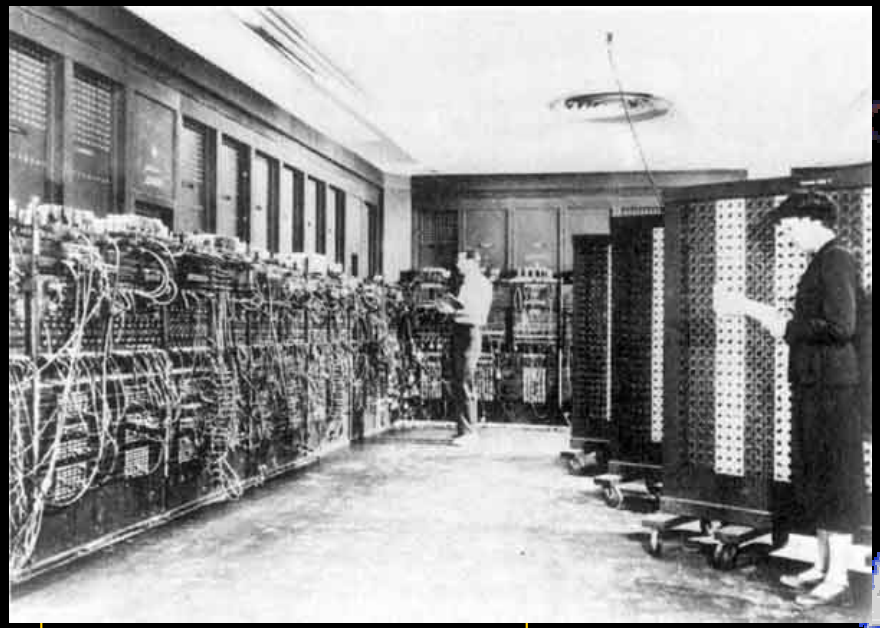
http://bb.nctu.edu.tw/

計算機概論



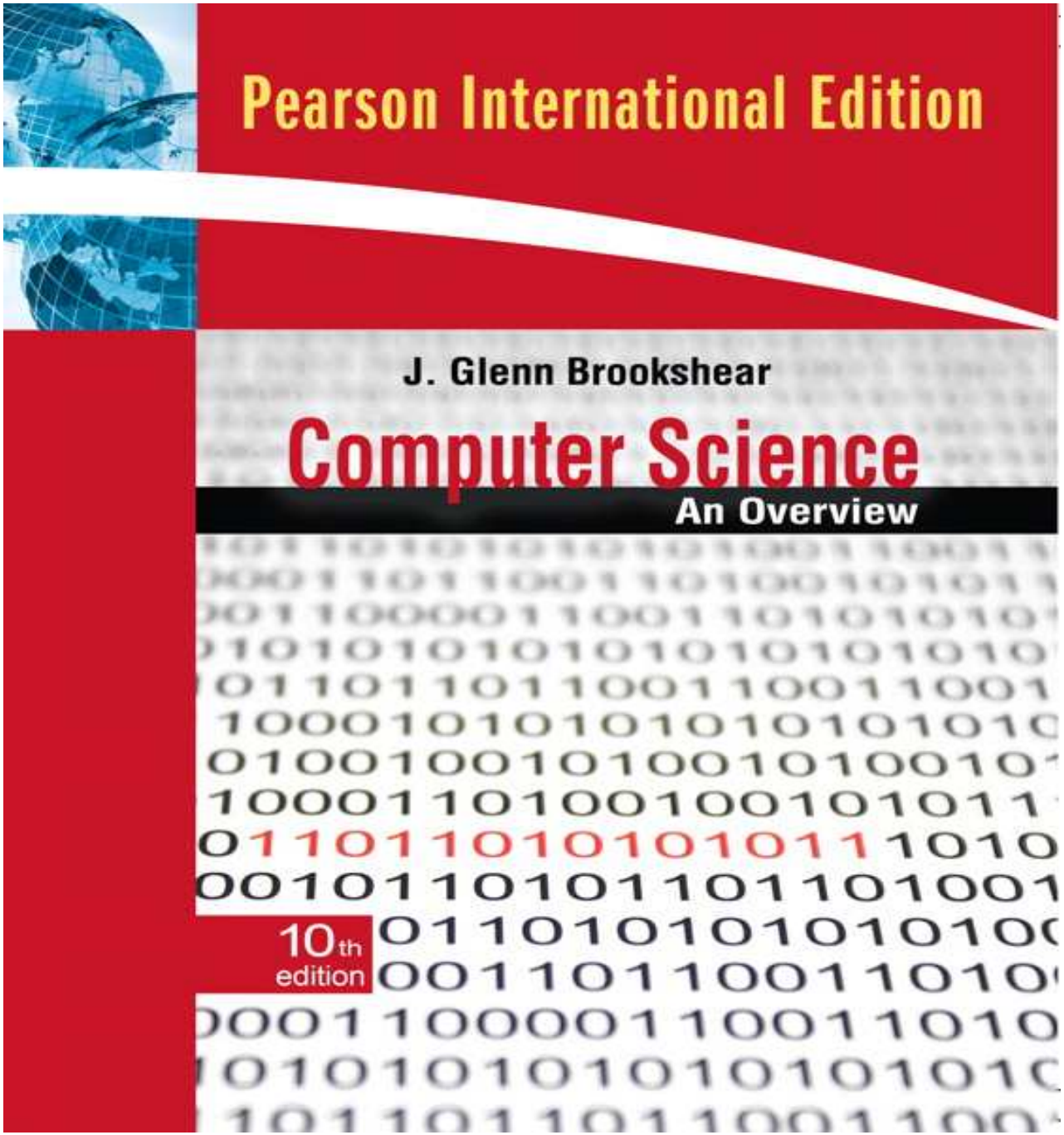
ENIAC

蔡文能

tsaiwn@csie.nctu.edu.tw

交通大學資訊工程學系

CHAPTER 1

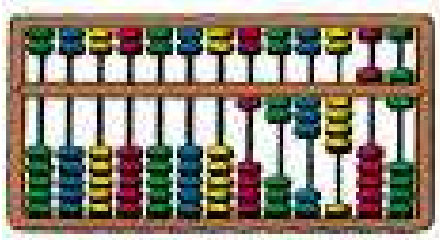


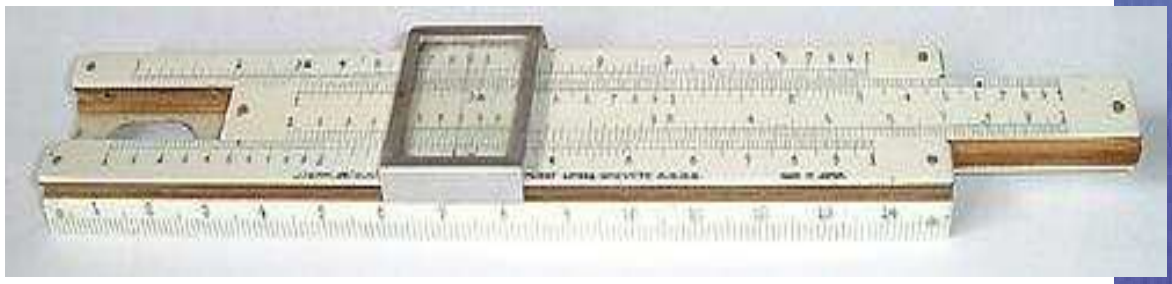
計算機科學概論?

- 概論？蓋論？　計算機概論 的 概 論
- 資訊素養(電腦常識)? BCC?(**Basic Computer Concept**)
- 電腦歷史 Computing History
  - 史上第一部大型電腦 ENIAC
  - **E**lectronic **N**umerical **I**ntegrator **A**nd **C**alculator
  - 誕生於61 年前的情人節： **1946 – 02 – 14**
- 傷腦筋的**頭字語**(Acronym)
- 電腦如何表示資料: **二進位**也可以很有趣 ☺
- 電腦硬體? 軟體? 作業系統? **演算法**? …

Computing History 電腦歷史

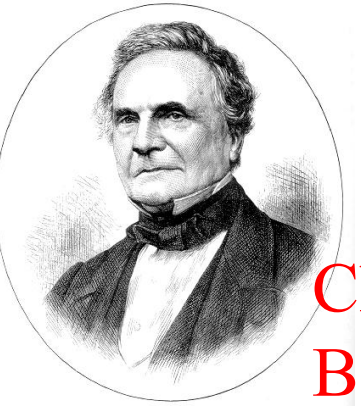
•0: BC 500: abacus , 1625: slide rule, by William Oughtred





• 1642: Pascal's adding machine, by Blaise Pascal

Blaise Pascal (1623 – 1662)



Charles Babbage

Pascaline: 機器可以幫人做事

CHAPTER 1 Data Storage

Bits and Bit Patterns

- **Bit (BIT):** Binary digIT (0 or 1)
- Bit Patterns are used to represent information.
  - Numbers
  - Text characters
  - Images
  - Sound
  - Video
  - Etc…

Boolean Operations

- **Boolean Operation:** An operation that manipulates one or more true/false values
- Specific operations
  - AND
  - OR
  - **NOT**
  - XOR (eXclusive OR)

Bitwise operations

因爲電晶體特性,
在硬體實作元件實務上
AND 是由
NAND 再 NOT 做出

NOR? NAND?

用 gogle.com 找看看 NAND 型 Flash與NOR型的差別

Operators (運算符號) in C/C++/Java Languages

- ( ) [ ] -> . 例如 p->id = xstu.id ; k = (2+3)*5;
- ! ~ ++ -- + - * & (cast_type) sizeof
- * / % (乘 除 取餘數)
- + -
- << >> (往左 shift, 往右 shift)
- < <= > >=
- == != (equal, not equal)
- & (bitwise and, mathematical and)
- ^ (xor)
- | (bitwise or, mathematical or)
- && (logical and, 邏輯 AND)
- || (logical or, 邏輯 OR)
- ?: (trinary operator)
- = += -= *= /= %= &=
- ^= |= <<= >>=
- , comma (逗號) operator, 例如 a=b+c, x=m+n, yy=a+x;

參考K&R課本2.12節

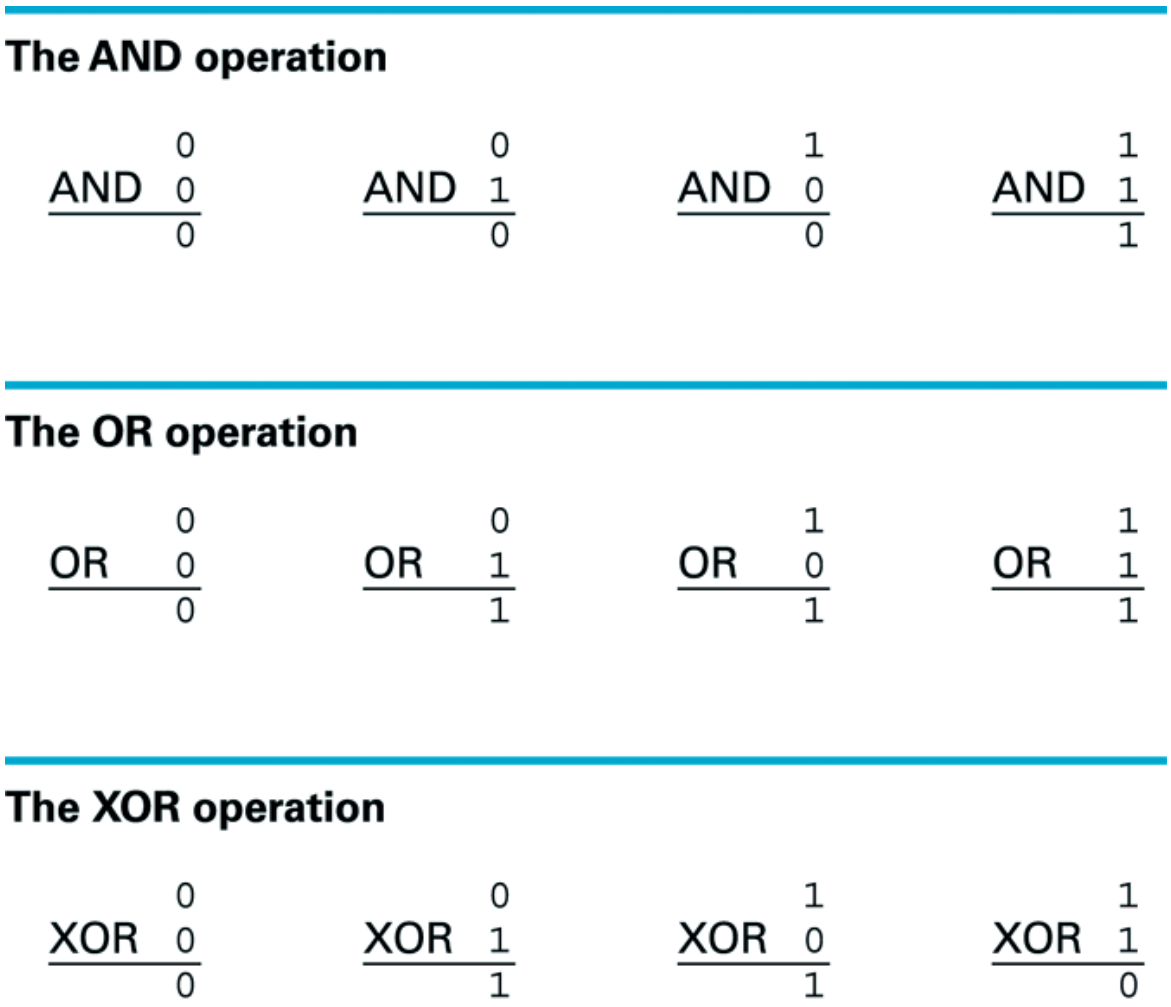
Operator precedence

Left association

Right association

見K&R第2章最後一頁

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



Gates 邏輯閘(Logic Gates)

- **Gate:** A device that computes a Boolean operation
  - Often implemented as (small) electronic circuits
  - Provide the building blocks from which computers are constructed
  - VLSI (Very Large Scale Integration) /ULSI

Bill Gates 比爾蓋茲 (William Henry Gates)
Bill is a common nickname for *William*.
Bob is a common nickname for *Robert*.
Dick is a common nickname for *Richard*.

Figure 1.2a: A pictorial representation of AND, OR, XOR, and NOT gates as well as their input and output values (1/2)

AND

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

OR

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

Figure 1.2b:A pictorial representation of AND, OR, XOR, and NOT gates as well as their input and output values (2/2)

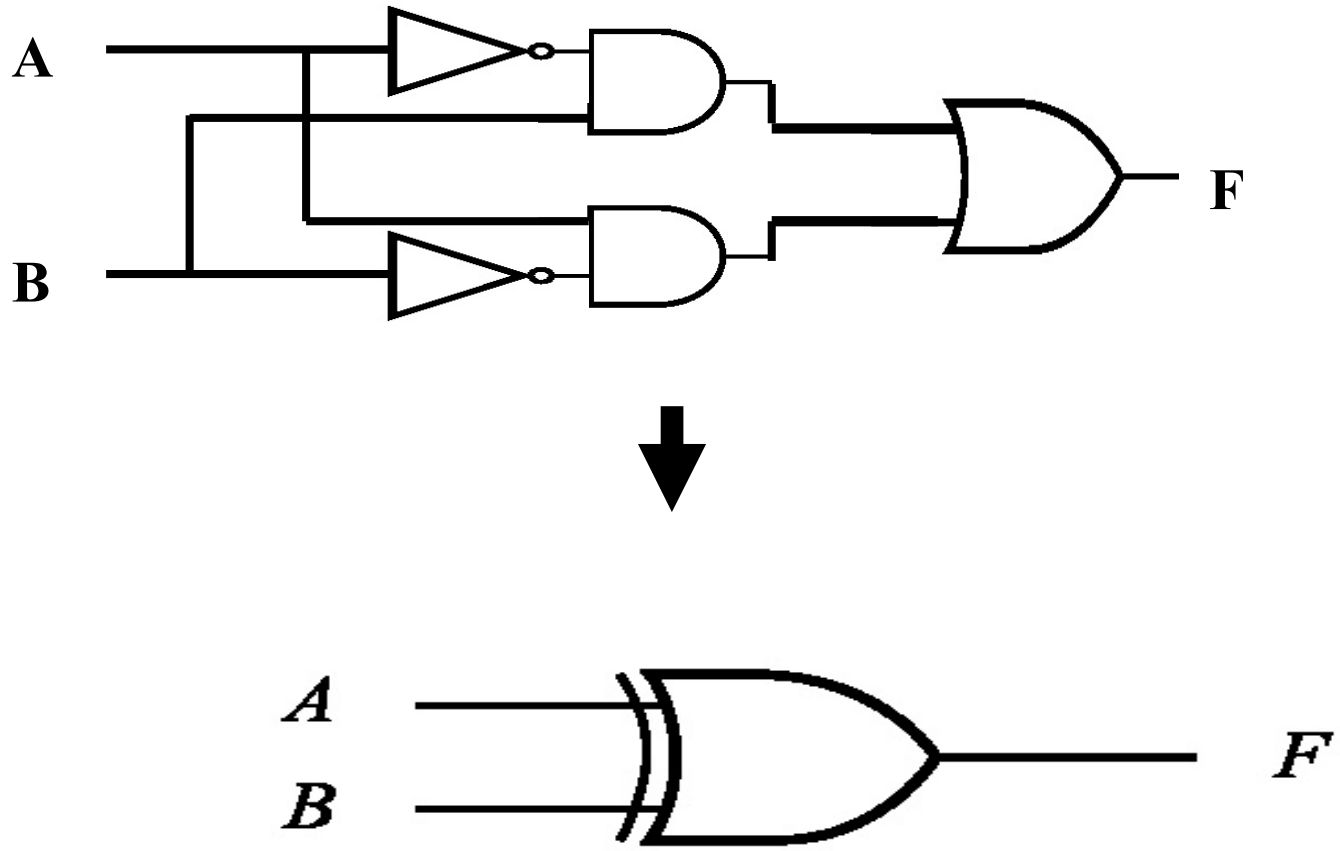
XOR

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

NOT

Inputs	Output
0	1
1	0

XOR Gate (互斥或閘 ;不相容或閘)



如何表示一個 bit ?

Flip-flops 正反器

- **Flip-flop:** A circuit built from gates that can store one bit.
  - One input line is used to set its stored value to 1
  - One input line is used to set its stored value to 0
  - While both input lines are 0, the most recently stored value is preserved

Figure 1.3:
A simple **flip-flop** circuit

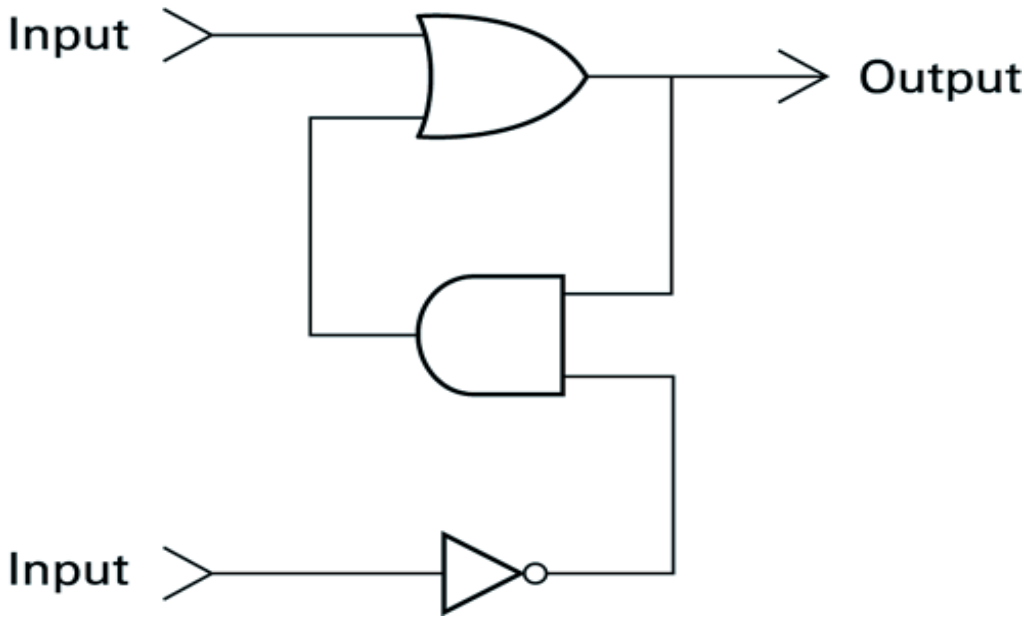
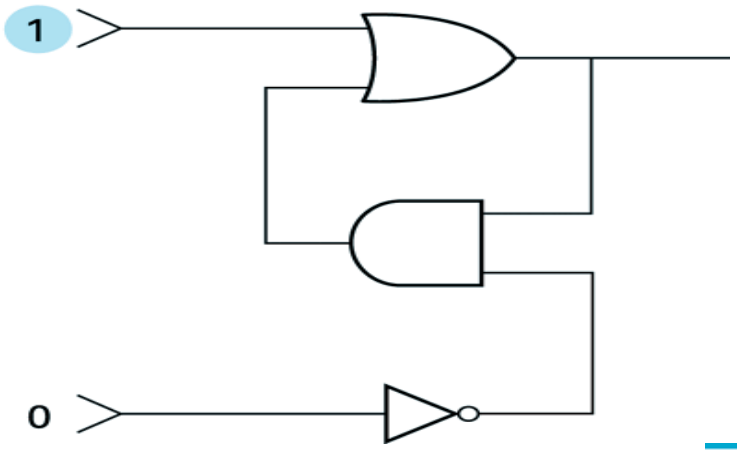


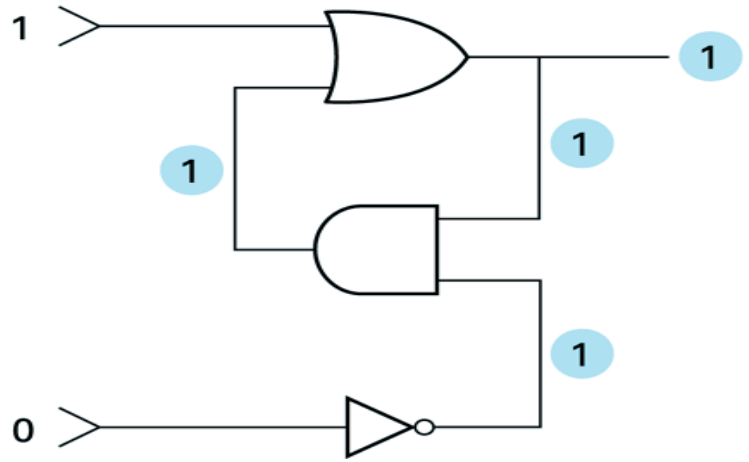
Figure 1.4:

Setting the output of a flip-flop to 1

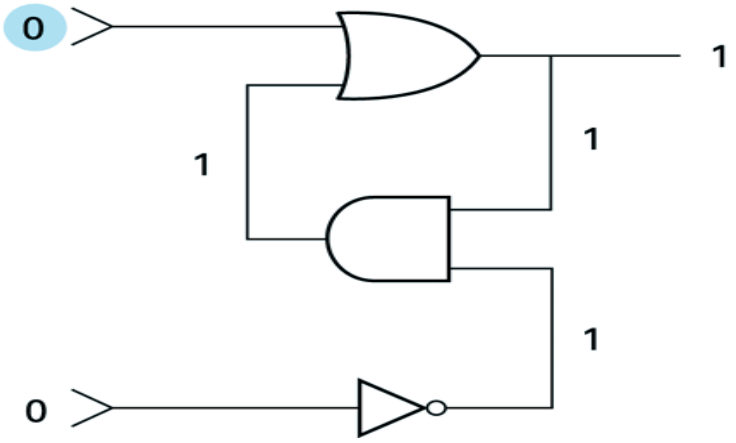
a. 1 is placed on the upper input.



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



c. The 1 from the AND gate keeps the OR gate from changing after the upper input returns to 0.

