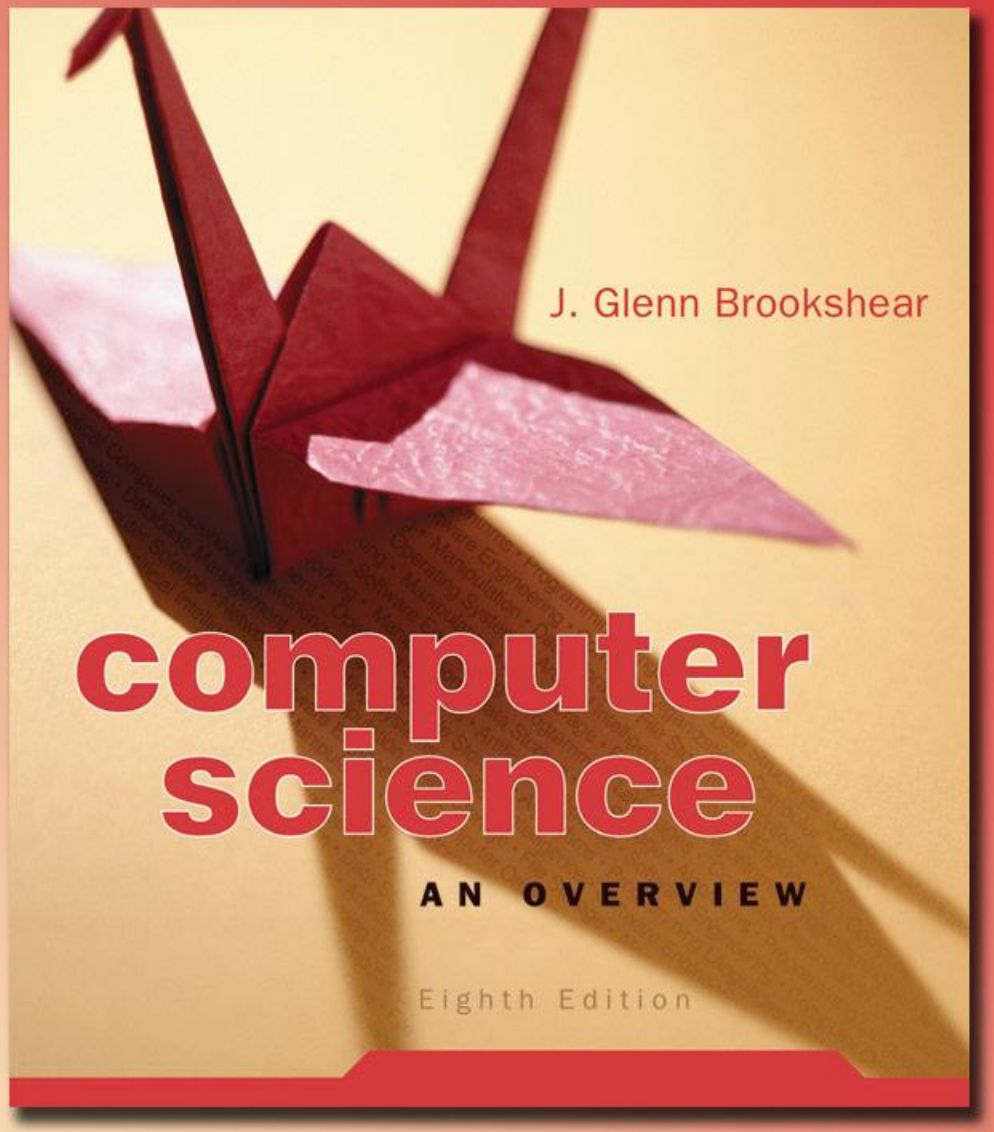
Data Storage



Data Storage

How to Think Like a Computer Scientist

Outline

- 1.1 Bits and Their Storage
- 1.2 Main Memory
- 1.3 Mass Storage
- 1.4 Representing Information as Bit Patterns
- 1.5 The Binary System
- 1.6 Storing Integers
- 1.7 Storing Fractions
- 1.8 Data Compression
- 1.9 Communications Errors

Data / Information

- Data is the raw representation of some physical quantity.
- Information is structured and interpreted data
- In computer science, algorithms use data and produce data for some end-user

Computer Algorithms

To be able to use computers,

- Algorithms and data must be coded in terms of electrical pulses (eventually).

- We will need to represent numbers and text data (letters,...) as electrical pulses.

- Results represented by the electrical pulses (inside computer) must be decoded into forms suitable for humans.

Computer Programs

Algorithm in coded form is a computer program, or program's code.

Translation from concepts to computer internal form is done in several phases, and each involves coding with a different style:

- From concepts to a high level language
- From high level language to low level language,
- From low level language to machine language,
- From machine language to internal (voltage) form

Data Representation

Data must be represented depending on its type and usage:

Numerical data must be represented according to a number system

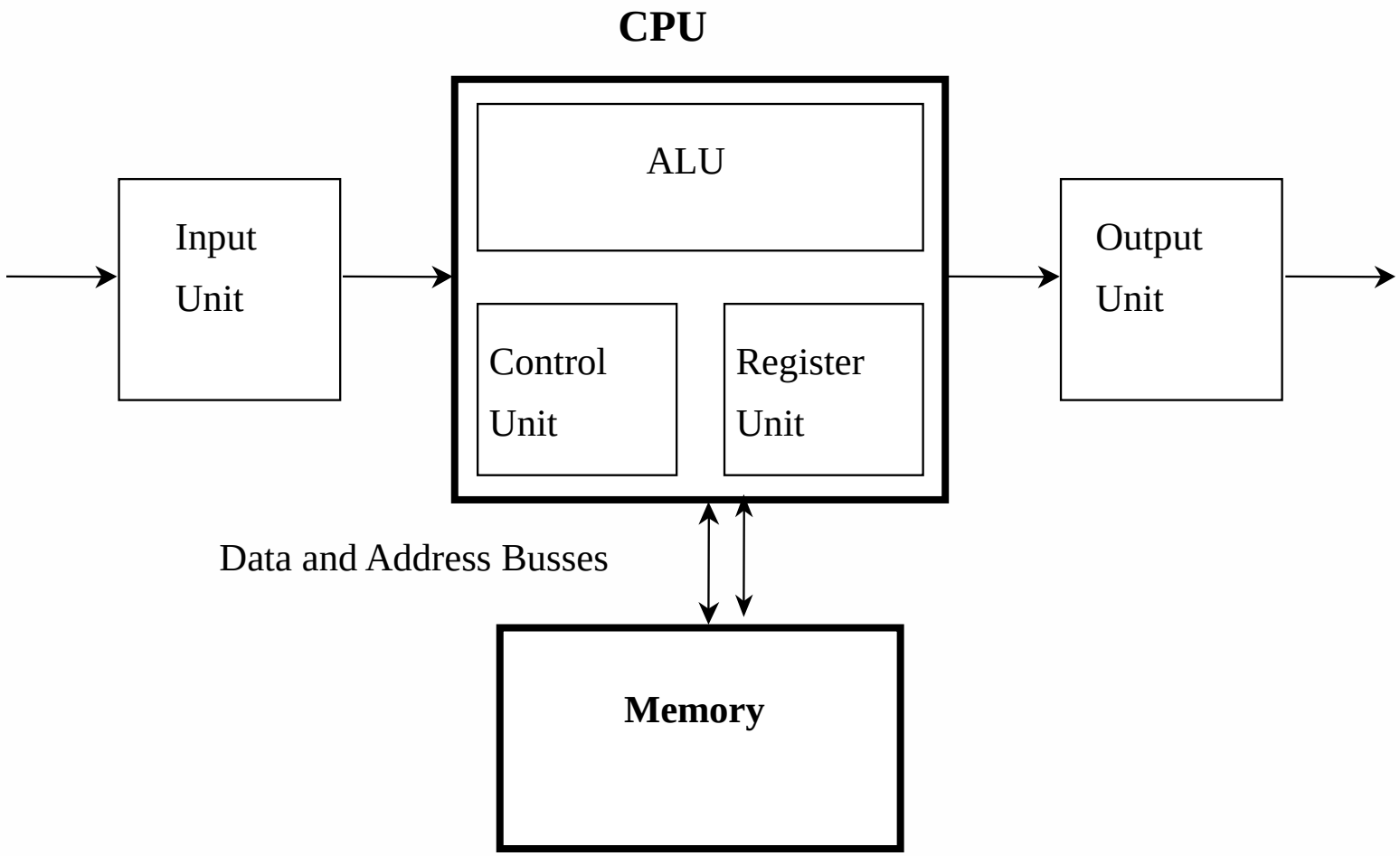
Text data must be represented according to a standard coding table.

A Simplified Computer

Logical View

- CPU (**C**entral **P**rocessing **U**nit)
- Memory (RAM, ROM, Cache)
- Input Devices (keyboard, mouse, ...)
- Output Devices (Screen, printer, ...)
- Mass Storage Devices (disks, tapes)

Computer Block Diagram



Terminology

Algorithm	Algoritma
Representation	Gösterim
Coding	Kodlama
Data (p.), Datum (s.)	Veri
Information	Bilgi
Programming language	Programlama dili
High-level language	Yüksek (üst) düzey dil
Low-level language	Alt düzey dil

Terminology

RAM (Random **A**ccess **M**emory**)**	Rastgele erisimli bellek
ROM (Read **O**nly **M**emory**)**	Salt-okunur bellek
Data Bus	Veri (iletim) yolu
Address Bus	Adres (iletim) yolu
Memory	Bellek
ALU (Arithmetic **L**ogic **U**nit**)**	Aritmetik mantık birimi
Control Unit	Denetim birimi
Input	Giris / Girdi
Output	Çıkıs / Çıktı
Device	Aygıt

Bits and their meaning

- **Bit** = Binary Digit = a symbol whose meaning depends on the application at hand.
- Some possible meanings for a single bit
  - Numeric value (1 or 0)
  - Boolean value (true or false)
  - Voltage (high or low)

Bit patterns

- *All data* stored in a computer are represented by patterns of bits:
  - Numbers
  - Text characters
  - Images
  - Sound
  - *Anything else…*

Boolean operations

- **Boolean operation** = any operation that manipulates one or more true/false values
  - Can be used to operate on bits
  - often referred to as a *logical operation*
- Specific operations
  - AND
  - OR
  - XOR (‘exclusive OR’: X or Y but not both)
  - NOT

Figure 1.1 The Boolean operations AND, OR, and XOR (exclusive or)

The AND operation

	0		0		1		1
AND	0	AND	1	AND	0	AND	1
	0		0		0		1

The OR operation

	0		0		1		1
OR	0	OR	1	OR	0	OR	1
	0		1		1		1

The XOR operation

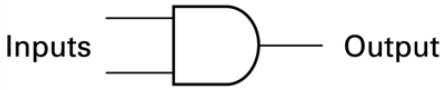
	0		0		1		1
XOR	0	XOR	1	XOR	0	XOR	1
	0		1		1		0

Gates

- **Gates** = devices that produce the outputs of Boolean operations when given the operations' input values
  - Often implemented as electronic circuits
  - Provide the building blocks from which computers are constructed

A pictorial representation of AND, OR, XOR, and NOT gates, as well as their input and output values

AND



Inputs	Output
0 0	0
0 1	0
1 0	0
1 1	1

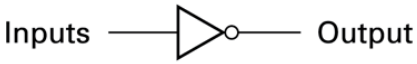
OR

Inputs	Output
0 0	0
0 1	1
1 0	1
1 1	1

XOR

Inputs	Output
0 0	0
0 1	1
1 0	1
1 1	0

NOT



Inputs	Output
0	1
1	0

Flip-flops

- **Flip-flop** = a circuit built from gates that can store one bit of data.
  - Has an input line which sets its stored value to 1
  - Has an input line which sets its stored value to 0
  - While both input lines are 0, the most recently stored value is preserved

