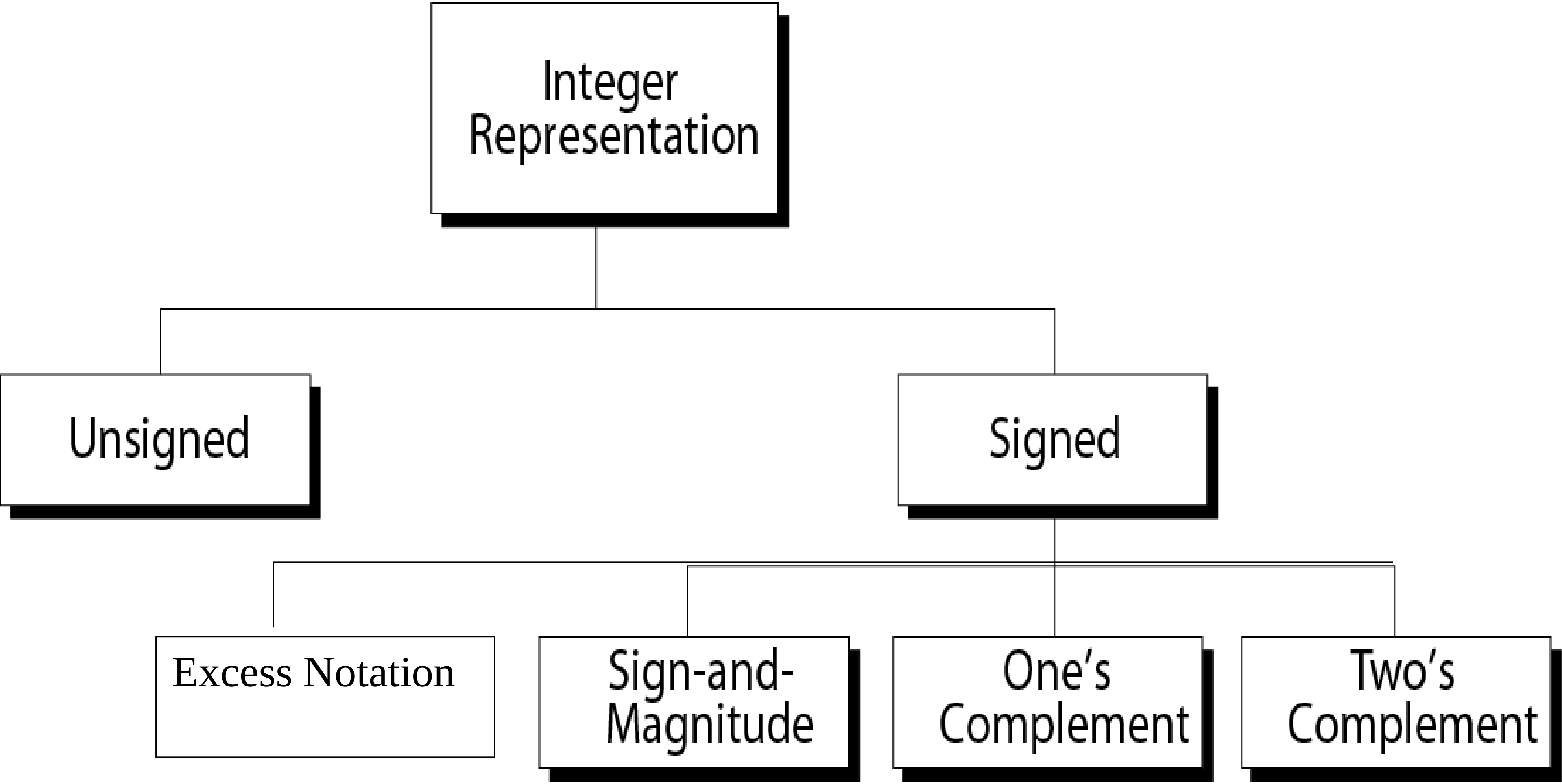
Exercise

Decimal	Binary	Octal	Hexa-decimal
29.8			
	101.1101		
		3.07	
			C.82

Exercise … Answer

Decimal	Binary	Octal	Hexa-decimal
29.8	11101.110011…	35.63…	1D.CC…
5.8125	101.1101	5.64	5.D
3.109375	11.000111	3.07	3.1C
12.5078125	1100.10000010	14.404	C.82

Binary Integers Representations



Sign-and-Magnitude

- The easiest representation
- The **leftmost bit** represents the sign of the number.