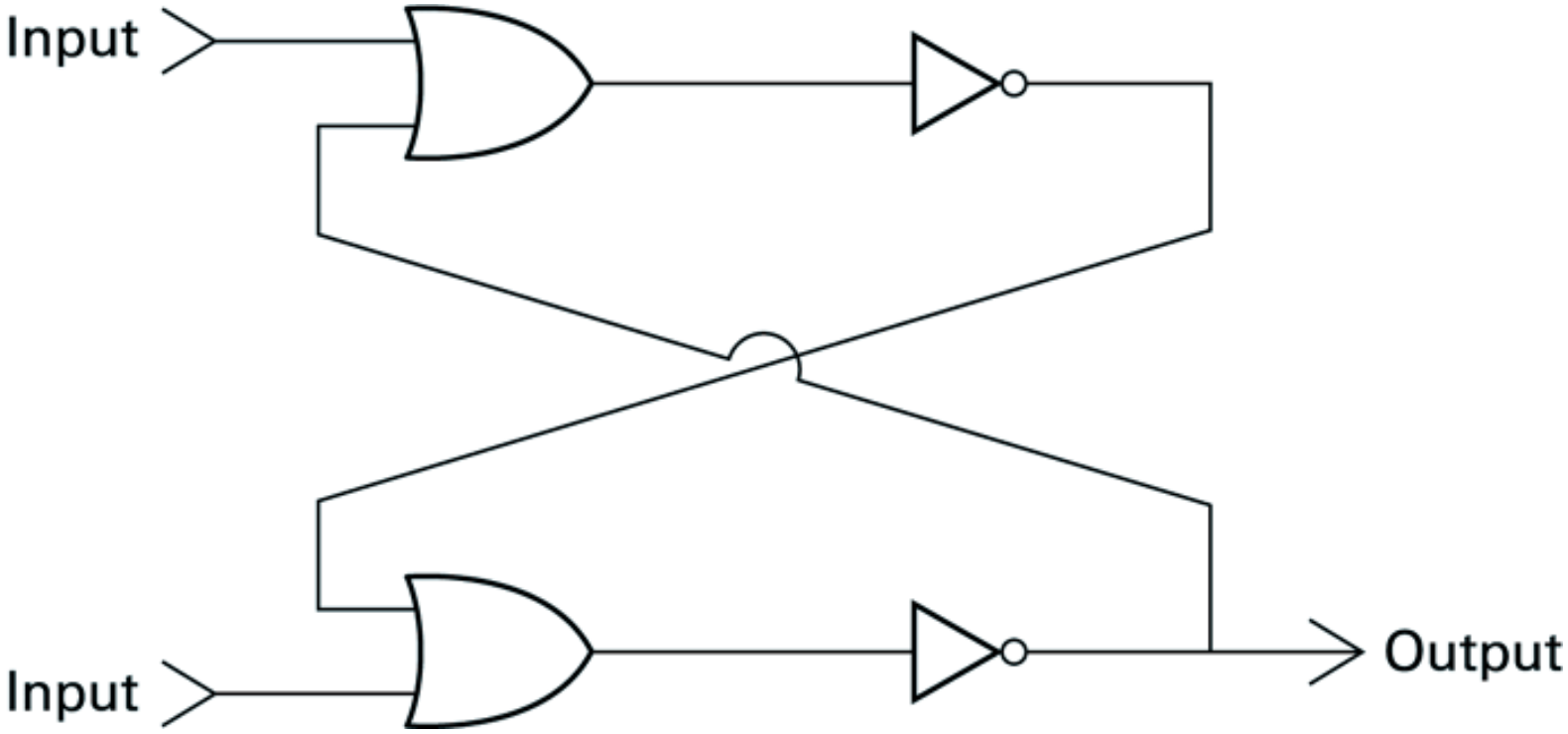


Flip-Flop:

如何記住 1？

如何變成 0？

Figure 1.5: Another way of constructing a **flip-flop**



SRAM 的原理是 Flip-Flop (正反器)

DRAM 的原理是 電容; 需要 refresh

Hexadecimal Notation

- **Hexadecimal notation:** A shorthand notation for long bit patterns
  - Divides a pattern into groups of four bits each
  - Represents each group by a single symbol
- Example: **1010**0011 becomes **A**3

- Octal 八進位制
- Octet 特別指 8-bit 的 Byte (因一 Byte 不一定幾個 bits)

Figure 1.6: The hexadecimal coding system

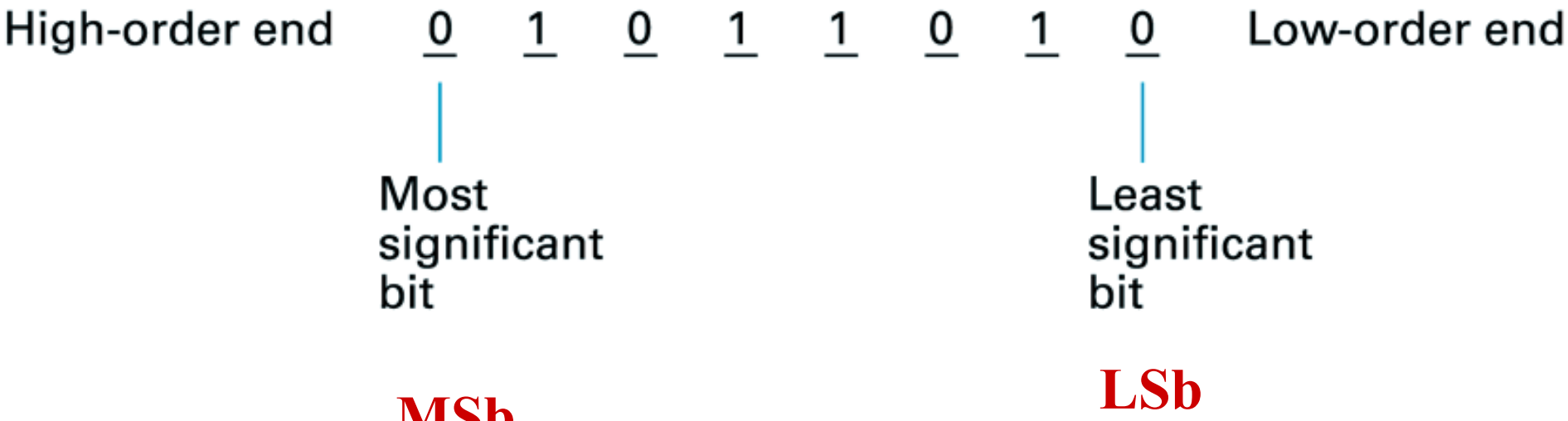
Bit pattern	Hexadecimal representation
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Main Memory Cells

- **Cell:** A unit of main memory (typically 8 bits which is one **byte**)
  - **Most Significant bit (MSb):** the bit at the left (highorder) end of the conceptual row of bits in a memory cell
  - **Least Significant bit (LSb):** the bit at the right (loworder) end of the conceptual row of bits in a memory cell

Figure 1.7: The organization of a byte-size memory cell

BYTE = BinarY TErm



最左邊叫 bit-7

最右邊叫 bit-0

Measuring Memory Capacity

- **Kilobyte:** 2^{10} bytes = 1024 bytes
  - Example: 3 KB = 3 times1024 bytes
  - Sometimes “kibi” rather than “kilo”
- **Megabyte:** 2^{20} bytes = 1,048,576 bytes
  - Example: 3 MB = 3 x 1,048,576 bytes
  - Sometimes “megi” rather than “mega”
- **Gigabyte:** 2^{30} bytes = 1,073,741,824 bytes
  - Example: 3 GB = 3 x 1,073,741,824 bytes
  - Sometimes “gigi” rather than “giga”

- **K*m*** = 千米=公里

- Tera = 1024 Giga
- Peta = 1024 Tera
- Exa = 1024 Peta

- Zeta = 1024 Exa
- Yotta = 1024 Zeta

μs = micro second

760 *mm* Hg (Atmospheric pressure)

- d = deci = 10^{-1}
- c = centi = 10^{-2}
- m = milli = 10^{-3}
- μ = micro = 10^{-6}
- n = nano = 10^{-9}
- p = pico = 10^{-12}

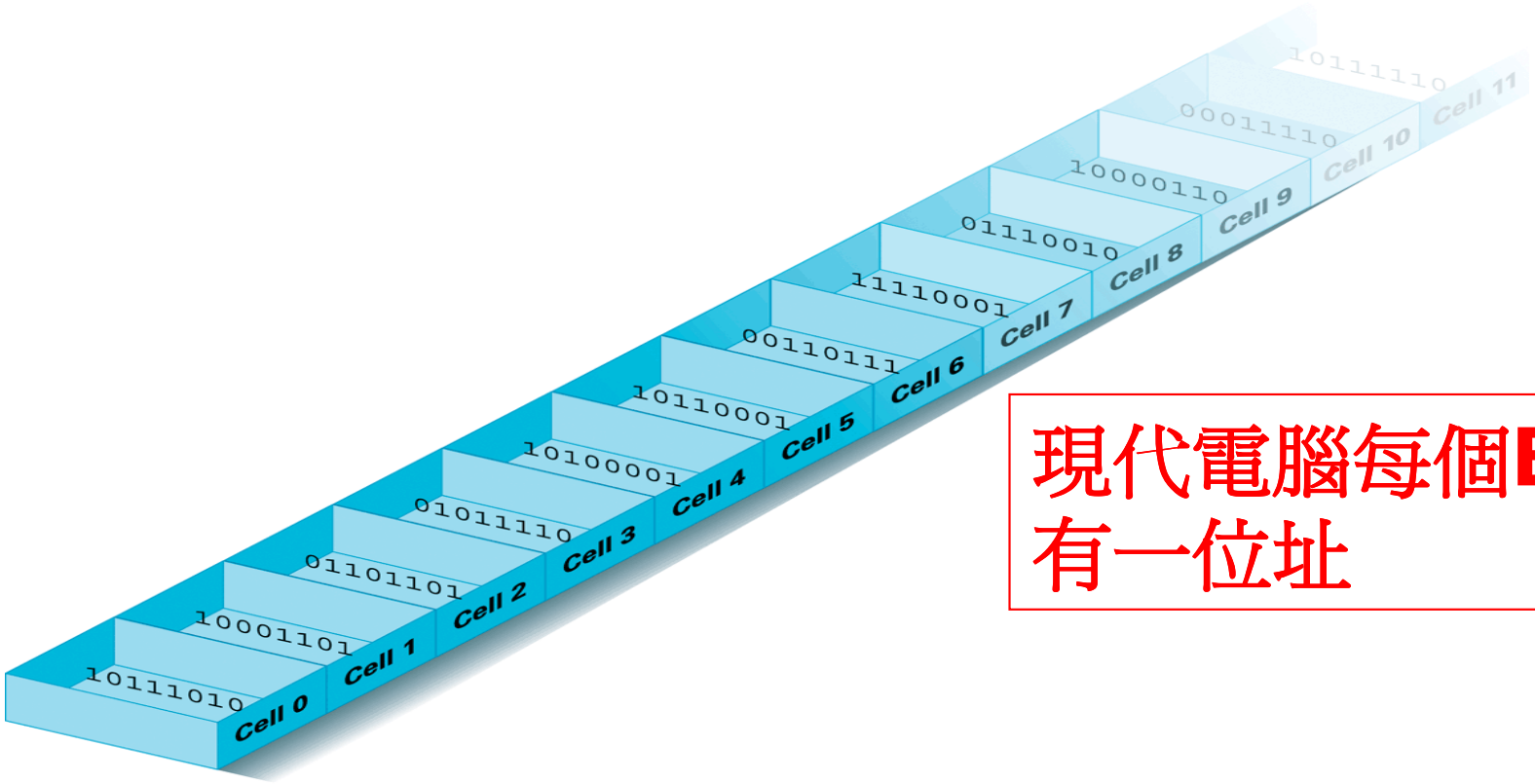
- f = femto = 10^{-15}
- a = atto = 10^{-18}
- z = zepto = 10^{-21}
- y = yocto = 10^{-24}

- *cm* = 公分=厘米

- *mm* = 毫米

http://en.wikipedia.org/wiki/Exa-

Figure 1.8: Memory cells arranged by address



現代電腦每個**Byte**有一位址

- **Byte order when using multi bytes to represent a number**
  - Little Endian 小印第安系統 (Intel CPU)
  - Big Endian 大印第安

Memory Terminology

- **Read Only Memory (ROM):** *nonvolatile*
  - **Mask ROM, PROM, EPROM, EEPROM,…**
- **Random Access Memory (RAM):** Memory in which individual cells can be easily accessed in any order; usually volatile
- **Dynamic RAM (DRAM):** RAM that need *refresh* (*刷新*; 複習) – 原理是電容
- **Static Memory (SRAM):** RAM that does NOT need to *refresh* – 原理是正反器

Memory Layout for Multiprogrammed System

- Programs must be loaded into main memory when running
- Address space

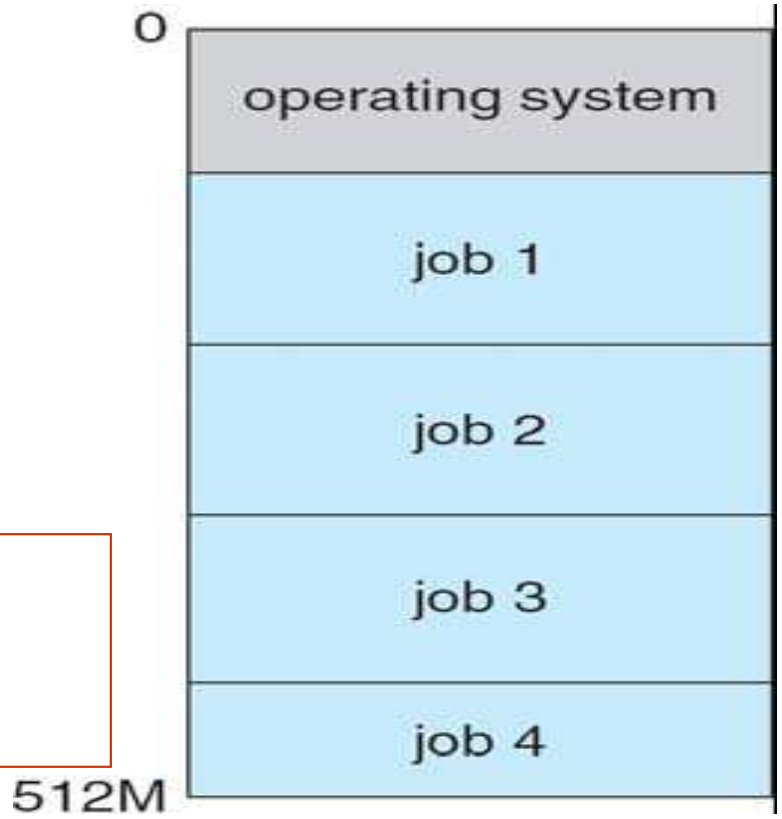
  Physical address space

  vs.

  Virtual address space

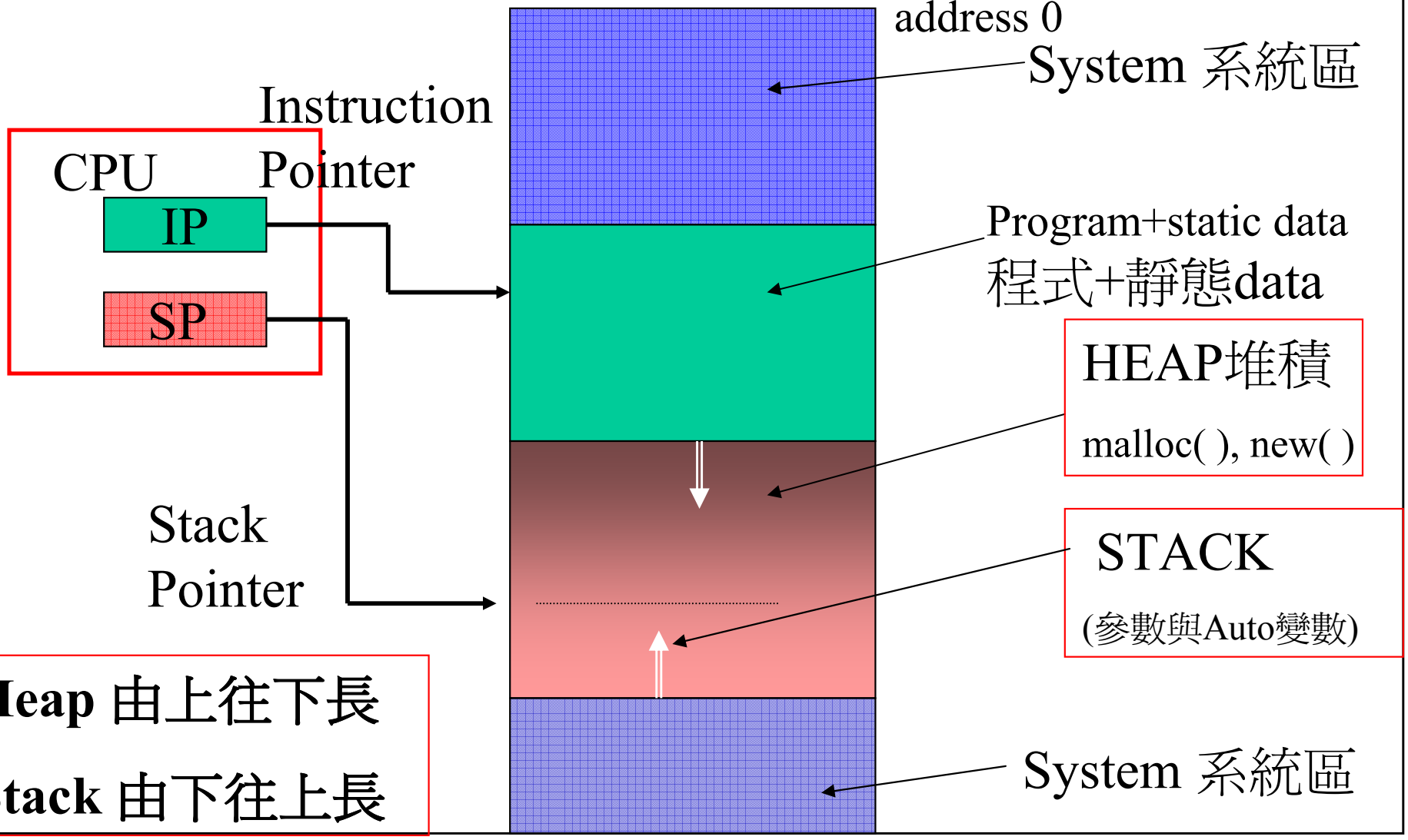
- Paging/Swapping ?

  http://en.wikipedia.org/wiki/Paging



Auto variables use STACK area memory

- **Auto 變數就是沒寫 static 的 Local 變數**



Heap 由上往下長

Stack 由下往上長

Memory operations with protection

- **Dual-mode** operation allows OS to protect itself and other system components
  - **User mode** and **kernel mode**
  - **Mode bit** provided by hardware
    - Provides ability to distinguish when system is running user code or kernel code
    - Some instructions designated as **privileged**, only executable in kernel mode
    - System call changes mode to kernel, return from call resets it to user
- **Intel CPU: Real mode, Protected mode, Virtual-86 mode, System Management Mode (SMM)**

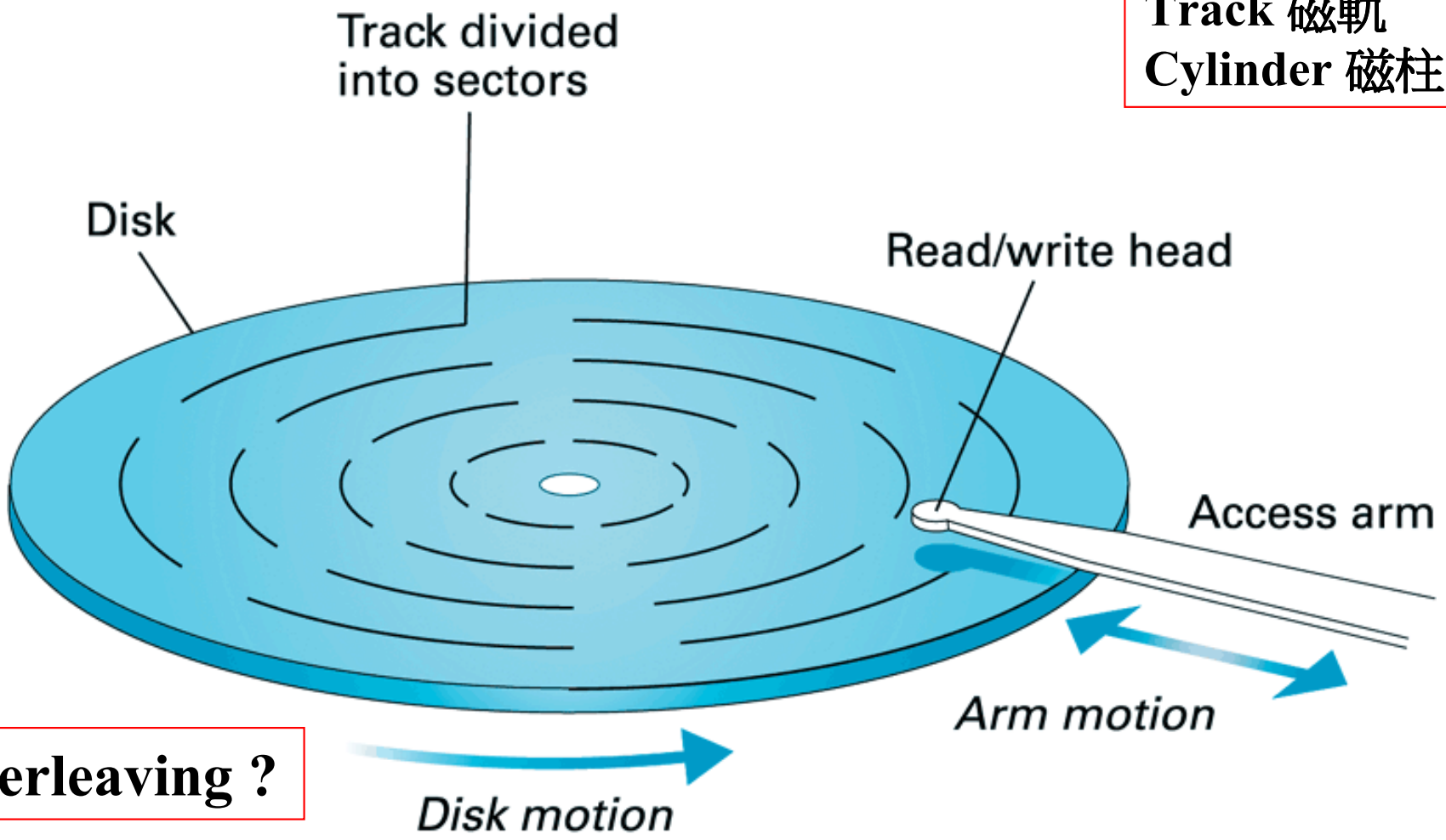
Mass Storage 大量儲存體

讀寫單位通常是 **Block**
—**Block** = 數十到數千**bytes**

- Characteristics
  - On-line versus off-line
  - Typically larger than main memory
  - Typically less **volatile** than main memory
  - Typically slower than main memory
- Types of Mass Storage Systems
  - Magnetic Systems: Disk (Floppy, HD), Tape
  - Optical Systems: CD, DVD
  - Flash Drives: Thumb Drive, Removable Drive,…

Figure 1.9: A **magnetic disk storage system**

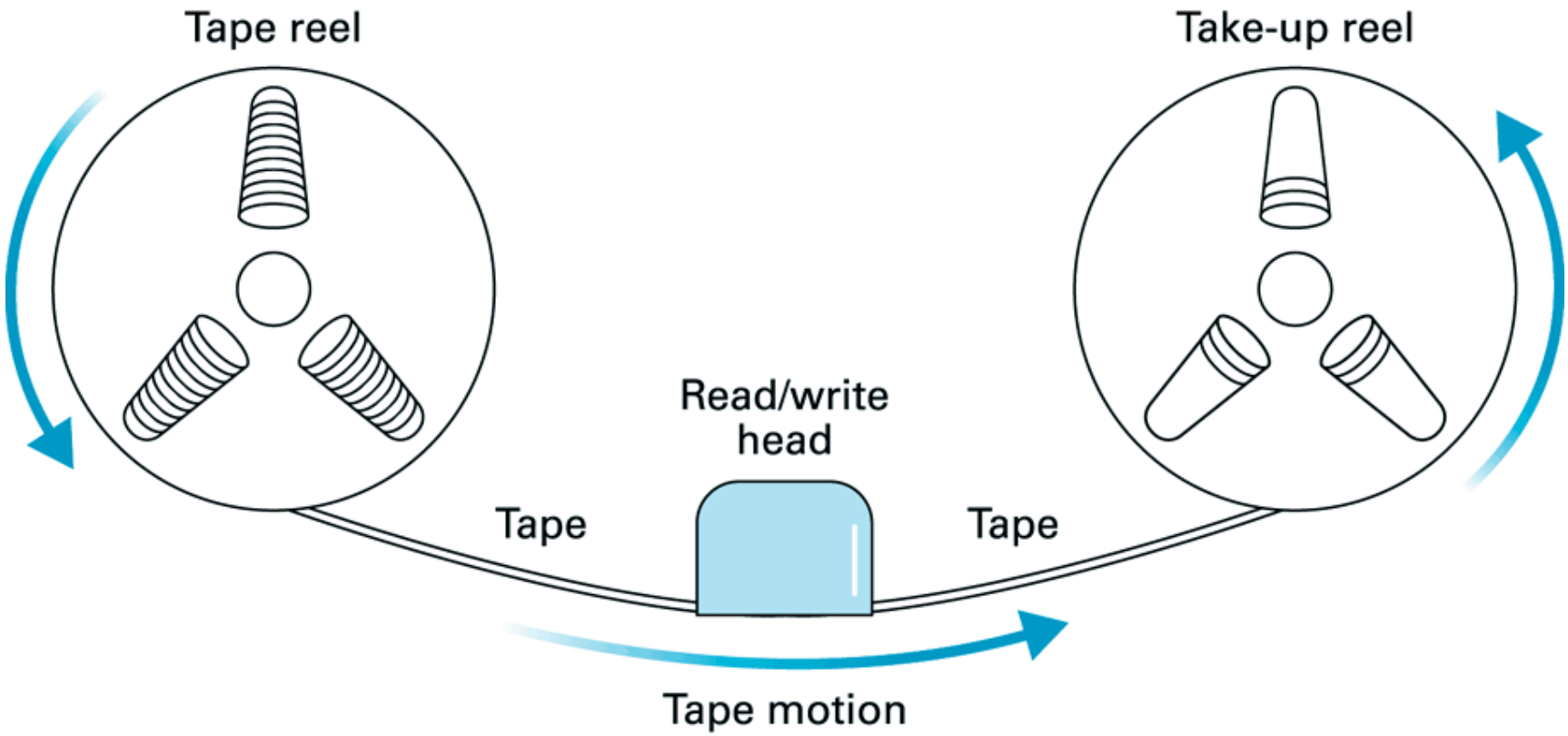
Sector 磁扇、磁區
Track 磁軌
Cylinder 磁柱



Interleaving ?