Figure 1.3 A simple flip-flop circuit

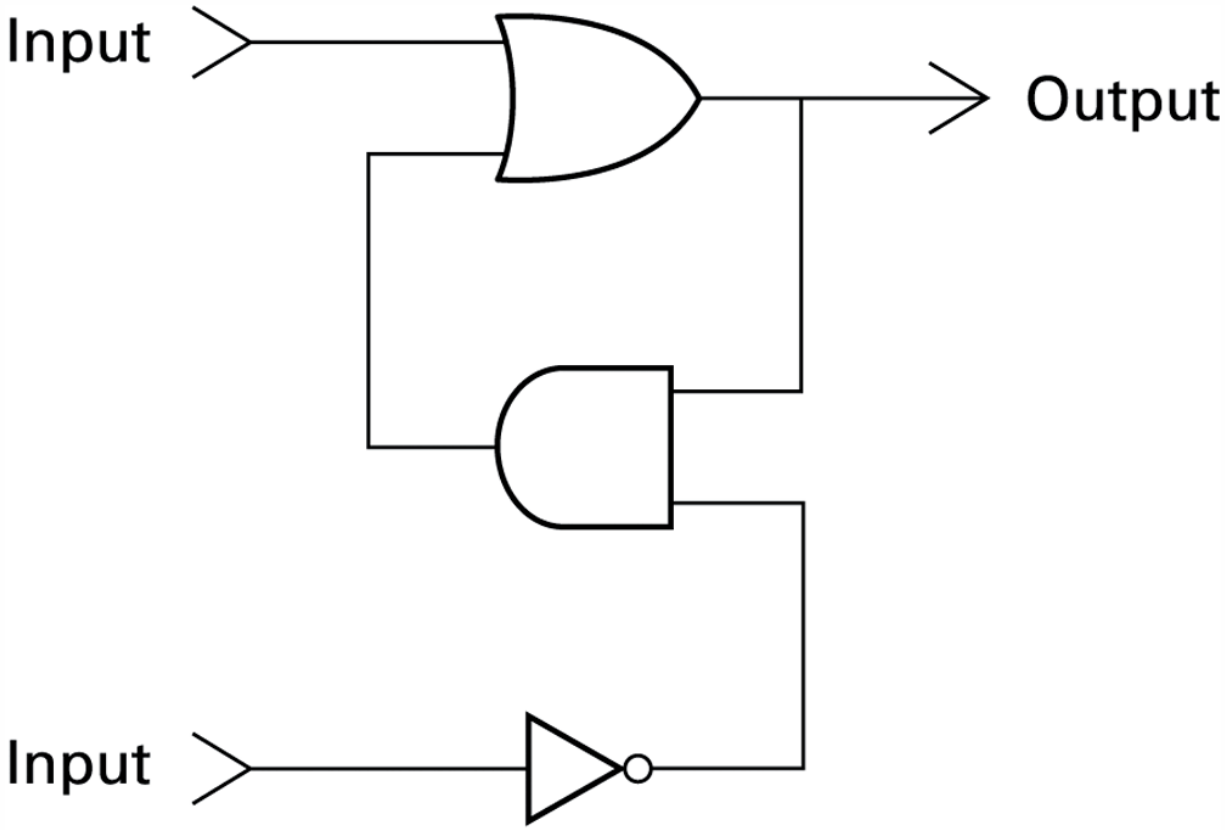
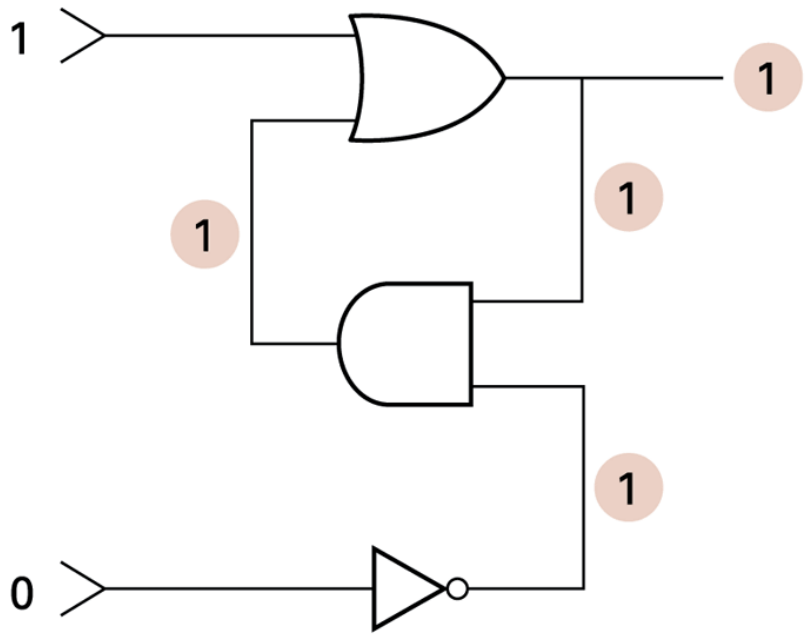


Figure 1.4 Setting the output of a flip-flop to 1 (cont'd)

b. This causes the output of the OR gate to be 1 and, in turn, the output of the AND gate to be 1.



Hexadecimal notation

- The **hexadecimal system** is a counting system using base 16 (c.f. decimal, which uses base 10 -see later notes)
- **Hexadecimal notation** = a shorthand notation for streams of bits.
  - Stream = a long string of bits.
  - Long bit streams are difficult to make sense of.
  - The lengths of most bit streams used in a machine are multiples of four.
  - Hexadecimal notation is more compact.
    - Less error-prone to manually read, copy, or write

Figure 1.6 The hexadecimal coding system

Bit pattern	Hexadecimal representation
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

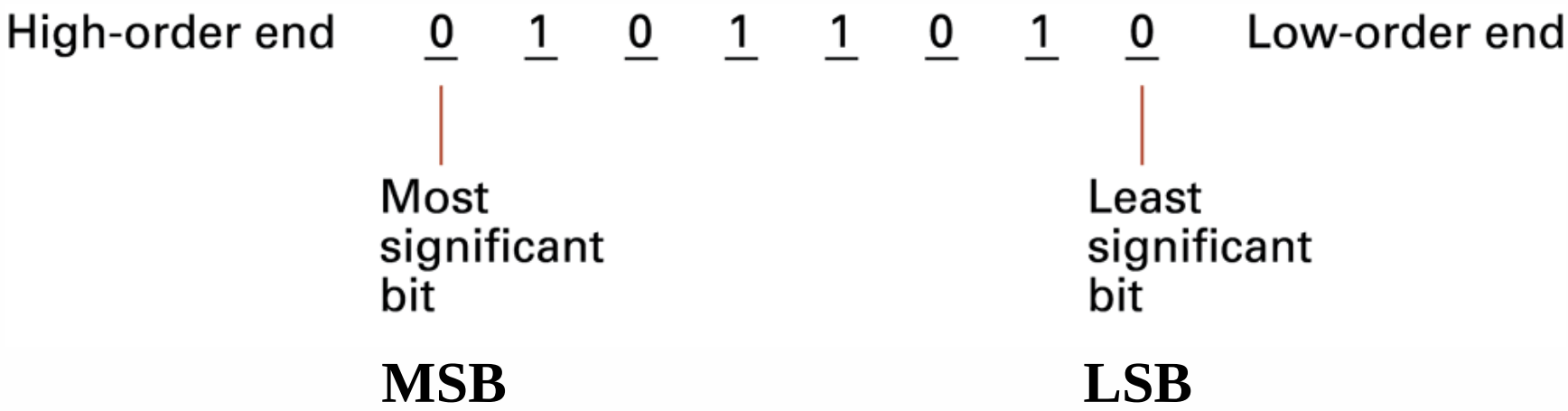
Octal encoding:
for encoding a bit stream in groups of three

Bit pattern	Octal representation
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

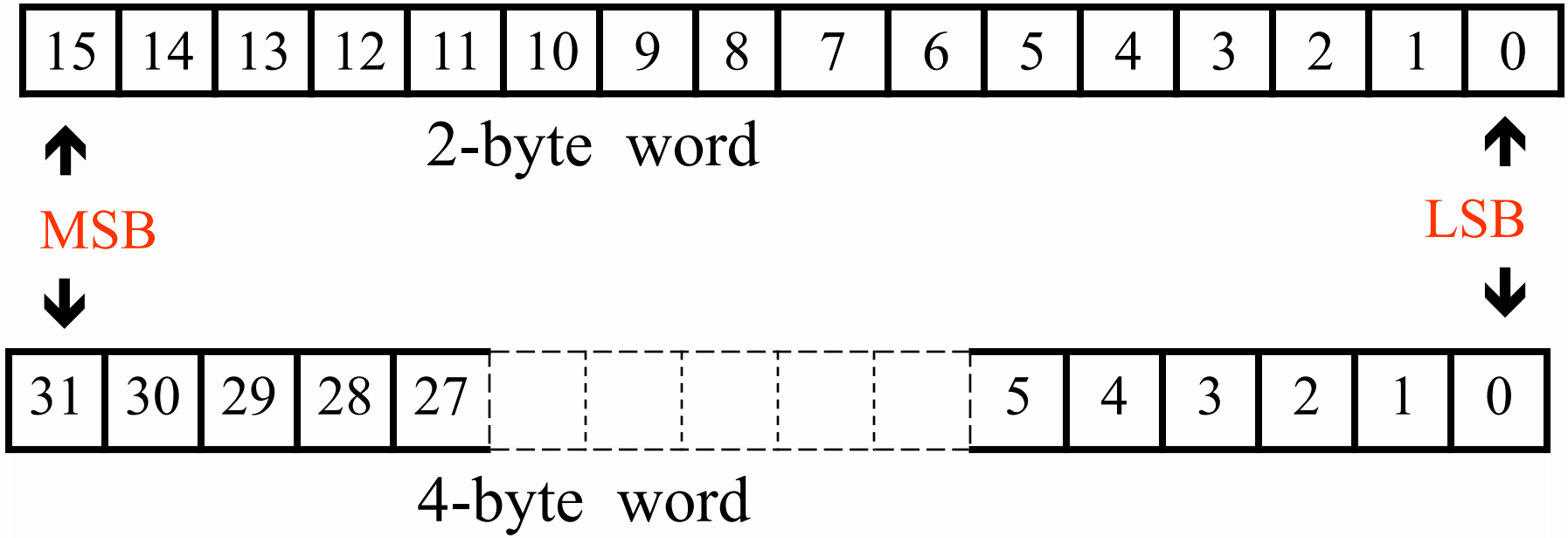
Main memory: cells

- **Cells** = manageable units (typically 8 bits) into which a computer's main memory is arranged.
- **Byte** = a string of 8 bits.
  - Smallest addressable unit of data storage
- **High-order end** = the left end of the conceptual row in which the contents of a cell are laid out.
  - **Most significant bit** = the last bit at the high-order end.
- **Low-order end** = the right end of the conceptual row in which the contents of a cell are laid out.
  - **Least significant bit** = the last bit at the low-order end.

Figure 1.7 The organization of a byte-size memory cell



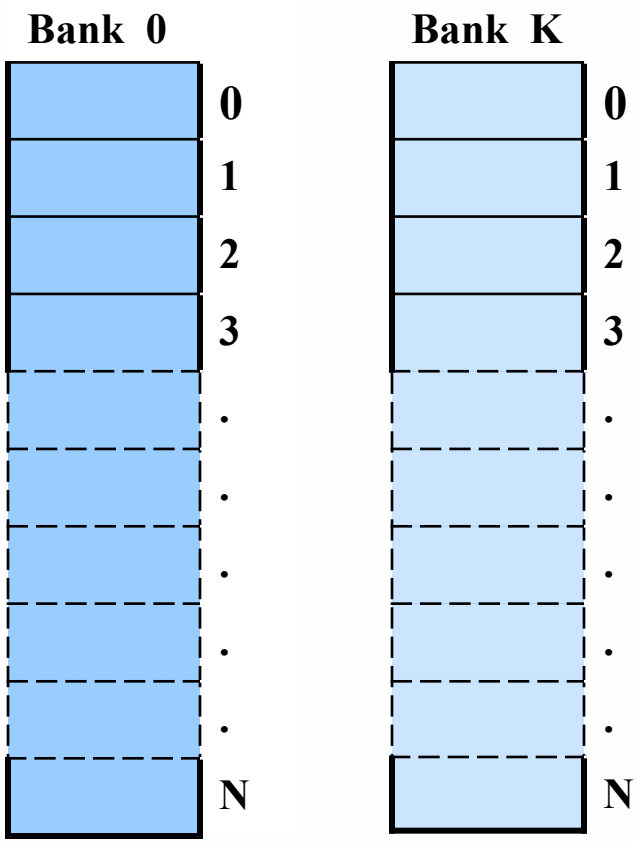
Word : Is a general term describing several consecutive bytes in memory (usually 2 or 4).



Main memory addresses

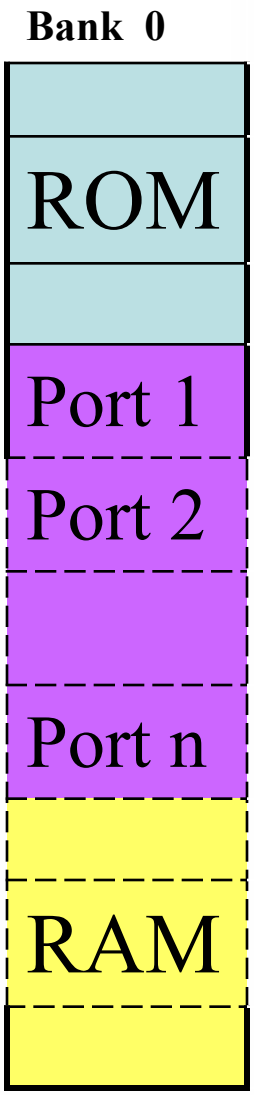
- **Address** = a “name” to uniquely identify one cell in the computer’s main memory
- The names for cells in a computer are consecutive numbers, usually starting at zero
- Cells have an order, so “previous cell” and “next cell” have reasonable meanings
- **Random Access Memory** = memory where any cell can be accessed independently

Main Memory, Physical view



- Memory is composed of a number of (modules) banks
- Each bank has a number of bytes arranged as an array
- Each byte has an address of the form (Bank#, byte#)

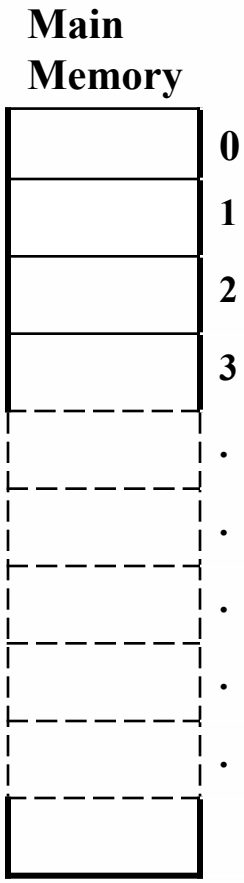
Main Memory, Physical view



- Some bytes are kept in RAM, some in ROM, some in I/O ports (special purpose registers).
- Address control logic circuits handle addressing correct bytes.
- Bytes in RAM can be read/written
- Bytes in ROM can be read only

Main Memory, Logical view

("Logical" means "As a programmer would imagine it")



- Memory is composed of a number of bytes
- Bytes are arranged as an array
- Each byte has an address (byte#)
- All bytes are R/W

