Foundations of Computer Science

Second Edition

BEHROUZ FOROUZAN

FIROUZ MOSHARRAF

Chapter 3
Data Storage

Outlines

- Data Types
- Storing numbers
- Storing text
- Storing audio
- Storing images
- Storing video

Objectives

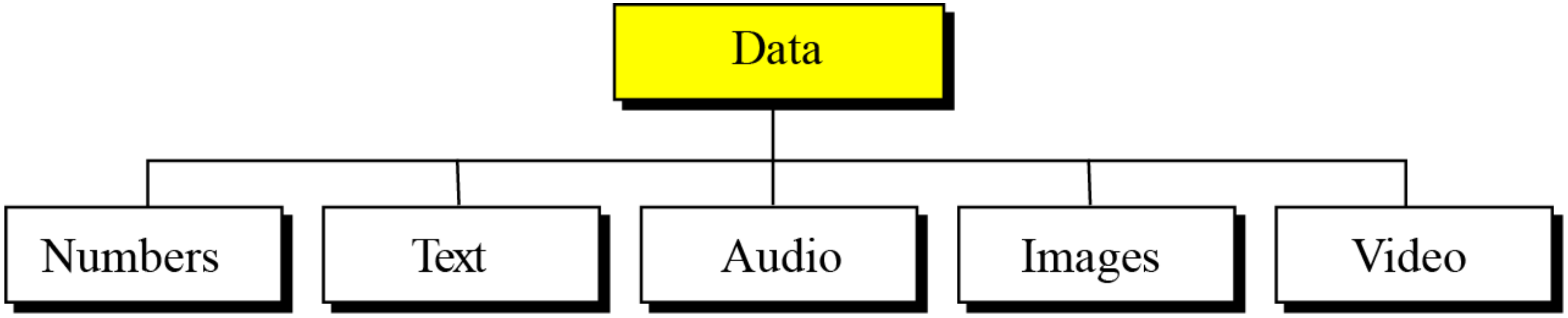
After studying this chapter, the student should understand

- Five different data types used in a computer.
- How different data is stored inside a computer.
- How integers are stored in a computer.
- How reals are stored in a computer.
- How text is stored in a computer.
- How audio is stored in a computer.
- How images are stored in a computer.
- How video is stored in a computer.

3.1 Data Types

Introduction

- Different types of data (Fig. 3.1)



The computer industry uses the term “multimedia” to define information that contains numbers, text, images, audio and video.

Data inside the computer

- All data types are transformed into a uniform representation when they are stored in a computer and transformed back to their original form when retrieved. This universal representation is called a **bit pattern**.

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

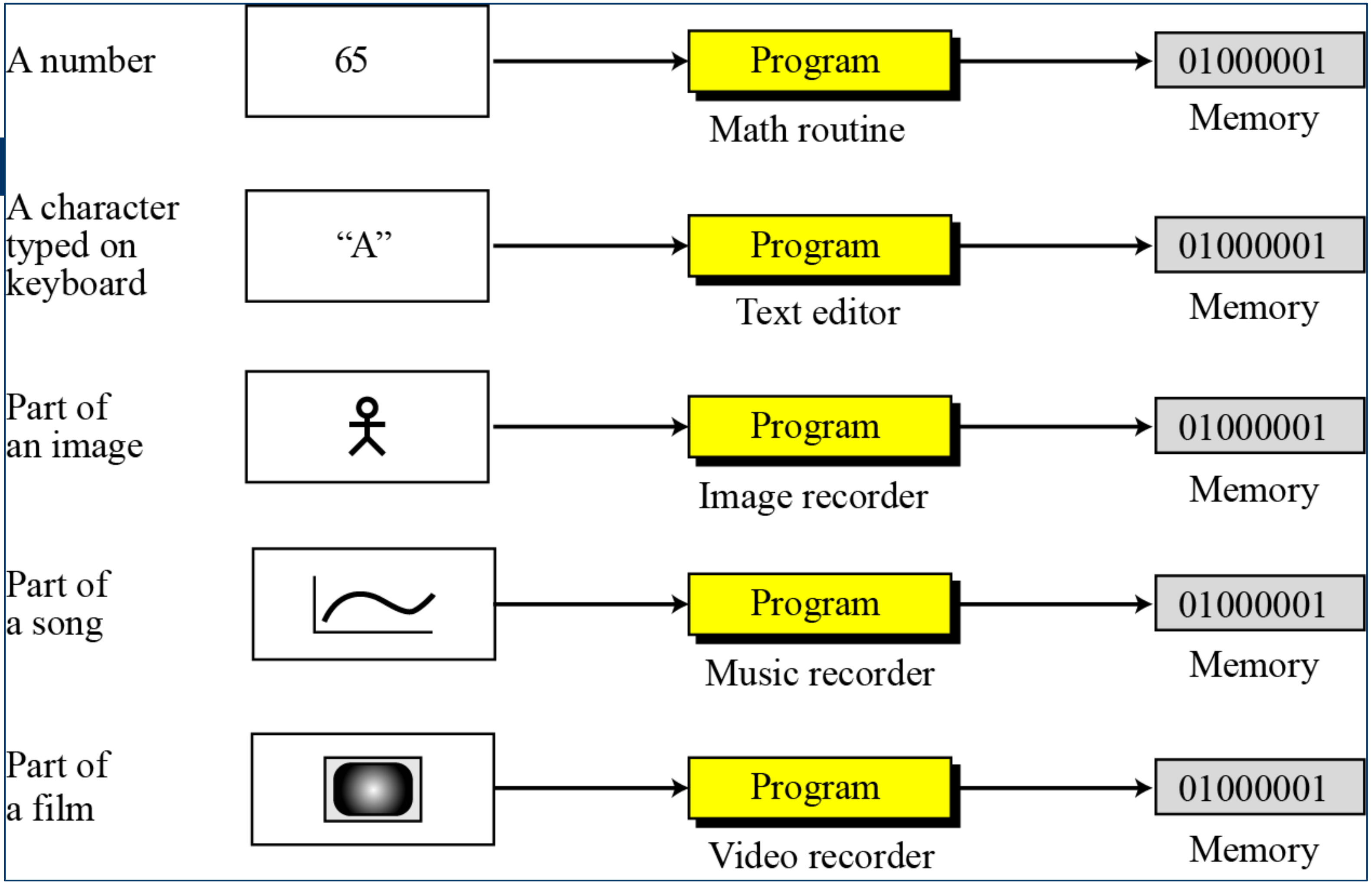


Figure 3.3 Storage of different data types

Data compression & Error detection and correction

- Data compression
  - To occupy less memory space, data is normally compressed before being stored in the computer.
  - Details in Chapter 15.
- Error detection and correction
  - Another issue related to data is the detection and correction of errors during transmission or storage.
  - Briefly in Appendix H.

3.2 Storing Numbers

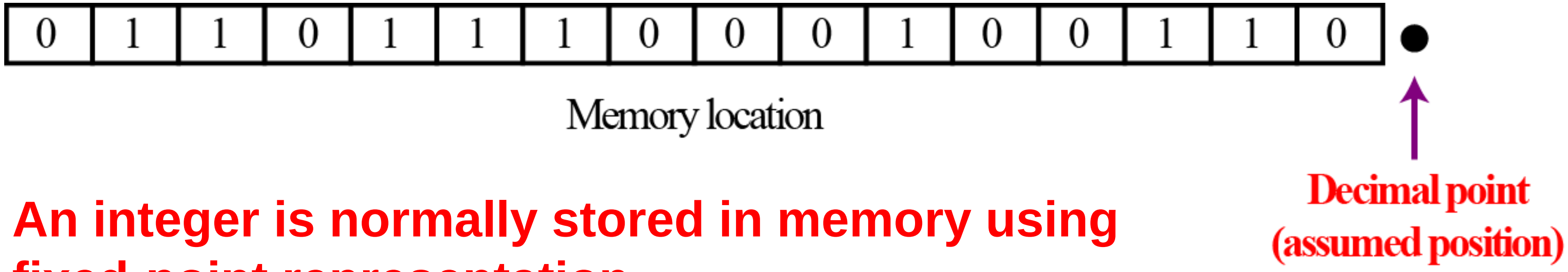
Overview

- A **number** is changed to the binary system before being stored in the computer memory, as described in Chapter 2.
- Problems:
  - How to store the sign of the number.
  - How to show the decimal point.

Storing Integers

- Integers (without a fractional part), for examples:

  134 and −125 are integers, but 134.23 and −0.235 are not
- An integer can be thought of as a number in which the position of the decimal point is fixed
- Fixed-point representation is used to store an integer (Fig. 3.4). The decimal point is assumed but not stored



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

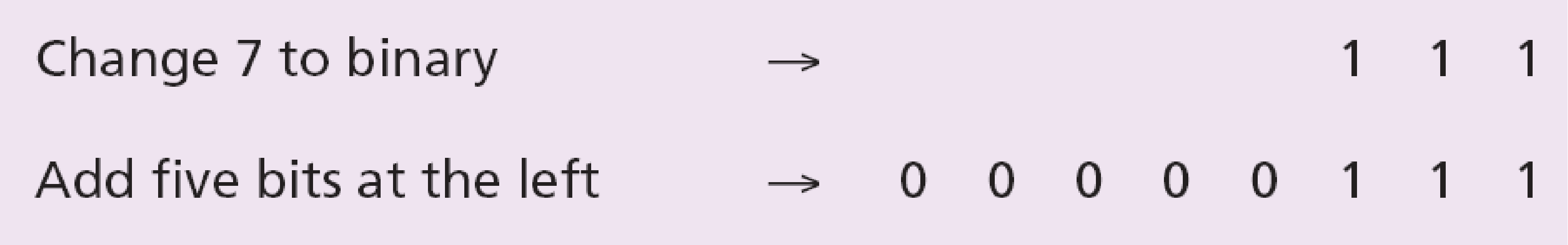
An integer is normally stored in memory using fixed-point representation.

Unsigned representation (1)

- An **unsigned integer** is an integer that can never be negative. Its range is between 0 and positive infinity.
- An input device stores an unsigned integer using the following steps:
  - The integer is changed to binary.
  - If the number of bits is less than *n*, 0s are added to the left.

Unsigned representation (2)

- Example 3.1 Store 7 in an 8-bit memory location using unsigned representation.
  - First change the integer to binary, $(111)_2$.
  - Add five 0s to make a total of eight bits, $(00000111)_2$.

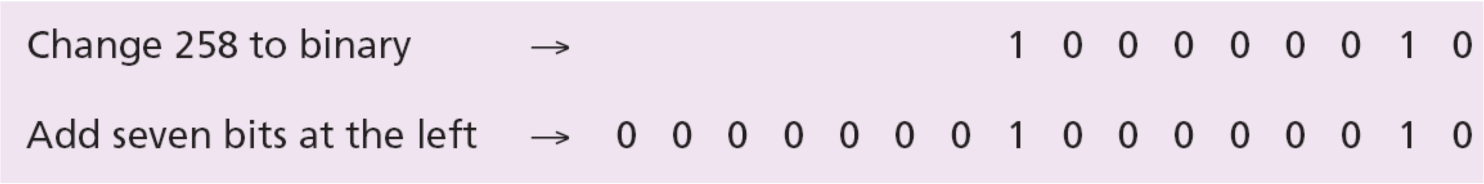


Change 7 to binary	→						1	1	1
Add five bits at the left	→	0	0	0	0	0	1	1	1

The integer is stored in the memory location, but the subscript is not stored in the computer

Unsigned representation (3)

- Example 3.2 store 258 in a 16-bit memory location.
  - First change the integer to binary $(100000010)_2$.
  - Add seven 0s to make a total of sixteen bits, $(0000000100000010)_2$.

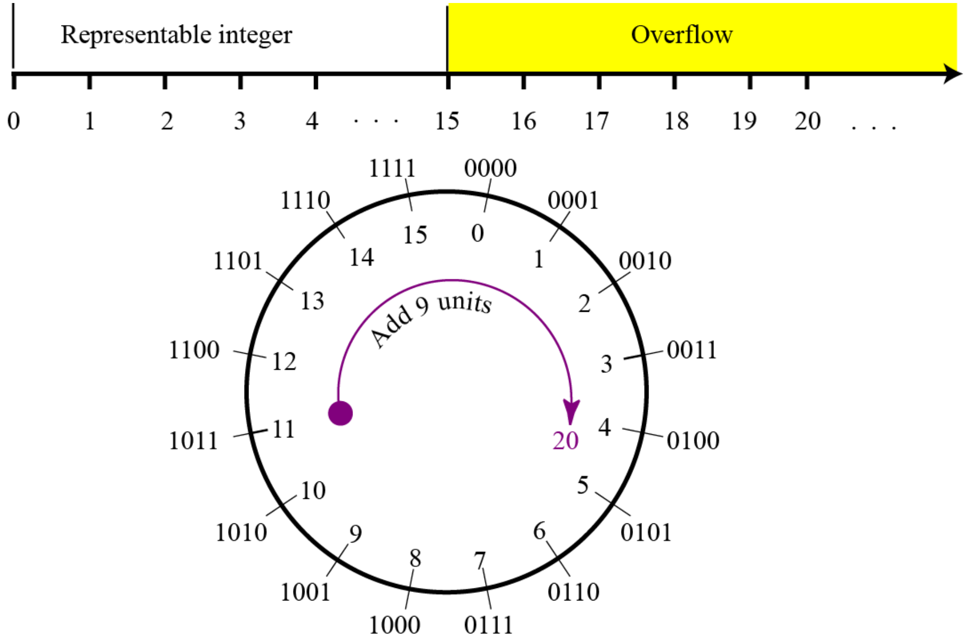


Change 258 to binary	→								1	0	0	0	0	0	0	1	0
Add seven bits at the left	→	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0

- Example 3.3
  - What is returned when retrieving the bit string 00101011 stored in memory as an unsigned integer?
  - The binary integer is converted to the unsigned integer 43.