注意 CD / DVD 儲存不是同心圓方式

Figure 1.10: A magnetic tape storage mechanism

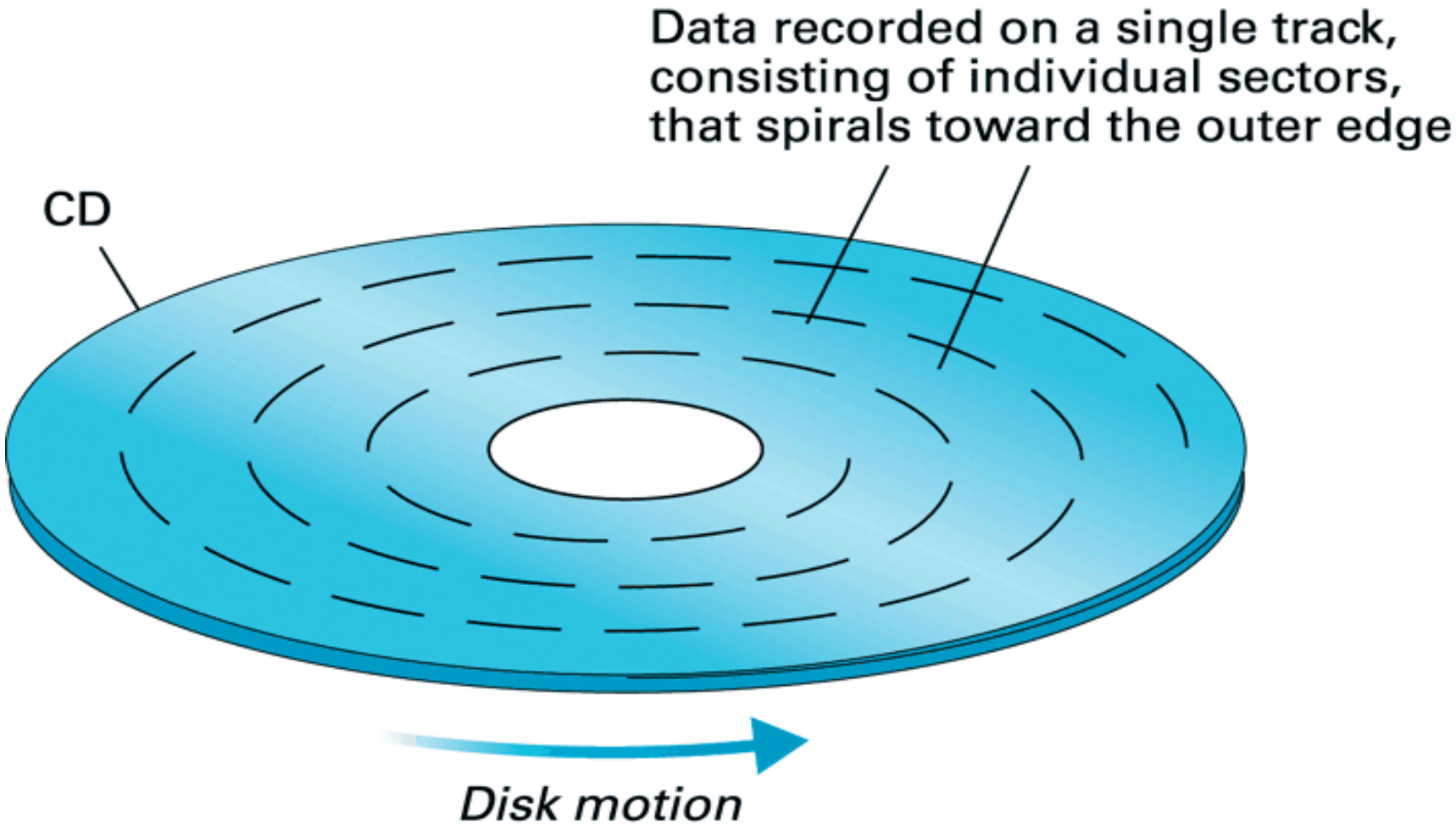


9 tracks vs. 7 tracks

6250 bpi? 1600 bpi (63 bpmm)?

Bit per inch / bit per milli meter

Figure 1.11: CD storage format



注意 CD / DVD 只有一個 track

Ideal storage device

- stable – contents do not corrupt
- fast – to match the speed of CPU
- large in capacity – 1000 GB (1TB)+
- cheap in per unit capacity
- small in size – portable ?
- low-power

Unfortunately, there is no single technology that fulfills all the requirements.

Storage Structure

- **Main memory (*volatile* and/or *nonvolatile*)**
  - *only large storage media that the CPU can access directly. (Magnetic Drum, **Core 磁蕊**, DRAM)*
- **Secondary storage**
  - *extension of main memory that provides large **nonvolatile** storage capacity.*
  - *Magnetic disks*
    - rigid metal or glass platters covered with magnetic recording material
    - Disk surface is logically divided into *tracks(磁軌)*, which are subdivided into *sectors(磁扇或磁區)*.
    - The *disk controller* determines the logical interaction

Core – by Dr. An Wang (1/2)

- 1949/9 An Wang (王安)filed his patent application for a "Pulse Transfer Controlling Device."
- His Pulse Transfer Controlling Devices were minuscule toroidal coils with a donut-shaped ferrite core magnetized in one of two possible directions: 0 or 1, the basic units of every bit of information that a computer or electronic calculator can handle. Its permanent but controllable nature made them the ideal substratum to

Core – by Dr. An Wang (2/2)

- Six years later, in May 17, 1955, the Patent Office issued patent 2,708,722 to Dr. Wang. It took little time for IBM, a company with big stakes on the field, to catch interest on this invention, and after a bitter negotiation IBM bought the patent.

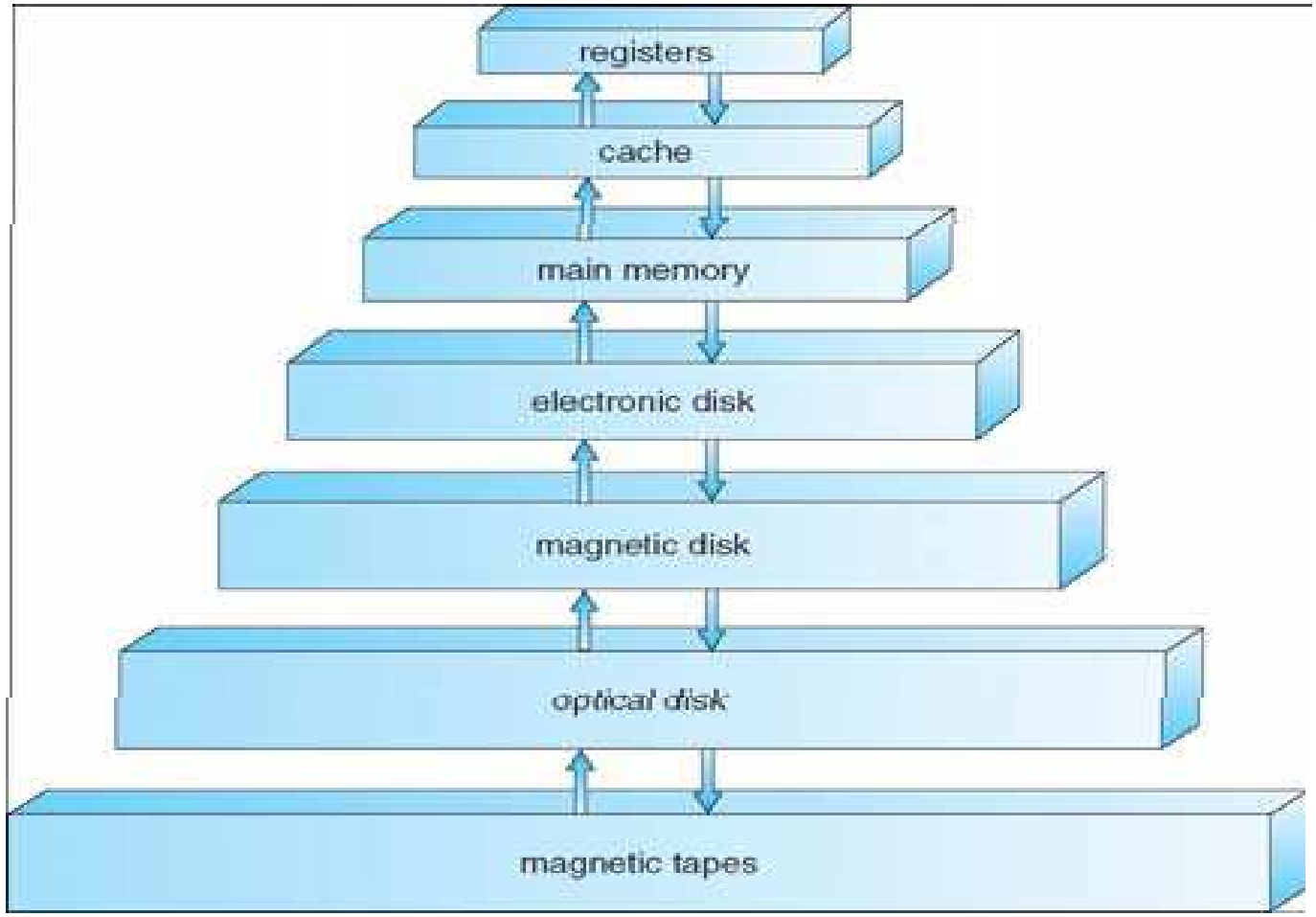
(傳說是 100萬美金! 王安用它來創業成立曾經盛極一時的王安電腦公司)

Storage Hierarchy

- Storage systems organized in hierarchy.
  - Speed
  - Cost
  - Volatility (揮發性) : 沒電後資料是否還在?
- *Caching* – copying information into faster storage system; main memory can be viewed as a last *cache* for secondary storage.

Cache (快取) 念作 cash (現鈔)

Storage-Device Hierarchy



RAID 磁碟陣列(Redundant Array of Independent Disks)

Performance of Various Levels of Storage

Level	1	2	3	4
Name	registers	cache	main memory	disk storage
Typical size	< 1 KB	> 16 MB	> 16 GB	> 100 GB
Implementation technology	custom memory with multiple ports, CMOS	on-chip or off-chip CMOS SRAM	CMOS DRAM	magnetic disk
Access time (ns)	0.25 – 0.5	0.5 – 25	80 – 250	5,000,000
Bandwidth (MB/sec)	20,000 – 100,000	5000 – 10,000	1000 – 5000	20 – 150
Managed by	compiler	hardware	operating system	operating system
Backed by	cache	main memory	disk	CD or tape

- Movement between levels of storage hierarchy can be explicit or implicit

Storage Management (1/2)

- Uniform, logical view of information storage
  - Abstracts physical properties to logical storage unit – **file**
  - Each medium is controlled by device (i.e., disk drive, tape drive
    - Varying properties include access speed, capacity data transfer rate access method (**sequential** or **random**)

On Unix, the size of a file may larger than a single Disk

Storage Management (2/2)

- File-System management
  - Files usually organized into directories
  - Access control on most systems to determine who can access what (UID; ACL: Access Control List)
  - Management activities include
    - Creating and deleting files and directories
    - Primitives to manipulate files and directories
    - Mapping files onto secondary storage
    - Backup files onto stable (non-volatile) storage media

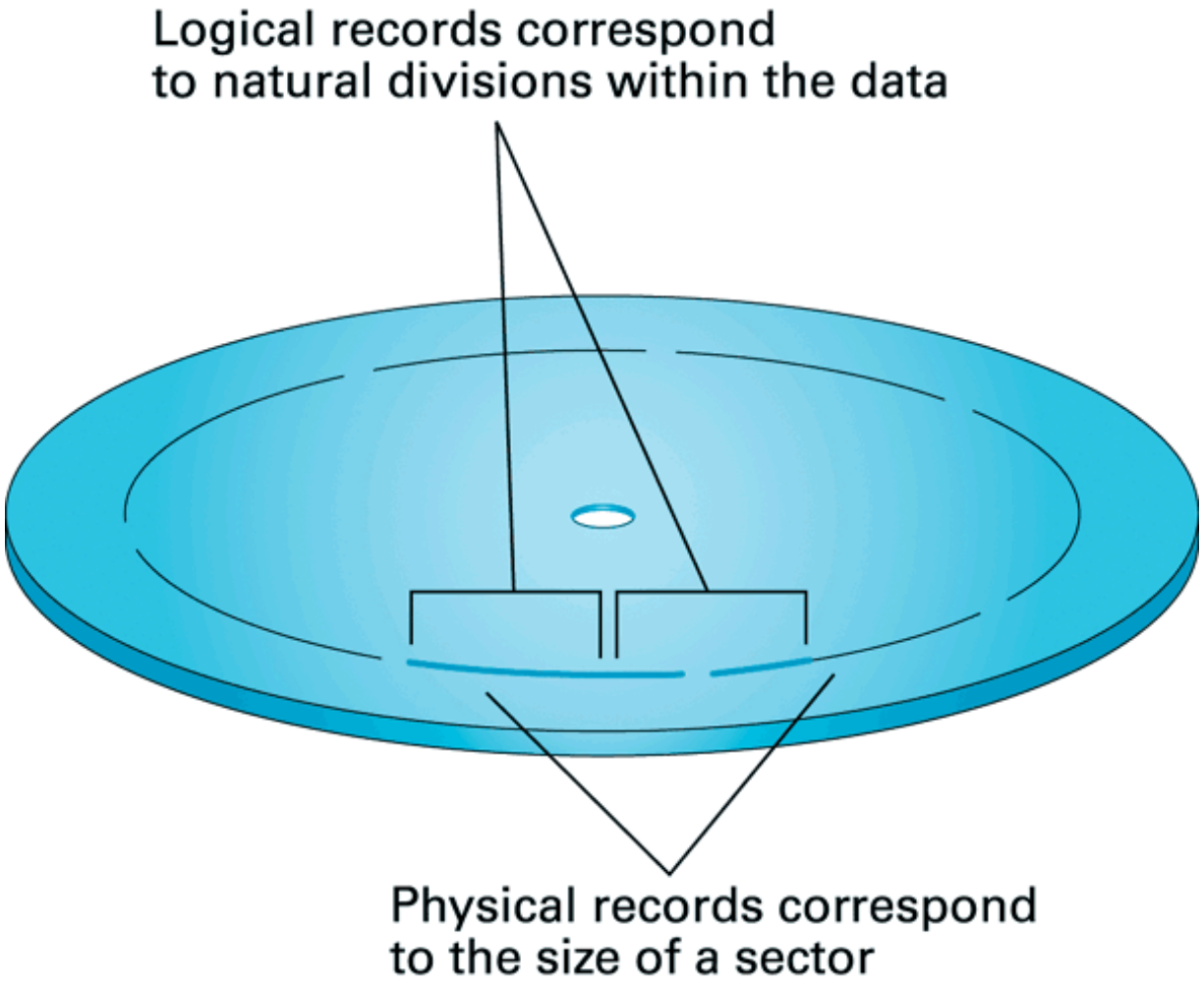
UID: User ID; GID: Group ID

Files

- **File:** A unit of data stored in mass storage system
  - **Fields** and **keyfields**
- Physical record versus Logical record
  - On Unix, the size of a file may larger than a single Disk
- **Buffer:** A memory area used for the temporary storage of data (usually as a step in transferring the data)

Buffer vs. **Cache**

Figure 1.12: Logical records versus physical records on a disk



Mass-Storage Management

- Usually disks used to store data that does not fit in main memory or data that must be kept for a “long” period of time.
- Proper management is of central importance
- Entire speed of computer operation hinges on disk subsystem and its algorithms
- OS activities
  - Free-space management
  - Storage allocation
  - Disk scheduling
- Some storage need not be fast
  - Tertiary storage includes Optical storage, Magnetic tape
  - Still must be managed
  - Varies between WORM (Write-Once, Read-Many-times) and RW (Read-Write)

Protection and Security (1/2)

- **Protection** – any mechanism for controlling access of processes or users to resources
- **Security** – defense of the system against internal and external attacks
  - Huge range, including Denial-of-Service (DoS), worms, **viruses**, identity theft, theft of service

Protection and Security (2/2)

- Systems generally first distinguish among users, to determine who can do what
  - User identities (**user IDs**, **UID**; security IDs) include name and associated number, one per user
  - User ID then associated with all files, processes of that user to determine access control
  - Group identifier (g**roup ID** , **GID**) allows set of users to be defined and controls managed, then also associated with each process, file
  - **Privilege escalation** allows user to change to Effective ID (**EUID**) with more rights
  - **SetUID/SetGID: UID vs. EUID; GID vs. EGID**

The Binary System

- The traditional decimal system is based on powers of ten.
- The Binary system is based on powers of two.

- The Octal system is based on powers of eight.
- The Ternary system is based on powers of three.

Hexadecimal, Decimal, Octal, Ternary, Binary

十六進制, 十進制, 八進制, 三進制, 二進制

BIT = Binary digIT 位元

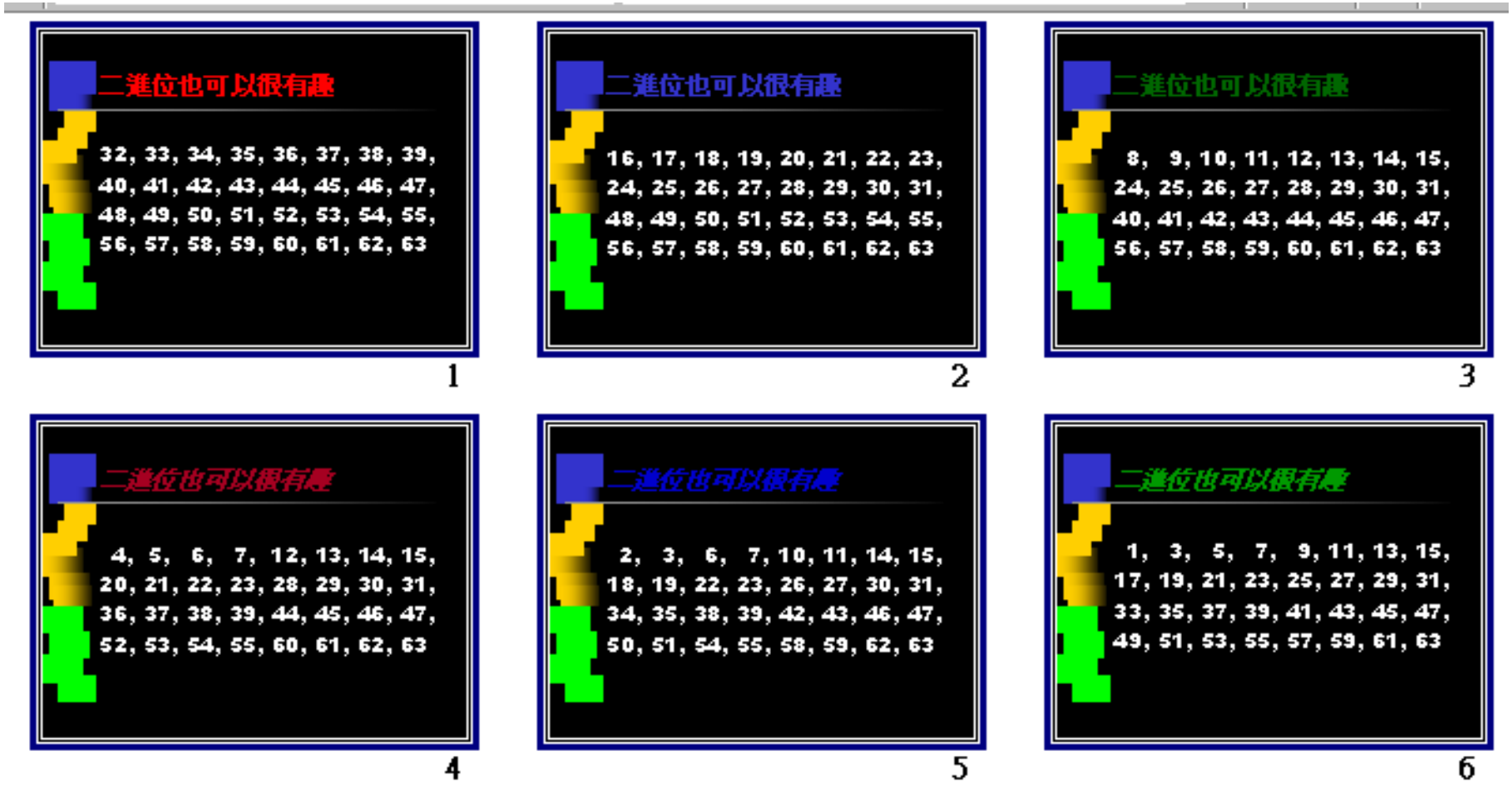
Data Representation

- Bit Patterns are used to represent information.
  - Numbers: Integer, Real (Float; see IEEE754/IEEE854)
  - Text characters: ASCII, EBCDIC, BIG5, …
  - Images: GIF, JPEG, …
  - Sound: WAV, MP3, …
  - Video: AVI, MPEG, …
  - Etc…
- Usually denoted as hexadecimal format for convenient to read by human

二進位也可以很有趣 ☺

想一個 0 到 63 間的任一個數，不要告訴我答案...

但是告訴我有沒有在以下哪幾張卡片中? 我可以很快找出來喔!



卡片玩法與原理：6 bits

請玩者心裡想一個0到63的整數不要告訴你(廢話), 然後你依第六張、第五張、第四張 ... 第一張拿給他, 請他告訴你上面有沒有他想的數, 爲了表示你是過目不忘, 交給他之前假裝看一看, 交給他後不要立刻拿回來, 只請他分別放不同手中作爲證據。

答案: 如果六張都沒有就是 0。

在每張給他之前看清楚第一個數, 只要他說有就加到答案中, 等到最後一張給他, 等他說出有沒有後, 答案也出來了。

原理:

其實每張紙都是代表一個二進位數 (binary number)。

有就是 1, 沒有就是 0。

與上一遊戲一樣, 他已經把答案告訴你了！

因每張紙上第一個數值就是該位的比重(weight, 加權)。

這遊戲的意義?

Representing Text

- **Each character (letter, punctuation, etc.) is assigned a unique bit pattern.**
  - **ASCII**: Uses patterns of 7-bits to represent most symbols used in written English text
  - Unicode: Uses patterns of 16-bits to represent the major symbols used in languages world wide
  - ISO standard: Uses patterns of 32-bits to represent most symbols used in languages world wide
- **ASCII: American Standard Code for Information Interchange**

BYTE: BinarY Term ： 一群二進位; 位元組

1 BYTE = 6 bits, **7 bits**, 8 bits

1 Octet = 8 bits

Figure 1.13: The message "Hello." in ASCII

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

CDC Display code: use 6 bits; lowercase letter? Uppercase?