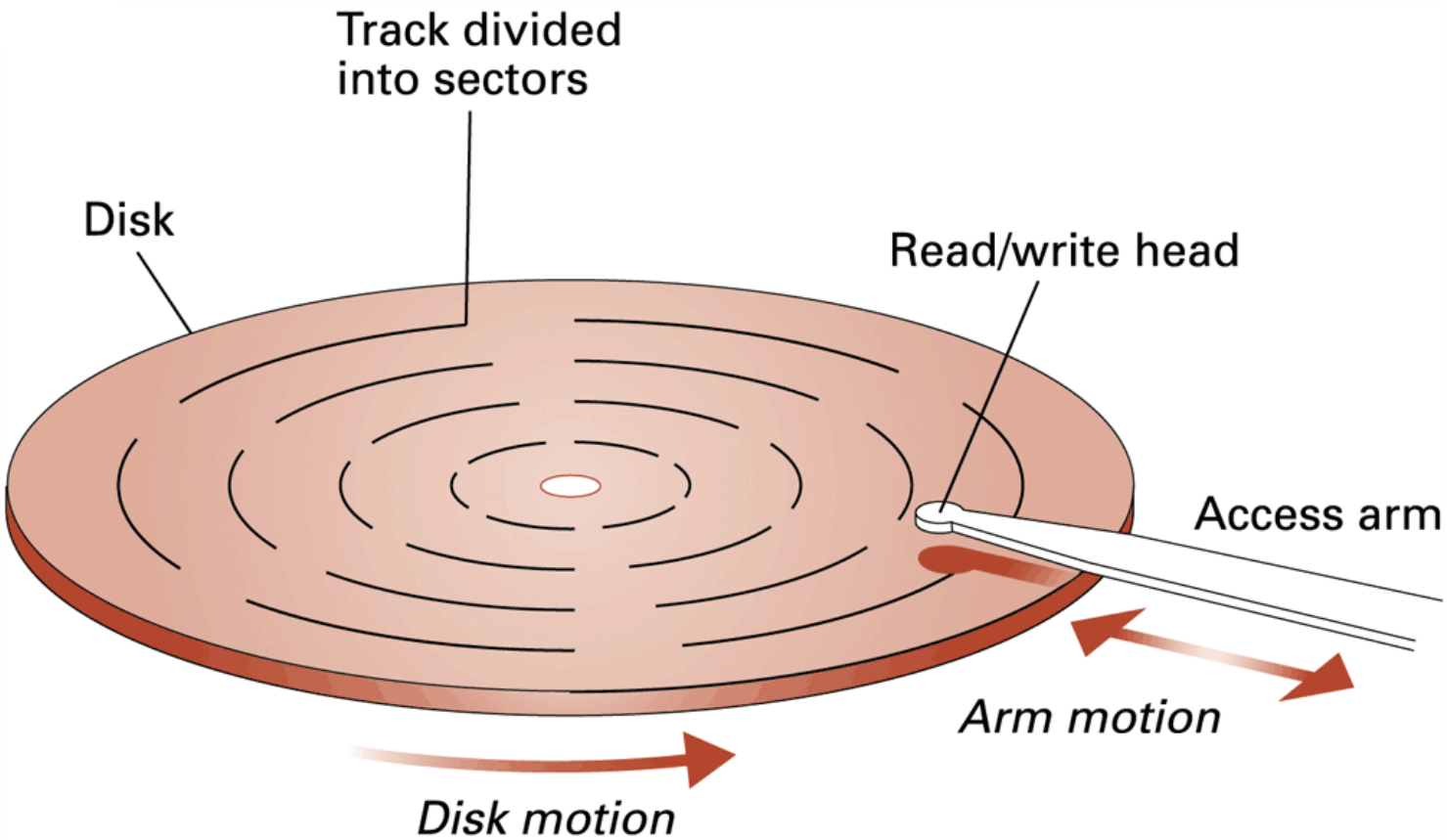
Measuring memory capacity: Not quite like the metric system

- "Kilo-" normally means 1,000;
  Kilobyte = 2^{10} = 1024
- "Mega-" normally means 1,000,000;
  Megabyte = 2^{20} = 1,048,576
- "Giga-" normally means 1,000,000,000;
  Gigabyte = 2^{30} = 1,073,741,824
- *BUT* disc capacity now sometimes specified using decimal definitions (10^3 , 10^6 , 10^9 …)!

Mass Storage Systems

- Non-volatile: data remains when computer is off
- Usually much bigger than main memory
- Usually rotating disks
  - Hard disk, floppy disk, CD-ROM
  - Much slower than main memory
    - Data access must wait for **seek time** (head positioning)
    - Data access must wait for **rotation delay** or **latency time** (time for desired data to rotate to read/write head)
    - Access time = seek time + latency time

Figure 1.9 A disk storage system



Each track contains same number of sectors, each with same number of bits – unless zoned-bit recording is used to add more sectors to longer, outer tracks

CD & DVD data storage

- One continuous spiral data track
  - in one revolution more data is read from outer part of track than from inner
- Uniform rate of data transfer can only be achieved by varying the rotation speed
- Most systems use constant rotation speed therefore the system must handle varying data transfer rates
- Typical capacity: 600-700 MB (CD)

  several GB (DVD)

Files

- **File** = the unit of data stored on a mass storage system.
  - **Logical record** natural groups of data within a file

    i.e. natural from the programmer point of view
  - **Field :** sub-unit of logical record
    - **Key field** holds identifier **key**
- **Physical record** = a block of data conforming to the physical characteristics of the storage device.
- **Buffer** = main memory area sometimes set aside for assembling logical records or fields of a file

Representing Data

External representation: (for people) to use data easily, data must be represented (visually, graphically, ...) on paper or on some other directly useable media.

Internal representation: (for computing machines) to support storage and computation, data must be represented as a sequence of 0's and 1's.

Representing Data

How to represent data on paper? in memory?

Coding is representing data according to a system, internally or externally.

Information is coded:

- For internal computations
- For presentation purposes (text, graphics, sound)
- For long-term archival storage (databases, archives)

Representing Data

In all cases, data must be coded as a data type, in order to be recognized later.

In a computer system:

Text data must be represented according to standard coding table(s),

Numerical data must be represented by some notation (a number system)

Representing Numeric Data

- **Internal number representation**:

  Always a sequence of 0's and 1's:

  ...01100111001010 ...

- **External number representation**:

Many notations (number representation systems)

- Context-dependent notations
- Positional (fixed point) notations
- Floating point notation

Representing Numeric Data (Externally)
Non-positional notation

Roman Numerals:

- Positions do not carry weights
- Digits are interpreted according to context

Numeral	means
I	1 or -1 (add 1 or subtract 1)
V	5 or -5
X	10 or -10
L	50 or -50
C	100 or -100
D	500 or -500
M	1000

XIV = 14

MCMLXXIV = 1974

Negative numbers had not been invented yet!

Representing Numeric Data (Externally)
Positional notation

For representing data **externally**, the most commonly used number systems:

Number System	Radix (base)	Digits used
Decimal	10	0123456789
Binary	2	01
Octal	8	01234567
Hexadecimal	16	0123456789 ABCDEF

Representing Non-numeric Data

When the computing device has to handle letters and special symbols, each symbol is coded separately.

Two dominant coding schemes are:

- ASCII (American Standard Code for Information Interchange)
- EBCDIC (Extended Binary Coded Decimal Interchange Code)

Representing Strings in ASCII

String: a sequence of bytes (characters).

Generally contains text characters,

Can have any number of characters,

The end point must be specified in some way.

A	N	K	A	R	A	...	

↑ Code for 'A' in ASCII table

Figure 1.13 The message “Hello.” in ASCII

01001000	01100101	01101100	01101100	01101111	00101110
H	**e**	**l**	**l**	**o**	**.**

Representing text

- Each printable character (letter, punctuation, etc.) is assigned a unique bit pattern.
  - ASCII = 7-bit values for most symbols used in written English text
    - Extended to 8-bit values

    (may be extended in different ways by different organisations!)
  - Unicode = 16-bit values for most symbols used in most world languages today
  - ISO proposed standard = 32-bit values

- *Text file* consists of sequence of symbols encoded using ASCII or Unicode

ASCII number in range [0,127]	HEX code	Symbol
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

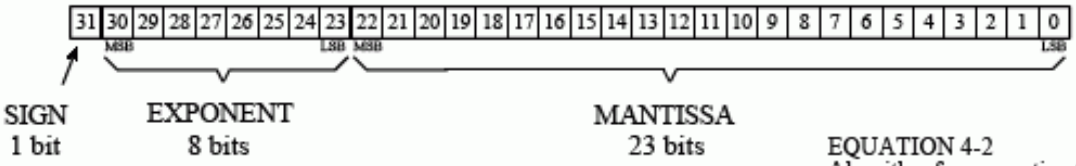
ASCII	HEX	Symbol
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O

ASCII	HEX	Symbol
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o

IEEE Single-Precision Floating Point Format

EQUATION 4-1
Equation for converting a bit pattern into a floating point number. The number is represented by v, S is the value of the sign bit, M is the value of the mantissa, and E is the value of the exponent.

$$v = (-1)^S \times M \times 2^{E-127}$$



EQUATION 4-2
Algorithm for converting the bit pattern into the mantissa, M, used in Eq. 4-1.

$$M = 1.m_{22}m_{21}m_{20}m_{19}\cdots m_2m_1m_0$$

Example 1

0 00000111 11000000000000000000000

$+$ 7 0.75

$$+1.75 \times 2^{(7-127)} = +1.316554 \times 10^{-36}$$

Example 2

1 10000001 01100000000000000000000

$-$ 129 0.375

$$-1.375 \times 2^{(129-127)} = -5.500000$$

FIGURE 4-2
Single precision floating point storage format. The 32 bits are broken into three separate parts, the sign bit, the exponent and the mantissa. Equations 4-1 and 4-2 shows how the represented number is found from these three parts. MSB and LSB refer to "most significant bit" and "least significant bit," respectively.

IEEE Single-Precision Floating Point Format

The term: $(-1)^S$, simply means that the sign bit, *S*, is 0 for a positive number and 1 for a negative number. The variable, *E*, is the number between 0 and 255 represented by the eight exponent bits. Subtracting 127 from this number allows the exponent term to run from to In other words, the exponent is stored in *offset binary* with an offset of 127.

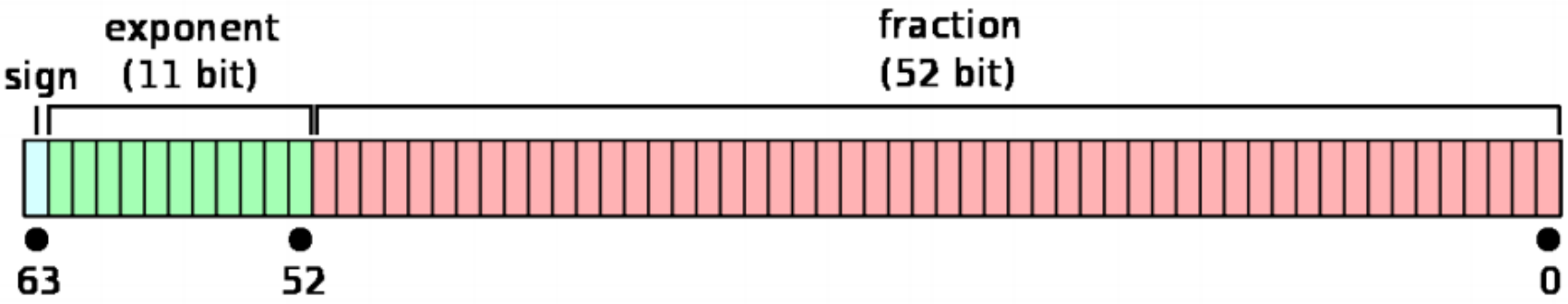
The mantissa, *M*, is formed from the 23 bits as a *binary fraction*. For example, the decimal fraction: 2.783, is interpreted: 2 + 7/10 + 8/100 + 3/1000. The binary fraction: 1.0101, means: 1 + 0/2 + 1/4 + 0/8 + 1/16. Floating point numbers are *normalized* in the same way as scientific notation, that is, there is only one nonzero digit left of the decimal point (called a *binary point* in

base 2). Since the only nonzero number that exists in base two is 1, the leading digit in the mantissa will always be a 1, and therefore does not need to be stored. Removing this redundancy allows the number to have an additional one bit of precision. The 23 stored bits, referred to by the notation: $m_{22}, m_{21}, m_{21}, \ldots, m_0$, form the mantissa according to:

IEEE Double-Precision Binary Floating Point Format

- Sign bit: 1 bit
- Exponent width: 11 bits
- Significand precision: 53 bits (52 explicitly stored)

The format is written with the significand having an implicit integer bit of value 1, unless the written exponent is all zeros. With the 52 bits of the fraction significand appearing in the memory format, the total precision is therefore 53 bits (approximately 16 decimal digits, $53 \log_{10}(2) \approx 15.955$). The bits are laid out as follows:



The real value assumed by a given 64 bit **double precision** data with a given biased exponent **e** and a **52 bit fraction** is $= (-1)^{sign}(1.b_{-1}b_{-2}...b_{-52})_2 \times 2^{e-1023}$ where more precisely we have :

$$value = (-1)^{sign}(1 + \sum_{i=1}^{52} \lceil b_{-i} \rceil 2^{-i}) \times 2^{(e-1023)}$$

Largest and Smallest Positive Floating Point Numbers

Largest Positive Number

0	11111110	11111111111111111111111
Sign 1 bit	Exponent 8 bits	Significand 23 bits

Significand: $1111 \ldots 1 = 1 + (1 - 2^{-23}) = 2 - 2^{-23}$.

Exponent: $(254 - 127) = 127$.

Largest Number $= (2 - 2^{-23}) \times 2^{127} \cong 3.403 \times 10^{38}$.

If the result of a computation exceeds the largest number that can be stored in the computer, then it is called an *overflow*.

Smallest Positive Number

0	00000001	00000000000000000000000
Sign 1 bit	Exponent 8 bits	Significand 23 bits

Significand = 1.0.

Exponent = 1 – 127 = –126.