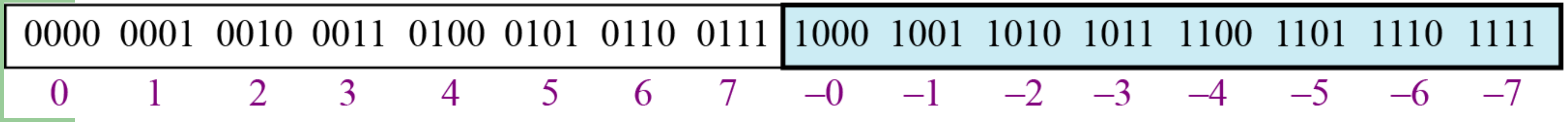
Unsigned representation (4)

- Figure 3.5 stores an integer that is larger than $2^4 - 1 = 15$ in a memory location that can only hold four bits. (Overflow)



Sign-and-magnitude representation (1)

- The range for unsigned integers (0 to $2^n - 1$) is divided into two equal sub-ranges.
  - The first half represents positive integers,
  - the second half, negative integers.

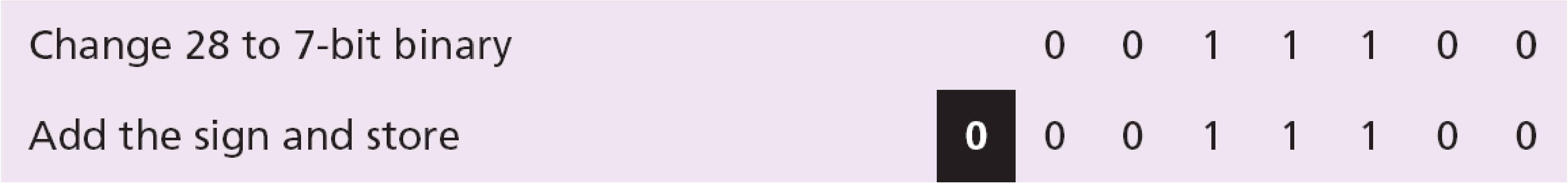


0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	1	2	3	4	5	6	7	–0	–1	–2	–3	–4	–5	–6	–7

The leftmost bit defines the sign. If it is 0, the integer is positive. If it is 1, the integer is negative.

Sign-and-magnitude representation (2)

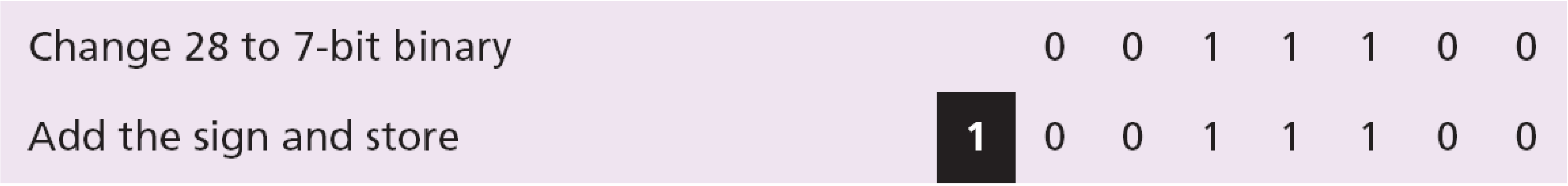
- Example 3.4 stores +28 in an 8-bit memory location
- Solution
  - Changing the integer to 7-bit binary
  - The leftmost bit is set to 0
  - The 8-bit number is stored.



Change 28 to 7-bit binary		0	0	1	1	1	0	0
Add the sign and store	0	0	0	1	1	1	0	0

Sign-and-magnitude representation (3)

- Example 3.5 stores -28 in an 8-bit memory location
- Solution
  - Changing the integer to 7-bit binary.
  - The leftmost bit is set to 1.
  - The 8-bit number is stored.



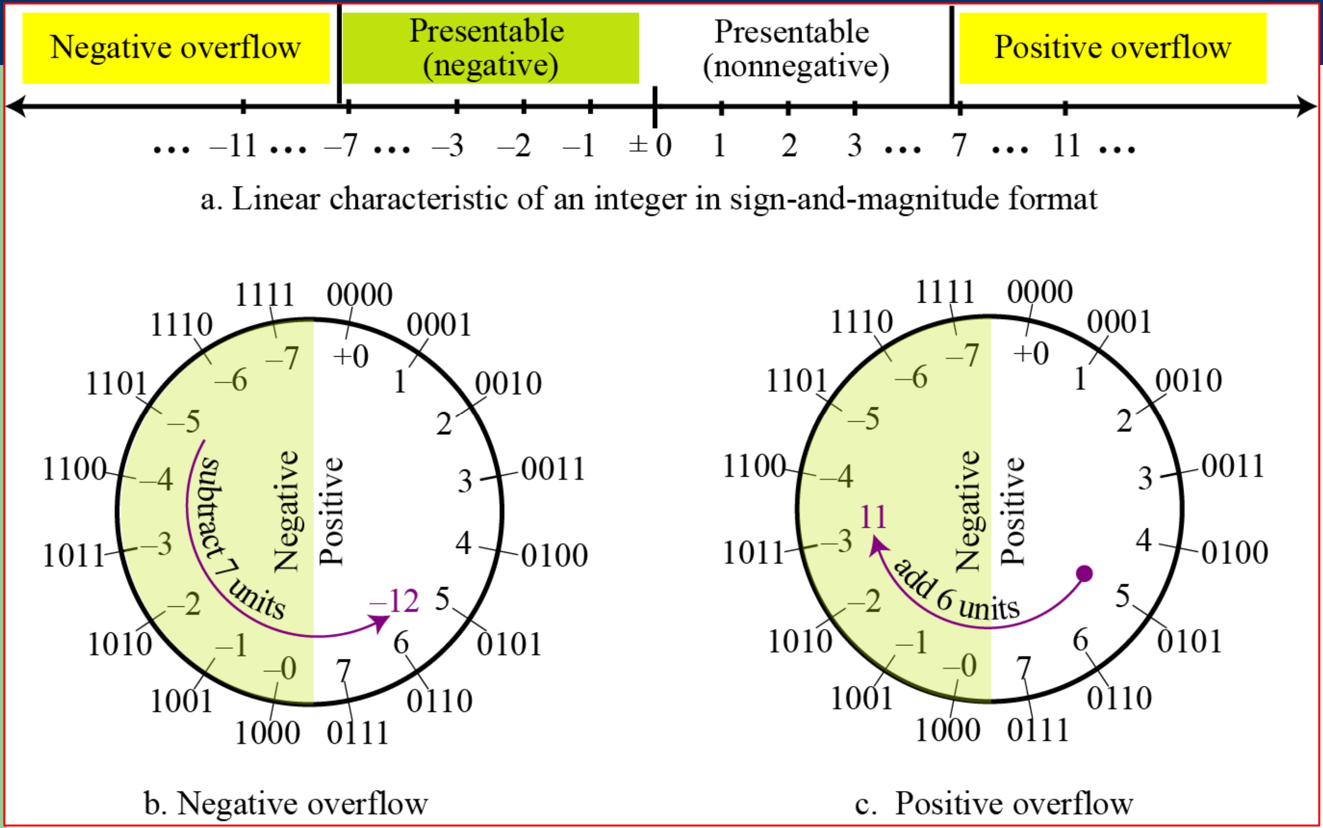
Change 28 to 7-bit binary		0	0	1	1	1	0	0
Add the sign and store	1	0	0	1	1	1	0	0

Sign-and-magnitude representation (4)

- Example 3.6 retrieves the integer that is stored as 01001101 in sign-and-magnitude representation.
  - Since the leftmost bit is 0, the sign is positive.
  - The rest of the bits (1001101) are changed to decimal (77).
  - After adding the sign, the integer is +77.

- Example 3.7 retrieves the integer that is stored as 10100001 in sign-and-magnitude representation.
  - Since the leftmost bit is 1, the sign is negative.
  - The rest of the bits (0100001) are changed to decimal (33).
  - After adding the sign, the integer is −33.

Sign-and-magnitude representation (5)



a. Linear characteristic of an integer in sign-and-magnitude format

b. Negative overflow

c. Positive overflow

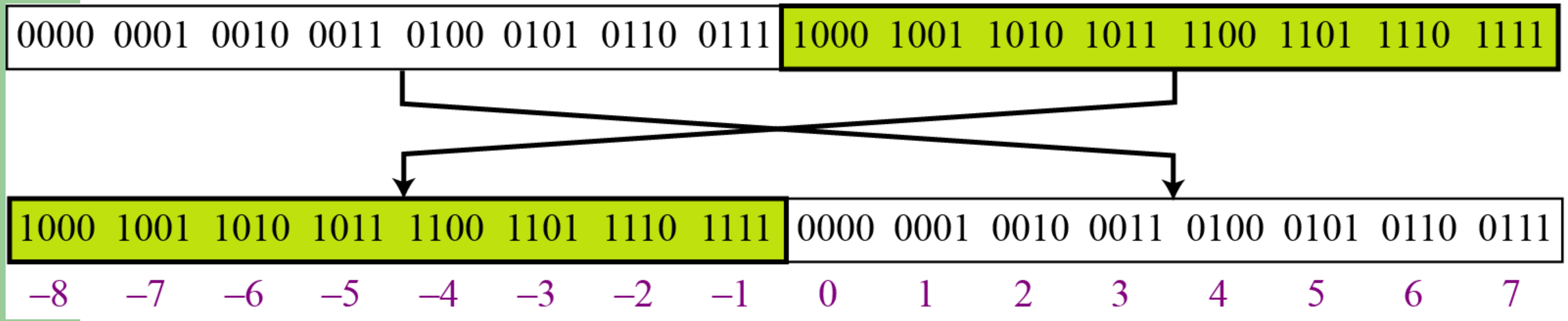
- Positive and negative overflow when storing an integer using a 4-bit memory location.

Two’s complement representation (1)

- Almost all computers use 2’s complement representation to store a signed integer in an *n*-bit memory location.
  - The available range for an unsigned integer of (0 to 2n – 1) is divided into two equal sub-ranges.
  - The first sub-range for nonnegative integers,
  - The second half for negative integers.

Two’s complement representation (2)

- The bit patterns are assigned to negative and nonnegative integers as shown in the Figure.



0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111

1000	1001	1010	1011	1100	1101	1110	1111	0000	0001	0010	0011	0100	0101	0110	0111
–8	–7	–6	–5	–4	–3	–2	–1	0	1	2	3	4	5	6	7

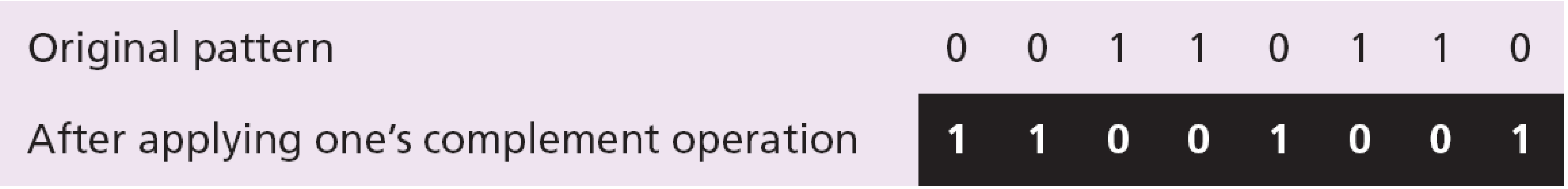
The leftmost bit defines the sign of the integer. If it is 0, the integer is positive. If it is 1, the integer is negative.

One’s Complementing (1)

- Taking the one’s complement of an integer.
- The operation can be applied to any integer, positive or negative.
- This operation simply reverses (flips) each bit. A 0-bit is changed to a 1-bit, a 1-bit is changed to a 0-bit.

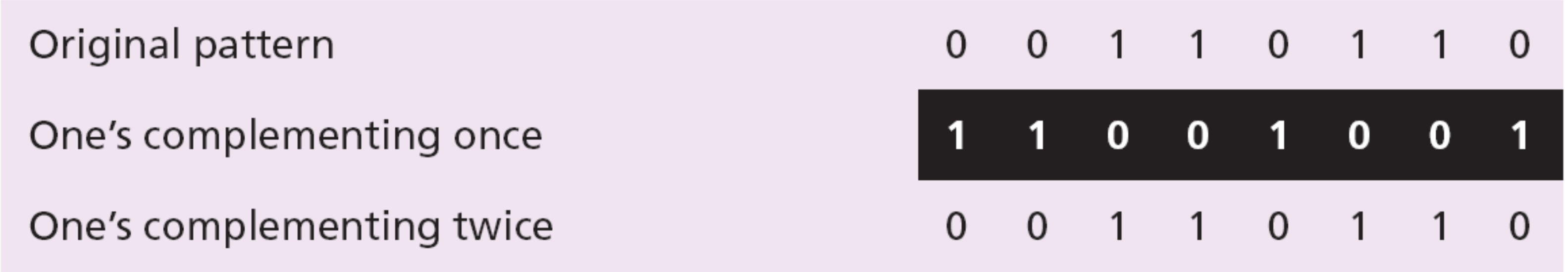
One’s Complementing (2)

- Example 3.8 shows the one’s complement of the integer 00110110.