`0 if positive and 1 if negative`

Example

a) $+7_{10} = \underline{\mathbf{0}}\ \underline{0}\ \underline{0}\ \underline{0}\ \underline{0}\ \underline{1}\ \underline{1}\ \underline{1}_2$

$(-7_{10} = \mathbf{1}0000111_2)$

b) $-10_{10} = \underline{\mathbf{1}}\ \underline{0}\ \underline{0}\ \underline{0}\ \underline{1}\ \underline{0}\ \underline{1}\ \underline{0}_2$

$(+10_{10} = \mathbf{0}0001010_2)$

One's Complement

- To get the negative numbers, subtract each digit from the (base-1).
- In the binary system, this gives us *one's complement*.
- It almost like inverting the bits.

One's Complement

- **Positive numbers are same as in sign-and-magnitude**

  Example: $+5_{10} = 00000101_2$ (8 bit)

$\Rightarrow$ as in sign-and-magnitude representation

- For **negative numbers**, their representations are obtained by changing *bit 0 $\rightarrow$ 1 and 1 $\rightarrow$ 0* from their positive numbers

  Example: $+5_{10} = 00000101_2$

  $-5_{10} = 11111010_2$ (8 bit)

Example

Get the representation of one's complement (6 bit) for the following numbers:

i) $+7_{10}$

ii) -10_{10}

Solution :

$(+7) = 000111_2$

Solution:

$(+10)_{10} = 0\ 0\ 1\ 0\ 1\ 0_2$

So,

$(-10)_{10} = 1\ 1\ 0\ 1\ 0\ 1_2$

Two's Complement

- Similar to one's complement, its **positive number is same as sign-and-magnitude**
- Representation of its **negative number** is obtained by **adding 1 to the one's complement of the number**.

Two's Complement

- Another way to get the two's complement is copy and complement method.
- In this method you copy numbers from the right till the first one and then you reverse (complement the rest)
- Example:
  - 24 is **00011000**
  - -24 in two's complement:

    11101000

Example

Convert –5 into two’s complement representation and give the answer in 8 bits.

Solution:

- ✓ First, obtain +5 representation in 8 bits ⇒ 00000101_2
- ✓ Obtain one’s complement for –5

  ⇒ 11111010_2
- ✓ Add 1 to the one’s complement number:

  ⇒ $11111010_2 + 1_2 = 11111011_2$
- ✓ –5 in two’s complement is $\mathbf{11111011_2}$

Example

- Obtain representation of two's complement (6 bit) for the following numbers

i) $+7_{10}$

ii) -10_{10}

Solution:

$(+7) = \mathbf{0}00111_2$ (same as sign-magnitude)

Solution:

$(+10)_{10} = 001010_2$

$(-10)_{10} = 110101_2 + 1_2$
$= 110110_2$

Arithmetic Operations for Ones Complement, Twos Complement, Sign-and-Magnitude

Sign-and-Magnitude

`Z = X + Y`

There are few possibilities:

i. **If both numbers, X and Y are positive**

Just perform the addition operation

(Assume the number is represented ***in 6 bit***)

Example:

$5_{10} + 3_{10} = 000101_2 + 000011_2$

$= 001000_2$

$= 8_{10}$

ii. **If both numbers are negative**

Add |X| and |Y| and set the sign bit = 1 to the result, Z

Example: $(–3)_{10} + (–4)_{10}$

$= 1\mathbf{00011}_2 + 1\mathbf{00100}_2$

Only add the magnitude, i.e.: $00011_2 + 00100_2 = 00111_2$

Set the sign bit of the result (Z) to 1 (–ve)

$= \mathbf{1}00111_2$

$= –7_{10}$

iii. **If signs of both numbers differ**

There will be 2 cases:

a) | +ve Number | > | –ve Number |

Example: (+5) + (–3), (–2) + (+4)

- Set the sign bit of the –ve number to 0 (+ve), so that both numbers become +ve.
- Subtract the number of smaller magnitude from the number with a bigger magnitude

Sample solution:

Change the sign bit of the –ve number to +ve

$$
\begin{aligned}
(-2) + (+4) &= \mathbf{1}00010_2 + 000100_2 \\
&= 000100_2 - 000010_2 \\
&= 000010_2 = 2_{10}
\end{aligned}
$$

b) | –ve Number | > | +ve Number |

– Subtract the +ve number from the –ve number

Example: $(+3_{10}) + (-5_{10})$

$$= 000011_2 + 100101_2$$

$$= 100101_2 - 000011_2$$

$$= 100010_2$$

$$= -2_{10}$$

One’s Complement

- In one’s complement, it is easier than sign-and-magnitude
- Just perform the addition operation on the one’s complement representation of the numbers
- !! However a situation called ***Overflow*** might occur when addition is performed on the following categories:

1. **If both are negative numbers**
2. **If they have different signs and**

   |+ve Number| > | –ve Number|

Overflow => the addition result exceeds the number of bits that was fixed

1. **Both are –ve numbers**

Example: $(-3_{10}) + (-4_{10})$

Solution:

- Convert -3_{10} and -4_{10} into one's complement representation:

$+3_{10} = 000011_2$ (6 bits)

$-3_{10} = 111100_2$

$+4_{10} = 000100_2$ (6 bits)

$-4_{10} = 111011_2$

- Perform the addition operation

(-3_{10}) => 111100 (6 bit)

$+(-4_{10})$ => 111011 (6 bit)

1 110111 (7 bit)

Overflow occurs. This value (called "End-Around Carry" or EAC) needs to be added to the rightmost bit.

$$\begin{array}{r} 110111 \\ + \qquad 1 \\ \hline \mathbf{111000_2} \end{array} \quad = -7_{10}$$

the answer

2. **| +ve Number| > |–ve Number|**

This case will also cause an ***overflow***

Example: (–2) + 4 = (–2) + (+4)

Solution:

- Change both numbers into one's complement representation

$-2 = 111101_2$ $+4 = 000100_2$

- Add both numbers

$$
\begin{array}{lrl}
(-2_{10}) & \Rightarrow & 111101 \text{ (6 bit)} \\
\underline{+\ (+4_{10})} & \underline{\Rightarrow} & \underline{000100 \text{ (6 bit)}} \\
 & & \mathbf{1}\,000001 \text{ (7 bit)}
\end{array}
$$

overflow

- Add it to the rightmost bit

$$
\begin{array}{r}
000001 \\
+ \qquad 1 \\
\hline
\mathbf{000010_2}
\end{array}
= +2_{10}
$$

the answer

Note:

For cases other than 1 & 2 above, ***overflow*** does not occur and there will be no EAC and the need to perform addition to the rightmost bit does not arise

Two’s Complement

Addition operation in two’s complement is same with that of one’s complement, i.e. ***overflow*** occurs if:

1. **If both are negative numbers**
2. **If both have different signs and**

   |+ve Number| > |–ve Number|

BUT, you do the operation and ignore the EAC

Example

$$-3_{10} - 4_{10} = (-3_{10}) + (-4_{10})$$

Solution:

- Convert both numbers into two's complement representation

$+3_{10} = 000011_2$ (6 bit)

$-3_{10} = 111100_2$ (one's complement)

$-3_{10} = 111101_2$ (two's complement)

$-4_{10} = 111011_2$ (one's complement)

$-4_{10} = 111100_2$ (two's complement)

- Perform addition operation and ignore the EAC.

$$
\begin{array}{r l}
111101 & (-3_{10}) \\
\underline{111100} & (-4_{10}) \\
\mathbf{1}\underbrace{111001} & \\
\end{array}
$$

Ignore the EAC

The answer

$= 111001_2$ (two's complement)

$= -7_{10}$

Note

In two’s complement, EAC is ignored (**do not need** to be added to the rightmost bit, like that of one’s complement)

Storing Integers inside a Computer

- Memory and CPU registers are accessed in *words*.
- A word is usually 4 bytes or *32 bits* or it can be *64 bits*.
- An integer number is usually stored in one full word.