The smallest normalized number is $= 2^{-126} \cong 1.1755 \times 10^{-38}$.

IEEE Single-Precision Floating Point Format

***Representation of Infinity*:** All 1s in the exponent field is assumed to represent infinity (∞). A sign bit 0 represents +∞ and a sign bit 1 represents –∞. Thus the representations of +∞ and –∞ are:

+∞

0	11111111	00000000000000000000000
Sign 1 bit	Exponent 8 bits	Significand 23 bits

–∞

1	11111111	00000000000000000000000
Sign 1 bit	Exponent 8 bits	Significand 23 bits

Representation of Non Numbers

When an operation is performed by a computer on a pair of operands, the result may not be mathematically defined. For example, if zero is divided by zero, the result is indeterminate. Such a result is called Not a Number (NaN) in the IEEE Standard.

When the result of an operation is not defined (i.e., indeterminate) it is called a Quiet NaN (QNaN). Examples are: 0/0, (infinity)

Representation of Non Numbers

The other type of NaN is called a Signalling Nan (SNaN).

This is used to give an error message.

When an operation leads to a floating point underflow, i.e., the result of a computation is smaller than the smallest number that can be stored as a floating point number, or the result is an overflow, i.e., it is larger than the largest number that can be stored, SNaN is used.

Representation of Non Numbers

When no valid value is stored in a variable name (i.e., it is undefined) and an attempt is made to use it in an arithmetic operation, SNaN would result.

QNaN is represented by 0 or 1 as the sign bit, all 1s as exponent, and a 0 as the left-most bit of the significand and at least one 1 in the rest of the significand.

SNaN is represented by 0 or 1 as the sign bit, all 1s as exponent, and a 1 as the left-most bit of the significand and any string of bits for the remaining 22 bits

Representation of Non Numbers

QNaN

0 or 1	11111111	00010000000000000000000
Sign 1 bit	Exponent 8 bits	Significand 23 bits

The most significant bit of the significand is 0. There is at least one 1 in the rest of the significand.

SNaN

0 or 1	11111111	10000000000001000000000
Sign 1 bit	Exponent 8 bits	Significand 23 bits

The most significant bit of the significand is 1. Any combination of bits is allowed for the other bits of the significand.

Representing Still Images

Two approaches:

(1) Bit map techniques

- Represent image as collection of dots (pixels)
- Black and white binary images
  - Pixel values either 0 (black) or 1 (white)
- Black and white (monochrome) images
  - Pixel values range from 0 to 255 (commonly)
- Colour images
  - Pixel represented as 3 colour components
    - Red, Green & Blue
    - Luminance, red chrominance & blue chrominance

- Problems with resizing, zooming

Representing Still Images

(2) Vector techniques

Image represented as set of lines and curves, specified as a description

Details of drawing left to the device

- Used in computer-aided design (CAD) systems
- Scalable fonts
  - TrueType (MS / Apple)
  - Postscript (Adobe)

Colour depth

24-bit colour

8-bit colour

4-bit colour

1-bit colour

- The colour depth of an image is determined by the available palette size
  - 1-bit images support 2 “colours” black/white binary images
  - 8-bit images support 256 grey levels
  - 24-bit images support 16 Million colours
- Many image formats:
  - .png - Portable Network Graphics Format – approved by W3C to replace the GIF format for the web
  - .bmp - Windows bitmap
  - .pic – PC Paint graphics format
  - .mac – Macintosh MacPaint format
  - .gif (Graphics Interchange Format) – common for graphics on the world wide web
  - .jpg – JPEG image (Joint Photographic Experts Group) – platform independent – used for photos
  - .tif (Tagged image file format (TIFF)

Representing Sound

- Two approaches:

(1) Sample sound wave over time to create a digital representation

(2) MIDI (musical instrument digital interface)

Encode directions for producing the sound on a synthesizer

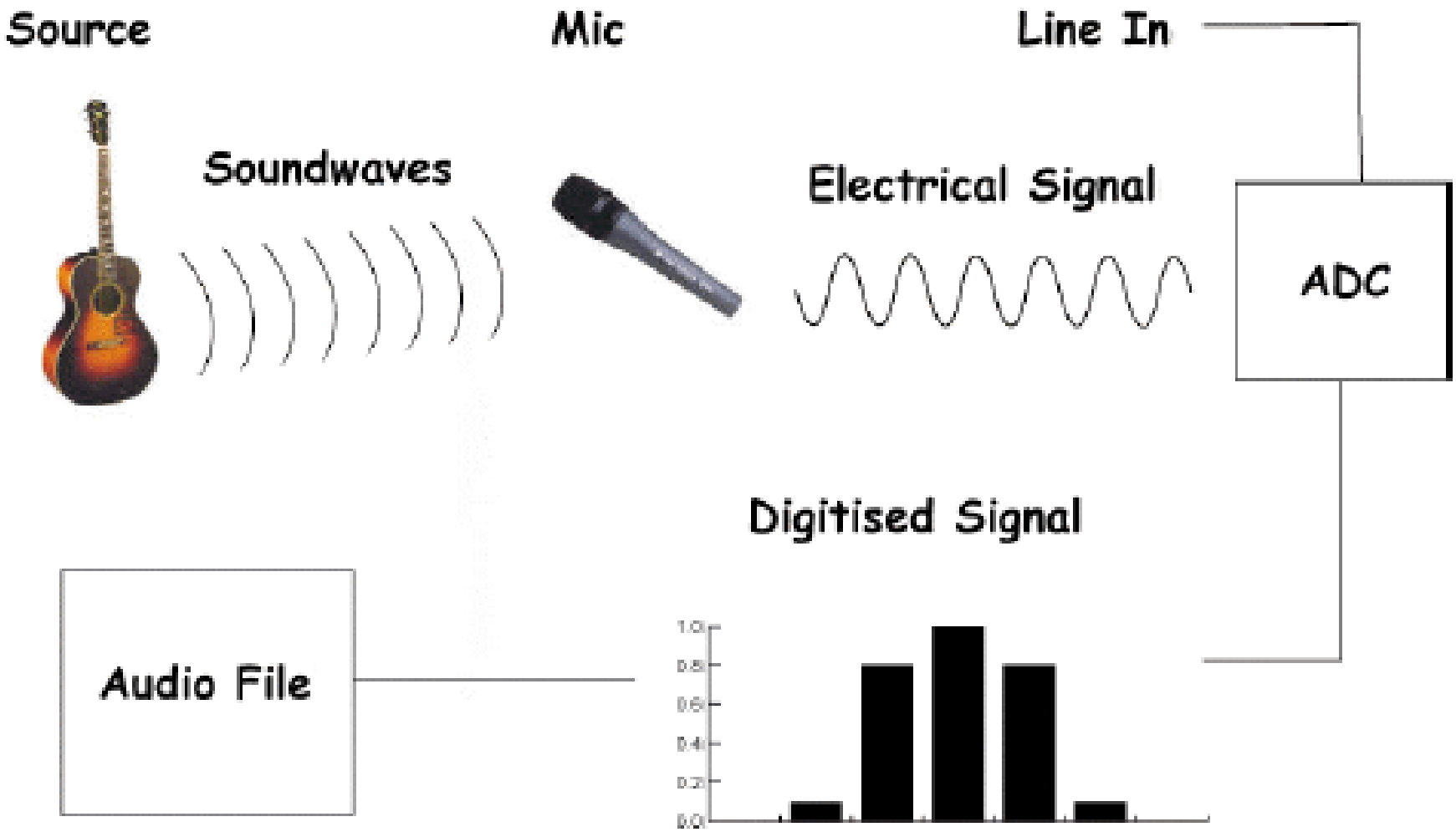
What instruments play which notes for what durations of time

Files take much less data

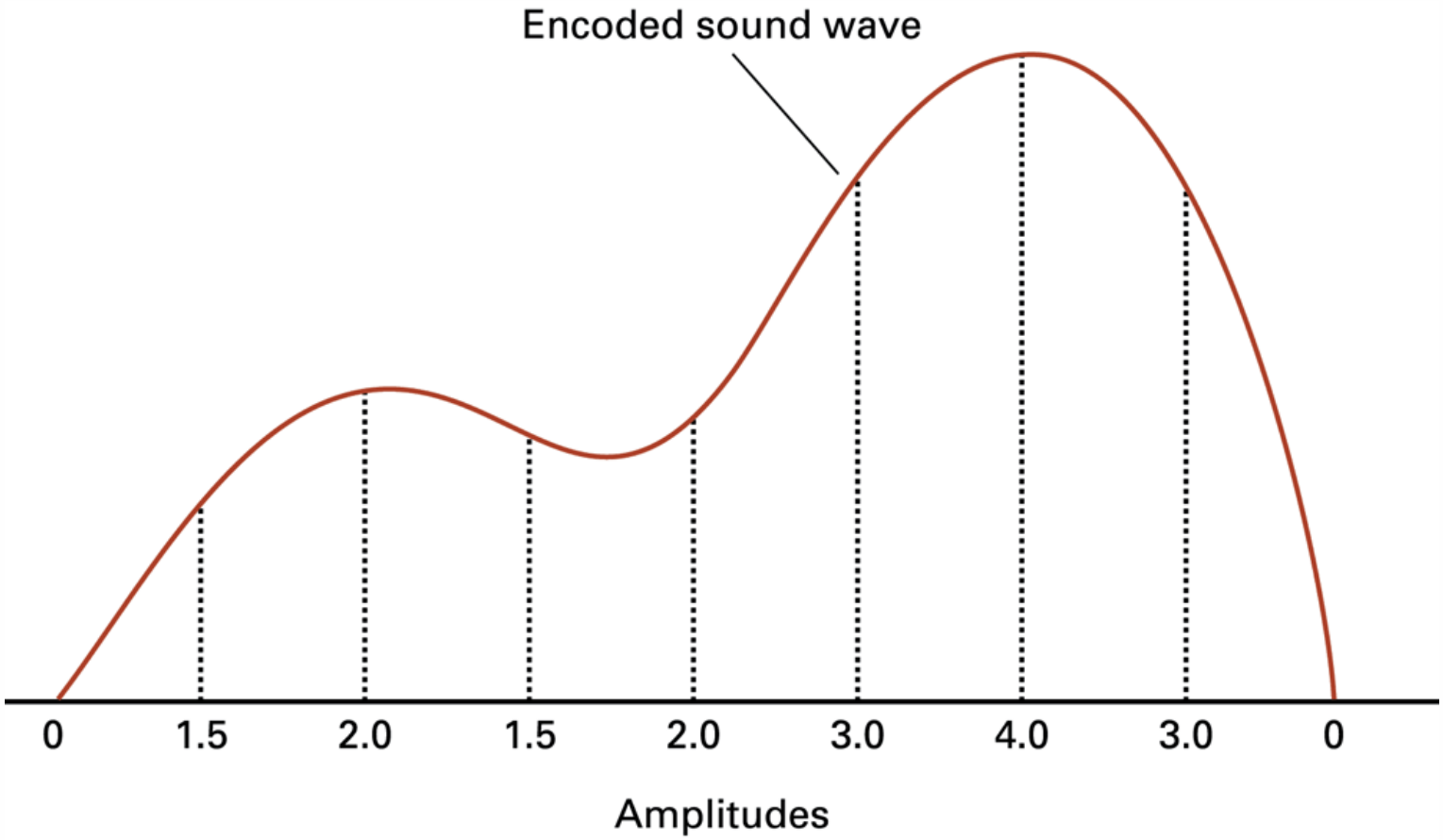
Sounds can be widely different performed on different synthesizers

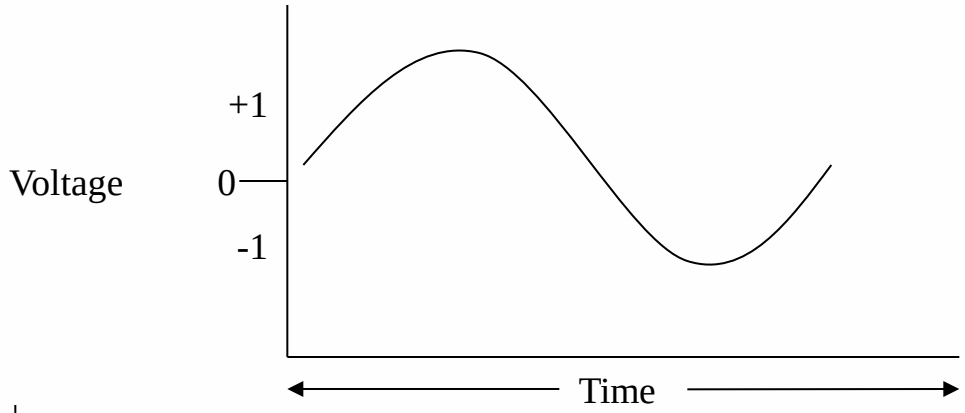
Recording Audio

Analogue to Digital Conversion (ADC)

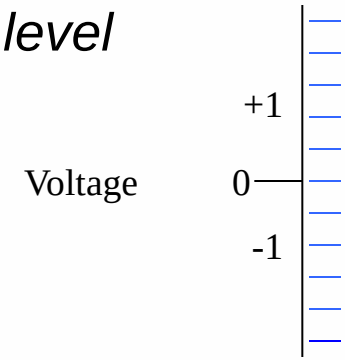


(1) Digitising The sound wave represented by the sequence 0, 1.5, 2.0, 1.5, 2.0, 3.0, 4.0, 3.0, 0