Original pattern	0	0	1	1	0	1	1	0
After applying one’s complement operation	1	1	0	0	1	0	0	1

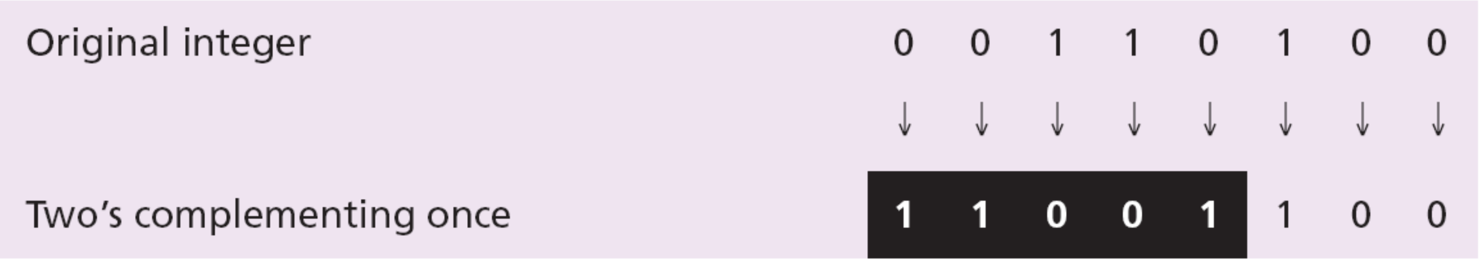
- Example 3.9 shows that we get the original integer if we apply the one’s complement operations twice.



Original pattern	0	0	1	1	0	1	1	0
One’s complementing once	1	1	0	0	1	0	0	1
One’s complementing twice	0	0	1	1	0	1	1	0

Two’s Complementing (1)

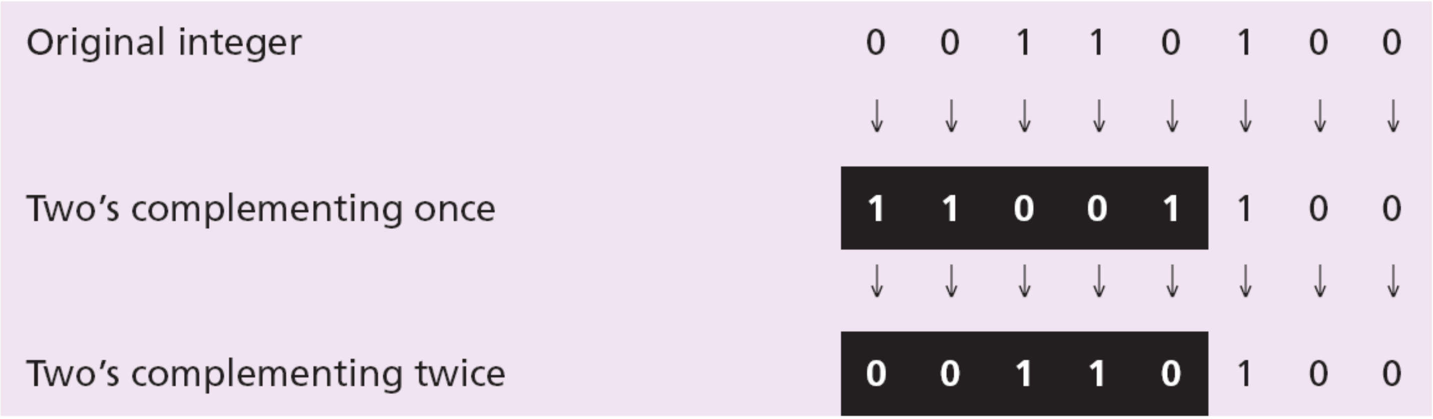
- Taking the two’s complement of an integer in binary.
- Two steps
  - First, copying bits from the right until a 1
  - Then, flipping the rest of the bits.
- Example 3.10 shows the two’s complement of the integer 00110100.



Original integer	0	0	1	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two’s complementing once	**1**	**1**	**0**	**0**	**1**	1	0	0

Two's Complementing (2)

- Example 3.11: we get the original integer if applying the 2's complement operation twice.

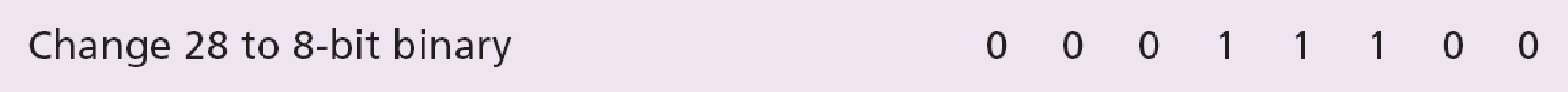


Original integer	0	0	1	1	0	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing once	1	1	0	0	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Two's complementing twice	0	0	1	1	0	1	0	0

An alternative way to take the 2's complement of an integer is to first take the 1's complement and then add 1 to the result.

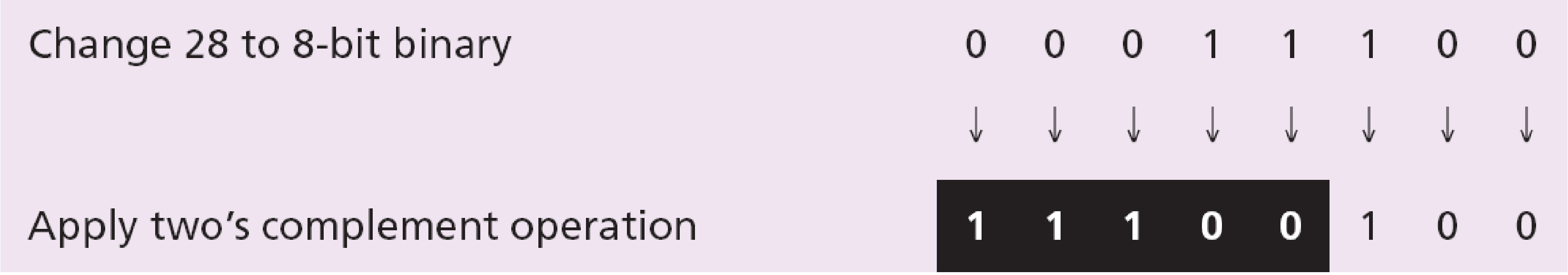
Two’s Complementing (3)

- Example 3.12 stores the integer 28 in an 8-bit memory location using 2’s complement representation.



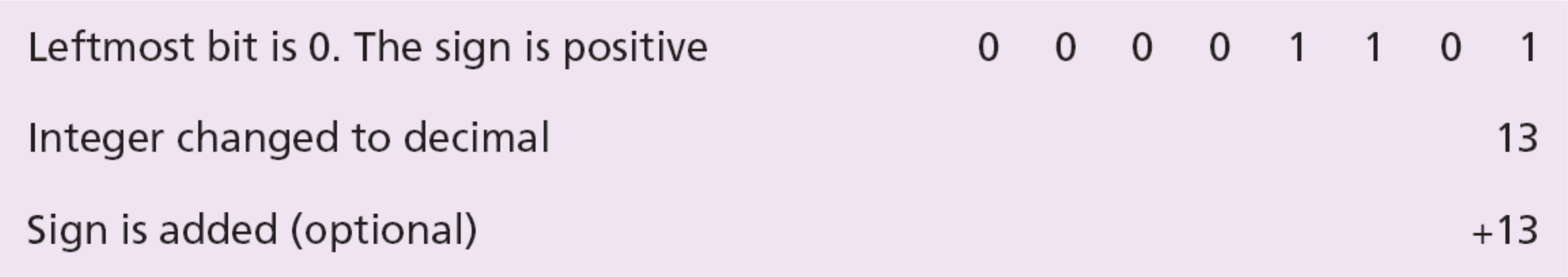
Change 28 to 8-bit binary	0	0	0	1	1	1	0	0

- Example 3.13 stores −28 in an 8-bit memory location using 2’s complement representation.



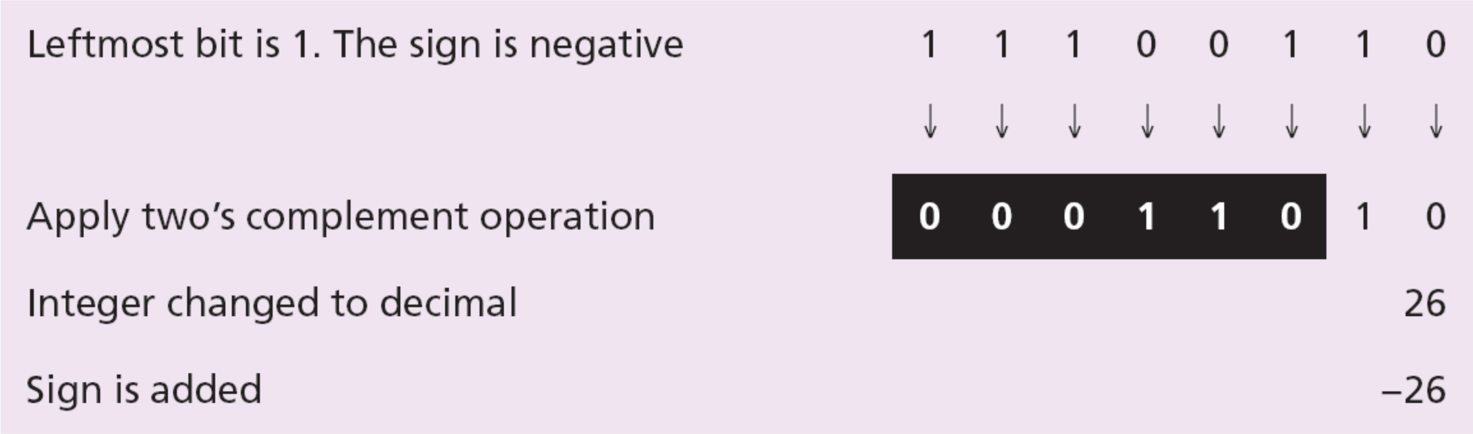
Change 28 to 8-bit binary	0	0	0	1	1	1	0	0
	↓	↓	↓	↓	↓	↓	↓	↓
Apply two’s complement operation	1	1	1	0	0	1	0	0

- Example 3.14 retrieves the integer that is stored as 00001101 in memory in two’s complement format.



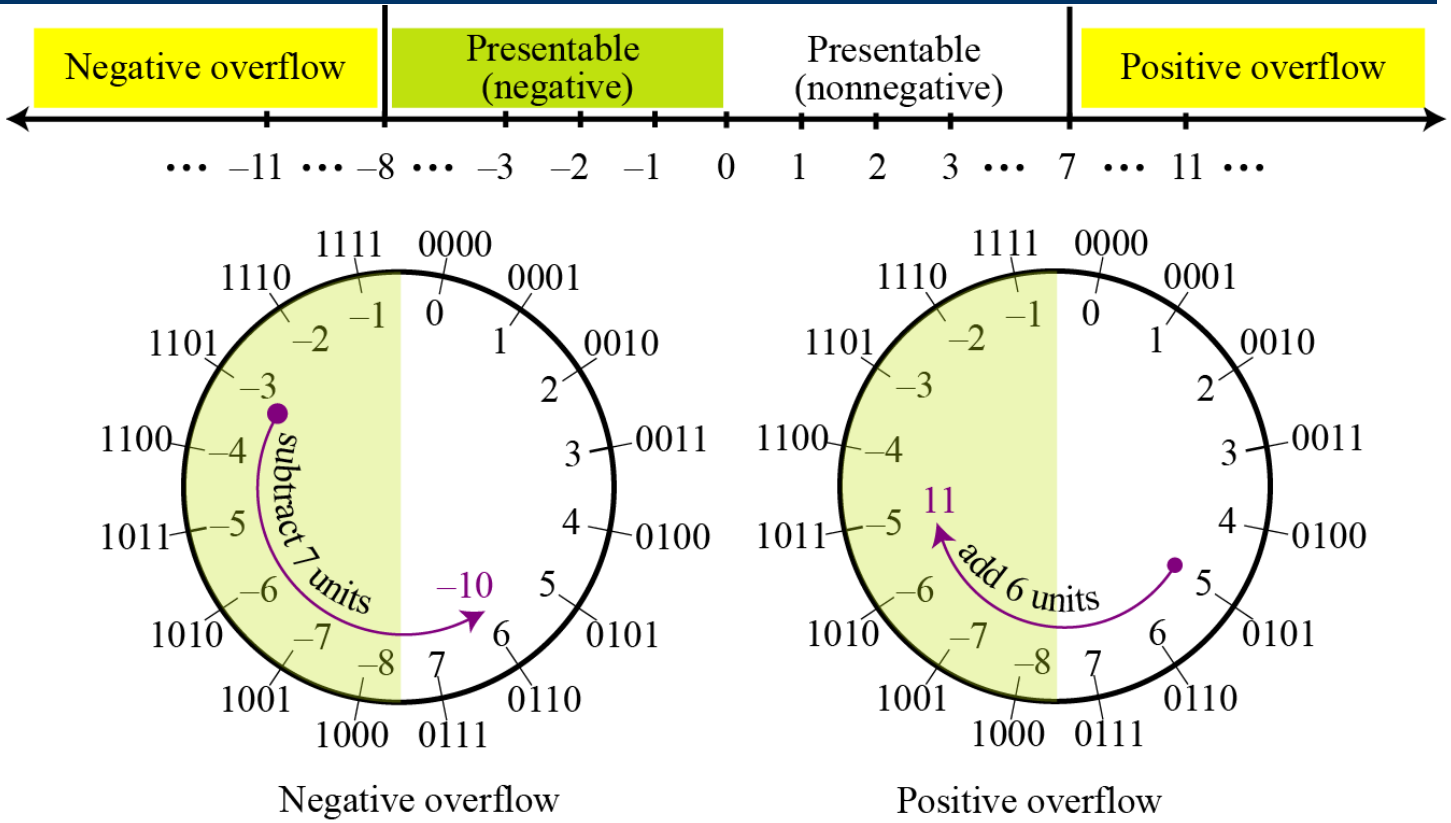
Leftmost bit is 0. The sign is positive	0	0	0	0	1	1	0	1
Integer changed to decimal								13
Sign is added (optional)								+13

- Example 3.15 retrieves the integer stored as 11100110 in memory using two’s complement format.