BCD: Binary Coded Decimal: use 6 bits

EBCDIC: Extended Binary Coded Decimal Interchange Code

http://en.wikipedia.org/wiki/EBCDIC

ANSI : American National Standard Institute

IEEE: Institute of Electrical and Electronics Engineers

ISO : International Organization for Standardardization

問題與思考 (ASCII code) (1/2)

```
#include<stdio.h>
main( ) {
    printf(" %c 的 ASCII code是%d\n",  '0',  '0');
    printf(" %c 的 ASCII code是%d\n",  'A',  'A');
    printf(" %c 的 ASCII code是%d\n", 'a',  'a');
}
```

0 的 ASCII code 是 48
A 的 ASCII code 是 65
a 的 ASCII code 是 97

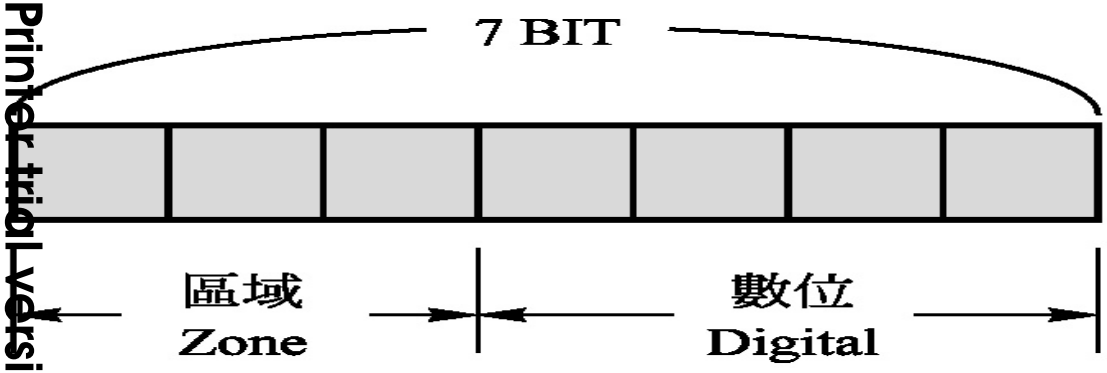
如果系統使用EBCDIC碼呢?

問題與思考 (ASCII code) (2/2)

```
#include<stdio.h>
main( ) {
    int i, k=0;
    for(i=65; i<=122; ++i) {
        printf(" %c 的 ASCII code", i);
        printf(" 是%d", i);
        ++k; printf(k%3==0? "\n" : "\t");
    } printf("\n" );
}
```

ASCII 美國標準資訊交換碼

- ASCII碼有七個位元可表示範圍2^7=128種
- 最左三個位元爲區域位元，右邊四個爲數字位元



區域位元	表示意義
000	保留給通訊控制用
001	保留給通訊控制用
010	特殊符號
011	阿拉伯數字及特殊符號
100	大寫英文字母A ~O
101	大寫英文字母P ~Z及特殊符號
110	小寫英文字母a ~o
111	大寫英文字母 p ~z

American Standard Code for Information Interchange

BCD交換碼 (Binary Coded Decimal;又稱8421碼)

- 是二進制的加權碼; 使用 6 個 bits 表示文字符號
- 注意其實應該稱 84-2-1碼, 因爲是:
  最右邊爲元的加權爲" –1"
  右邊算來第二位的加權 爲" -2"

十進制數字	84-2-1碼
0	0000
1	0111
2	0110
3	0101
4	0100
5	1011
6	1010
7	1001
8	1000
9	1111

EBCDIC:擴展BCD交換碼 (by IBM)

- IBM電腦用8個BIT進行編碼, 故將BCD交換碼擴展爲八個位元可表示範圍2^8=256種
- 最左邊四個位元爲區域位元，右邊四個爲數字位元

EBCDIC： Extended BCD Interchange Code

EBCDIC區域位元的分配

區域位元	表示意義
00XX	保留
01XX	特殊符號
1000	小寫英文字母 a ~ i
1001	小寫英文字母 j ~ r
1010	小寫英文字母 s ~ z
1011	保留
1100	大寫英文字母 A ~ I
1101	大寫英文字母 J ~ R
1110	大寫英文字母 S ~ Z
1111	阿拉伯數字0-9

問題與思考 (中文碼?) (1/2)

```
#include<stdio.h>
unsigned char x[9] = { 0 };
main( ) {
    int m = 0xa4,  n=0x6a;
    x[0] = m;  x[1]=n;
    x[2] = 0xae,  x[3]=97;
    x[4] = 0xa6,  x[5]=0x6e;
    printf("==%s==\n", x);
}
```

問題與思考 (中文碼?) (2/2)

先把前面程式存入 testc.c

```
ccbsd2: tsaiwn> gcc testc.c
ccbsd2: tsaiwn> ./a.out
==大家好==

ccbsd2: tsaiwn>
```

編譯與連結

執行

```
'A'   ==   65
 'a'   ==   97
```

```
 '0'   ==   48
'\a'   ==    7
```

Representing Numeric Values

- Binary notation: Uses bits to represent a number in base two
- Limitations of computer representations of numeric values
  - **Overflow** – occurs when a value is too big to be represented
  - Truncation occurs when a value cannot be represented accurately (**precision** problem)

Floating point **underflow** ???

Number Systems

Numbers can be represented in any base (humans use base 10)

- Symbols for a number system of base B are 0, 1, 2, …, B –1
- decimal (base 10) 0, 1, 2, .., 9 binary (base 2) 0, 1
- notation "$number_B$" (375 in decimal is written 375_{10}, 1011 in binary is written 1011_2)
- Value of i^{th} digit d is "d * B^i" where i starts from 0 and increases from right to left

2 1 0	**i**	**positional notation**
3 7 5	**data**	
5 * 10^0	**=**	**5**
7 * 10^1	**=**	**70**
3 * 10^2	**=**	**300**

Three hundred and seventy five

Conversion from binary to decimal

Convert 1011_2 to decimal == 8 + 0 + 2 + 1 == 11

3 2 1 0 bit-i

1 0 1 1 data

$= (1 * 2^0) + (1 * 2^1) + (0 * 2^2) + (1 * 2^3)$

$= 1 + 2 + 0 + 8$

$= 11_{10}$

This process can be used for conversion from any number system to decimal (TRY convert 123_8 to decimal)

Conversion from decimal to binary

Step 1: divide value by 2 and record remainder (每次除以2取餘數)

Step 2: as long as quotient not zero, continue to divide the newest quotient by 2 and record the remainder

Step 3: when obtain a zero as quotient, binary representation consists of remainders listed from right to left in order

Example: Convert 13_{10} to binary

Operation	*Quotient*	*remainder*
13 by 2	6	1
6 by 2	3	0
3 by 2	1	1
1 by 2	0	1

$$13_{10} = 1101_2$$

Figure 1.15: The base ten and binary systems

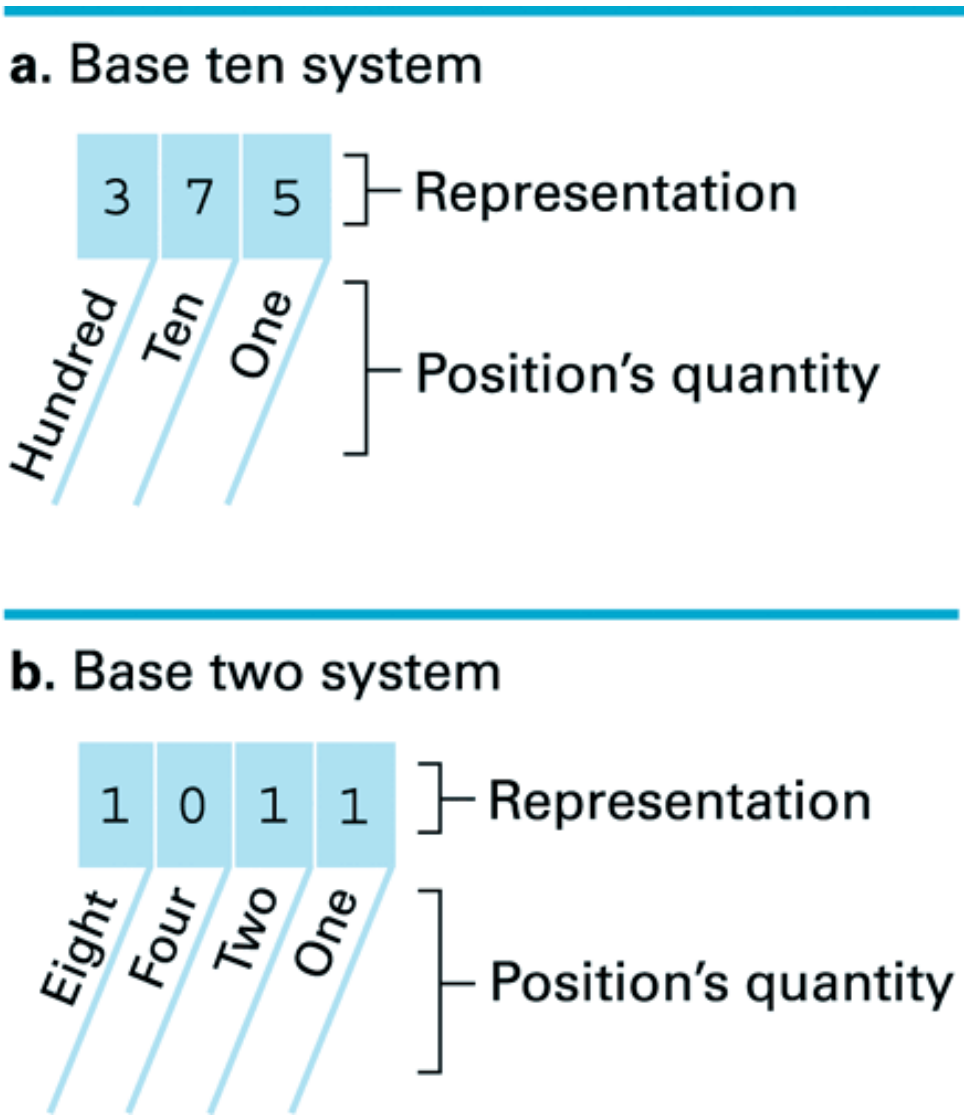


Figure 1.16: Decoding the binary representation 100101 = 37

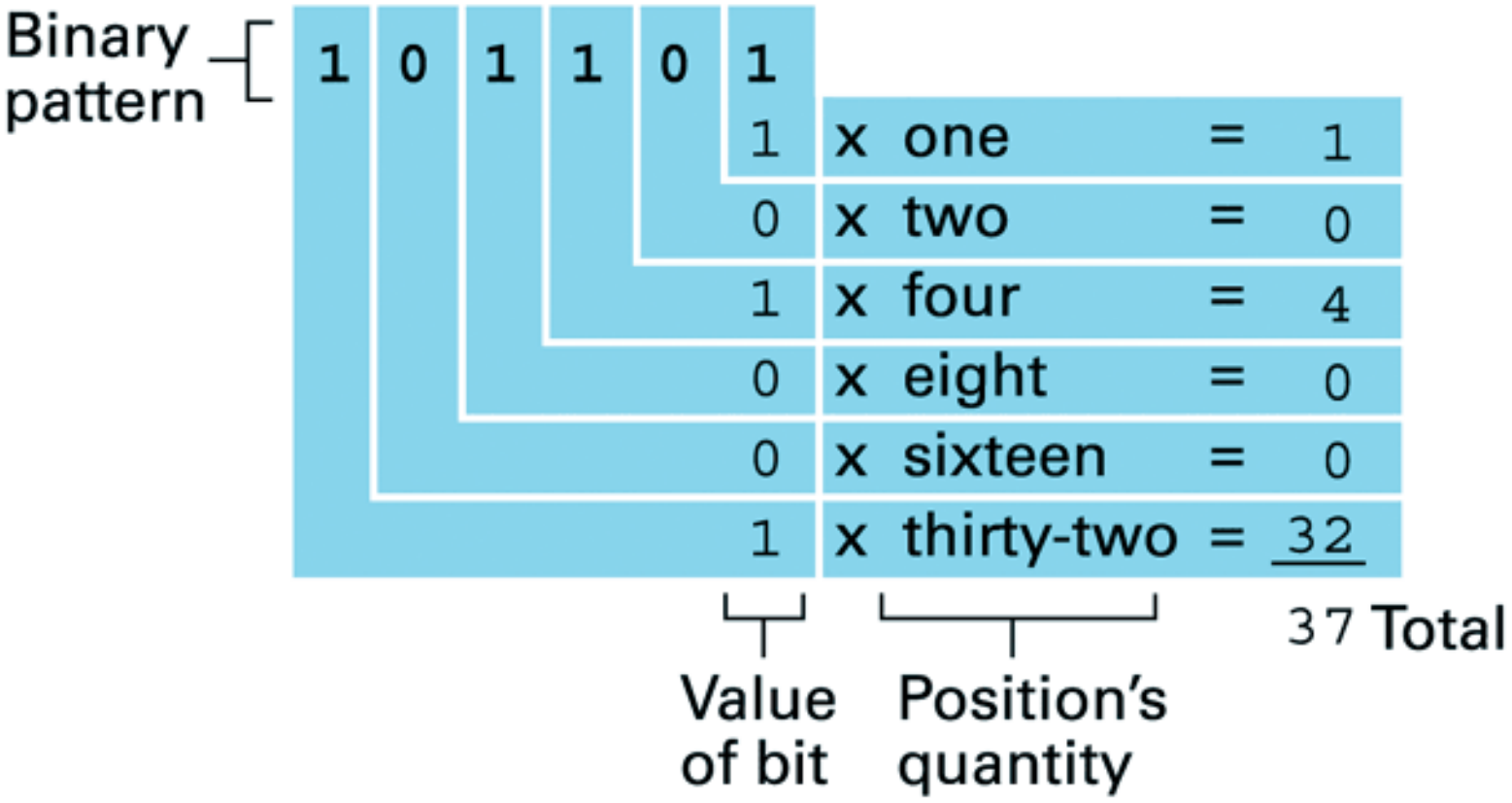


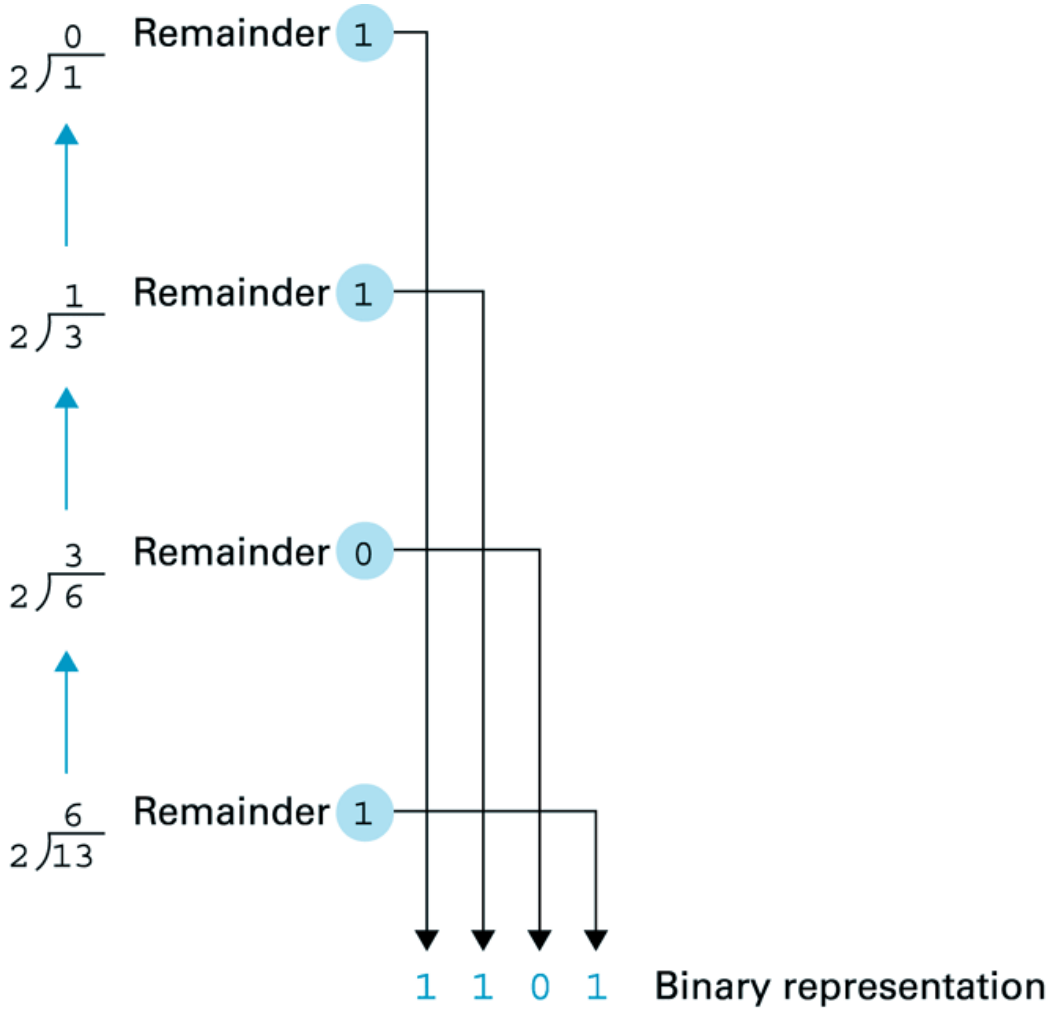
Figure 1.17: An algorithm for finding the binary representation of a positive integer

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

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

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

Figure 1.18: Applying the algorithm in Figure 1.17 to obtain the binary representation of thirteen (13)



Other Number Systems

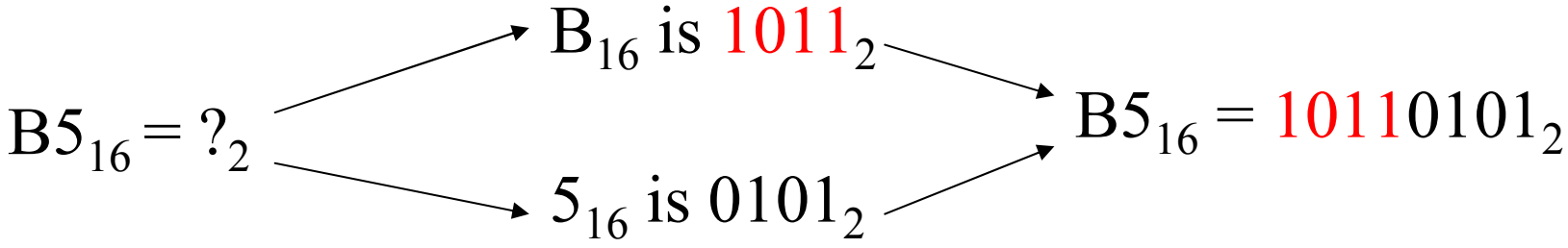
- **Octal** (base 8)

  Symbols (0, 1, 2, 3, 4, 5, 6, 7)

- Working with too long binary numbers is a problem
- Hexadecimal (base 16)

  Symbols (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F)

- Byte = 8 bits = 2 hex digits (1 hex digit is 4 bits)



Hexadecimal (十六進位)

- Hexadecimal (HEX or $_{16}$) is a notation normally used as a shorthand version of binary ($_2$).
- Each digit in a HEX number contains the information of 4 bits of a binary number, eg.

  $01100111_2 = 67_{16}$ and $01011010_2 = 5A_{16}$ and

  $1111111111111111_2 = FFFF_{16}$.
- It is easy enough to convert between the two bases:
  - To convert from binary to hex, simply group the bits 4 at a time from the right and then convert each group into a hex digit, using A for the digit value of 10, through to F for 15.
  - To convert from hex to binary, convert each digit to a 4 bit binary value and then concatenate the groups of 4 bits (in order).
- Often we are not interested in the base 10 value of a quantity (eg. for a memory address or input signal), and HEX can be a convenient way of writing the quantity.

Conversion from binary to hex

Convert 1101001110_2 to hex

Divide binary number into 4 bits groups from right to left

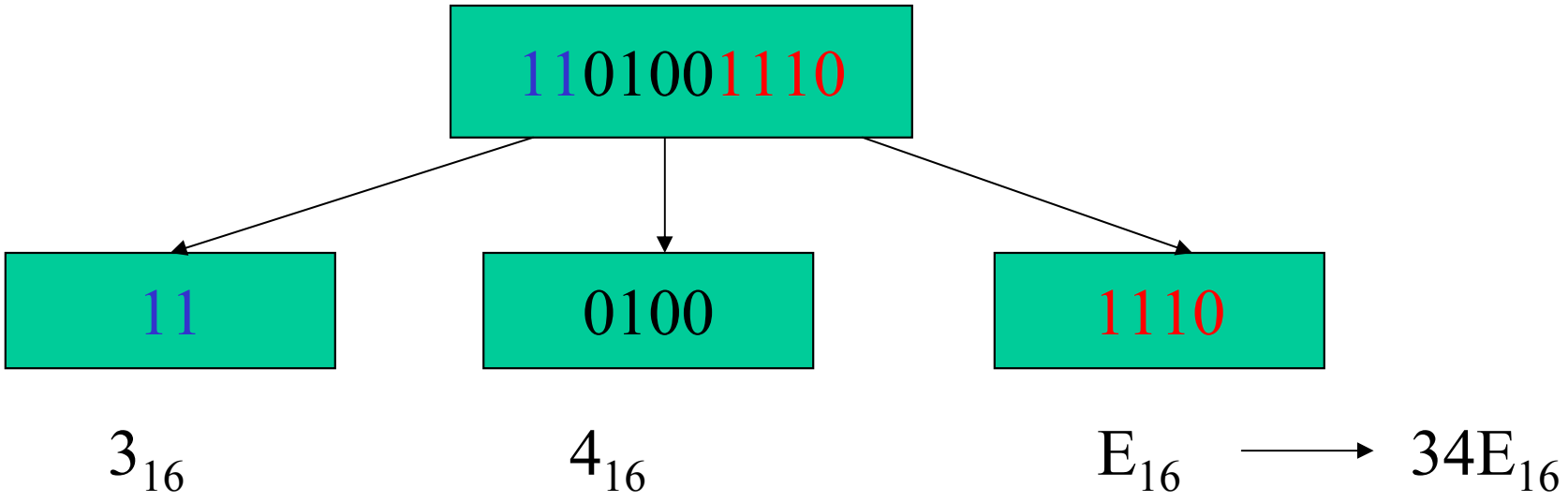


Figure 1.6 (new): The hexadecimal coding system

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

$2^0 = 1$	$2^7 = 128$
$2^1 = 2$	$2^8 = 256$
$2^2 = 4$	$2^9 = 512$
$2^3 = 8$	$2^{10} = 1024$
$2^4 = 16$	$2^{11} = 2048$
$2^5 = 32$	$2^{12} = 4096$
$2^6 = 64$	$2^{13} = 8190$

Figure 1.19: The binary addition facts

```
  0      1      0      1
+ 0    + 0    + 1    + 1
-----  -----  -----  -----
  0      1      1     10
```

Truth Table (真值表)

A	B	carry	sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Storing Integers

- **Two's complement notation:** The most popular means of representing integer values
- **Excess notation:** Another means of representing integer values
  - Usually used in Exponent part when representing floating point numbers (real numbers)
- Both can suffer from overflow errors.

- **one's complement notation?**
- **sign magnitudes** ? (符號 + 絕對值)

What about negative numbers如何表示負數?