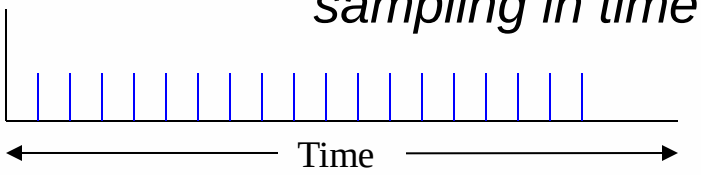


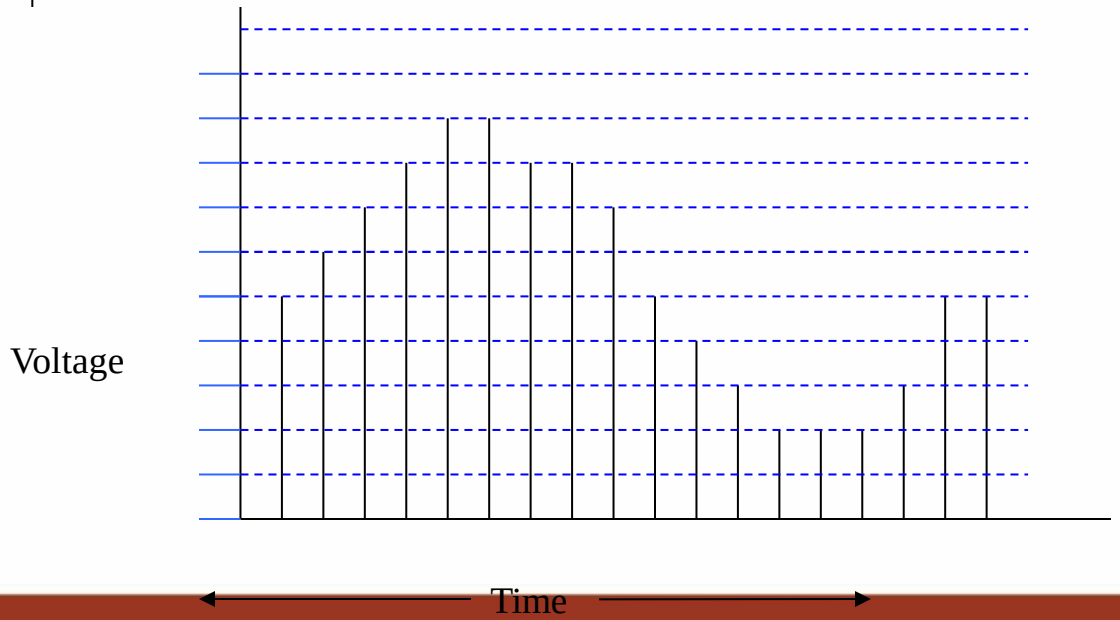


quantization of level



sampling in time





Representing numeric values

- Binary notation – uses bits to represent a number in base two
- Limitations of computer representations of numeric values
  - Overflow – happens when a number is too big to be represented. If the bit sequence is cut off to fit the space available, this is truncation
  - Truncation also happens when a number is between two representable numbers

Types of Numbers

N : Natural or Whole numbers. 0, 1, 2, 3, 4, ….

Z : Integers. … -3, -2, -1, 0, 1, 2, 3, …

Q : Rational numbers. $\{ p/q \mid p, q \in Z, q \neq 0 \}$

R : Real numbers

irrational numbers = **R \ Q**

C : Complex numbers

(irrationals cannot be expressed as a quotient of whole numbers, hence their decimal or binary representation can never terminate)

base ten (decimal - or *denary*) & base two (binary) systems

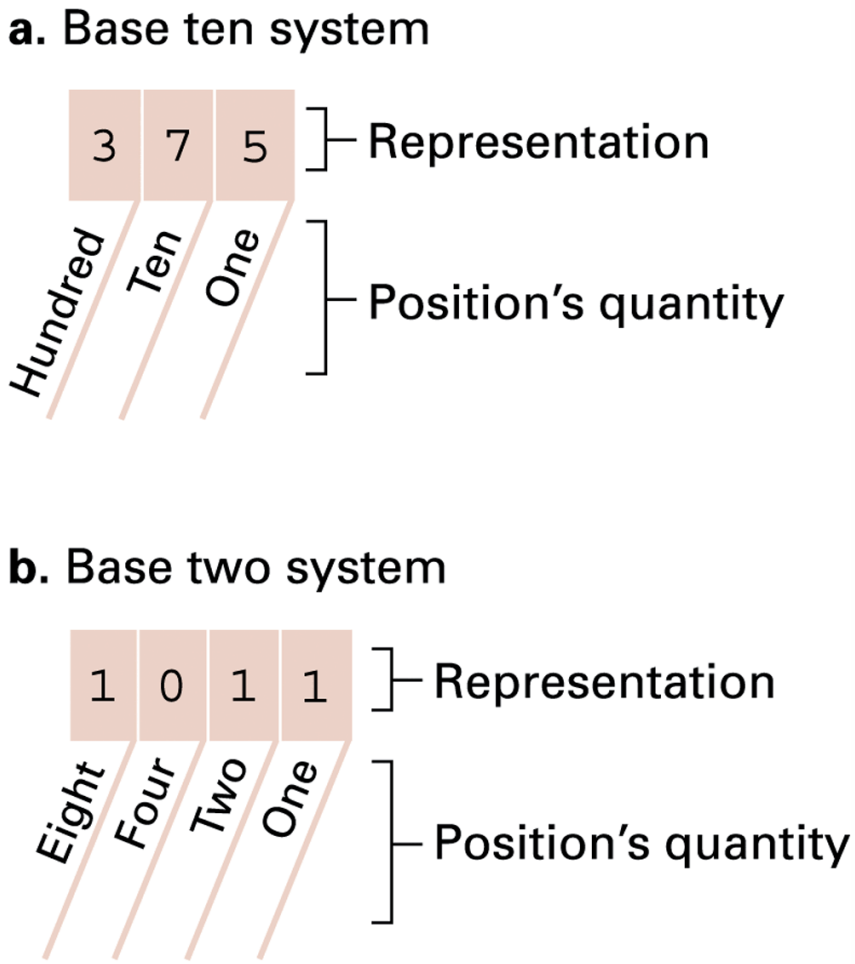


Figure 1.16 Decoding the binary representation 100101

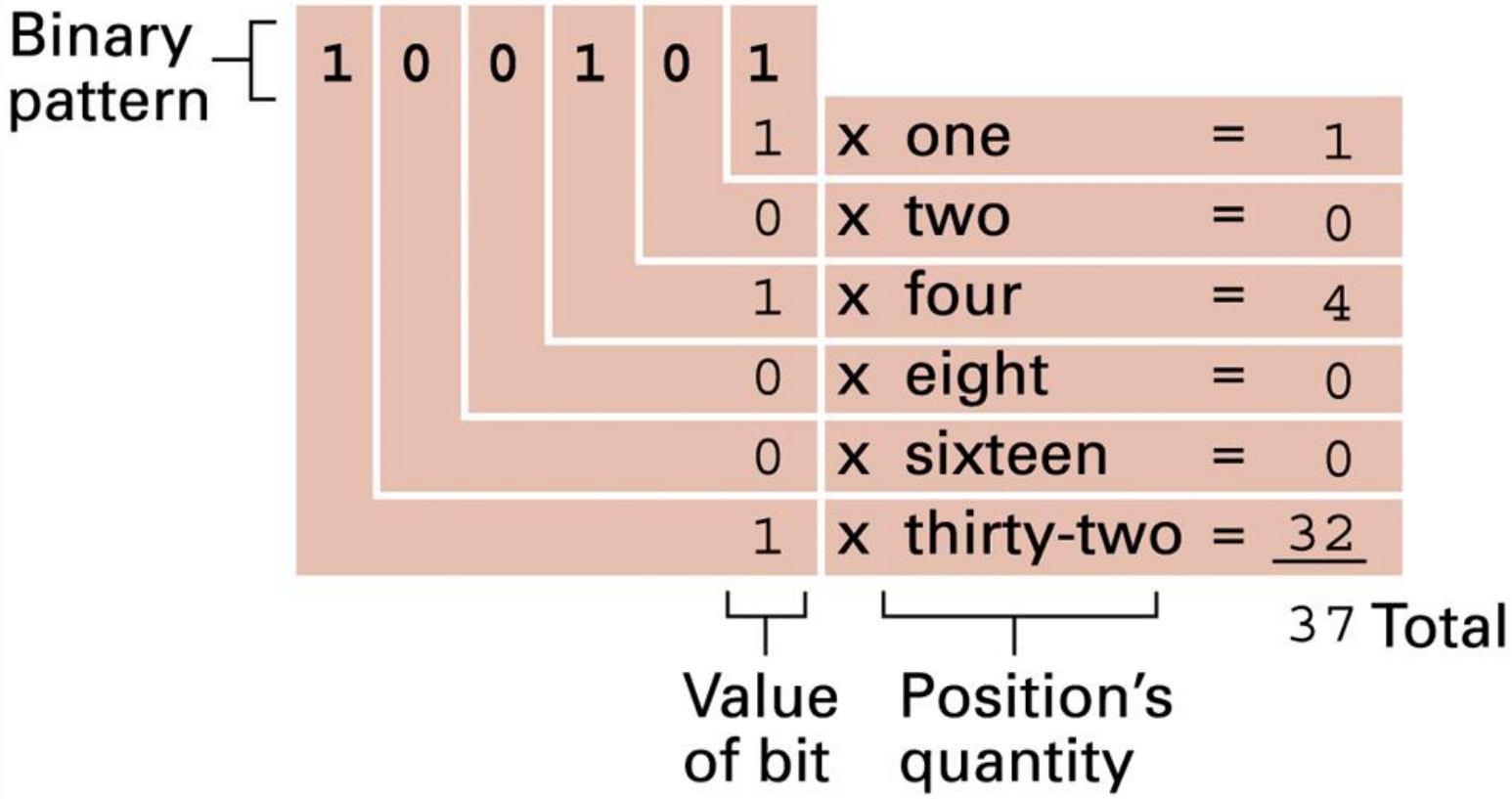


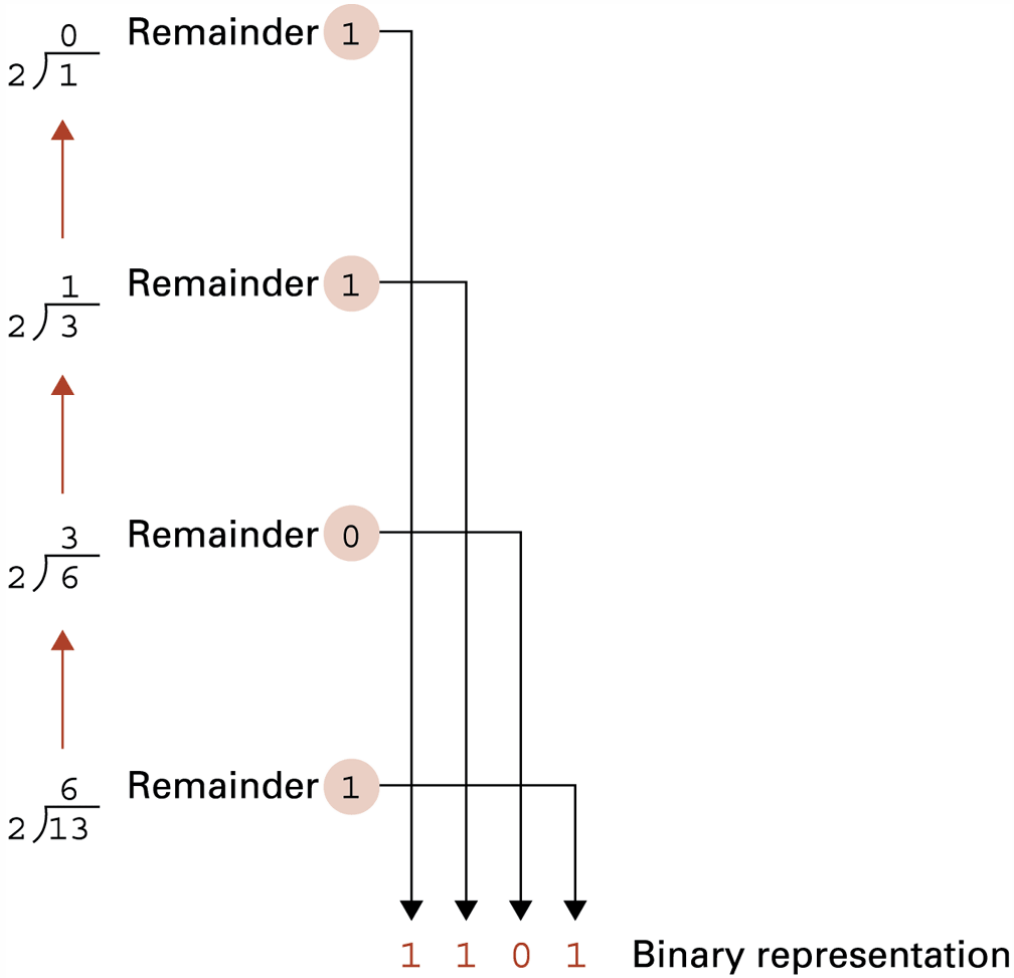
Figure 1.17 An algorithm for finding the binary representation of a positive integer

Step 1. Divide the value by two and record the remainder.

Step 2. As long as the quotient obtained is not zero, continue to divide the newest quotient by two and record the remainder.

Step 3. Now that a quotient of zero has been obtained, the binary representation of the original value consists of the remainders listed from right to left in the order they were recorded.

Figure 1.18 Applying the algorithm in to obtain the binary representation of thirteen



Informal restatement of the algorithm

Finding the binary representation of a positive integer P:

Repeatedly divide P by 2, recording the remainders from right to left. Stop when you get 0 in the result (not as the remainder)

Example: P=13

13 = 2 x 6 +1	1.
6 = 2 x 3 + 0	01.
3 = 2 x 1 + 1	101.
1 = 2 x 0 + 1	1101. STOP

Figure 1.19 The binary addition facts

$$\begin{array}{r} 0 \\ +\,0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ +\,0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 0 \\ +\,1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\,1 \\ \hline 10 \end{array}$$

Decimal addition

Addition in base 10:

$$
\begin{array}{rcccc}
 & 1 & 8 & 0 & 9 \\
+ & 0 & 3 & 2 & 3 \\
\hline
 & 2 & 1 & 3 & 2 \\
 & & \bullet & & \bullet \quad \text{(carry)}
\end{array}
$$

- We start with a carry value of 0.
- Whenever the sum gets larger than 9, we add 1 to the carry.
- The carry is added at the next position.