Leftmost bit is 1. The sign is negative	1	1	1	0	0	1	1	0
	↓	↓	↓	↓	↓	↓	↓	↓
Apply two’s complement operation	**0**	**0**	**0**	**1**	**1**	**0**	1	0
Integer changed to decimal								26
Sign is added								−26

Two's Complementing (4) Overflow



There is only one zero in two's complement notation.

Comparison

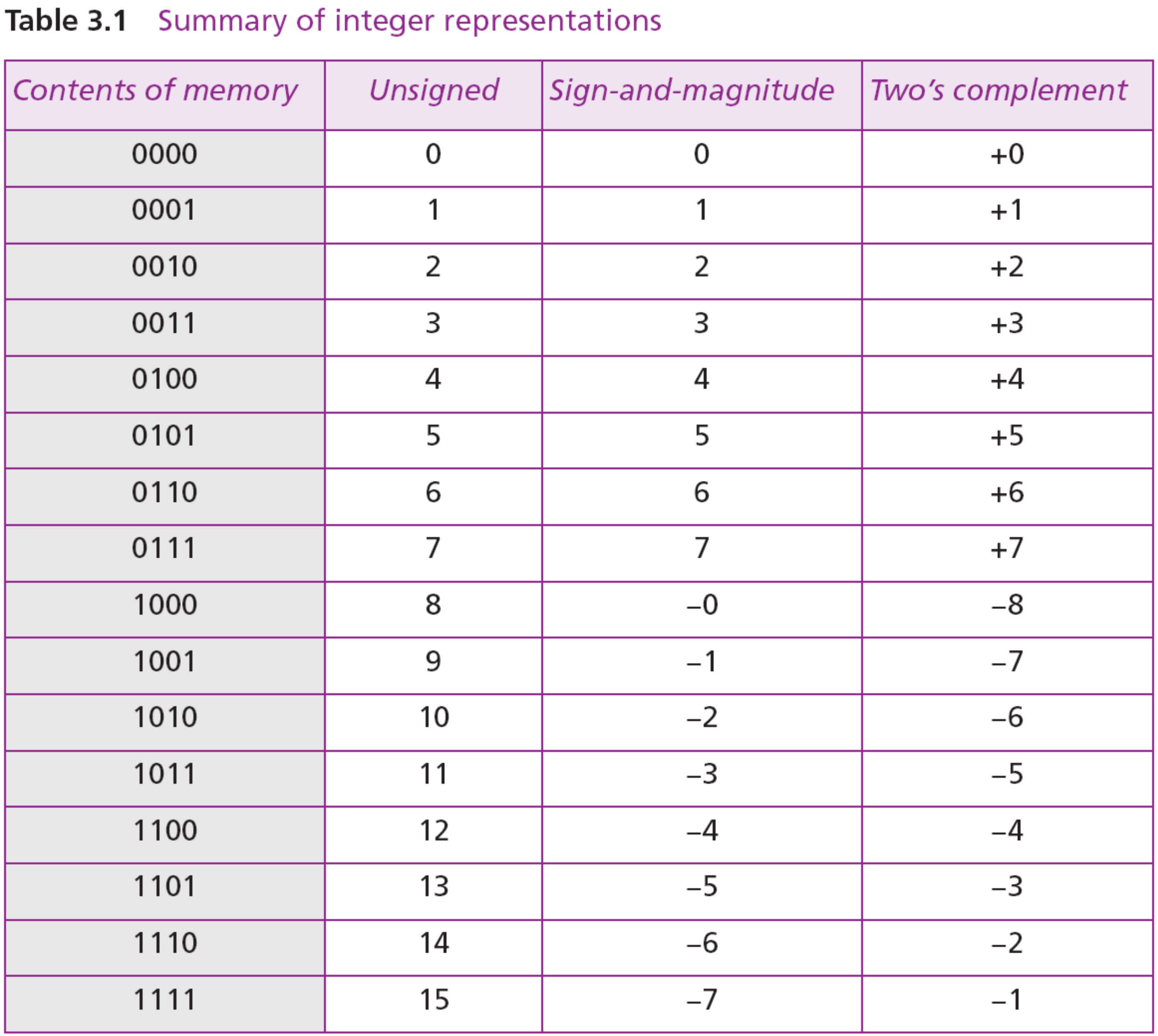


Table 3.1 Summary of integer representations

Contents of memory	*Unsigned*	*Sign-and-magnitude*	*Two's complement*
0000	0	0	+0
0001	1	1	+1
0010	2	2	+2
0011	3	3	+3
0100	4	4	+4
0101	5	5	+5
0110	6	6	+6
0111	7	7	+7
1000	8	–0	–8
1001	9	–1	–7
1010	10	–2	–6
1011	11	–3	–5
1100	12	–4	–4
1101	13	–5	–3
1110	14	–6	–2
1111	15	–7	–1

Storing Reals (1)

- A real is a number with an integral part and a fractional part.

- Although a fixed-point representation can be used to represent a real number, the result may not be accurate or it may not have the required precision. The next two examples explain why.

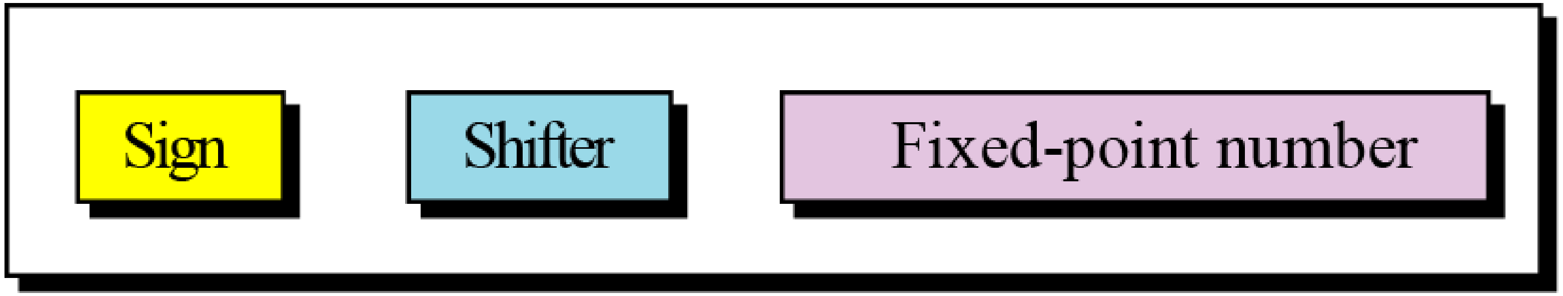
- Real numbers with very large integral parts or very small fractional parts should not be stored in fixed-point representation.

Storing Reals (2)

- Example 3.16 tries to represent a decimal number such as 1.00234 to a total of sixteen digits.
  - Two digits at the right of the decimal point
  - Fourteen digits at the left of the decimal point
  - The system stores the number as 1.00; The precision of a real number in this system is lost
- Example 3.17 tries to represent a decimal number such as 236154302345.00 to a total of sixteen digits.
  - Six digits to the right of the decimal point
  - Ten digits for the left of the decimal point,
  - The system stores the number as 6154302345.00; The accuracy of a real number in this system is lost

Floating-point representation (1)

- The solution for maintaining accuracy or precision is to use floating-point representation.
- Three parts: a sign, a shifter and a fixed-point number. (Figure 3.9)



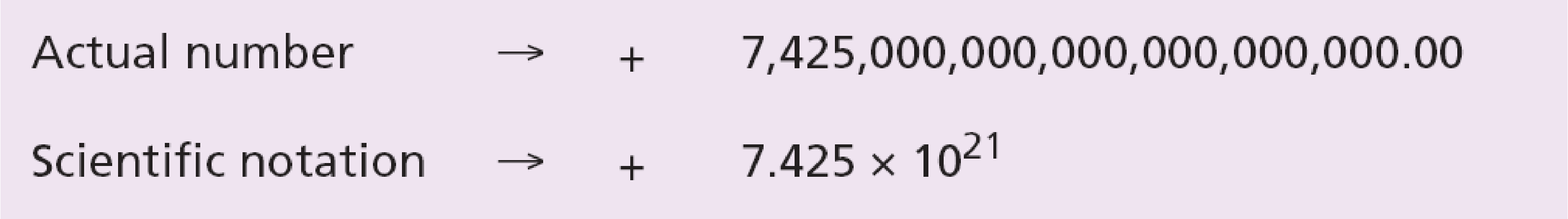
Floating-point pepresentation

Floating-point representation (2)

- Example 3.18 shows the decimal number in floating-point representation.

7,452,000,000,000,000,000,000.00

- The three parts are
  - The **sign** (+), the **shifter** (21), and the **fixed-point part** (7.425).
  - Note that the shifter is the exponent.



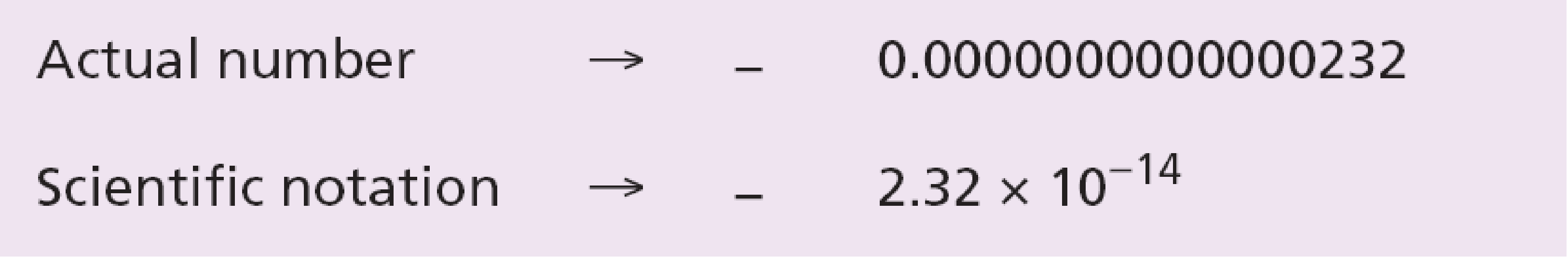
Actual number	→	+	7,425,000,000,000,000,000,000.00
Scientific notation	→	+	7.425×10^{21}

Floating-point representation (3)

- Example 3.19 shows the number in floating-point representation.

−0.0000000000000232

- The three parts are
  - The sign (-), the shifter (-14), and the fixed-point part (2.32)

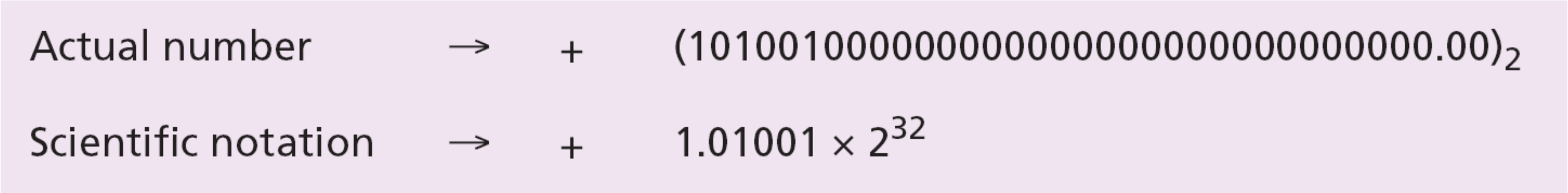


Actual number	→	−	0.0000000000000232
Scientific notation	→	−	2.32×10^{-14}

Floating-point representation (4)

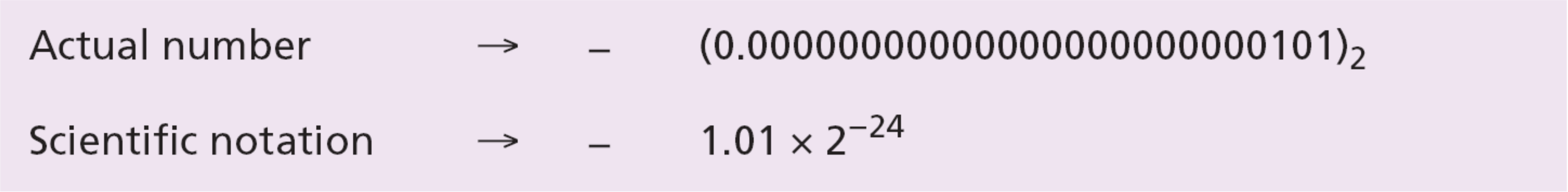
- Examples (3.20 and 3.21) show the numbers in floating-point representation.
  - Keeping only one digit to the left of the decimal point.