- 通常用最左的 bit 表示正負
  - 0 == 正
  - 1 == 負
- 但它的值呢? 至少有以下四種方法:
  - Sign magnitude 符號絕對值法
  - One's complement 一的補數
  - Two's complement 二的補數
  - Excess method 減某值法

Negative numbers 1/5

Sign and magnitude

Left hand bit is used to determine the sign of the number

Left hand bit 0 = positive number

Left hand bit 1 = negative number

0010 = +2

1010 = -2

Using 4 bits with sign and magnitude, largest positive number represented is 7

(-7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7)

15 values overall can be represented

Negative numbers 2/5

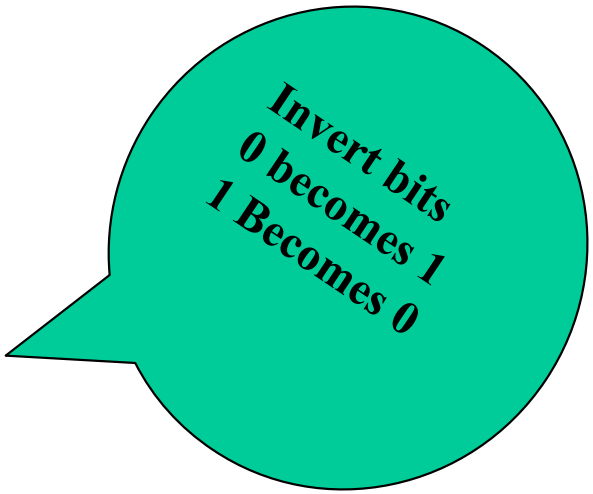
One's Complement

- +2 added to –2 in sign magnitude using regular binary addition does not equal 0 (make sure of that!)
- First bit indicates a negative value but also bears a position value
- For a positive number, the two's complement representation is itself
- For a negative number, complement positive value

3 in one's complement is 011

-3 in one's complement is

Invert bits 011 becomes 100



Negative numbers 3/5

Two's Complement

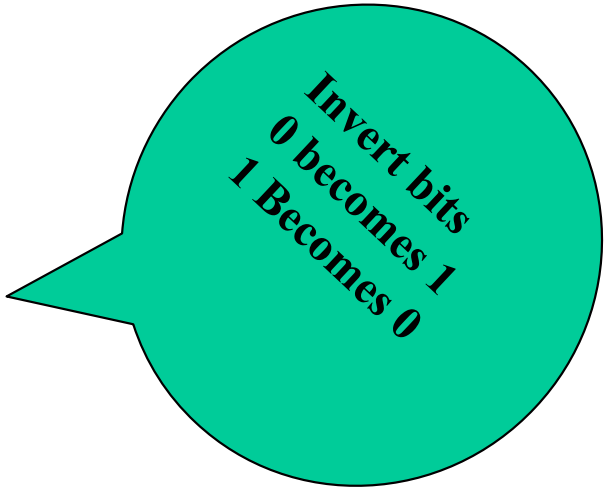
- +2 added to –2 in sign magnitude using regular binary addition does not equal 0 (make sure of that!)
- First bit indicates a negative value but also bears a position value
- For a positive number, the two's complement representation is itself
- For a negative number, complement positive value, then add 1

3 in two's complement is 011

-3 in two's complement is

Invert bits 011 becomes 100

Add 1 100 + 1 = 101



Negative numbers 4/5

What is 1010 in two's complement?

It is a negative number since left hand bit is a 1

Invert bits 1010 becomes 0101

Add 1 0101 + 1 = 0110 (+6)

Then the original number is -6

Negative Numbers 5/5

Two's complement addition

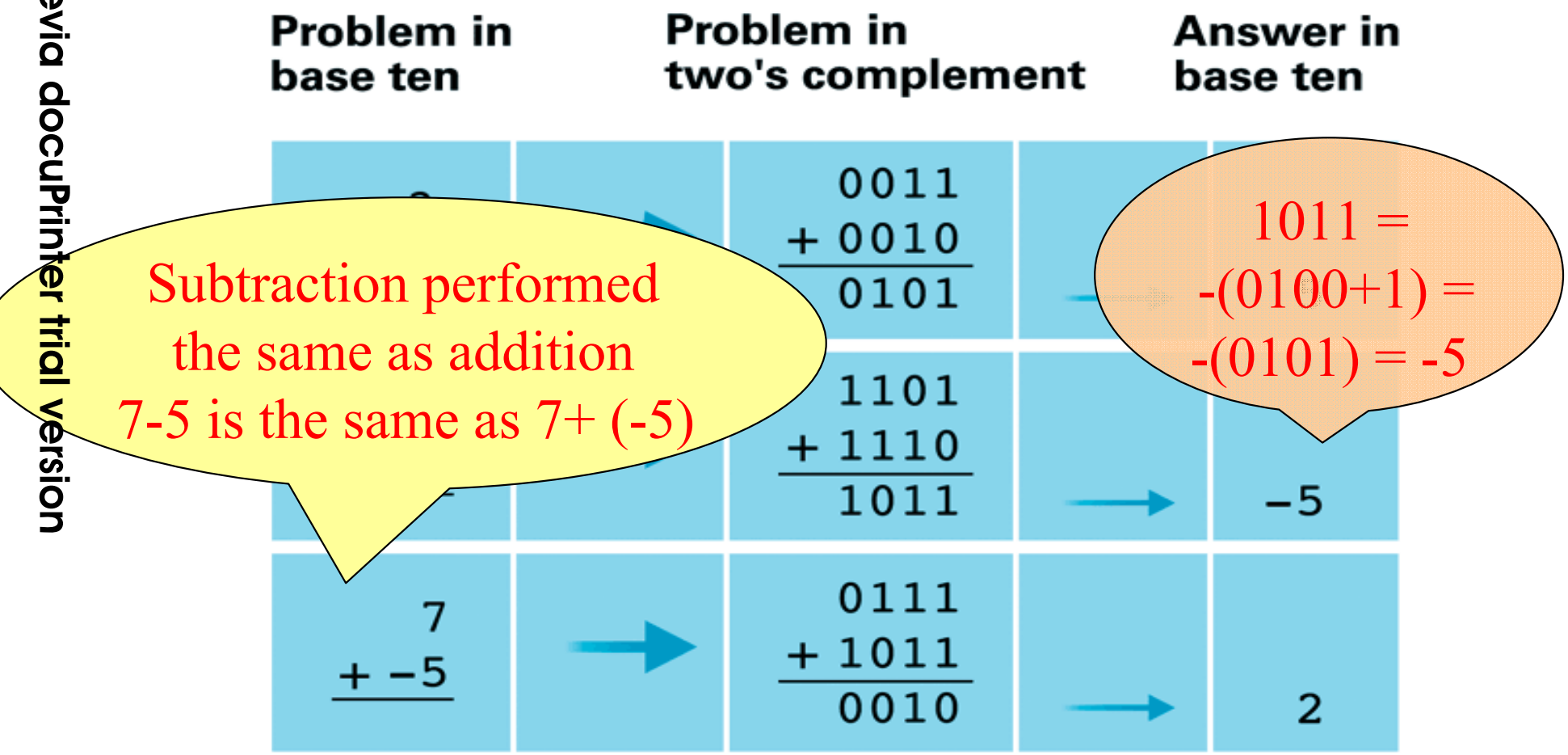


Figure 1.21: Two's complement notation systems

a. Using patterns of length three

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

b. Using patterns of length four

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

Figure 1.22: Coding the value -6 in two's complement notation using four bits

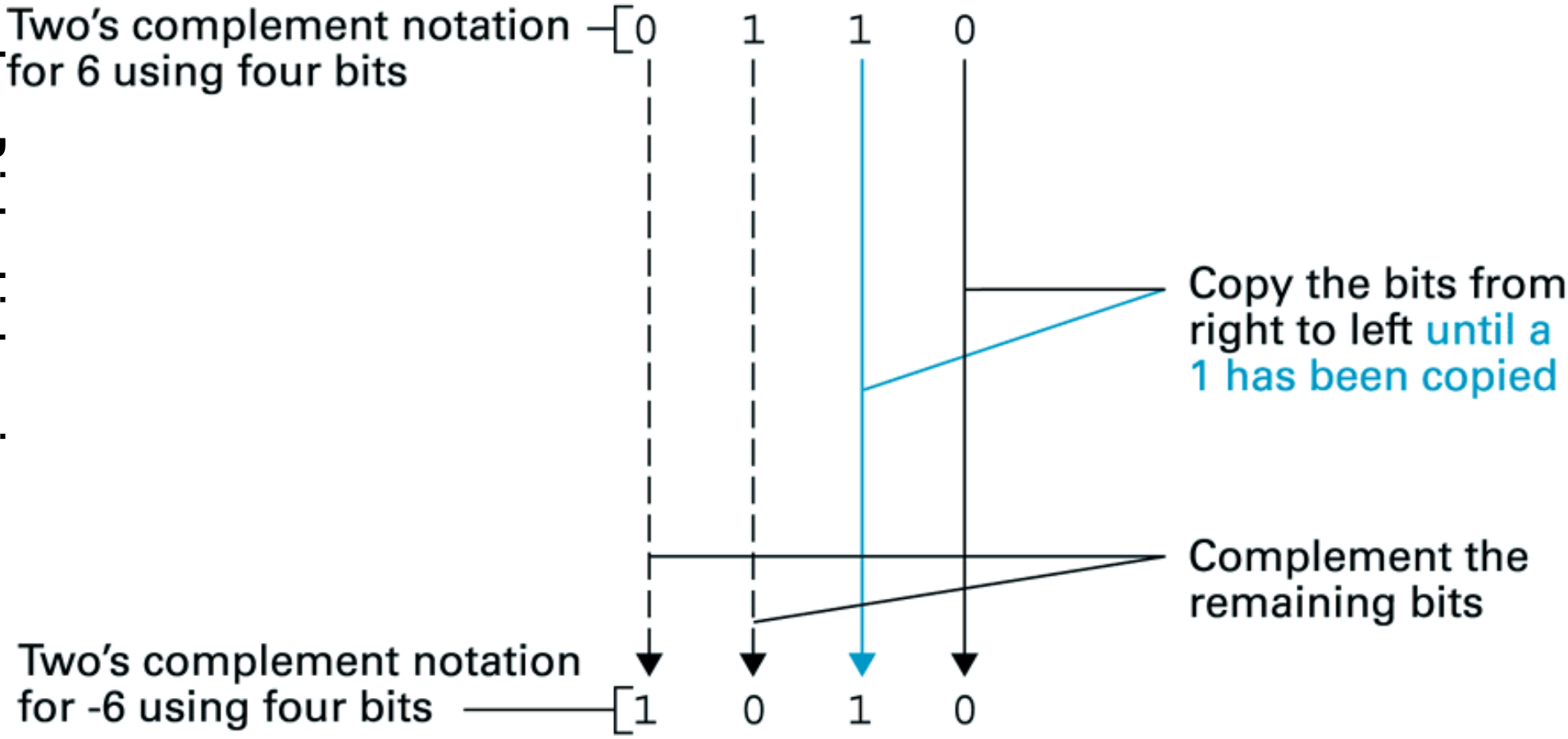
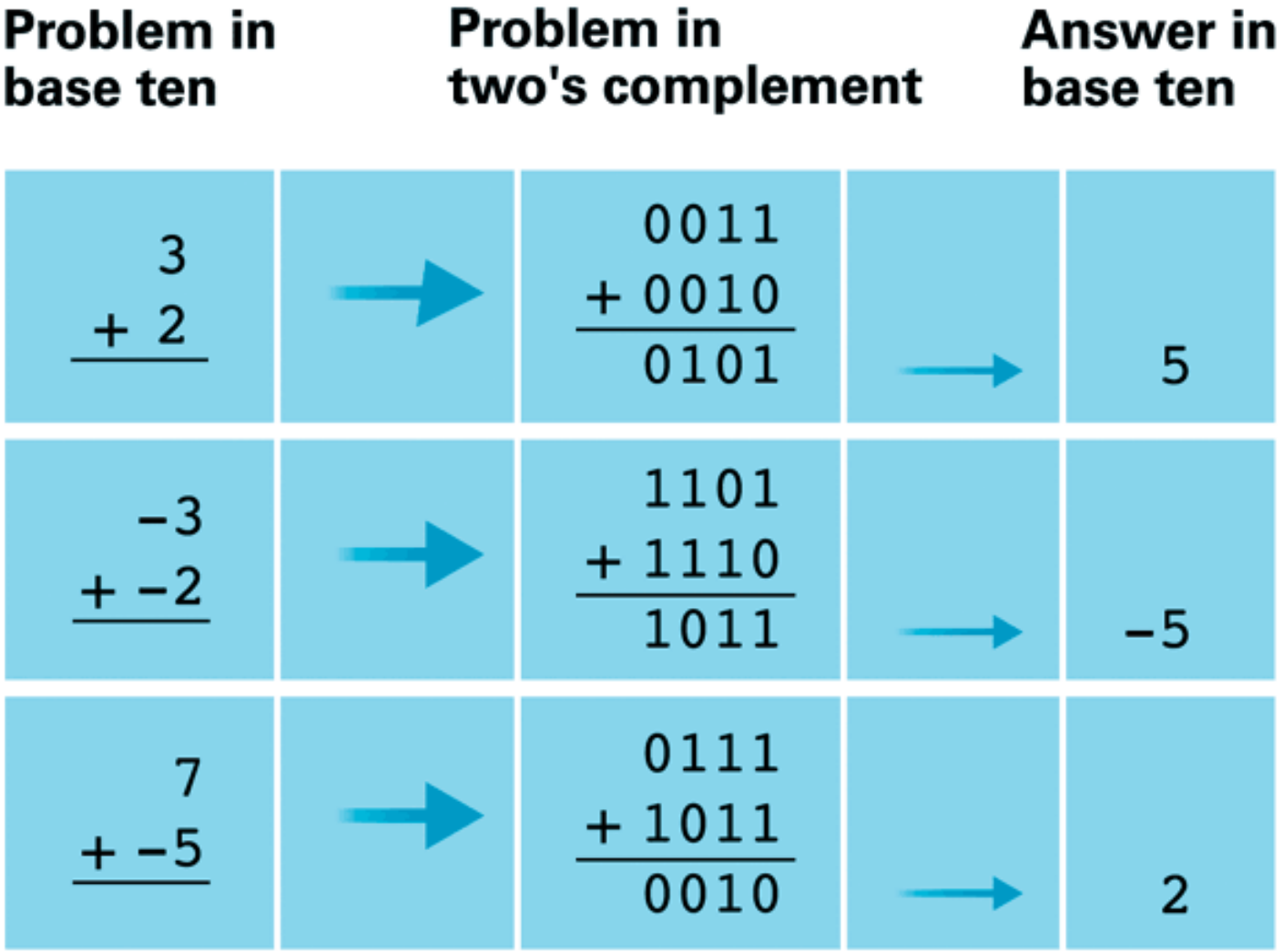


Figure 1.23: Addition problems converted to two's complement notation



Overflow Problem

Try adding 5 + 4 in 4-bit two's complement notation

5	0101	
+4	0100	
9	1001	→ **Overflow**. Result is Negative value

Actually –(0110+1) = -(0111) = -7

There is a limit to the size of the values that can be represented according to how many bits are used.

Normally integers are represented in 32-bit patterns.

Figure 1.24: An excess eight conversion table

Excess 8 = 過 8

= 減去8

就是說你看到的其實要 減去 8 後才是它真正的值

例如 1111 是 15 代表 7 == 15 – 8

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

Figure 1.25: An excess notation system using bit patterns of length three

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

3 bits 有 8 種組合

採用 excess 4 則可以表示 -4 到 3 共是8個值

若採用 excess 3 呢?

➔ 可以 -3 到 5

Fractional Values (小數值)

- What does a number like 908.543 actually mean: **$9x10^2+0x10^1+8x10^0+5x10^{-1}+4x10^{-2}+3x10^{-3}$**.
- Addition is easy, just line up the decimal points and go.
- However this is difficult to implement in a computer so the system known as floating point is used. This represents 908.543, or as $0.908543x10^3$, or as $9.08543x10^2$, ie
- $(9x10^0+0x10^{-1}+8x10^{-2}+5x10^{-3}+4x10^{-4}+3x10^{-5})x10^2$
- This is called floating point, and splits the number into a *fractional* part (9.08543 - the **mantissa**) and a "*power of the current base*" part (2 - the **exponent**).
- The same system works for binary numbers, but in base 2, ie. 0.1_2 means $2^{-1} = 0.5$, 0.01_2 means $2^{-2} = 0.25$, etc, so
- $11.0101_2 = (1x2^0+1x2^{-1}+0x2^{-2}+1x2^{-3}+0x2^{-4}+1x2^{-5})x2^1$.

Representing float in Binary

- $101.101_2 = (1x2^0+0x2^{-1}+1x2^{-2}+1x2^{-3}+0x2^{-4}+1x2^{-5})x2^2$ gives a means of storing fractional quantities in a binary format, in base 2 we write the number as
- **1.01101_2 times 2 to the power of 2** (2的2次方)
  and then store the number in two parts:
- **101101 as the *mantissa*** and **2 as the *exponent***.
- The number of bits allocated to the mantissa give the required accuracy, the number of bits allocated to the exponent (and the range of exponent values) give the required range of numbers (largest and smallest) that can be represented.
- If a number gets too large then there is overflow. It is also possible for a number to be too small (underflow).

mantissa = 假數

exponent = 指數

Storing Fractions

- **Floating-point Notation:** Consists of 3 fields: a sign bit, an Exponent field, and a mantissa (or fraction or significand) field.
- Related topics include:
  - Normalized form
  - Truncation errors (Precision problem)

mantissa = 假數

```
float x, y=1234567.2;    x = y + 0.0001;
```

```
float x=1234567.7, y=1234567.8;    if(x==y) …
```

IEEE 754/ IEEE854:
http://en.wikipedia.org/wiki/IEEE_754-1985

Figure 1.20: Decoding the binary representation 101.101

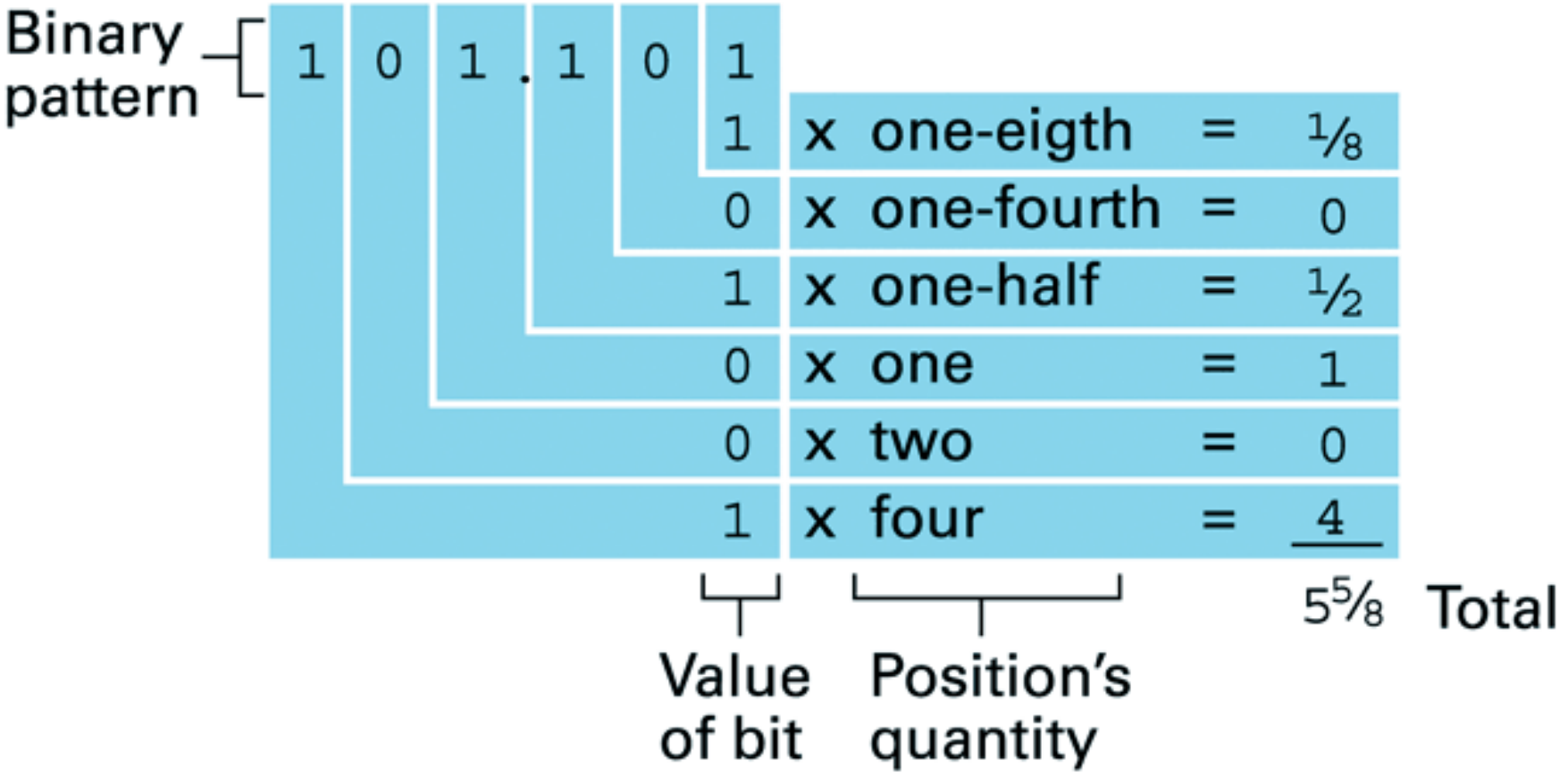
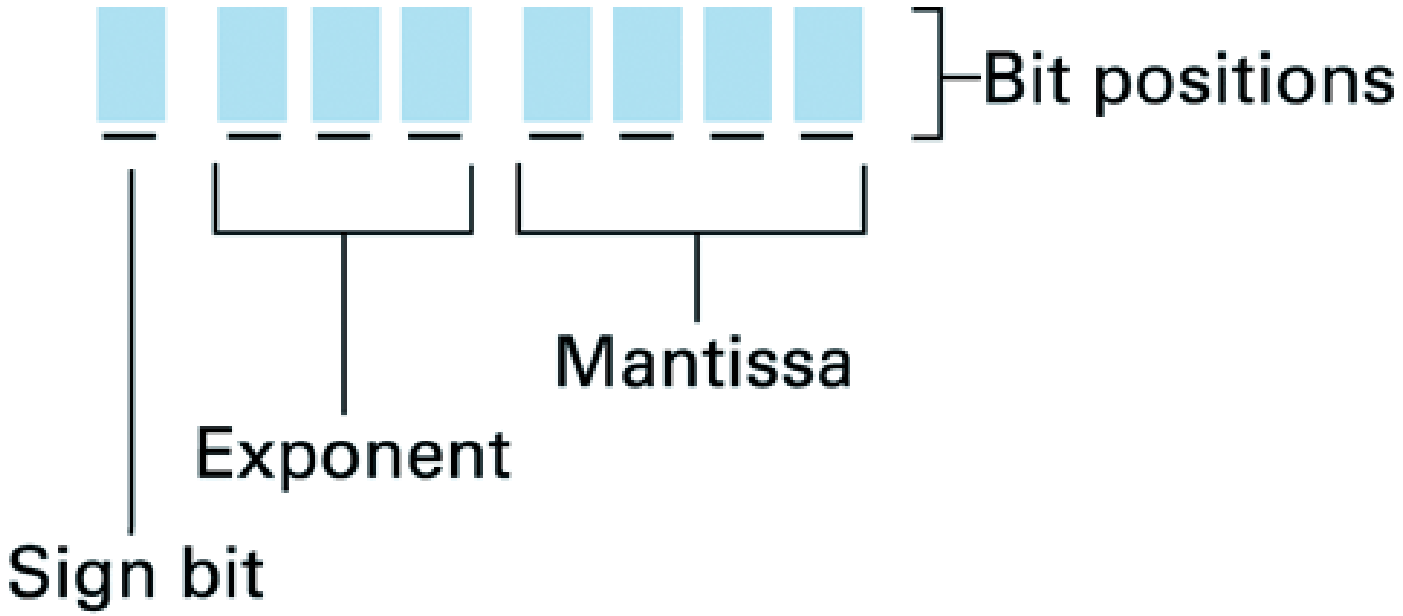


Figure 1.26: Floating-point notation components

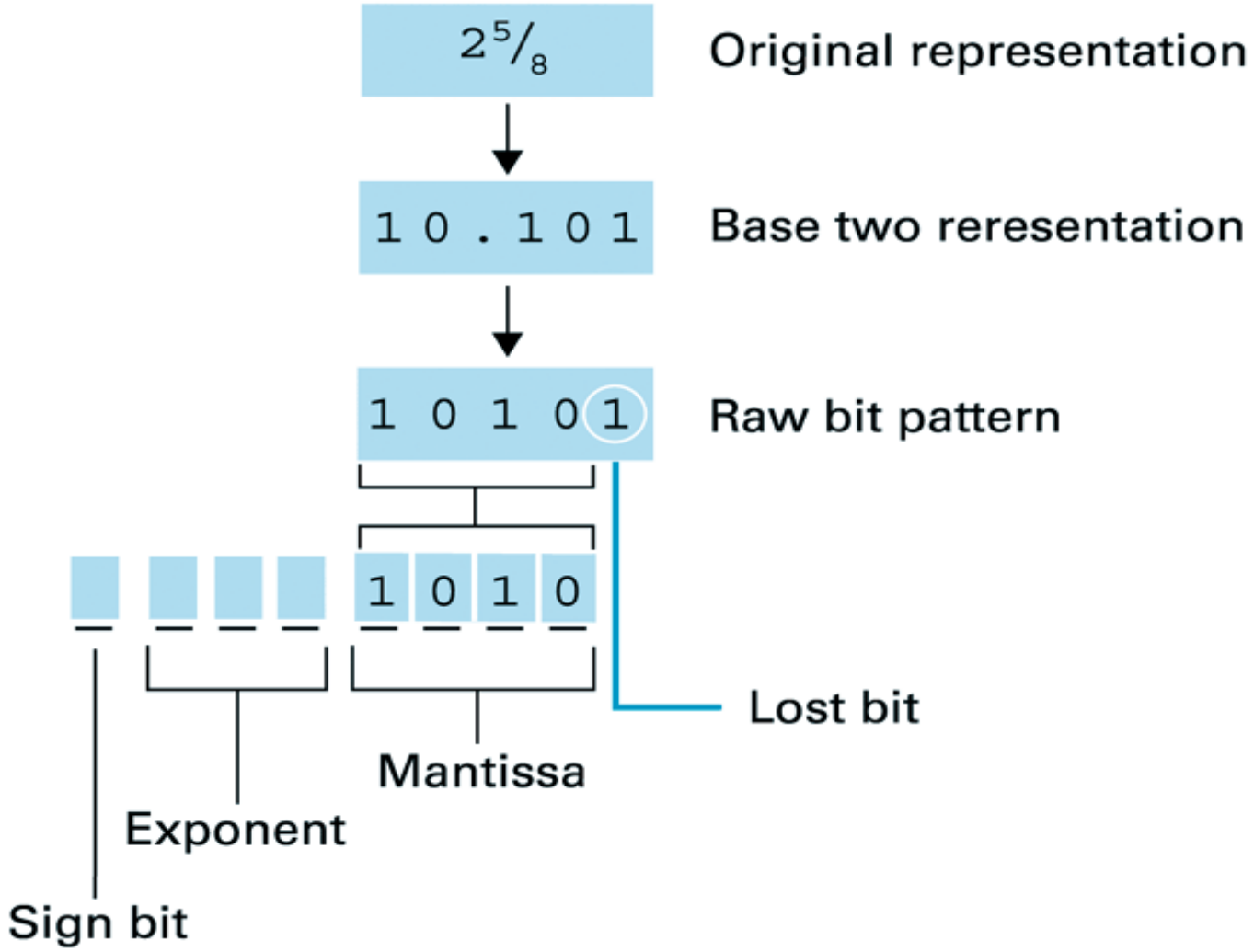


這是課本上附錄中的假想電腦儲存實數的方法; 有幾位準確度呢?