Largest integer we can store

- For unsigned numbers the range of numbers that can be stored in a certain number of bits ***n*** is:

  $0 \rightarrow 2^n-1$

- For signed numbers represented as sign and magnitude the range of numbers to be stored in ***n*** bits is:

  $-(2^{n-1}) \rightarrow (2^{n-1}-1)$

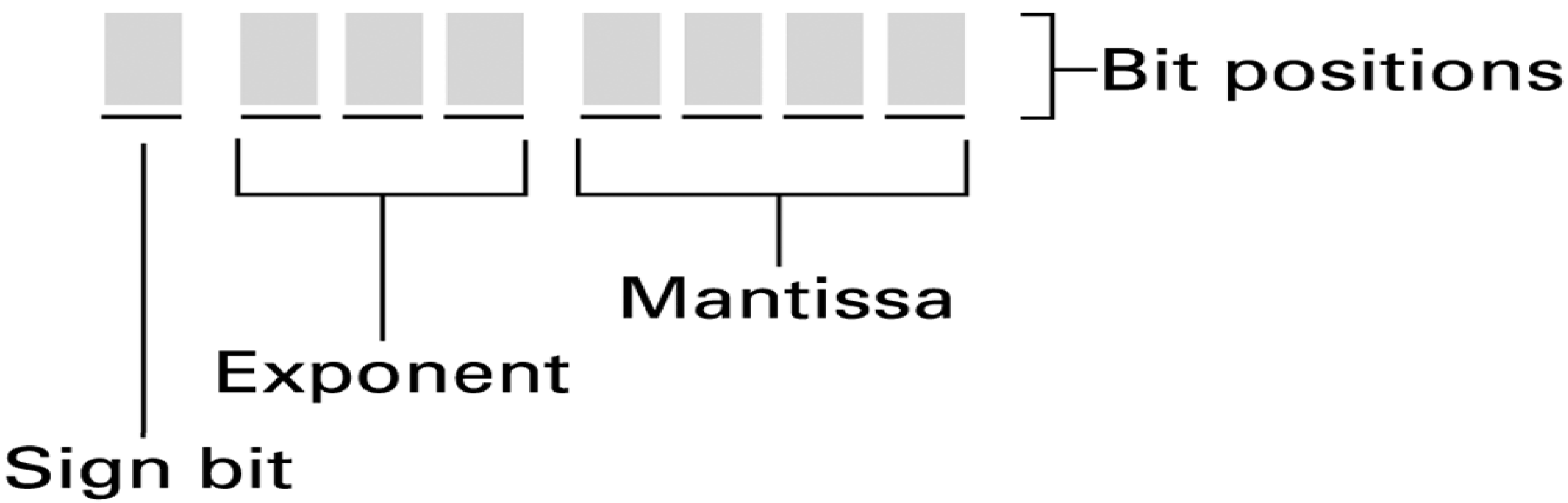
- 4-bits word unsigned:
  - 0000, ….., 1111 = 16 values
  - Max is number $15 = 2^4 - 1$
- 4-bits word signed magnitude:
  - Min / Max numbers (1000 / 0111)

    $-8 / +7 = 2^{4-1} / 2^{4-1} - 1$

Floating-point notation components

- Computers use a form of scientific notation for floating-point representation

- **Floating-point Notation** has a sign bit, a mantissa (significand) field, and an exponent field.



Convert a decimal number into floating point

1. Convert the absolute value of the number to **binary**,
2. Normalize the number to get the **mantissa**.
3. Find the exponent and express it as a **3 bits excess notation**
4. Set the sign bit, **1** for **-ve**, **0** for **+ve**, according to the sign of the original number
5. Write the **sign** in **1 bit**, **exponent** number in next **3 bits**, then write **mantissa** in next **4 bits**.

Excess Notation

- Mainly used in floating-point representation of fractions.
- Each value is represented by a bit pattern of the same length.
  - First select the pattern length
  - Write down all the different bit patterns of that length
  - Next, we observe that the first pattern with a 1 as its MSB most significant bit appears approximately halfway through the list.
  - We pick this pattern to represent zero

Bit pattern	Value represented
111	3
110	2
101	1
100	0
011	-1
010	-2
001	-3
000	-4

An excess notation system using bit patterns of length three (Excess 4)

Example

How to represent 2.75 ?