Eg. 3.20: $(101001000000000000000000000000000.00)_2$



Actual number	→	+	$(101001000000000000000000000000000.00)_2$
Scientific notation	→	+	1.01001×2^{32}

Eg. 3.21: $-(0.00000000000000000000000101)_2$

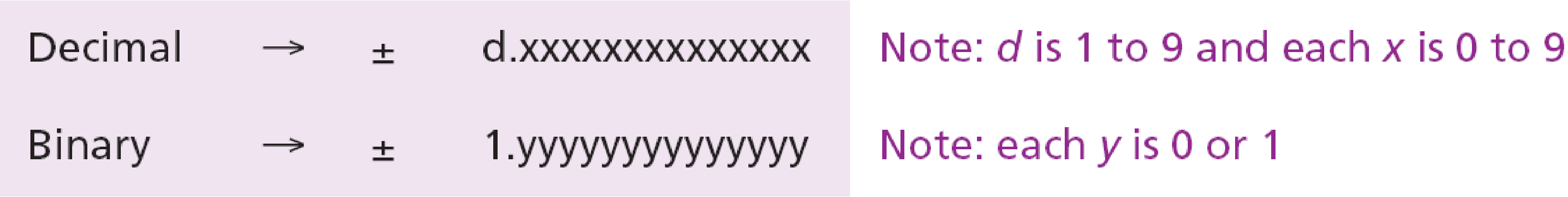


Actual number	→	−	$(0.00000000000000000000000101)_2$
Scientific notation	→	−	1.01×2^{-24}

Normalization (1)

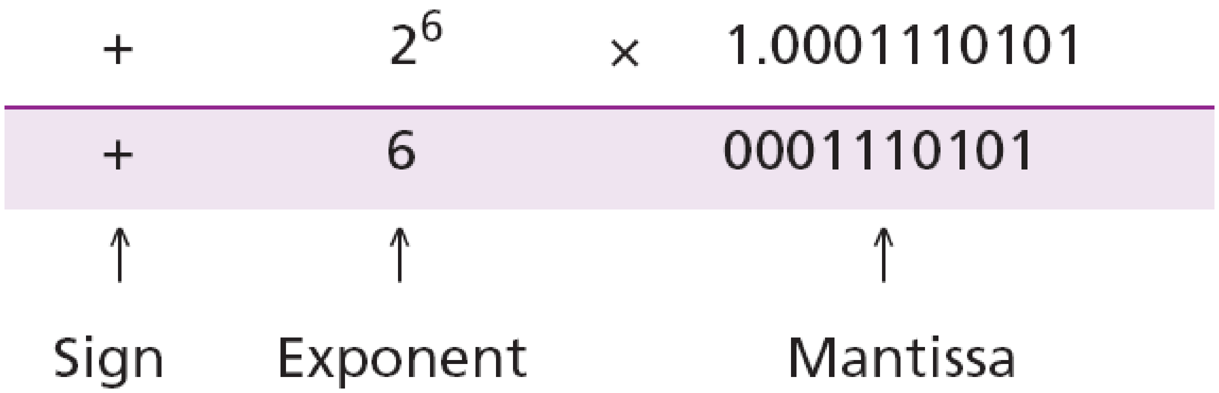
- Both the scientific method (for the decimal system) and the floating-point method (for the binary system) use only one non-zero digit on the left of the decimal point.

- In the decimal system this digit can be 1 to 9, while in the binary system it can only be 1.



Decimal	→	±	d.xxxxxxxxxxxxxx	Note: *d* is 1 to 9 and each *x* is 0 to 9
Binary	→	±	1.yyyyyyyyyyyyyy	Note: each *y* is 0 or 1

Normalization (2)



$+\ 2^6 \times 1.0001110101$

+	6	0001110101
Sign	Exponent	Mantissa

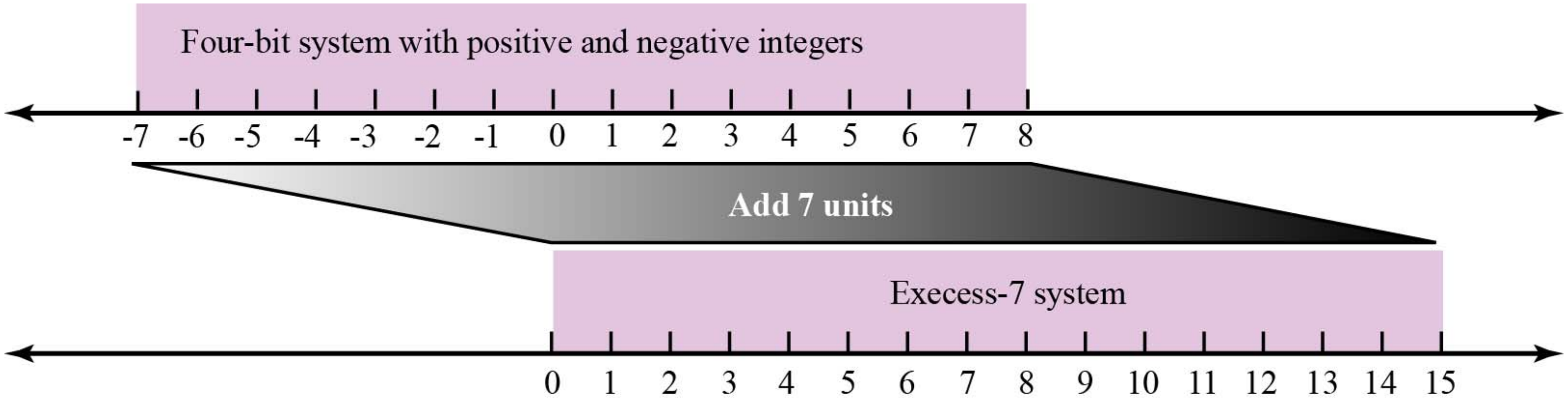
- Note that the point and the bit 1 to the left of the fixed-point section are not stored—they are implicit.
- The mantissa is a fractional part that, together with the sign, is treated like an integer stored in sign-and-magnitude representation.

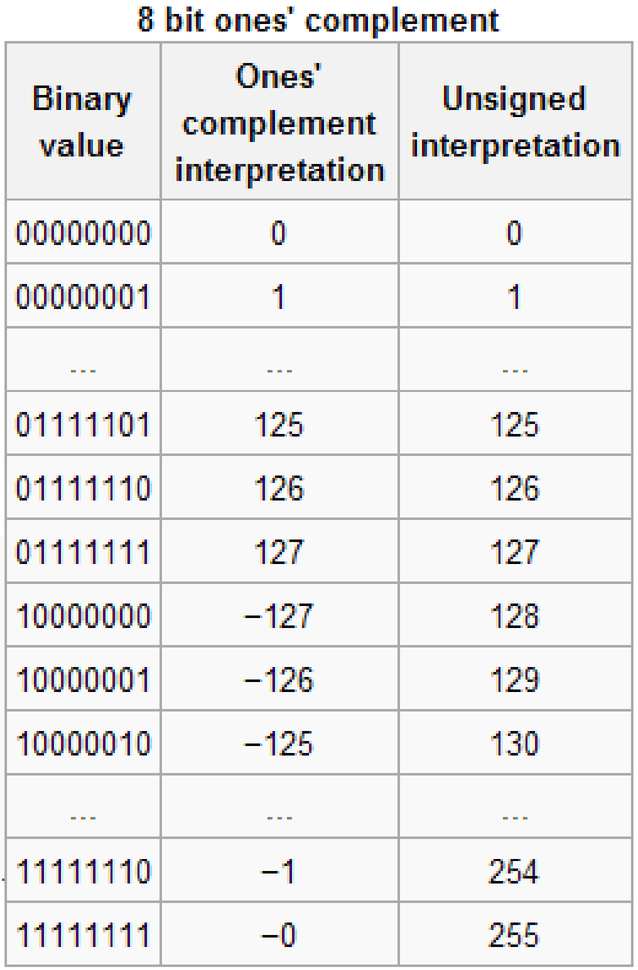
Excess System (1)

- Exponent
  - To show how many bits the decimal point should be moved to the left or right.
  - Being a signed number (使用Excess system)
- Excess System
  - Positive and negative integers are stored as unsigned integers.
  - A positive integer (called a **bias**) is added to each number to shift them uniformly to the non-negative side.
- The value of this bias is $2^{m-1} - 1$, where *m* is the size of the memory location to store the exponent.

Excess System (2)

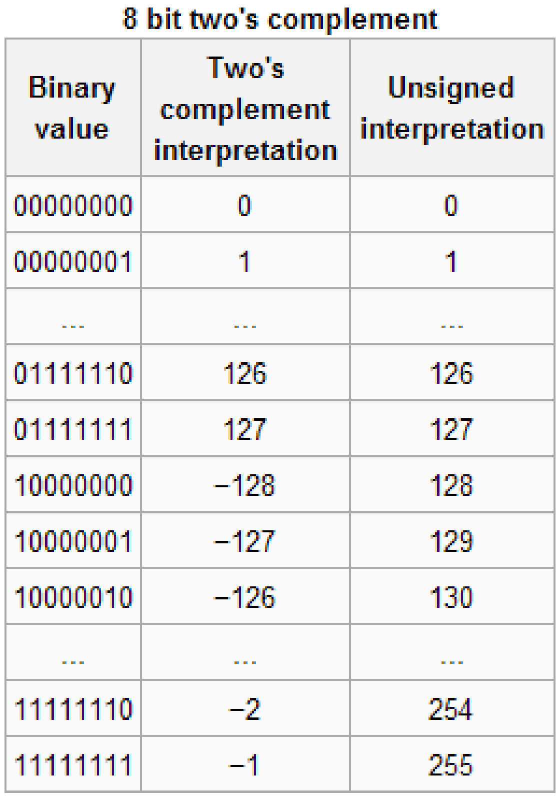
- Example 3.22 shows sixteen integers with 4-bit allocation.
  - By adding seven units to each integer, uniformly translating all integers to the right.
  - The new system is referred to as Excess-7, or biased representation with biasing value of 7.





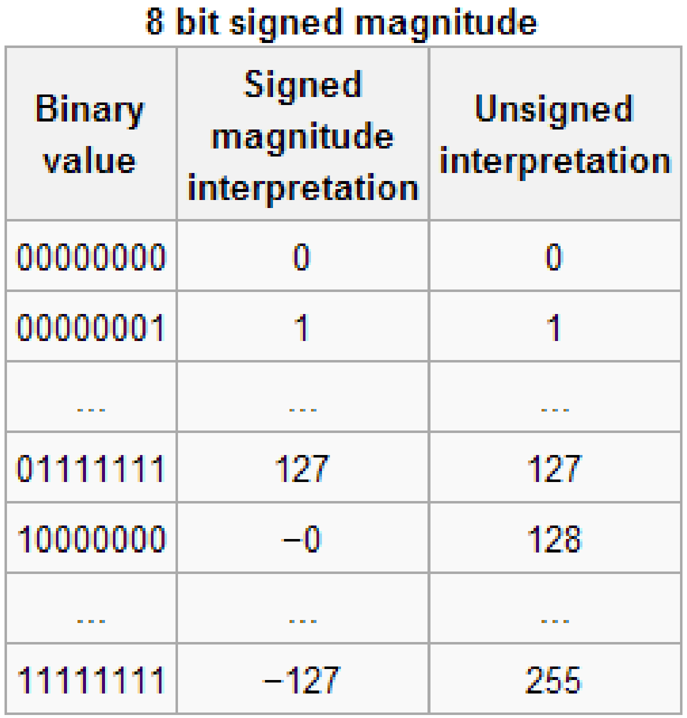
8 bit ones' complement

Binary value	Ones' complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
...	...	...
01111101	125	125
01111110	126	126
01111111	127	127
10000000	−127	128
10000001	−126	129
10000010	−125	130
...	...	...
11111110	−1	254
11111111	−0	255



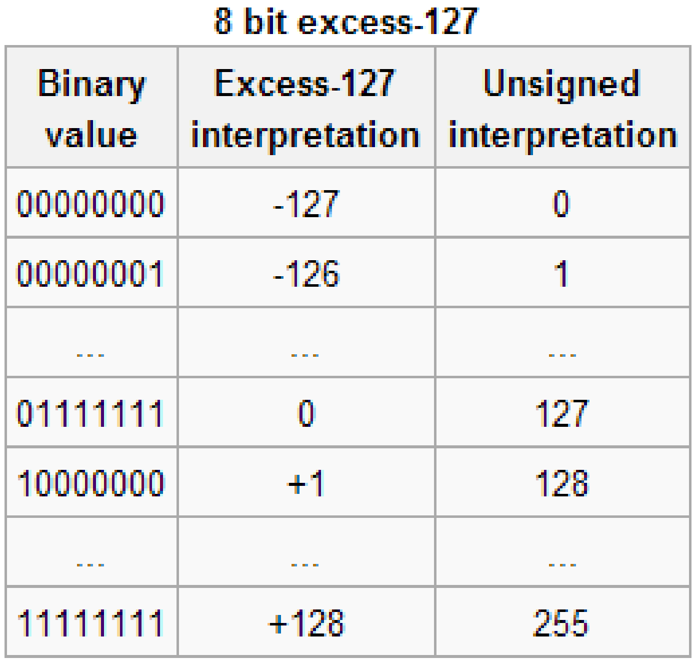
8 bit two's complement

Binary value	Two's complement interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
...	...	...
01111110	126	126
01111111	127	127
10000000	−128	128
10000001	−127	129
10000010	−126	130
...	...	...
11111110	−2	254
11111111	−1	255



8 bit signed magnitude

Binary value	Signed magnitude interpretation	Unsigned interpretation
00000000	0	0
00000001	1	1
...	...	...
01111111	127	127
10000000	−0	128
...	...	...
11111111	−127	255



8 bit excess-127

Binary value	Excess-127 interpretation	Unsigned interpretation
00000000	-127	0
00000001	-126	1
...	...	...
01111111	0	127
10000000	+1	128
...	...	...
11111111	+128	255