準確度 2進位 4 位 = 十進位 4*0.3010位

Mantissa(假數) 有些稱 significand 或是 fraction
(但CDC大電腦爲 integer mantissa)

Figure 1.27: Encoding the value 2 5/8



IEEE 754/854 如何表示 float? double?

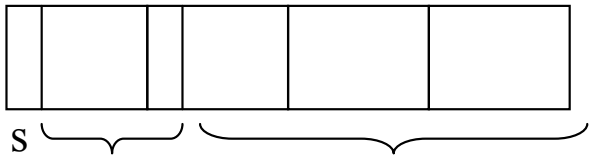
IEEE 754/854 Floating Point

- There have been many schemes for representing floating point numbers. Every manufacturer used to have one.
- The big push for standardisation came when Intel, who were developing their 8087 co-processor became involved in the IEEE process. There ensued a typical "fight" between their proposal and Digital (DEC), with the current standard IEEE 754 being adopted in 1985, being effectively the proposal from the "Intel people".
- The biggest debate between the schemes was on how to handle underflow. IEEE 854 now supplements 754.
- IEEE 754 has both 32 bit (single precision) and 64-bit (double precision) formats, and an 80-bit *double-extended* format.

用 gogle.com 輸入 IEEE 754 854 找資料看看

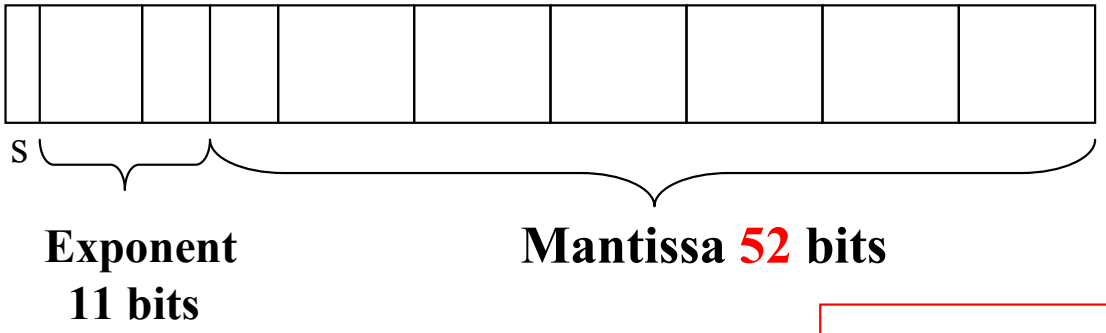
IEEE 754 Floating Point Formats

Significand precision: 24 (23 explicitly stored) bit



- Single precision 32-bit format (**float**)
- Exponent: **excess 127**

✓ 注意 **hidden bit 在假想的小數點左邊**



Double precision
64-bit format
Exponent: excess 1023

- **There is also an 80-bit double-extended format with 64 bits of mantissa & 15 bits of exponent. (long double?)**

The mantissa gives an extra effective bit (24 & 53 bits for the respective formats) as the implicit leading "1" bit is not stored. (hidden bit)

IEEE 754 Floating Point Formats specification

long double

Scheme	Single:	Double:	Double-Ext:
No of bits	32	64	80
Sign	1	1	1
Mantissa	23(24)	52(53)	64
Exponent	8	11	15
Exponent Bias	127	1023	16383
Significant Figures	6(~7)	15(~16)	19
Largest Number	3.4×10^{38}	1.79×10^{308}	1.2×10^{4932}
Smallest Number	1.17×10^{-38}	2.23×10^{-308}	$\sim 10^{-4932}$
Epsilon	1.19×10^{-7}	2.22×10^{-16} (ie. $1.0+\varepsilon \neq 1.0$)	
Underflow	gradual, uses leading mantissa 0 bits.		

✓ Turbo C++ 和很多PC上C/C++ 的 long double 是 Double-Ext

✓ 注意 Double-Ext 沒有 hidden bit

Examples of IEEE 754 float

We'll stick to 32-bits, the 64 and 80 bit formats work in a similar way.

E.g. 1)

$-3.625 = -11.101_2 = -1.1101 \times 2^1$, sign = −ve, exponent = 1

so: sign-bit = 1, exponent = 1+127 = 128 = 10000000_2

mantissa = 110 100 000 000 000 000 000 00 so representation is:

1 **10000000** 11010000000000000000000

E.g. 2)

$0.5625 = 0.1001_2 = 1.001 \times 2^{-1}$, sign = +ve, exponent = −1

so: sign-bit = 0, exponent = −1+127 = 126 = 01111110_2

mantissa = 001 000 000 000 000 000 000 00 so representation is:

0 **01111110** 00100000000000000000000

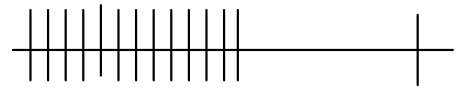
Note that all we are doing is taking these bit patterns and defining them to be -3.625 and 0.5625. In integer representation they would be −1066926078 and +2114977792 (you check these).

注意標準化成二進位正或負的 **1.xxx..** 但 **1.** 不存 (稱 hidden bit)

http://www.binaryconvert.com/convert_float.html

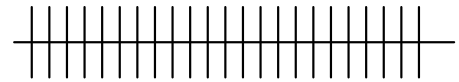
Gradual floating Underflow

- Very small floating point values can cause problems in arithmetic, as eventually the number system wants to record them as **zero**, which means that calculations lose accuracy.
- A problem arises because the smallest number is larger than the smallest difference between two numbers.



Nx1.17x10^{-38} 3.4x10^{-37} ???

- Gradual underflow uses sub-normal values to make the approaching of zero more "gradual", leaving a smaller "hole" for calculations to fall into.
- Use leading zeros on the mantissa of **float** number to allow the **effective exponent to go below -127**.



Nx1.17x10^{-38} 0

✓ **Underflow**: 實數的絕對值很小, 小到趨近於 0 則會被電腦當作 0

整數 沒有 Underflow ! Why?

float實數 underflow(虧失)實例

```
/* s eeeeeee fffffff gggggggg hhhhhhhh */
/* +/- 1.fffffffgggggggghhhhhhhh * 2 **(eeeeeeee - 127)   */
/*  eeeeeeee: Exponent base 2 excess 127 */
#include<stdio.h>
#define  DLOOP   146
int main( ) {    int k;
    float ans = 1.0;
    for(k=1; k<= DLOOP; ++k) ans = ans/2.0;
                /* ans = pow(2, -146);*/
    for(k=1; k<= 7; ++k){
        printf("= %12.8g\n", ans );
        ans /= 2.0;
    } return 0;
}
```

```
= 1.1210388e-44
= 5.6051939e-45
= 2.8025969e-45
= 1.4012985e-45
=           0
=           0
=           0
```

整數 沒有 Underflow ! Why?

Implications of Floating Point

- **When a calculation is done, it may no longer be exact. Even simple numbers like 0.1_{10} cannot be stored exactly in the binary floating point format.**
- $0.1_{10} = 0.000110011001\ldots_2$ which means that the very act of storing 0.1_{10} in floating point format loses some (small) amount of **accuracy**. Every time a calculation is done, more accuracy may be lost from the result.
- **It is possible for a calculation to become so inaccurate that the floating point bit pattern of the result is nonsense.**

- A software developer MUST take this "**loss of precision**" into account when storing and using data values and outputting results in floating point format.
- YOU ARE RESPONSIBLE FOR YOUR SOFTWARE'S OUTPUT ...
- But more than this ...
- YOU ARE ALSO RESPONSBILE FOR THE WAY IN WHICH THIS OUTPUT IS (REASONABLY) USED BY OTHERS ...

Implications: An Example

依下列規定動手算:

- Limit yourself to 3 significant (decimal) figures and use decimal arithmetic to add the following numbers.
- 0.007 0.103 0.205 0.008 3.12 0.006 0.876 0.005 0.015
- You may only add 2 numbers at a time, so you must keep a running total (initialized to 0.000) and add in the numbers one at a time.
- Now sort the numbers into ascending order and repeat the calculation again.
- **Is there a difference? If so, why?**

實數與準確度(precision)

```
#include<stdio.h>
float x, xdelta; int i;   /*precision.c */
main( ) { double y;
    x = 1234567.2,  xdelta = 0.0001;
    printf("Before loop, x=%f\n", x);
    for(i=1; i<= 8000; i++){
        y = x + xdelta;     /******/
        if(i == 1) printf("first y = %f\n", y);
        x = y;
   }
   printf("After loop, x=%f\n", x);
}
```

float 實數準確度七位多

```
ccbsd2:precision/> gcc precision.c
ccbsd2:precision/> ./a.out
Before loop, x=1234567.250000
first y = 1234567.250100
After loop, x=1234567.250000
ccbsd2:precision/>
```

float 實數佔用 32 bits

✓ 把1234567.2 改爲 1234567.3 再 Run 看看

float x = 1234567.2 ;float y = 1234567.3;

if(x == y) … ?

- float 1234567.2

  0 **10010011** **00101101011010000111010**

  - **0x 4996B43A**
- float 1234567.3

  0 **10010011** **00101101011010000111010**

  - **0x 4996B43A**
- float 1234567.7

  0 **10010011** **00101101011010000111110**

  - **0x 4996B43E**
- float 1234567.8

  0 **10010011** **00101101011010000111110**

  - **0x 4996B43E**

double 實數準確度

```
#include<stdio.h>
double x, xdelta; int i;   /*precdbl.c */
main( ) { double y;
    x = 1234567.2,  xdelta = 0.0001;
    printf("Before loop, x=%f\n", x);
    for(i=1; i<= 8000; i++){
        y = x + xdelta;     /******/
        if(i == 1) printf("first y = %f\n", y);
        x = y;
   }
   printf("After loop, x=%f\n", x);
}
```

double 實數準確十五位多

```
ccbsd2:precision/> gcc precdbl.c
ccbsd2:precision/> ./a.out
Before loop, x=1234567.200000
first y = 1234567.200100
After loop, x=1234568.000001
ccbsd2:precision/>
```

double 實數佔用 64 bits

整數 overflow (溢位)

```
#include<stdio.h>
int main( ) {
    short ans = 32765;
    int k;
    for(k=1; k<= 6; ++k) {
        printf ("= %d\n", ans++ );
    }
}
```

```
= 32765
= 32766
= 32767
= -32768
= -32767
= -32766
```

物極必反?

short k=-32768; k = -k; // what is the value of k ?

float實數 overflow (溢位)

```
/* s eeeeeee fffffff gggggggg hhhhhhhh */
/* +/- 1.fffffffgggggggghhhhhhhh * 2**(eeeeeeee - 127) */
/*  eeeeeeee: Exponent base 2 excess 127 */
#include<stdio.h>
#include<math.h>
int main( ) {
    float ans = pow(2,125);   /* 2 的 125 次方; pow在程式庫 */
    int k;
    for(k=1; k<= 6; ++k) {
        printf("= %f\n", ans );
        ans *= 2.0;
    }
}
```

= 42535295865117307932921825928971026432.000000

= 85070591730234615865843651857942052864.000000

= 170141183460469231731687303715884105728.000000

= Inf

= Inf

= Inf

Inf = 無窮大

Figure 1.28:
Decompressing xyxxyzy (5, 4, x)

a. Count backward 5 symbols.

x y x x y z y

b. Identify the four-bit segment to be appended to the end of the string.

x y x x y z y x x y z

c. Copy the four-bit segment onto the end of the message.

x y x x y z y x x y z x

d. Add the symbol identified in the triple to the end of the message.

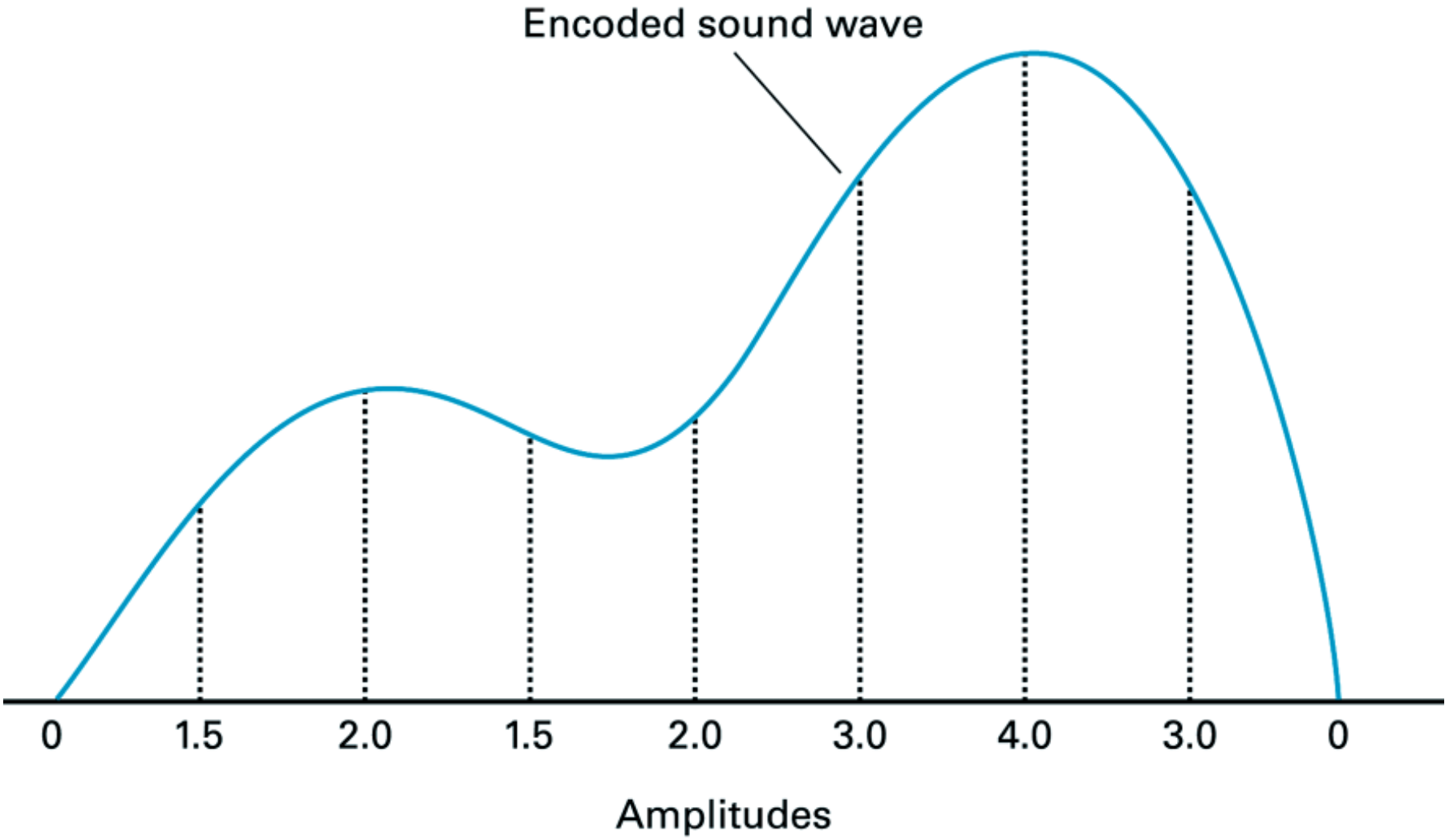
Representing Images

- Bit map techniques
  - Pixel: short for "picture element"
  - RGB
  - Luminance and chrominance
- Vector techniques
  - Scalable
  - TrueType and PostScript

Representing Sound

- Sampling techniques
  - Used for high quality recordings
  - Records actual audio
  - MP3 : MPEG_Layer3
- MIDI
  - Used in music synthesizers
  - Records “musical score”

Figure 1.14: The sound wave represented by the sequence 0, 1.5, 2.0, 1.5, 2.0, 3.0, 4.0, 3.0, 0



Data Compression

- Lossy versus lossless
- Run-length encoding
- Frequency-dependent encoding (Huffman codes)
- Relative encoding
- Dictionary encoding (Includes adaptive dictionary encoding such as LZW encoding.)
  - http://zh.wikipedia.org/wiki/LZW

Huffman codes

23x8bits =184bits

- Given ‘xyx xyz xyx xyw xyx xyw‘ with table

code	char
00	space
01	x
10	y
110	w
111	z

01 10 01 00 01 10 111 00 01 10 01 00
01 10 110 01 10 01 00 01 10 110

6x(8+8)+47=143bits

LZW (Lempel-Ziv-Welsh) encoding

23x8bits =184bits

- Given ‘xyx xyx xyx xyw xyx xyw‘ with table

#	words
0	space
1	x
2	y
3	w
4	xyx
5	xyw

121040401230405

16x8+15x3=173bits

- ‘x’ → 1
- ‘xyx’ → 121
- ‘xyx ’ → 1210
- ‘xyx xyx’ → 12104
- …

Compressing Images

- GIF (Graphic Interchange Format)
  - a palette of 256 colors for each file
  - 256x256x256 choices of colors
  - Good for cartoons
- **JPEG** (Joint Photographic Experts Group):
  - **Good for photographs**
- TIFF (Tagged Image File Format):
  - Good for image archiving

Compressing Audio and Video

- MPEG (Motion Picture Experts Group)
  - ISO
  - Relative techniques
  - High definition television broadcast
  - Video conferencing
- MP3 (MPEG Layer 3)
  - Temporal masking
  - Frequency masking
  - near CD quality

MP4 ? MPEG-4? MPEG-1, MPEG-2, MPEG-7, MPEG-47

MPEG-21 (ISO/IEC 21000) (DRM: Digital Rights Management)

http://en.wikipedia.org/wiki/MPEG-21

Figure 1.28: Decompressing xyxxyzy (5, 4, x)

a. Count backward 5 symbols.

x y x x y z y

b. Identify the four-bit segment to be appended to the end of the string.

x y x x y z y x x y z

c. Copy the four-bit segment onto the end of the message.

x y x x y z y x x y z x

d. Add the symbol identified in the triple to the end of the message.

Communication Errors

- Parity bits (even parity versus odd parity)
- Checkbytes / Checksum

Weighted checksum

Taiwan 身分證號碼最後一碼 (檢查碼)

Cyclic Redundancy Checks (CRCs)

MD5 : Message Digest algorithm 5

http://zh.wikipedia.org/zh-tw/MD5

http://zh.wikipedia.org/zh-tw/SHA-2

- **Error Correcting Codes (ECC)**
  http://www.eccpage.com/

Parity (同位) codes

- Simple example - (8,7) code is a parity bit appended to a 7-bit code
  - *Odd parity* - extra bit gives odd no. of 1's
  - *Even parity* - extra bit gives even no. of 1's
  - Hamming distance of 2, hence able to detect single-bit errors
  - Actually capable of detecting any odd number of bit errors
- (11,7) Hamming code has distance of 3:
  - Detects 2-bit errors and corrects 1-bit error
  - Widely used in memory ECC codes

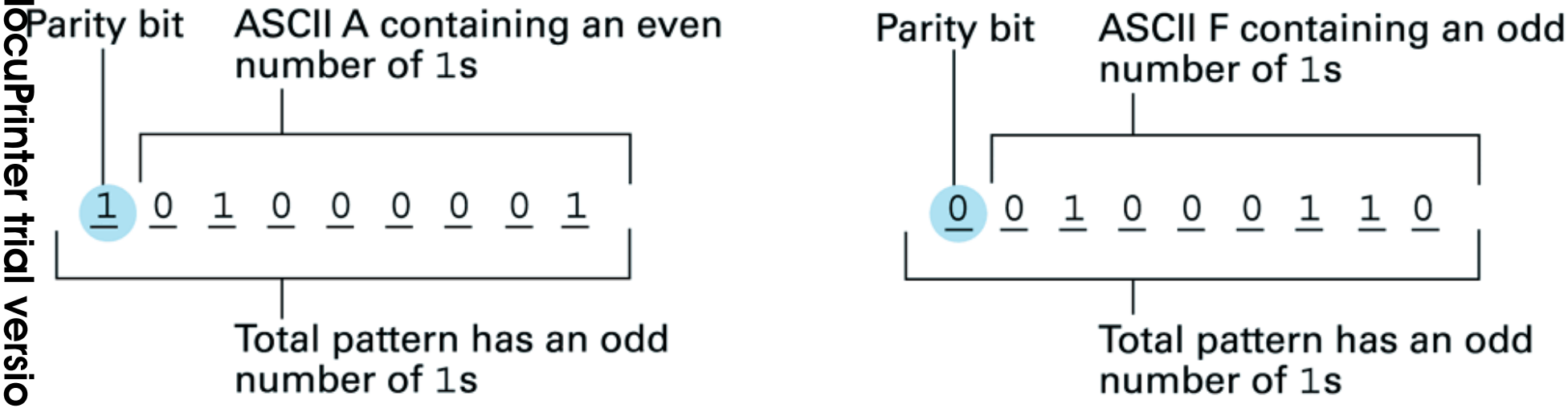
Hamming distance: minimum number of bits by which two valid *n*-bit codes differ

(*n*,*k*) code has *k* bits of information coded in *n* bits (*n* > *k*)

Hamming distance of *d* can detect errors up to (*d*-1) bits

Figure 1.29: The ASCII codes for the letters A and F adjusted for odd parity

example - (9,8) code is a parity bit appended to a 8-bit code



Odd parity? Even parity?