1. Convert 2.75 to binary

   $(2.75)_{10} = (10.11)_2$

2. Normalize the number to get mantiss

   $10.11=0.1011*2^2$ mantissa: 1011

3. Find the exponent and express it as a
   notation

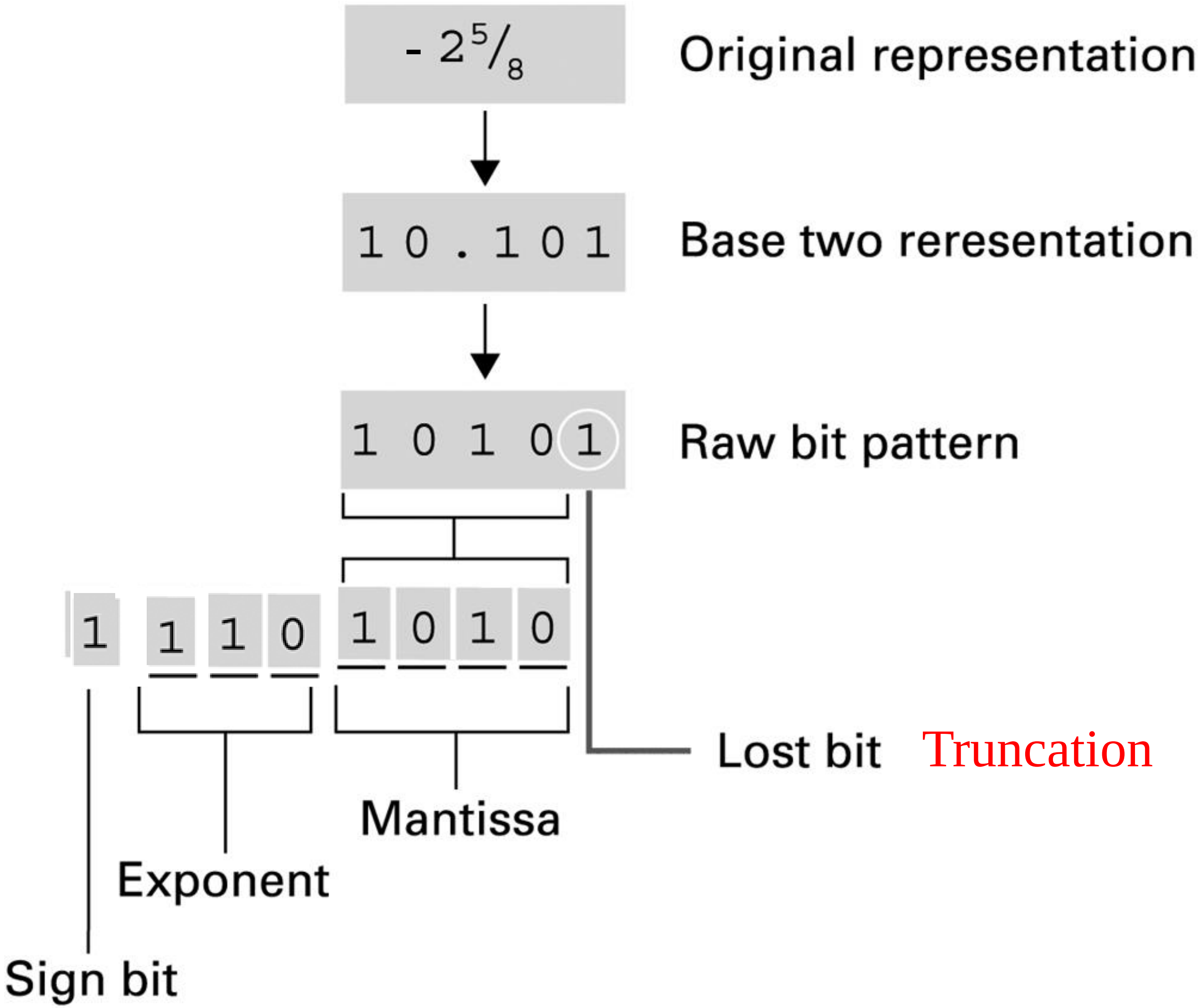
   Exponent = +2 → 110 (3-bit excess n

4. Find the floating point representation

   01101011

Bit pattern	Value represented
111	3
110	2
101	1
100	0
011	−1
010	−2
001	−3
000	−4

Encoding the value -2 ⅝



Problems when Representing Numeric Values

- Limitations of computer representations of numeric values
  - Overflow: occurs when a value is **too big** to be represented
  - Truncation: occurs when a value cannot be represented **accurately**