IEEE Standard

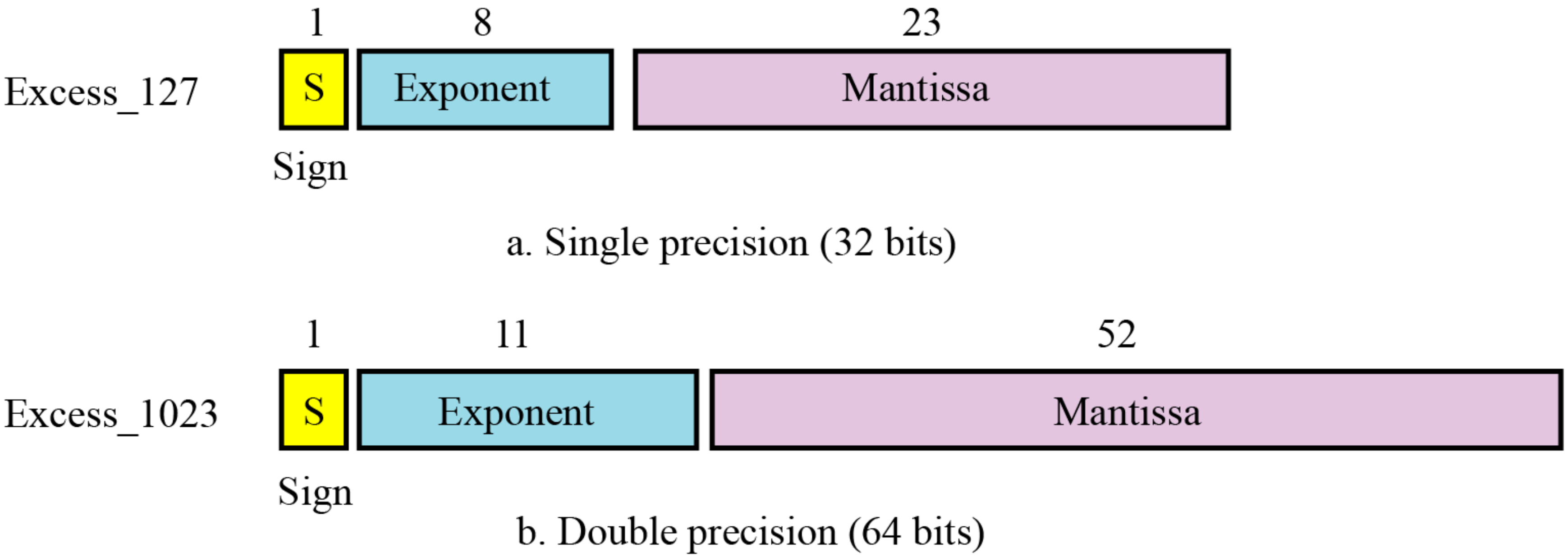


Figure 3.12 **IEEE standards for floating-point representation**

IEEE Specifications

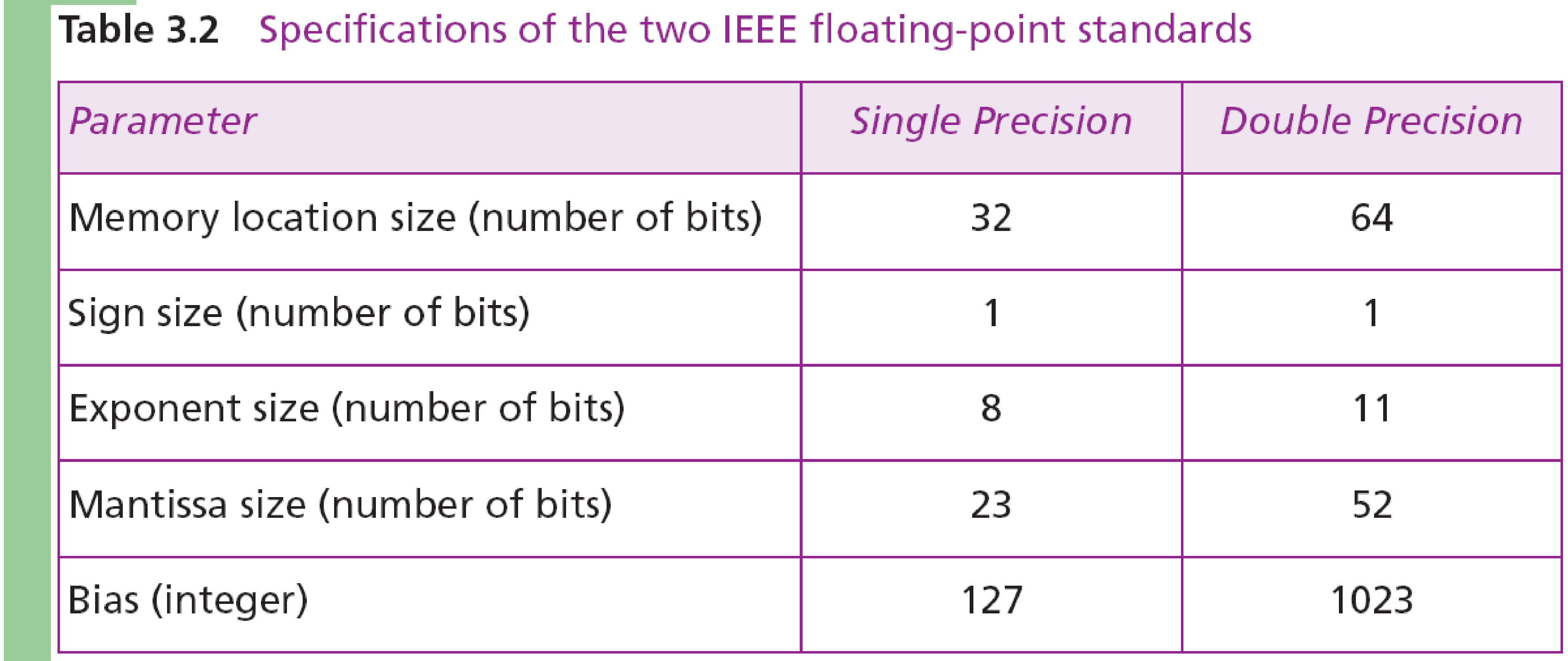


Table 3.2 Specifications of the two IEEE floating-point standards

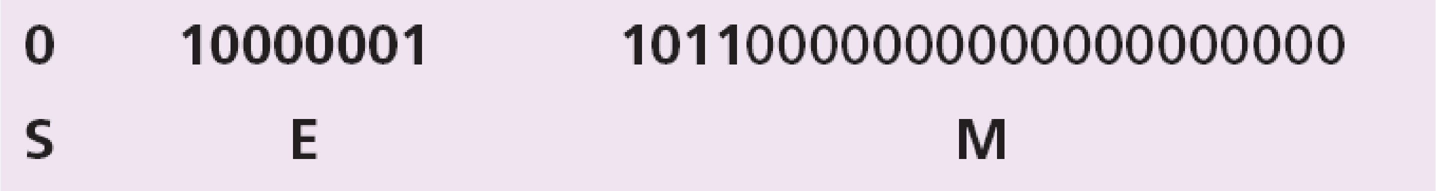
Parameter	*Single Precision*	*Double Precision*
Memory location size (number of bits)	32	64
Sign size (number of bits)	1	1
Exponent size (number of bits)	8	11
Mantissa size (number of bits)	23	52
Bias (integer)	127	1023

Excess System Example (1)

- Example 3.23 show the Excess_127 (single precision) representation of the decimal number 5.75.

Solution

- The sign is positive, so S = 0.
- Decimal to binary transformation: $5.75 = (101.11)_2$.
- Normalization: $(101.11)_2 = (1.1011)_2 \times 2^2$.
- $E = 2 + 127 = 129 = (10000001)_2$, $M = 1011$.
- We need to add nineteen zeros at the right of M to make it 23 bits.



0	10000001	10110000000000000000000
S	E	M

The number is stored in the computer as

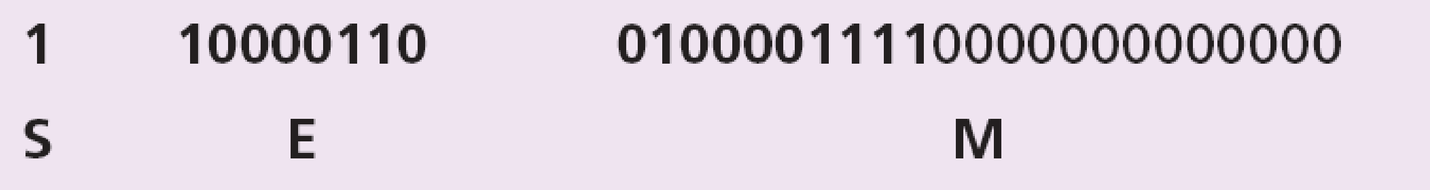
01000000110110000000000000000000

Excess System Example (2)

- Example 3.24 shows the Excess_127 representation of –161.875.

Solution:

- The sign is negative, so S = 1.
- Decimal to binary: 161.875= $(10100001.111)_2$.
- Normalization: $(10100001.111)_2 = (1.0100001111)_2 \times 2^7$.
- E = 7 + 127 = 134 = $(10000110)_2$ and M = $(0100001111)_2$.



1	10000110	01000011110000000000000
S	E	M

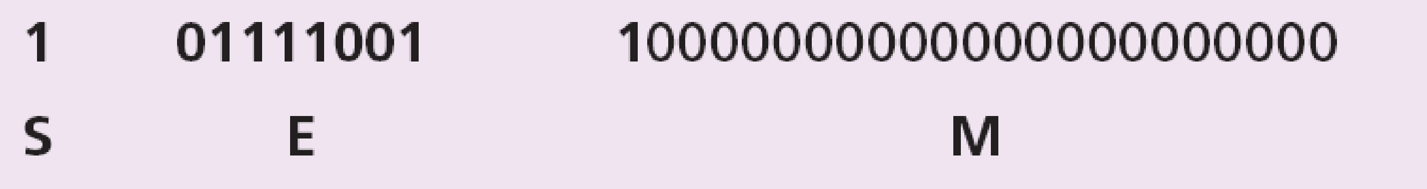
The number is stored in the computer as
11000011010000111100000000000000

Excess System Example (3)

- Example 3.25 shows the Excess_127 representation of –0.0234375.

Solution

- S = 1 (the number is negative).
- Decimal to binary: 0.0234375 = $(0.0000011)_2$.
- Normalization: $(0.0000011)_2 = (1.1)_2 \times 2^{-6}$.
- $E = -6 + 127 = 121 = (01111001)_2$ and $M = (1)_2$.



1	01111001	10000000000000000000000
S	E	M

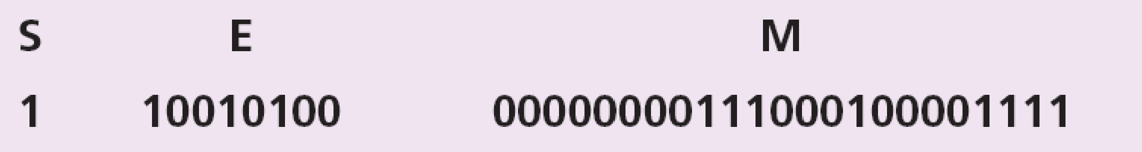
The number is stored in the computer as

10111100110000000000000000000000

Excess System Example (4)

- Example 3.26 show the value in decimal, when The bit pattern is stored in Excess_127 format:

$(11001010000000000111000100001111)_2$



S	E	M
1	10010100	00000000111000100001111

Solution

- The first bit is S, the next eight bits, E and the remaining 23 bits, M.

b. The shifter = E − 127 = 148 − 127 = 21.

c. This gives us $(1.00000000111000100001111)_2 \times 2^{21}$.

d. The binary number is $(1000000001110001000011.11)_2$.

e. The absolute value is 2,104,378.75.

f. The number is −2,104,378.75.

Overflow and Underflow

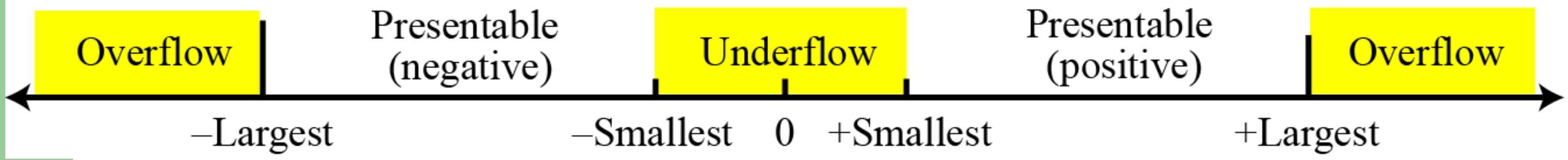
- **Overflow and underflow in floating-point representation of reals (Figure 3.12)**

– Largest: $-(1 - 2^{-24}) \times 2^{+128}$

+ Largest: $+(1 - 2^{-24}) \times 2^{+128}$

– Smallest: $-(1 - 2^{-1}) \times 2^{-127}$

+ Smallest: $+(1 - 2^{-1}) \times 2^{-127}$



Storing Zero

- A real number with an integral part and the fractional part set to zero, that is, 0.0, cannot be stored using the steps discussed above. To handle this special case, it is agreed that in this case the sign, exponent and the mantissa are set to 0s.

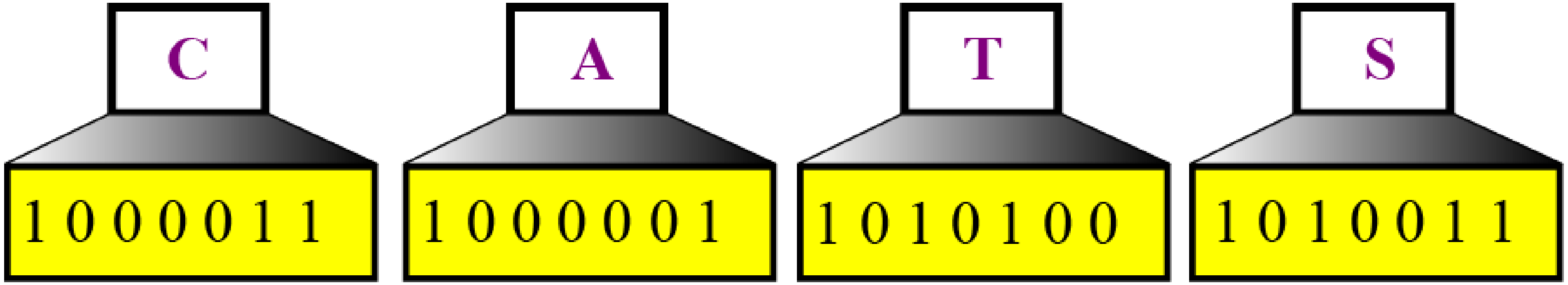
3.3 Storing Text

Overview (1)

- A section of text in any language is a sequence of symbols used to represent an idea in that language.
- For example, the English language uses
  - 26 symbols (A, B, C,…, Z) to represent uppercase letters,
  - 26 symbols (a, b, c, …, z) to represent lowercase letters,
  - nine symbols (0, 1, 2, …, 9) to represent numeric characters
  - symbols (., ?, :, ; , …, !) to represent punctuation.
- Other symbols such as blank, newline, and tab are used for text alignment and readability.