Binary addition

Addition in base 2:

$$
\begin{array}{rccccc}
 & & 1 & 1 & 0 & 1 \\
+ & & 0 & 1 & 0 & 1 \\
\hline
 & 1 & 0 & 0 & 1 & 0 \\
 & & \bullet & \bullet & & \bullet \quad \text{(carry)}
\end{array}
$$

- We start with a carry value of 0.
- Whenever the sum gets larger than 1, we add 1 to the carry.
- The carry is added at the next position.

Decimal multiplication

Multiplication in base 10:

$$\begin{array}{r} 3\ 1\ 4\ 1 \\ \times\quad 1\ 0\ 0 \\ \hline 3\ 1\ 4\ 1\ 0\ 0 \end{array}$$

When multiplying with $\mathbf{10^n}$ (n=1, 2, ..) simply add **n** zeros to back of the first number.

When dividing into $\mathbf{10^n}$ (n=1, 2, ..) simply delete **n** digits from the back of the first number.

If we want to keep fractional part of answer then move the radix point **n** steps left

Binary multiplication

Multiplication in base 2:

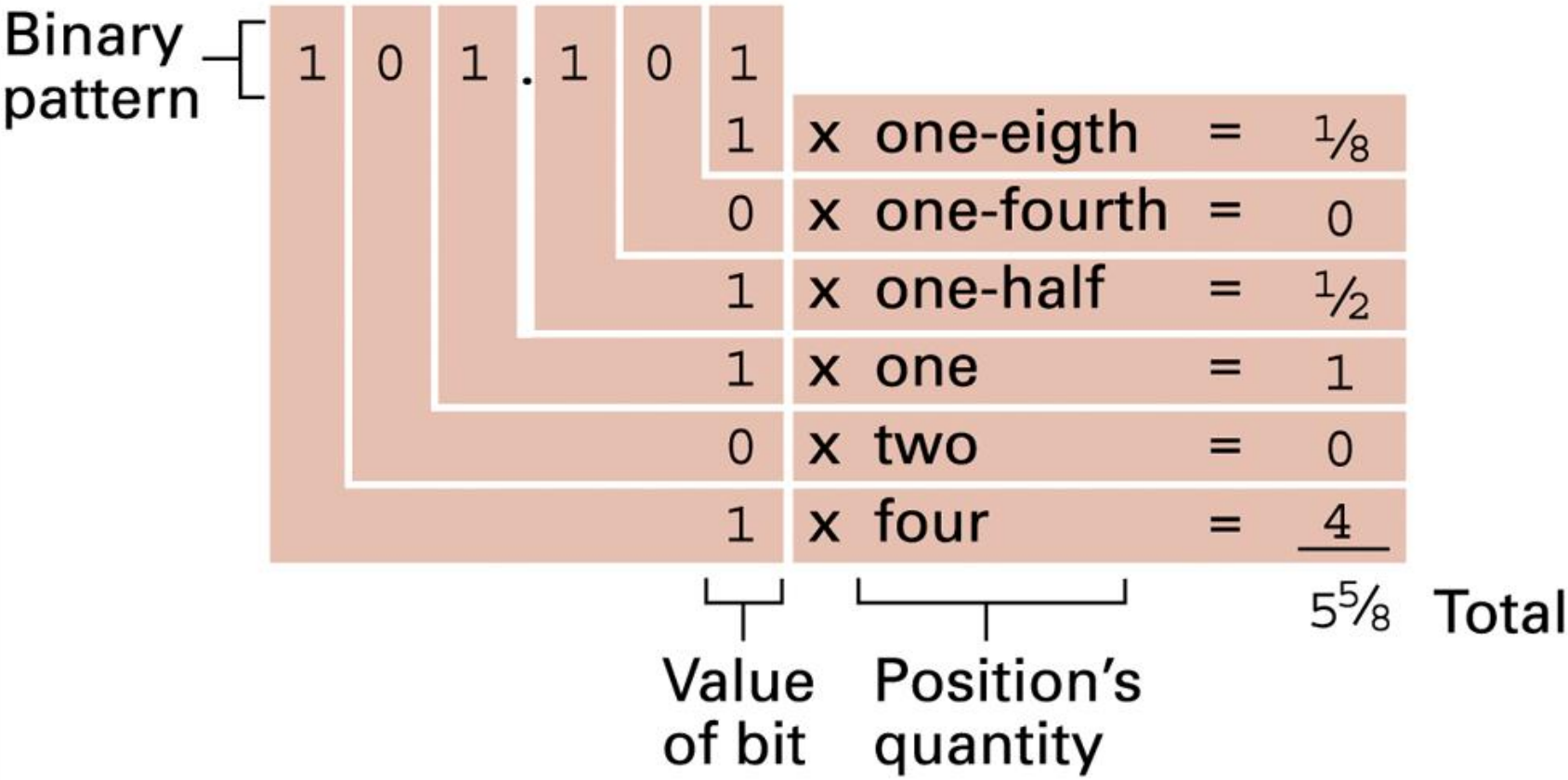
$$
\begin{array}{r}
1\ 1\ 0\ 1 \\
\times\quad 1\ 0\ 0 \\
\hline
1\ 1\ 0\ 1\ 0\ 0\ 0
\end{array}
\qquad (= 2^2)
$$

When multiplying with $\mathbf{2^n}$ (n=1, 2, ..) simply add **n** zeros to back of the first number.

When dividing into $\mathbf{2^n}$ (n=1, 2, ..) simply delete **n** zeros from the back of the first number.

Fractions in binary:

Decoding the binary representation 101.101



Finding the binary representation of a fraction

<u>Informal algorithm</u> for encoding:

Repeatedly subtract successive *negative* powers of 2, recording the success (1) or failure (0) of the subraction, from left to right after the radix point. Stop when subtraction gives result 0

Example: N = 5/8

5/8 = 1/2 +1/8	.1	
1/8 = 0 x 1/4 + 1/8	.10	subtraction failed
1/8 = 1/8 + 0	. 101	STOP

$2^{-1} = 1/2$

$2^{-2} = 1/2^2 = 1/4$

$2^{-3} = 1/8$

binary representation of a fraction
- same method

Example, M = 0.33

0.33 = 0 x 0.5 + 0.33	.0
0.33 = 1 x 0.25 + 0.08	.01
0.08 = 0 x 0.125 + 0.08	.010
0.08 = 1 x 0.0625 + 0.0175	.0101
0.0175 = 0 x 0.03125 + 0.0175	.01010
0.0175 = 1 x 0.015625 + …..	.010101

Process may not terminate.
Stop when available space filled
i.e. truncate the representation

Representing Integers

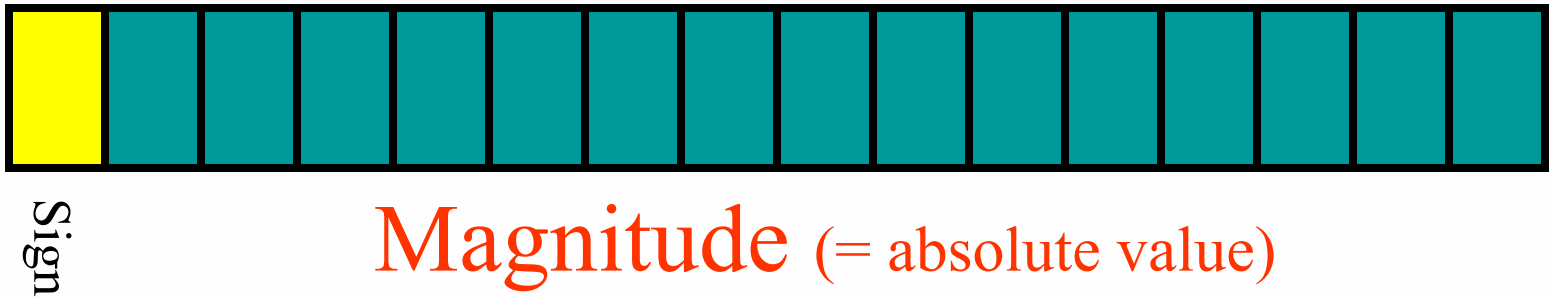
- Unsigned integers can be represented in base two
- Integers occupy 2^n bytes in memory (n usually is 1, 2, 3, 4)

  It is usual that the sign information is explicitly encoded

- Signed integers = whole numbers that can be *positive* or *negative*
- An integer can be coded in:
  - Sign-magnitude notation
  - One's complement notation
  - Two’s complement notation — the most popular representation
  - Excess notation — another, less popular representation, often used for exponent values in floating point representations

Representing Integers (Internally) Sign-magnitude

MSB carries sign **0 : +** **1 : -**



-6 : 1000 0000 0000 0110

6 : 0000 0000 0000 0110

Representing Integers (Internally) Sign-magnitude

Two representations for 0:

-0 : 1000 0000 0000 0000

+0 : 0000 0000 0000 0000

What are the maximum and the minimum values that can fit in S/M in 2-bytes?

Representing Integers (Internally) Sign-magnitude

What are the maximum and the minimum values that can fit in S/M in 2-bytes?

16 bits: 1 sign bit + 15 bits for magnitude

Max magnitude: $2^0 + 2^1 + \ldots + 2^{14} = 2^{15}-1$

$= 1 + 2 + \ldots + 16384 = 32767$

maximum value = 32767

minimum value = - 32767

Representing Integers (Internally) Complement notations

Complement notation provides another way of representing negative numbers.

Two complement types are used frequently, differing in how negative numbers are coded:

1's Complement

Subtract each digit of the number from (base-1)

i.e. 0 becomes 1, 1 becomes 0.

2's Complement

Subtract each digit of the number from 1, as above

Add 1 to the result

Ranges of values that can be represented in n bits are:

1's Complement

$-2^{n-1} -1$ to $2^{n-1} -1$

e.g. For one byte, from -127 to +127

2's Complement

-2^{n-1} to $2^{n-1} -1$

e.g. For one byte, from -128 to +127

The 1's complement representation of a *positive* integer is the same as the original binary, with MSB is set to 0

- *i.e.* Same as sign/magnitude representation

The 1's complement of a *negative* integer is obtained by subtracting its magnitude from 2^n -1 where *n* is the number of bits used to store the integer in binary.

e.g. One byte, *n*=8, subtract from 1111 1111

-1 becomes:

```
   1111 1111
  -0000 0001
=  1111 1110
```

Ones' complement

example using 8 bits (1 byte)

- Has two representations of zero: 00000000 (+0) and 11111111 (-0).
- To add two numbers represented in this system, one does a conventional binary addition, but it is then necessary to add any resulting carry back into the resulting sum: "end-around carry"
- e.g. add -1 and 2

  `11111110 + 00000010 = 00000000`

  with carry 1.

  Add this to LSB, giving result `00000001`

- One’s complement was popular in older computers (e.g UNIVAC, PDP-1)
- The problems of multiple representations of 0 and the need for the end-around carry are eliminated using **two's complement**.
- **2’s Complement of negative integer:**
  - Subtract each digit of its magnitude from 1
  - Add 1 to the result

Representing Integers (Internally) 2's complement notation

Representing binary numbers (here, in one byte only)

2's complement of -17

Binary code for +17 = **0001 0011**

1's complement => **1111 1111 - 0001 0011**
= 1110 1100

2's complement = **1110 1100 + 1** = **1110 1101**

2’s Complement

The 2's complement of a *positive* integer is the same as the original binary, with MSB set to 0

- i.e. Same as sign/magnitude representation

For a *negative* integer, begin as with 1’s complement: Subtract each digit of the number’s magnitude from 1