強迫使得 1 的 bit 是**奇數**個, 稱 odd parity 奇數同位

Redundancy for Error detection

- Helps to recognise that information is damaged
- (n,k) code has k bits of information coded in n bits ($n > k$)
- 2^n possible codes with 2^k meaningful values
- Remaining codes only result from damage, e.g. **0001**
  - Even if only 1-bit error, it cannot be corrected as it could be 1001 or 0011
- Usually discard and retransmit
- An important concept is *Hamming distance (海明距離)*

(n,k) code 編碼
Example: $(4,2)$ code
只有 2 bit 有意義

Valid codes

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

Hamming distance (*海明距離*)

- *Def n*: **minimum number of bits by which two valid *n*-bit codes differ**
- **Hamming distance of *d* can detect errors up to (*d*-1) bits**
- Can also correct errors affecting up to $floor\left(\frac{d-1}{2}\right)$ bits

- Example (4,2) code has Hamming distance 2
  - Can detect errors of up to 1 bit
  - No correction possible

0011 1100

0110 1001

- **Hamming distance depends on the difference between *k* and *n***

(*n*,*k*) code has *k* bits of information coded in *n* bits (*n* > *k*)

Figure 1.30: An Error-Correcting Code

ECC : Error Correcting Code

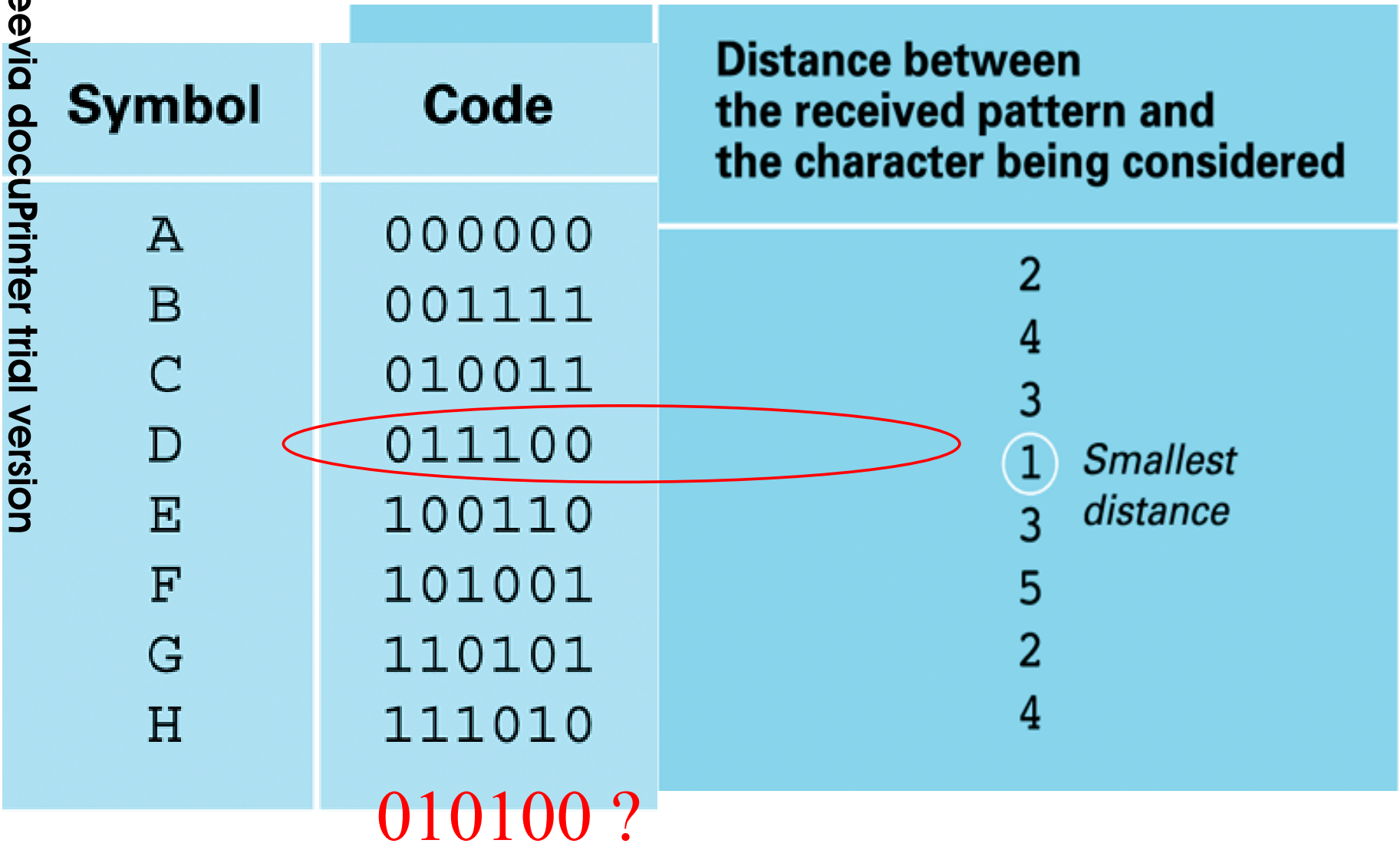
如何做到錯 1個bit 時可以自動更正?

➔Hamming distance 須爲 3

➔(3-1)/2 == 1

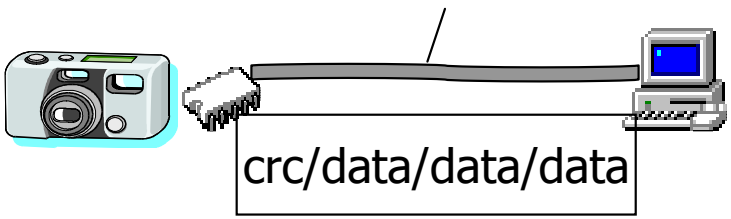
Symbol	Code
A	000000
B	001111
C	010011
D	011100
E	100110
F	101001
G	110101
H	111010

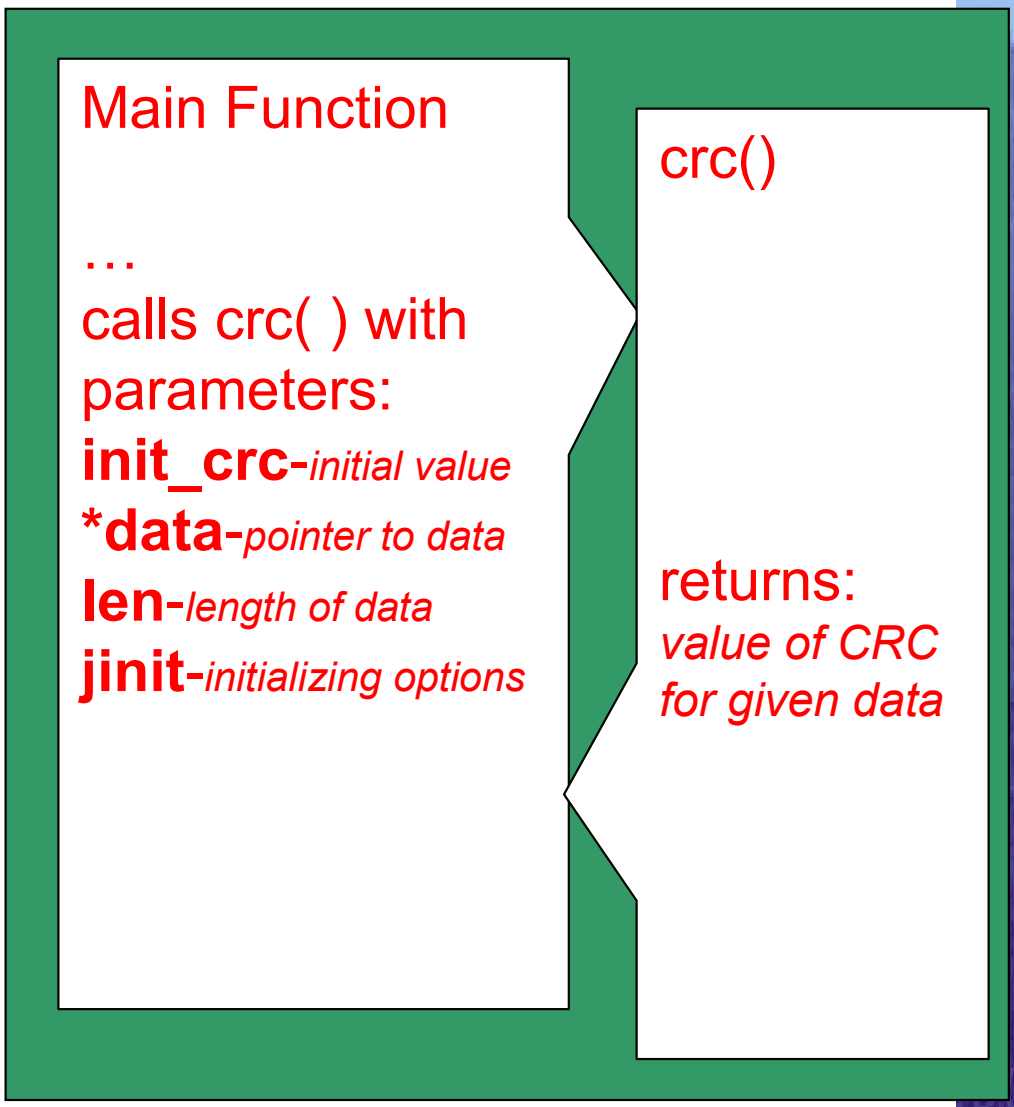
Figure 1.31: Decoding the pattern 010100 using the ECC code in Figure 1.30



CRC Algorithms

- CRC – Cyclic Redundancy Check
  - Used for error checking during communication, stronger than parity
  - Mathematically, divides a constant into the data and saves the remainder





Different Algorithms – CRC in Hardware

```
char crc_hw(…)
{
   unsigned short j , crc_value = init_crc;
   unsigned short new_crc_value;
   if (jinit >= 0) crc_value=((uchar) jinit) | (((uchar) jinit) << 8);
   for (j=1;j<=len;j++) {
     new_crc_value = bit(4,data[j]) ^ bit(0,data[j]) ^ bit(8,crc_value) ^ bit(12,crc_value); // bit 0
     new_crc_value = new_crc_value |
      (bit(5,data[j])^bit(1,data[j])^bit(9,crc_value)^bit(13,crc_value))<<1;
     new_crc_value = new_crc_value |
      (bit(6,data[j])^bit(2,data[j])^bit(10,crc_value)^bit(14,crc_value))<< 2;
.     … continue for bits 3 through 7 …
.  }
   return (new_crc_value);
}
```

- Hardware Version
  - Knowing the generator polynomial, one can calculate the XOR's for each individual bit
  - Each CRC value is the result of bit-wise XOR's with the data and the previous CRC value
  - Synthesizes to hw very nicely; but getting bits and shifting are inefficient in sw

Different Algorithms – CRC in Software

Software Version

- Before doing any calculations, create an initialization table that calculates the CRC for each individual character
- Use data as index into initialization table and execute two XOR's
- Requires lookups, but faster for a sequential calculation

```
char crc_sw(…) // Source: Numerical Recipes in C
{
  unsigned short initialize_table(unsigned short crc,
      unsigned char one_char);
  static unsigned short icrctb[256];
  unsigned short tmp1, j , crc_value = init_crc;
  if (!init) {
    init=1;
    for (j=0;j<=255;j++) {
      icrctb[j]=initialize_table(j << 8,(uchar)0);
    }
  }
  if (jinit >= 0) crc_value=((uchar) jinit) | (((uchar) jinit) <<
      8);
    for (j=1;j<=len;j++) {
      tmp1 = data[j] ^ HIBYTE(crc_value);
      crc_value = icrctb[tmp1] ^ LOBYTE(crc_value) << 8;
    }
  }
return (crc_value);
}
```

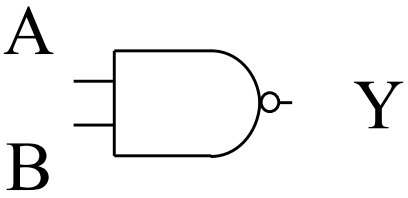
More about Logical gates

- De Morgan theory
- Using NAND gates ONLY
- Using NOR gates ONLY
- Adder
  - Half Adder
  - Full Adder

Karnaugh maps ?

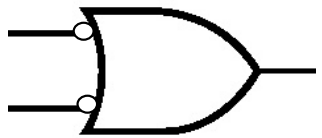
De Morgan theory (De Morgan Law)

- A ***NAND*** gate:



  - A ***NAND*** is equivalent to an OR gate with two NOT gates

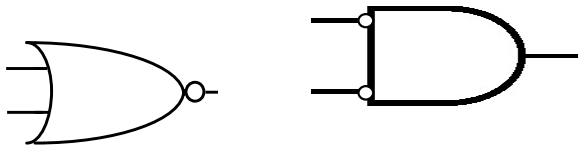
$$Y = \overline{A \cdot B} = \overline{A} + \overline{B}$$



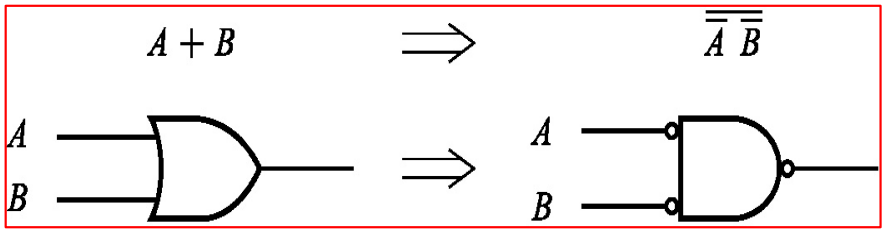
$$(A \cap B)^C = A^C \cup B^C$$

- A NOR gate can be constructed by using one AND gate with two Inverters (NOT gate)

$$Y = \overline{A+B} = \overline{A} \cdot \overline{B}$$



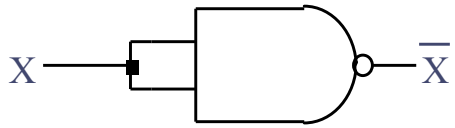
$$(A \cup B)^C = A^C \cap B^C$$



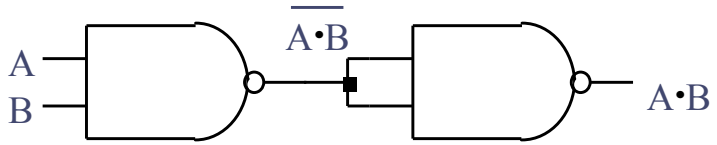
Using NAND Gates ONLY (1/2)

- It is possible to implement **any boolean expression** using only NAND gates

NOT

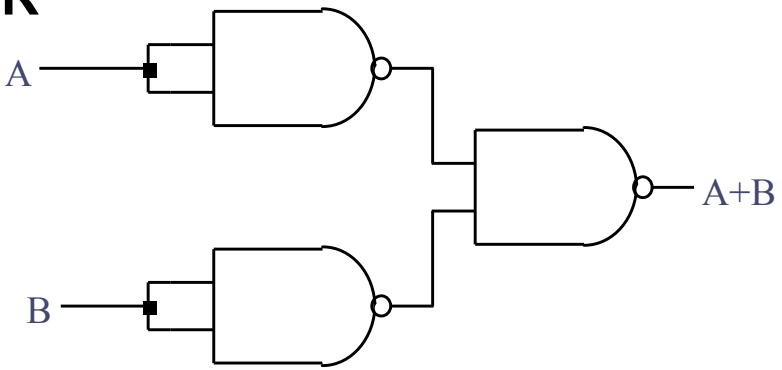


AND



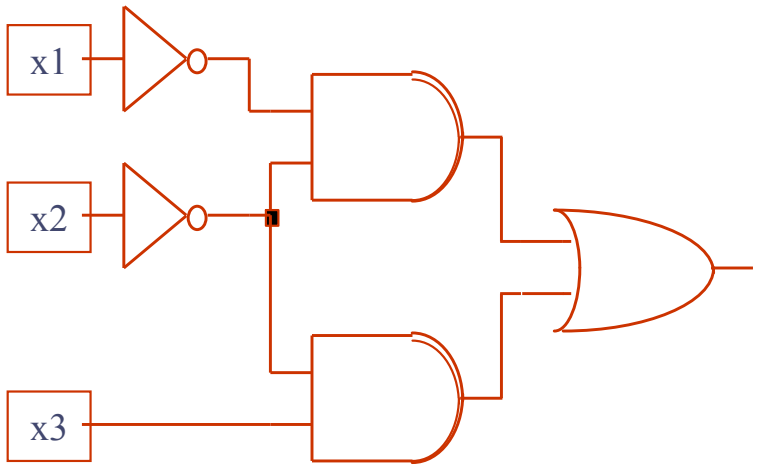
OR

$$A + B = \overline{\overline{A}.\overline{B}}$$



Using NAND Gates ONLY (2/2)

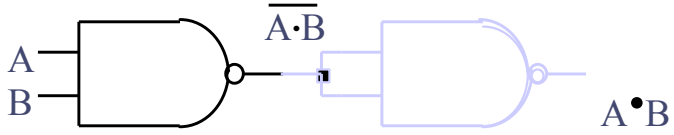
- Example of NAND Gate representation
  - Implement the following circuit using only NAND gates

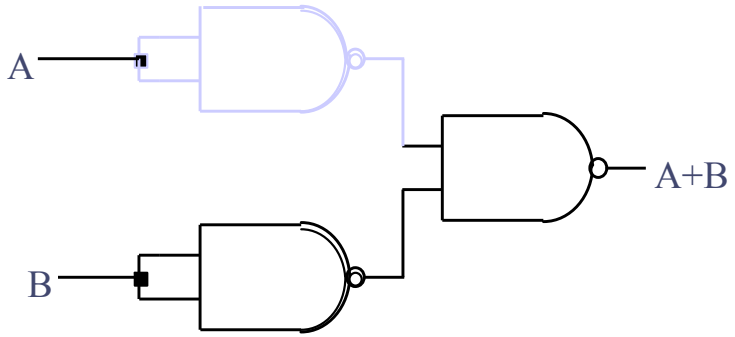


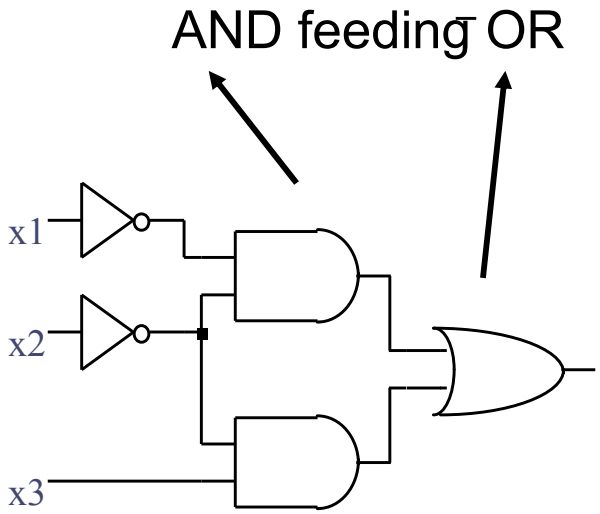
Hint: using De Morgan theory

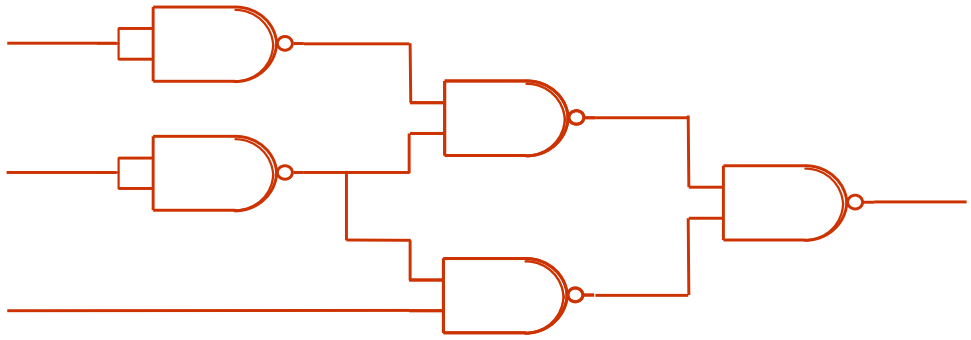
Answer to previous page problem

- Tips: two NOTs together can be removed.



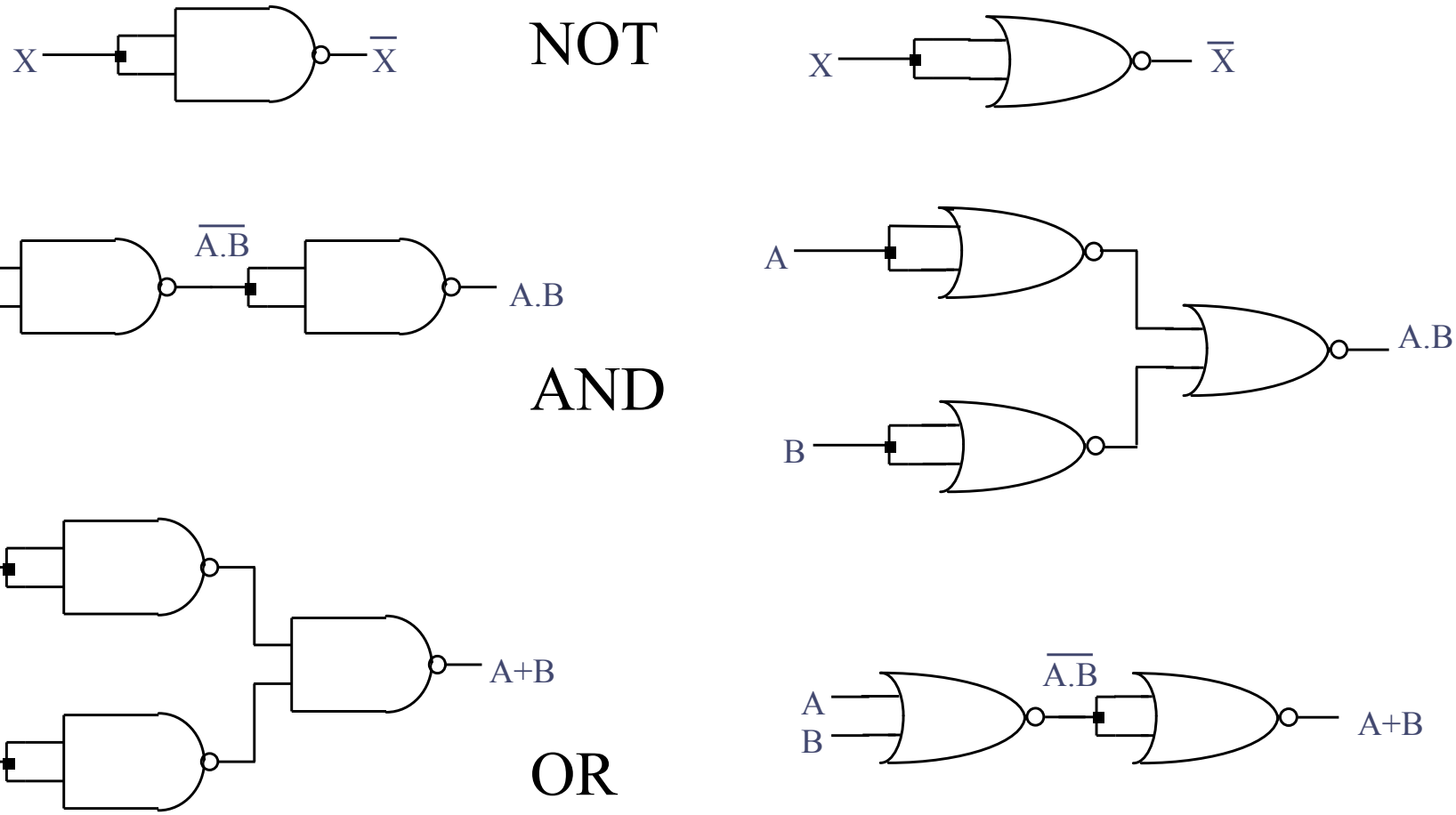






Using NOR Gates ONLY

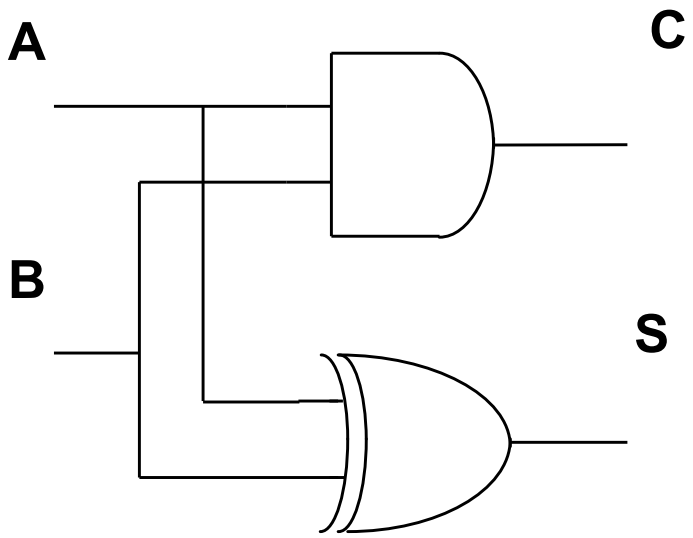
- Similar pattern to using NAND gates



Half adder (do adding A, B)

- The sum is XOR operation
- The carry is their AND value

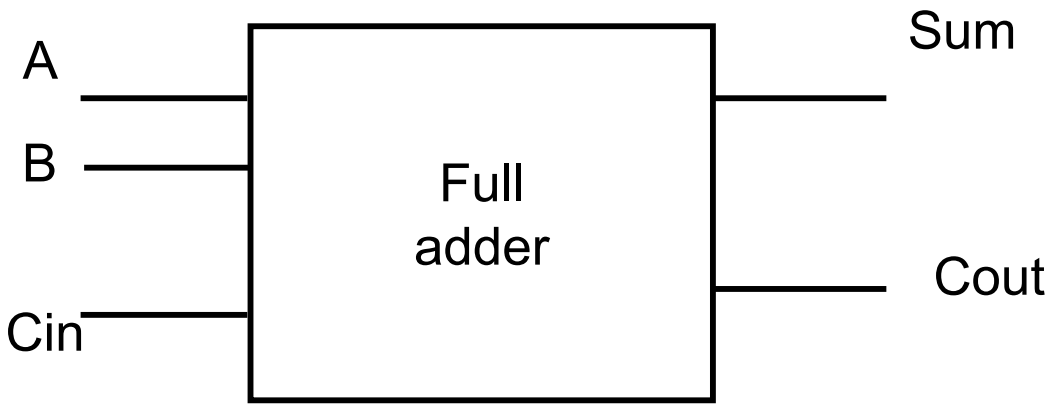
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