Overview (2)

- Representing each symbol with a bit pattern.
- Text such as “CATS”, which is made up from four symbols, can be represented as four *n*-bit patterns, each pattern defining a single symbol



C A T S

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

Figure 3.13 **Representing symbols using bit patterns**

Overview (3)

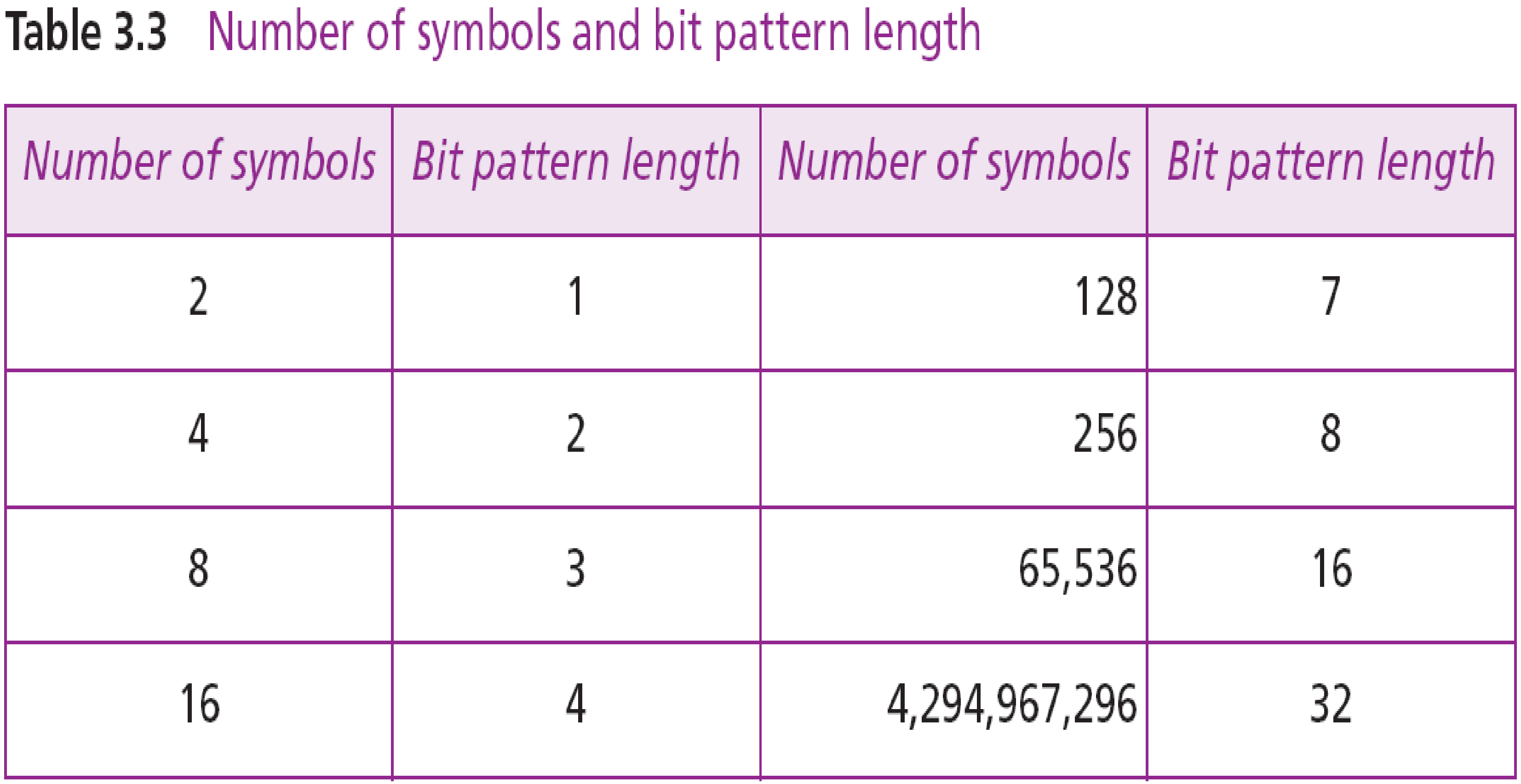


Table 3.3 Number of symbols and bit pattern length

Number of symbols	*Bit pattern length*	*Number of symbols*	*Bit pattern length*
2	1	128	7
4	2	256	8
8	3	65,536	16
16	4	4,294,967,296	32

Overview (4)

- Codes (See Appendix A)
  - ASCII
  - Unicode
  - Other Codes

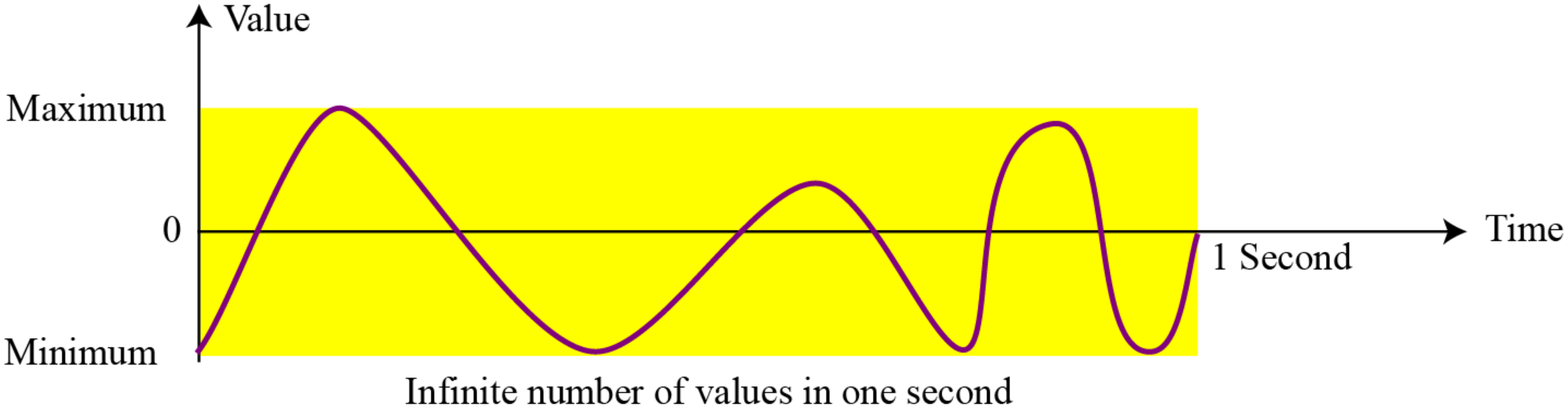
3.4 Storing Audio

Overview (1)

- Audio
  - A representation of sound or music.
  - Different to the numbers or text.
    - Text is composed of countable entities (characters); Text is an example of digital data.
  - Being not countable.
  - An example of analog data.
    - Even if we can measure all its values in a period of time, we cannot store these in the computer's memory, as we would need an infinite number of memory locations.
- How to store audio data
  - Sampling → Quantization → Encoding

Overview (2)

- Figure 3.15 shows the nature of an analog signal, such as audio, that varies with time.



Sampling

- Sampling means that we select only a finite number of points on the analog signal, measure their values, and record them.

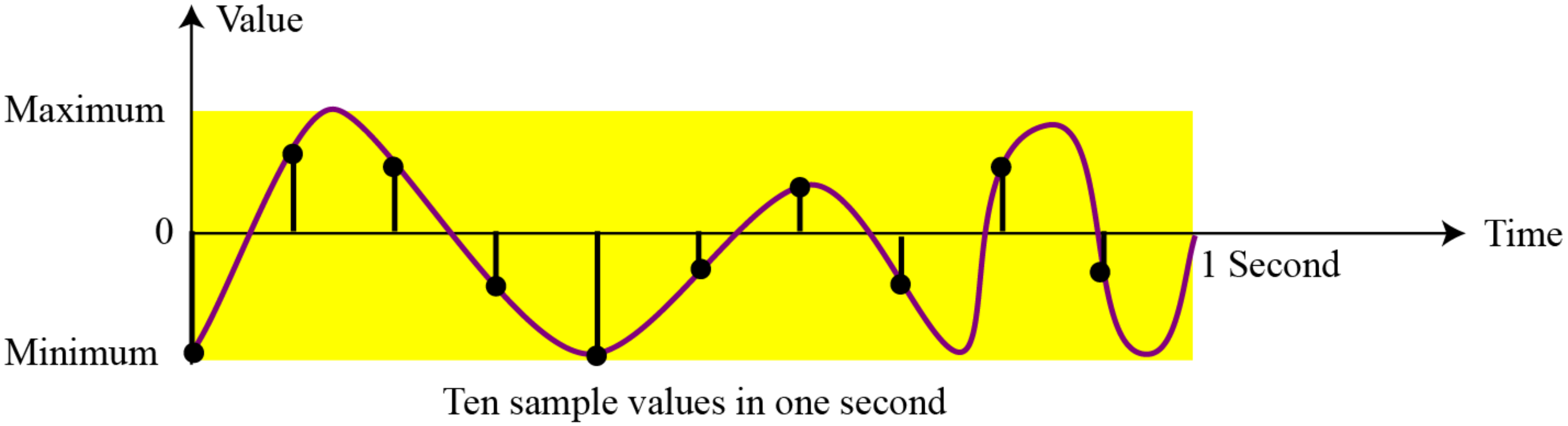


Figure 3.16 **Sampling an audio signal**

Quantization

- The value measured for each sample is a real number.
- However, it is simpler to use an unsigned integer (a bit pattern) for each sample.
- Quantization refers to a process that rounds the value of a sample to the closest integer value.
- For example,
  - if the real value is 17.2, it can be rounded down to 17: if the value is 17.7, it can be rounded up to 18.

Encoding

- Quantized sample values are encoded as bit patterns.
  - Some systems assign positive and negative values to samples,
  - Some shift the curve to the positive part and assign only positive values.
- Number of bits per sample B (bit depth), and the number of samples per second, S:
  - **To store S × B bits for each second of audio**
- This product is sometimes referred to as bit rate, R. For example: 40,000 samples per second and 16 bits per each sample, the bit rate is

$$R = 40{,}000 \times 16 = 640{,}000 \text{ bits per second}$$

Standards for sound encoding

- **MP3** (short for **MPEG Layer 3**).
  - Today the dominant standard for storing audio.
  - A modification of the **MPEG** (**Motion Picture Experts Group**) compression method used for video.
  - Using 44100 samples per second and 16 bits per sample.
- Lossy compression (Chapter 15)
  - The result is a signal with a bit rate of 705,600 bits per second, which is compressed using a compression method that discards information that cannot be detected by the human ear.
  - As opposed to lossless compression

3.5 Storing Images

Overview (1)

- Images are stored in computers using two different techniques:
  - Raster graphics and Vector graphics.
- Raster graphics (or bitmap graphics)
  - Be used when storing an analog image such as a photograph
  - A photograph consists of analog data, similar to audio information. The difference is that the intensity of data varies i space instead of in time.
  - Data must be sampled. However, sampling in this case is called scanning. The samples are called pixels (picture elements).

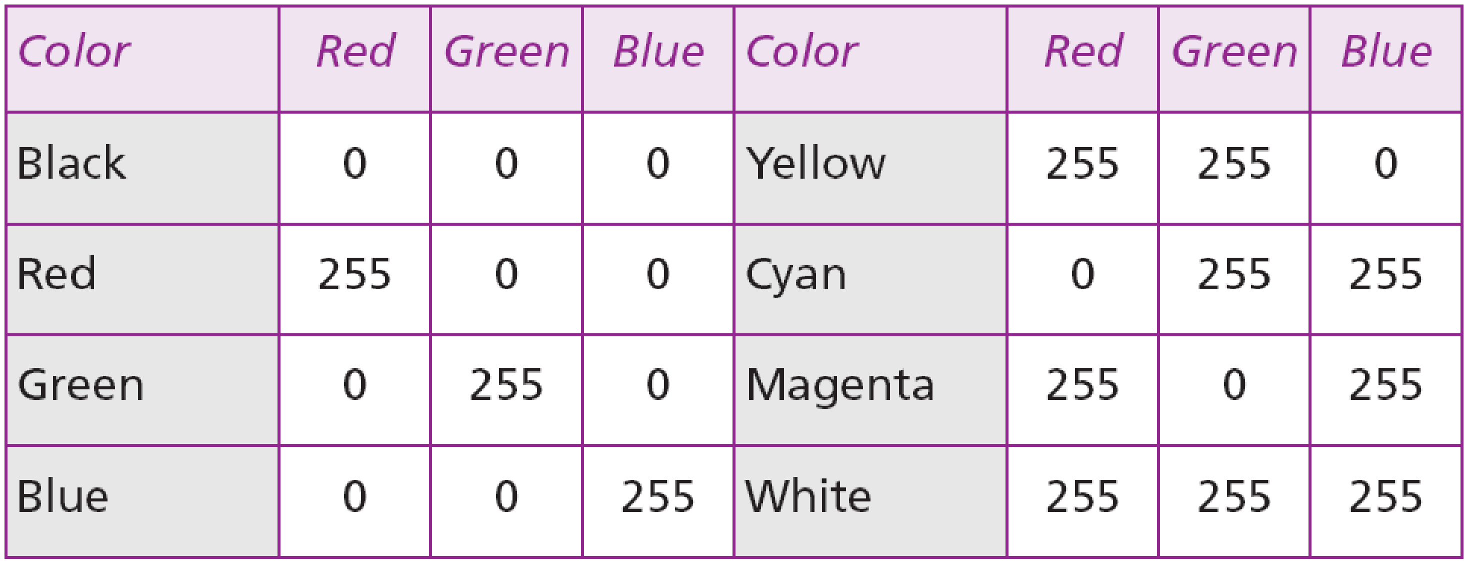
Overview (2)

- When Image scanning, decide how many pixels we should record for each square or linear inch.
- The scanning rate is called Resolution.
- The number of bits used to represent a pixel, its Color Depth, depends on how the pixel's color is handled by different encoding techniques.
- The perception of color is
  - How our eyes respond to a beam of light.
  - Our eyes have different types of photoreceptor cells: some respond to the three primary colors (RGB), while others merely respond to the intensity of light.