Then *add 1 to the result*

e.g. One byte, n=8,

-1 becomes: `0000 0001` with bits complemented

= `1111 1110`

& finally `1111 1111` after adding 1

Figure 1.21 Two's complement notation systems

a. Using patterns of length three

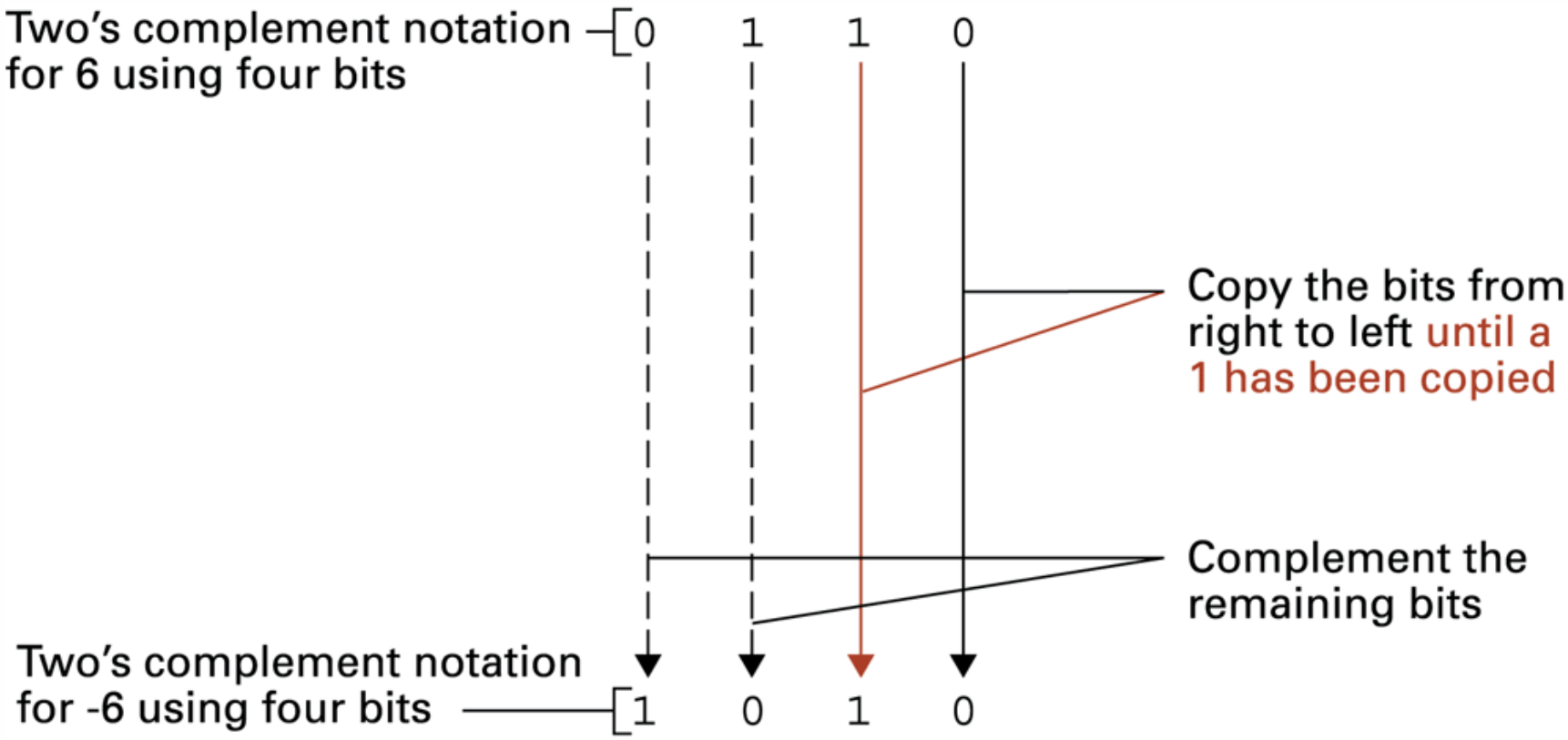
Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

b. Using patterns of length four

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

Figure 1.22 Coding the value -6 in two's complement notation using four bits (using an alternative algorithm)

Start with binary representation of the +ve value:



two's-complement

There is only one zero (00000000).

Negating a number (whether negative or positive) is done by inverting all the bits and then adding 1 to that result.

Addition of a pair of two's-complement integers is the same as addition of a pair of unsigned numbers (except for detection of overflow, if that is done).

Figure 1.23 Addition problems converted to two's complement notation

Problem in base ten	Problem in two's complement	Answer in base ten
3 + 2	0011 + 0010 0101	5
−3 + −2	1101 + 1110 1011	−5
7 + −5	0111 + 1011 0010	2

two's-complement addition

- The largest positive integer that can be stored in N bits in 2's complement form is 2^{N-1} -1, whose binary bit pattern is one 0 followed by N-1 1s
  - e.g one byte, N=8,

    largest +ve value = 2^7-1 = 127

    $127_{10} = 0111\ 1111_2$

- Therefore, if two positive 2's complement integers are added and their sum is greater than +127 an **overflow** will occur.

```
e.g. 100 + 30 :      01100100
                +    00011110
                   =  10000010
```

MSB (sign bit) indicates –ve value

2-complement addition

Rule for overflow detection:

2's complement overflow occurs when the carry into the MSB is not equal to the carry out from the MSB. (Overflow detector circuit is used)

e.g. in previous example, one carry into MSB, but no carry-out → overflow

```
e.g. 100 + 30 :      01100100
                +    00011110
                  =  10000010
                      .....     ← carries
```

2-complement addition

second example:

```
e.g. -60 -68
     -68   :     10111100
     -60       + 11000100
               = 10000000
                 ......
```

one carry into MSB, but one carry-out

– no overflow: answer *is* -128

Excess-N representation

A value is represented by the unsigned number which is *N* greater than the intended value. Thus 0 is represented by *N*, and −*N* is represented by the all-zeros bit pattern.

e.g. with 4 bits use excess-8 [4th bit is 8 in binary]

NB MSB = 0 for –ve values
1 for +ve values

Figure 1.24 An excess eight conversion table

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

Figure 1.25 An excess notation system using bit patterns of length three (excess-4)

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

Representing Integers (Internally)

External data is converted into internal form, with the aid of the programming language being used:

Using C, a programmer would:

- Select concrete type for the data (according to rules)

  int, long, char, double ...

- Select a name for the data (to be able to use it):

```
int   A ;
double  F;
```

- Specify the value (in external notation) to be used

  A = 0xFF; or A = 255;

Representing Integers (Internally)

During compilation, compiler will:

- Check for correctness, overflows, underflows
- Select the size of data (2-byte, 4 byte)
- Convert value into internal form (sign magnitude, 2's complement, Floating point ...)
- Allocate memory and place data in memory
- Generate code to correctly manipulate data (integers in 2's complement arithmetic, floats in FP arithmetic, etc)

Representing Integers (Internally)

Algorithm in coded form is a computer program, or program's code.

Translation from concepts to computer internal form is done in several phases, and each involves coding with a different style:

From concepts to a high level language (*)
From high level language to low level language,
From low level language to machine language,
From machine language to internal (voltage) form

(*) only this is the programmer's task.

Terminology

Data type	Veri türü
Base	Taban
Representation	Gösterim
Notation	Yazım
Sign-Magnitude (representation)	İşaret/büyüklük (gösterimi)
One's complement	1’in tümleri
Two's complement	2’nin tümleri
Prefix	Ön-ek

Fractional (Real) numbers

- Radix (base) : Number of different digit symbols.
- Positions are numbered, around the dot (if any).
- Each digit resides in a weighted position and the weight of each position is a power of the radix.

- An **N.M** digit number:

D_{N-1}	D_{N-1}	...	D_2	D_1	D_0	●	D_{-1}	D_{-2}	D_{-3}	...	D_{-M}
N-1	N-2	...	2	1	0		-1	-2	-3	...	-M

Representing Numeric Data (Externally)
Positional notation

D_{N-1}	D_{N-2}	...	D_2	D_1	D_0	●	D_{-1}	D_{-2}	D_{-3}	...	D_{-M}
N-1	N-2	...	2	1	0		-1	-2	-3	...	-M

Integer part $= (D_{N-1}\, r^{N-1}) + (D_{N-2}\, r^{N-2}) + \ldots + (D_0\, r^0)$

Fractional part $= (D_1\, r^{-1}) + (D_2\, r^{-2}) + \ldots (D_M\, r^{-M})$

Value = $$\sum_{i=-M}^{N-1} D_i\, r^i$$

r = radix (base)

D = digits

N: # of digits (integer)

M: # of digits (fraction)

Representing Numeric Data (Externally)

Positional notation — Base 10 (Decimal)

Example

Numeral: **809.25**

r =10

N = 3

M = 2

Integer part = $8 \times 10^{2} + 0 \times 10^{1} + 9 \times 10^{0} =$ **809.**

Fractional part = $2 \times 10^{-1} + 5 \times 10^{-2} =$ **.25**

Representing Numeric Data (Externally)

Positional notation — Base 2 (Binary)

r = 2, D= 0 or 1

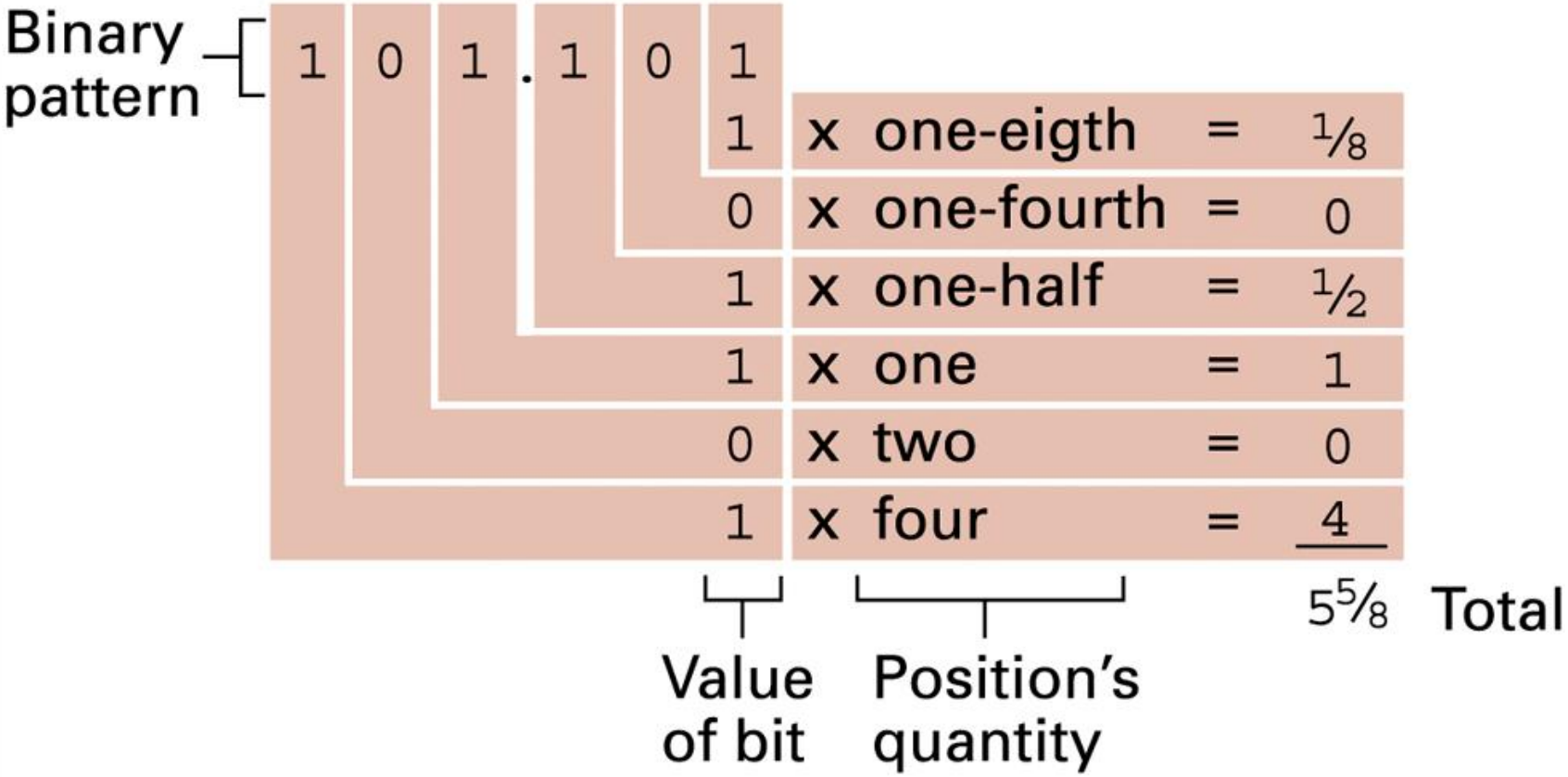
D_{N-1}	D_{N-2}	...	D_2	D_1	D_0	●	D_{-1}	D_{-2}	D_{-3}	...	D_{-M}
N-1	N-2	...	2	1	0		-1	-2	-3	...	-M

Integer part = $(D_{N-1}\, 2^{N-1}) + (D_{N-2}\, 2^{N-2}) + \ldots + (D_0\, 2^0)$

Fractional part = $(D_1\, 2^{-1}) + (D_2\, 2^{-2}) + \ldots (D_M\, 2^{-M})$

Value = $$\sum_{i=-M}^{N-1} D_i\, 2^i$$

Fractions: Decoding the binary representation 101.101



Fixed Point notation

- A number of bits sufficient for the precision and range required must be chosen to store the fractional and integer parts of a number. For example, using a 32-bit format, 16 bits might be used for the integer and 16 for the fraction.
  - used for applications where high precision is required, e.g. finance

Floating-point

- In the decimal system, we are familiar with floating-point numbers of the form:

  $1.1030402 \times 10^5 = 1.1030402 \times 100000 = 110304.02$

  or, more compactly: 1.1030402E5

  where E5 can be viewed as an instruction to move the decimal point 5 places to the right

Representing Numeric Data

Floating point notation

To manipulate very large or small numbers.

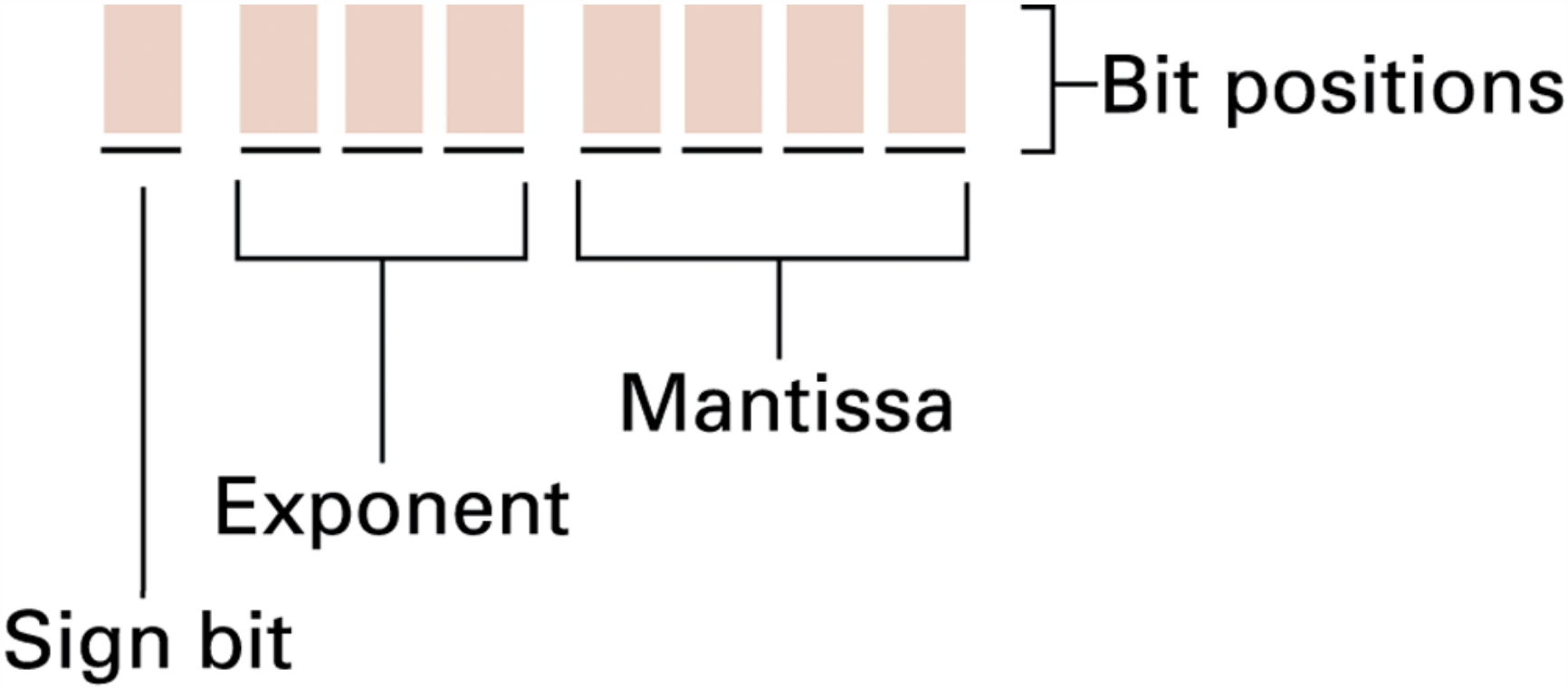
General form is:

$$N = \text{sign} \times M \times R^{E}$$

sign: - , + , or not shown(+)
M : Mantissa, a fixed-point number
R: Radix (base) usually 10
E: Exponent, an integer

Only the red parts need to be stored; the rest are implied

Figure 1.26 Floating-point notation components



Floating point notation

- Bits considered as three fields of *fixed size*:
  - Sign bit (MSB)
  - Exponent field
  - Mantissa field

- Previous example using 1 byte

  1 sign bit; 3 bits for exponent; 4 bits for mantissa

- Zero is encoded as string of 0s, e.g. 0000 0000

Encoding a number in floating-point notation

Example: -4 (decimal)

Convert to binary: -100.

Sign bit = 1, magnitude = 100. with Exponent 0

Copy magnitude into Mantissa field, from left to right, starting with leftmost 1, padding with 0s if necessary:

```
1 EEE 1000
```

!!But we must imagine radix point to left of leftmost 1

Encoding a number in floating-point notation (continued)

Example: -4 (decimal) = -100. (binary)

With sign bit set and magnitude copied to mantissa:

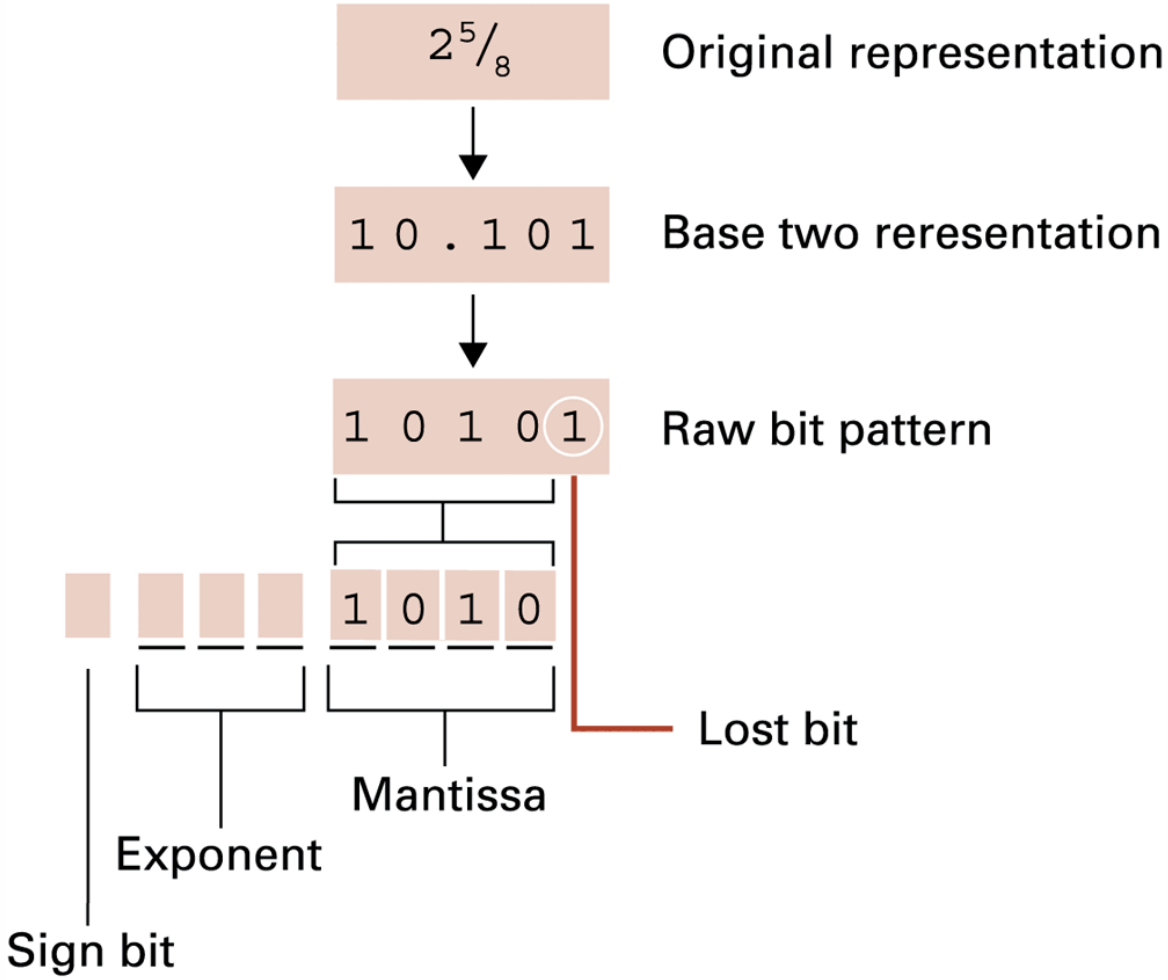
```
1 EEE 1000
```

But the implicit radix point to left of leftmost 1 would indicate the value 1/2 for this pattern

To get the correct value of 4 we would need to multiply by 8, i.e. 2^3 (representing moving radix point 3 places left). This gives us the necessary exponent value 3.

Enter this in excess-4 notation:

```
1 111 1000
```


$2^5/_8$
Original representation
10.101
Base two reresentation
1 0 1 0 1
Raw bit pattern
1 0 1 0
Lost bit
Mantissa
Exponent
Sign bit

Normalised form

<u>RULE: Leading digit in mantissa field must be 1</u>

– Increment / decrement exponent to compensate for each shift of one position

- Example: 3/8 (decimal) = 0.011

$0.011 \times 2^{0} = 0.11 \times 2^{-1}$

Store as 0 011 1100 and not 0 100 0110

exponent is -1 in excess-4 notation, not zero

Normalised form

- Normalised form prevents the possibility of multiple representations for the same value
  - e.g. 00111100 and 01000110 would both decode to the value 3/8, but only the first is normalised
- The only exception to the normalised form rule that the mantissa starts with 1 is the special case of the value zero
  - zero is represented by a bit pattern of all 0s

Decoding a number in floating-point notation (SEEEMMMM)

Example: 11101110

Sign bit = 1,
Magnitude = 0.1110 !!Note radix point inserted on left
Exponent = 110 in excess-4 notation = 2

We could decode at this point by converting 0.111 to decimal = $^1/_2 + {}^1/_4 + {}^1/_8 = {}^7/_8$ and then multiplying by 2^2

$^7/_8$ X 4 = 7/2 = $3^1/_2$

But remember sign bit: final answer is $-3^1/_2$

Decoding a number in floating-point notation (SEEEMMMM) continued

Example: 11101110

Sign bit = 1, Magnitude = 0.1110 Exponent = 2

But it is easier to apply the exponent before converting to decimal. Applying exponent of 2 simply shifts the radix point in the magnitude 2 places right, giving:

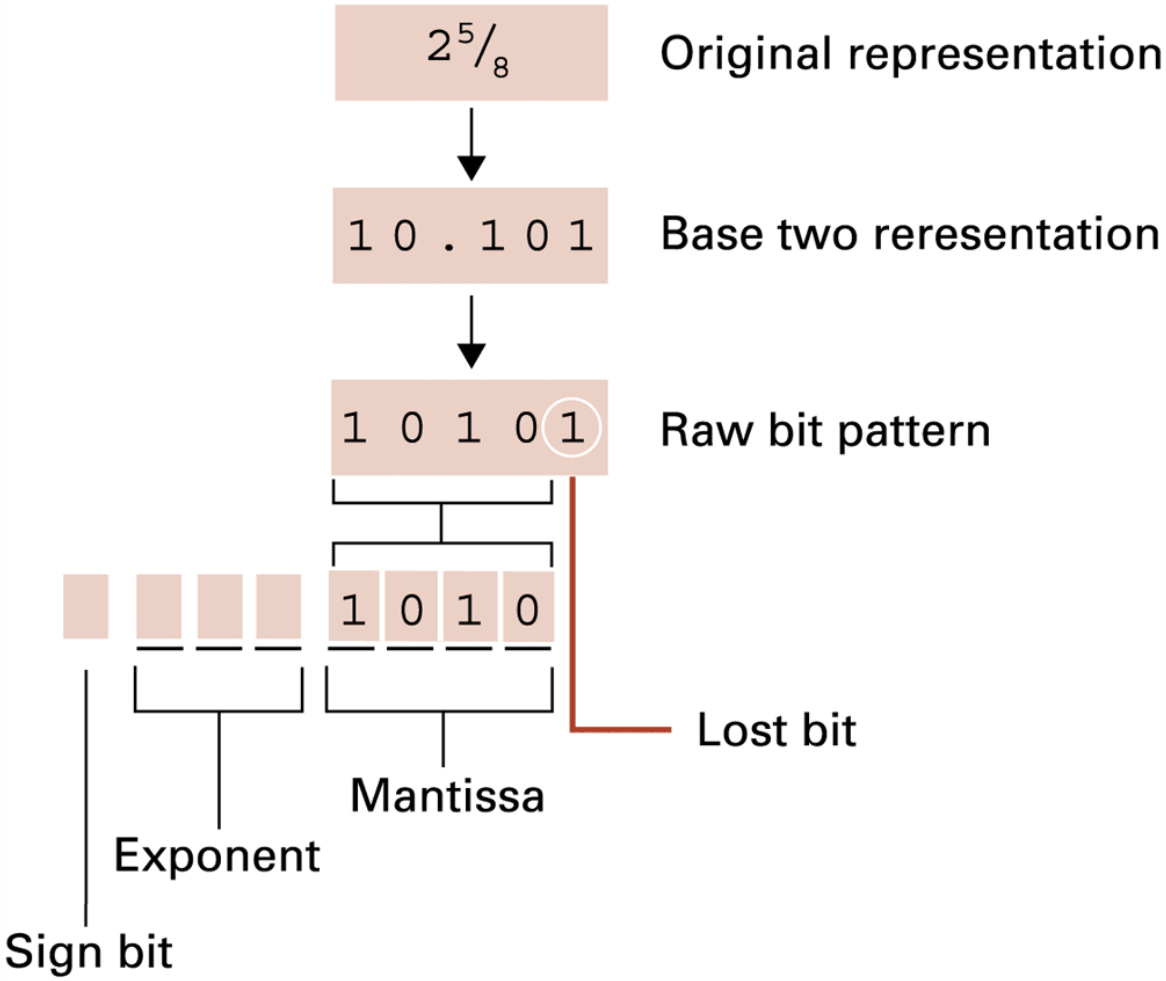
Sign bit = 1, Magnitude = 11.10 Exponent = 0

Now convert magnitude to decimal, giving $2+1+{}^{1}/_{2} = 3{}^{1}/_{2}$

Apply sign bit: final answer = $-3{}^{1}/_{2}$

Truncation Errors

e.g Coding the value $2^5/_8$



Floating point encoding

The fixed mantissa field size can lead to truncation error (round-off error)

Truncation errors can be reduced by using longer mantissa fields

Common representation uses at least 32 bits for sign bit + mantissa + exponent. But errors can still happen.

Floating point encoding

1/10 does not terminate as a binary fraction
(just as 1/3 does not terminate as a decimal:
0.33333333…)

Truncation errors in numerical calculations

Truncation may occur in adding very large and very small values.
Example $2^1/_2 + 1/8 + 1/8$, using 1 byte representation (S EEE MMMM): $2^1/_2 + 1/8$ is truncated, result is $2^1/_2$
Repeatedly adding 1/8 has no effect !

Truncation Errors

Minimise the problem by adding small numbers first, then add to larger numbers

Such strategies to reduce errors can be hidden from the user.

Modern spreadsheet software should not give problems unless range of values >= 10^{16}

Normalised form

<u>RULE: Leading digit in mantissa field must be 1</u>

– Increment / decrement exponent to compensate for each shift of one position

- Example: 3/8 (decimal) = 0.011

$0.011 \times 2^{0} = 0.11 \times 2^{-1}$

Store as 0 011 1100 and not 0 100 0110

exponent is -1 in excess-4 notation, not zero