True-Color

- One of the encoding techniques
- Using **24** bits to encode a pixel.

Table 3.4 Some colors defined in True-Color



Color	*Red*	*Green*	*Blue*	*Color*	*Red*	*Green*	*Blue*
Black	0	0	0	Yellow	255	255	0
Red	255	0	0	Cyan	0	255	255
Green	0	255	0	Magenta	255	0	255
Blue	0	0	255	White	255	255	255

Indexed Color (Palette color) (1)

- Using only a portion of these colors.

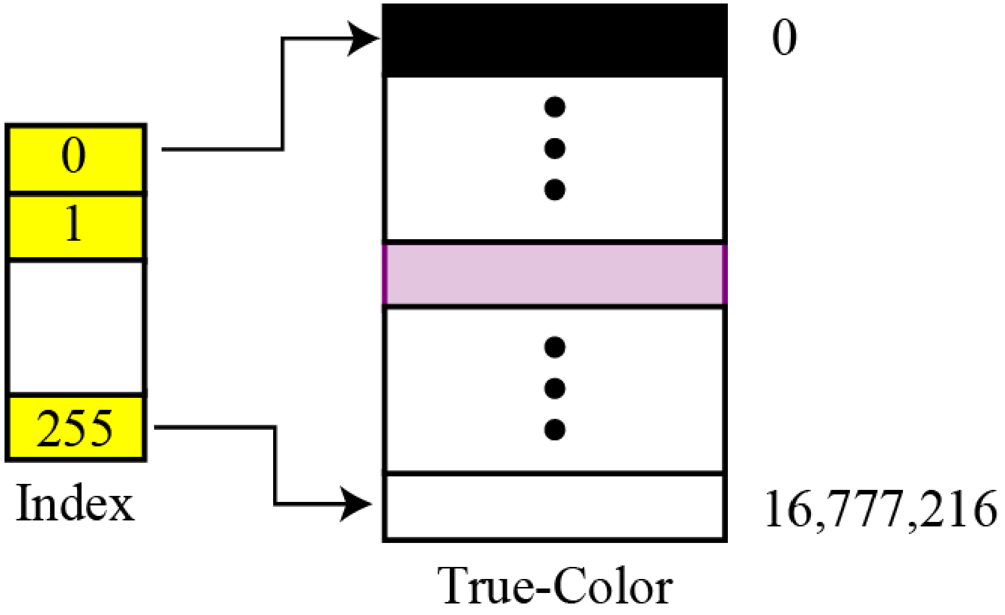
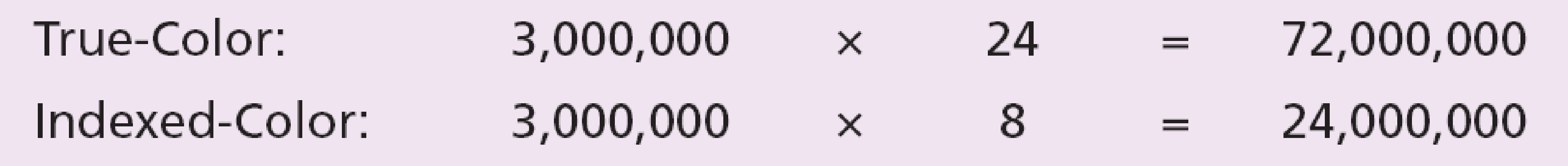


Figure 3.17 **Relationship of the indexed color to the True-Color**

Indexed Color (Palette color) (2)

- For example, a high-quality digital camera uses almost three million pixels for a 3 × 5 inch photo.
- The following shows the number of bits that need to be stored using each scheme:



True-Color:	3,000,000	×	24	=	72,000,000
Indexed-Color:	3,000,000	×	8	=	24,000,000

Standards for image encoding

- Several de facto standards for image encoding are in use.
- **JPEG** (**Joint Photographic Experts Group**) uses the True-Color scheme, but compresses the image to reduce the number of bits (Chapter 15).
- **GIF** (**Graphic Interchange Format**), on the other hand, uses the indexed color scheme.

Vector graphics

- **Raster graphics** has two disadvantages:
  - the file size is big and rescaling is troublesome.
  - To enlarge a raster graphics image means enlarging the pixels, so the image looks ragged when it is enlarged.
- The **vector graphic** image encoding method, however, does not store the bit patterns for each pixel. An image is decomposed into a combination of geometrical shapes such as lines, squares or circles.
- For example, consider a circle of radius r. The main pieces of information a program needs to draw this circle are:
  - **The radius *r* and equation of a circle.**
  - **The location of the center point of the circle.**
  - **The stroke line style and color.**
  - **The fill style and color.**

3.5 Storing Video

Overview

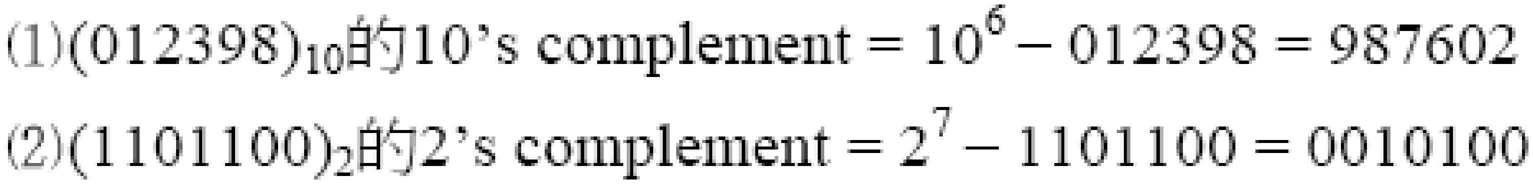
- Video is a representation of images (called frames) over time.
- A movie consists of a series of frames shown one after another.
  - Video is the representation of information that changes in space and in time.
  - So, if we know how to store an image inside a computer, we also know how to store video:
  - each image or frame is transformed into a set of bit patterns and stored.
  - The combination of the images then represents the video.
- See Chapter 15 for video compression.

補充

- r’s and (r-1)’s complement
- IEEE 754

r’s Complement

- 若一無號的數字N(N $\neq$ 0)，基底（base）是r，整數部分的位數爲n，則它的r’s complement定義爲(r^n – N)，且令N = 0時，N的r補數爲0。舉例說明如下：



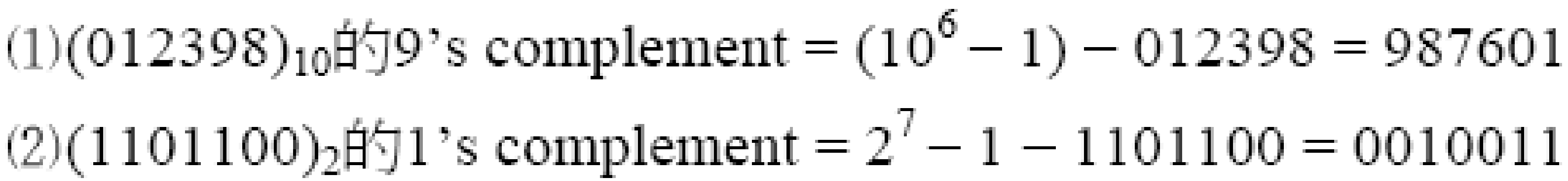
(1)$(012398)_{10}$的10’s complement = $10^6 - 012398 = 987602$

(2)$(1101100)_2$的2’s complement = $2^7 - 1101100 = 0010100$

e.g 6-digit Hex. number N= ABD4F5, What is the 16’s complement of N?

(r – 1)’s Complement

- 若一無號的數字N(N $\neq$ 0)，基底（base）是r，整數部分的位數爲n，則它的(r–1)補數爲$(r^n - 1) - N$，且令N = 0時，N的(r–1)補數爲0。舉例說明如下：



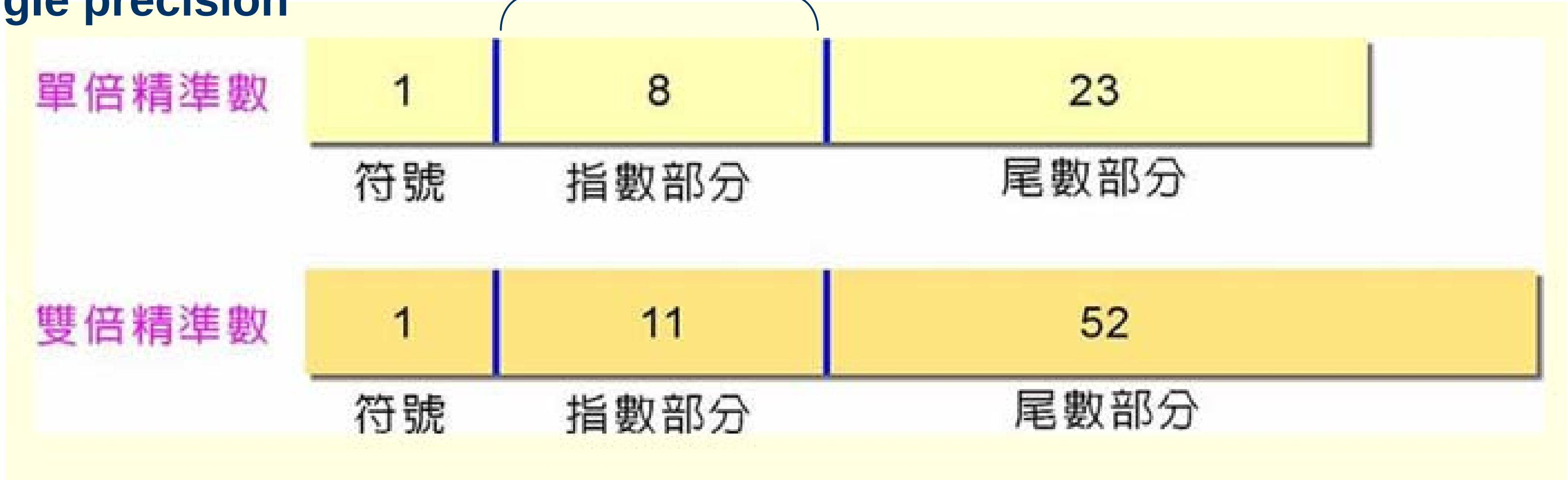
(1)$(012398)_{10}$的9’s complement = $(10^6 - 1) - 012398 = 987601$

(2)$(1101100)_2$的1’s complement = $2^7 - 1 - 1101100 = 0010011$

IEEE 754

Excess 127

single precision



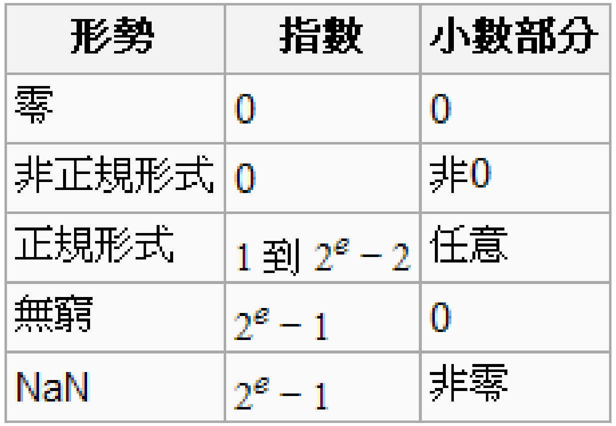
單倍精準數	1	8	23
	符號	指數部分	尾數部分
雙倍精準數	1	11	52
	符號	指數部分	尾數部分

IEEE 754

- 小數部分最高有效位由*exp*部分決定。如果*指數*在0 < *exponent* < 2*e* − 1之間，那麼小數部分*最高有效位*將是1，而且這個數將被稱爲正規形式。
- 如果*exp*是0，*有效數*最高有效位將會是0，並且這個數將被稱爲非正規形式。
- 有三個特殊值需要指出：

IEEE 754

- 如果 *exp* 是0 並且 *小數部分* 是0, 這個數±0 (和符號位相關)
- 如果 *exp* = $2^e - 1$ 並且 *小數部分* 是0, 這個數是 ±無窮大 (同樣和符號位相關)
- 如果 *exp* = $2^e - 1$ 並且 *小數部分* 非0, 這個數表示爲不是一個數(NaN).



形勢	指數	小數部分
零	0	0
非正規形式	0	非0
正規形式	1 到 $2^e - 2$	任意
無窮	$2^e - 1$	0
NaN	$2^e - 1$	非零

IEEE 754

- 0的公訂表示法爲 00000000000000000000000000000000
- 10000000000000000000000000000000 也是0（代表-0）
- 指數部分的-127（00000000）和+128（11111111）做爲特殊用途
- 最小的正數爲 00000000100000000000000000000000 其數值爲$+2^{-126}$；
- 最大的正數爲 01111111011111111111111111111111 其數值爲（$2-2^{-23}$）$x2^{127}$