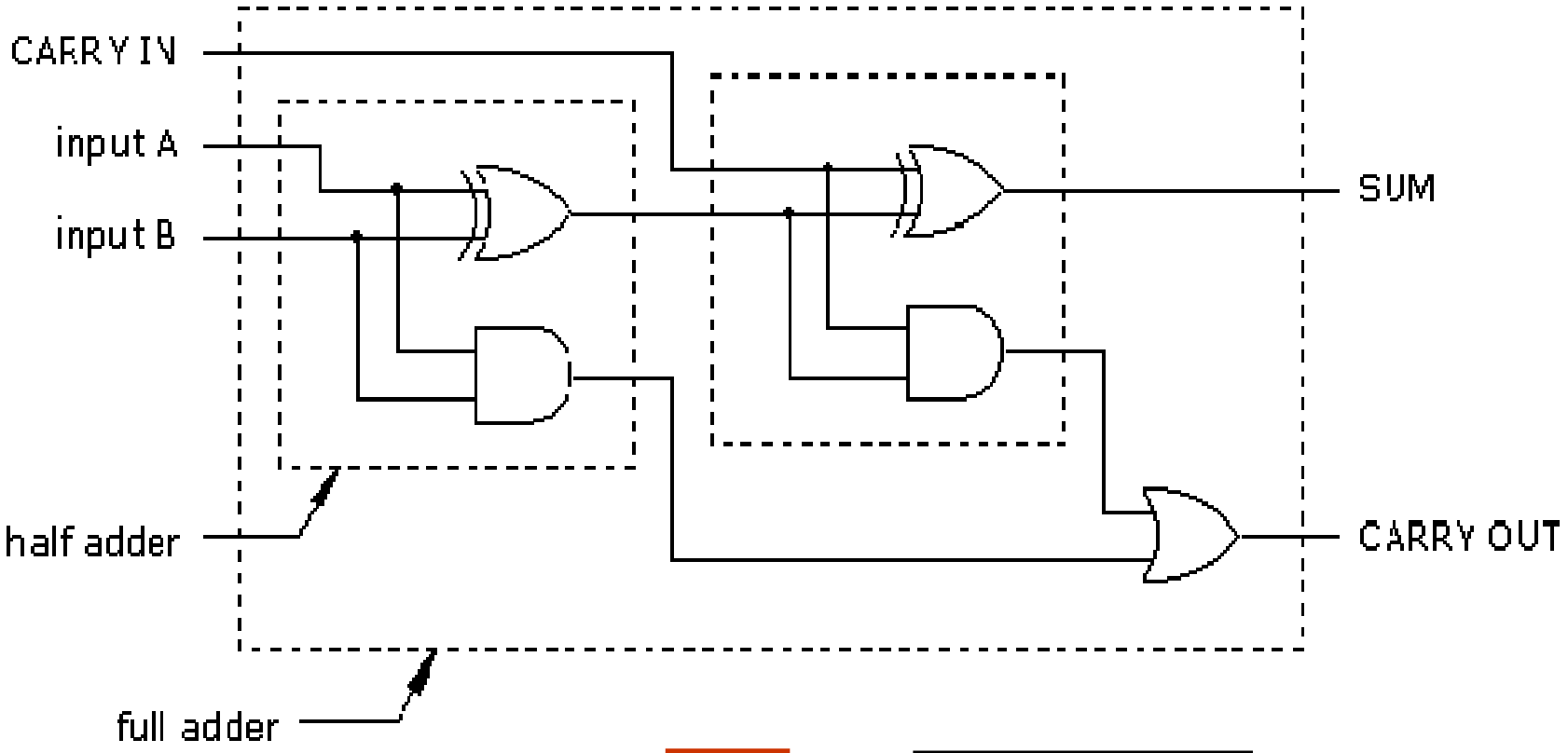
Full Adder

- Develop a truth table and Boolean expressions for the full adder (adding A, B, and Carry in.)

Cin	A	B	S	C
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		



A Full adder can be constructed using 2 half adder



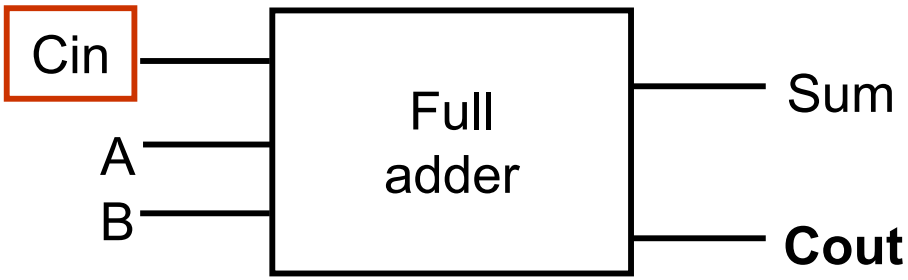
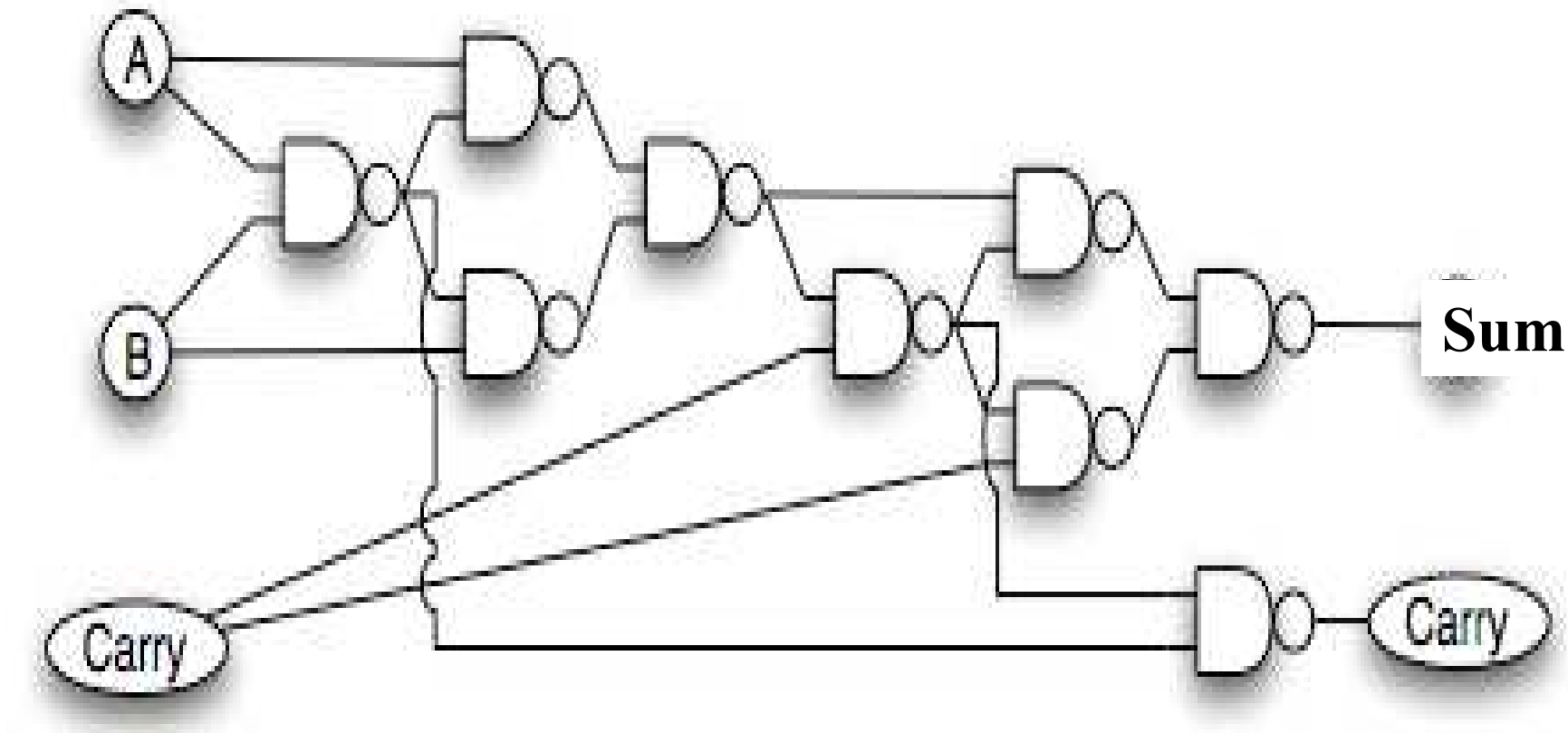


Fig.B.2 in Appendix B

A Full Adder using only NAND gates



Truth table for full adder

Cin	A	B	S	Cout
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

Hint:

Complete the **Karnaugh maps** for the Sum and the Carry out columns

Karnaugh maps for sum and carry

Sum – 1 when odd number of inputs is 1 = XOR gate

AB / Cin	00	01	11	10
0		1		1
1	1		1	

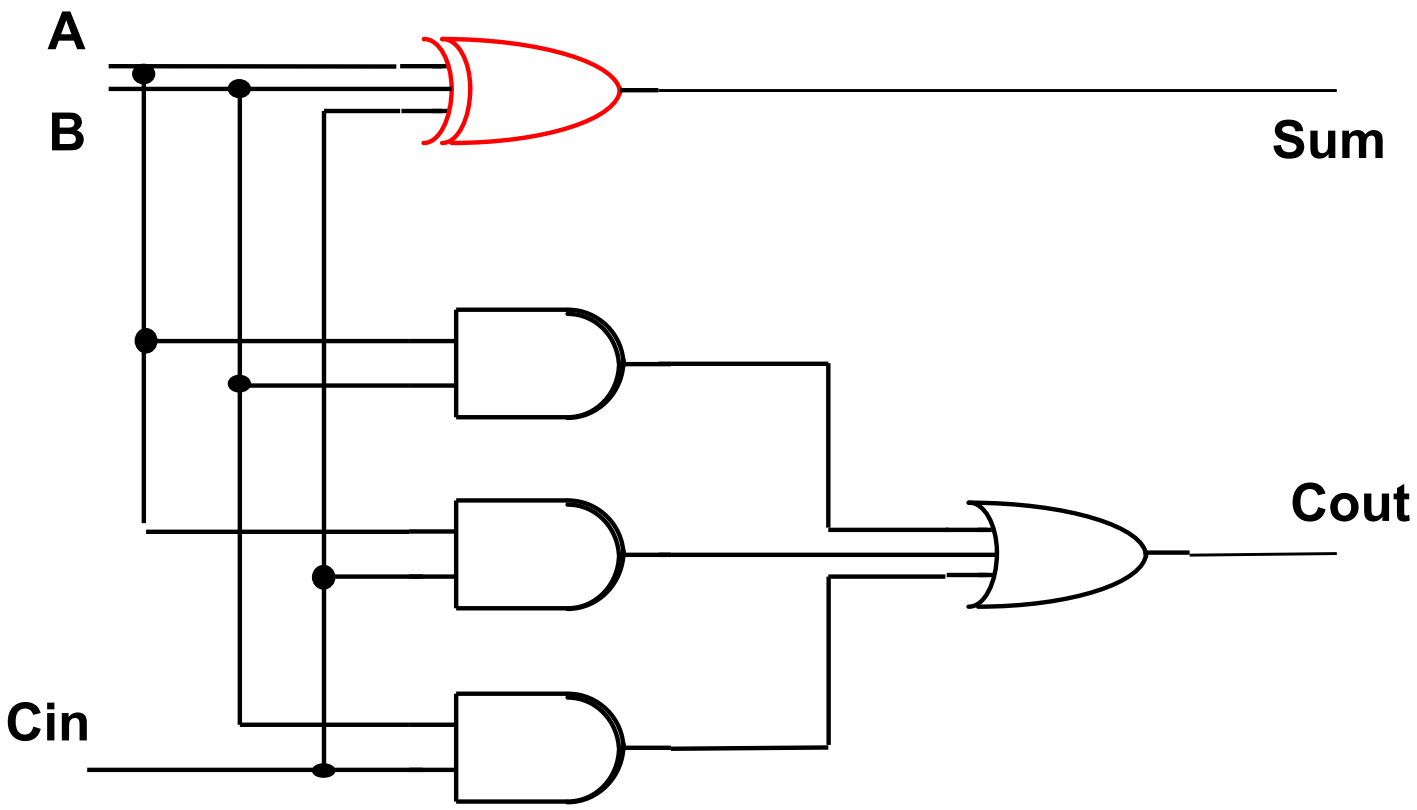
Sum = Cin xor A xor B

Carry out - simplifies to 3 pairs

AB / Cin	00	01	11	10
0			1	
1		1	1	1

Cout = A.B + A.Cin + B.Cin

The Full adder circuit (化簡後)



Sum = Cin **xor** A **xor** B

Cout = A.B + A.Cin + B.Cin

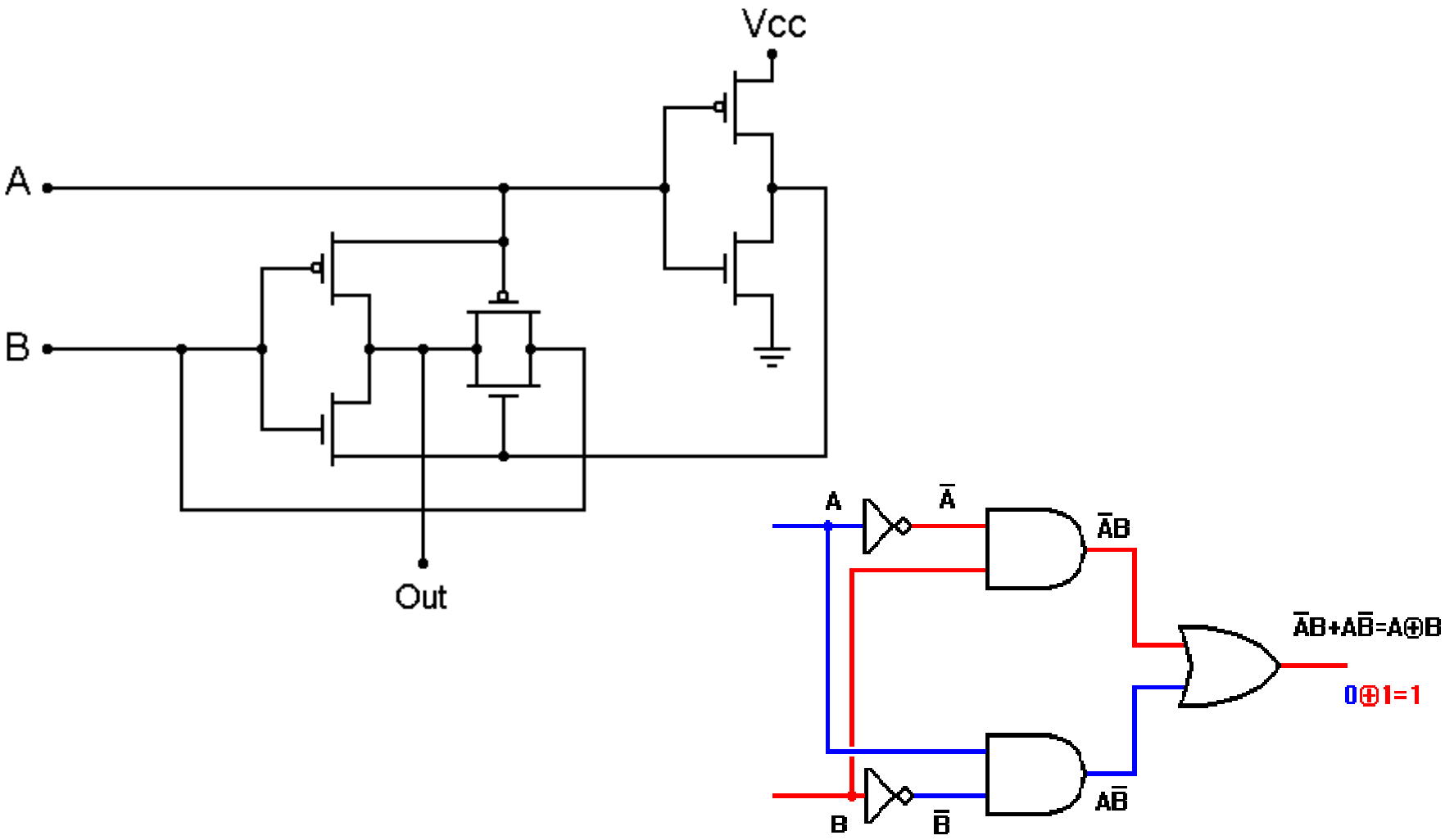
Exercise

- Write a function (in any of C/C++/Java) that takes 2 long int as parameters, and return the **sum** of these 2 long integers. But in this function, any + - * / % operations (including negative -) are **NOT** allowed!!!
- Using the function above to write a subtraction function that takes 2 long integers as parameters.
- Then write a testing driver (a small main program) to test your functions.

Hint: Please refer to the Full Adder circuit on previous page

XOR gate CMOS wiring

http://en.wikipedia.org/wiki/XOR_gate



XOR using NAND only

((a NAND b) NAND
((a NAND a) NAND (b NAND b)))
NAND
((a NAND b) NAND
((a NAND a) NAND (b NAND b)))

De Morgan's laws

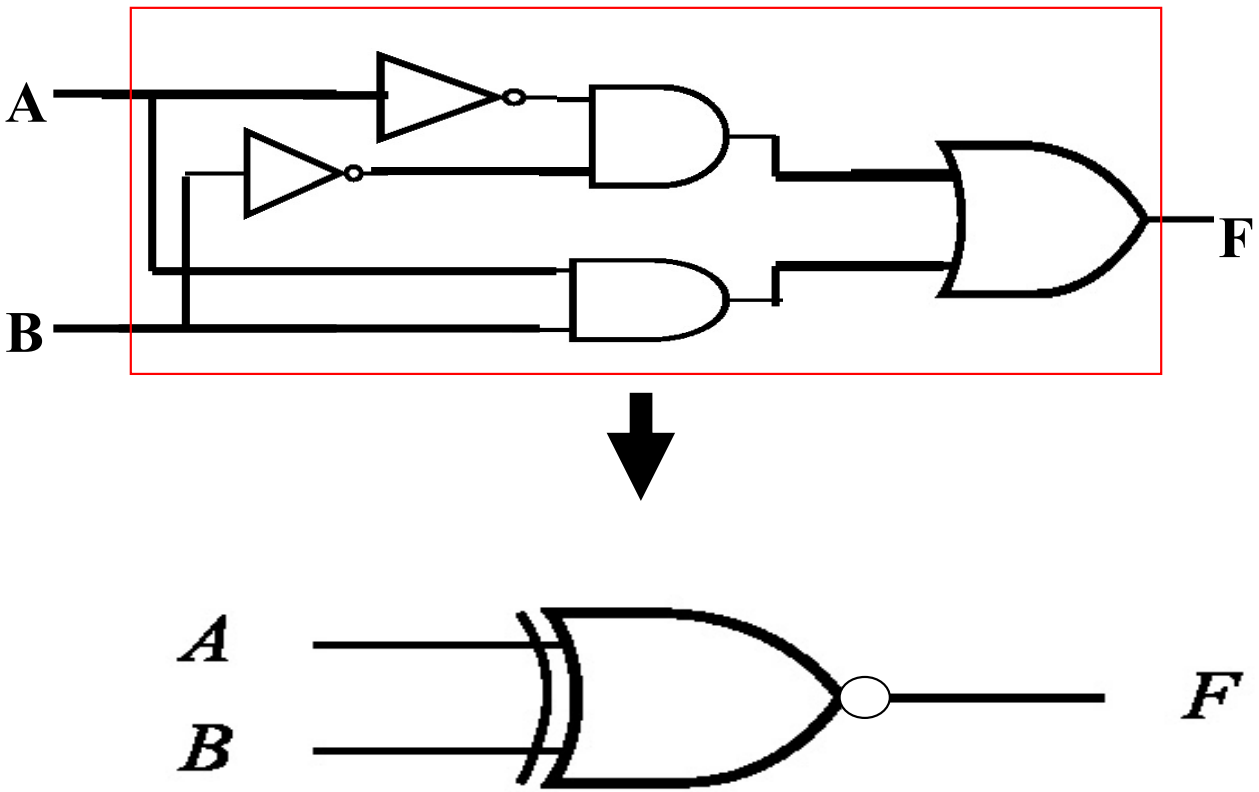
NOT (P OR Q) = (NOT P) AND (NOT Q)
NOT (P AND Q) = (NOT P) OR (NOT Q)

a OR b = (a NAND a) NAND (b NAND b)

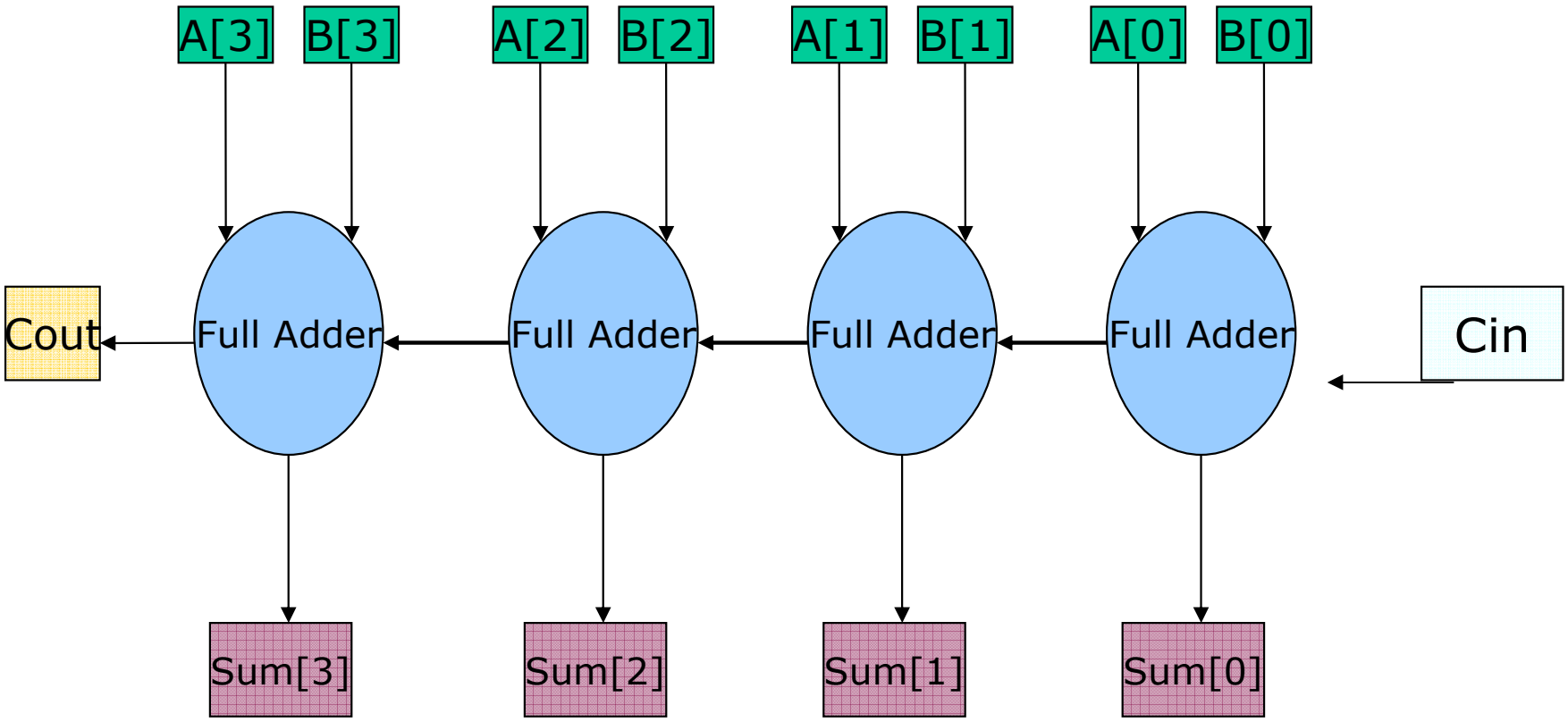
a AND b = (a NAND b) NAND (a NAND b)

a XOR b = (a NAND b) AND (a OR b)

XNOR gate (互斥反或閘)



Construct 4-Bit Full Adder using 4 1-bit Full Adder (sequential)

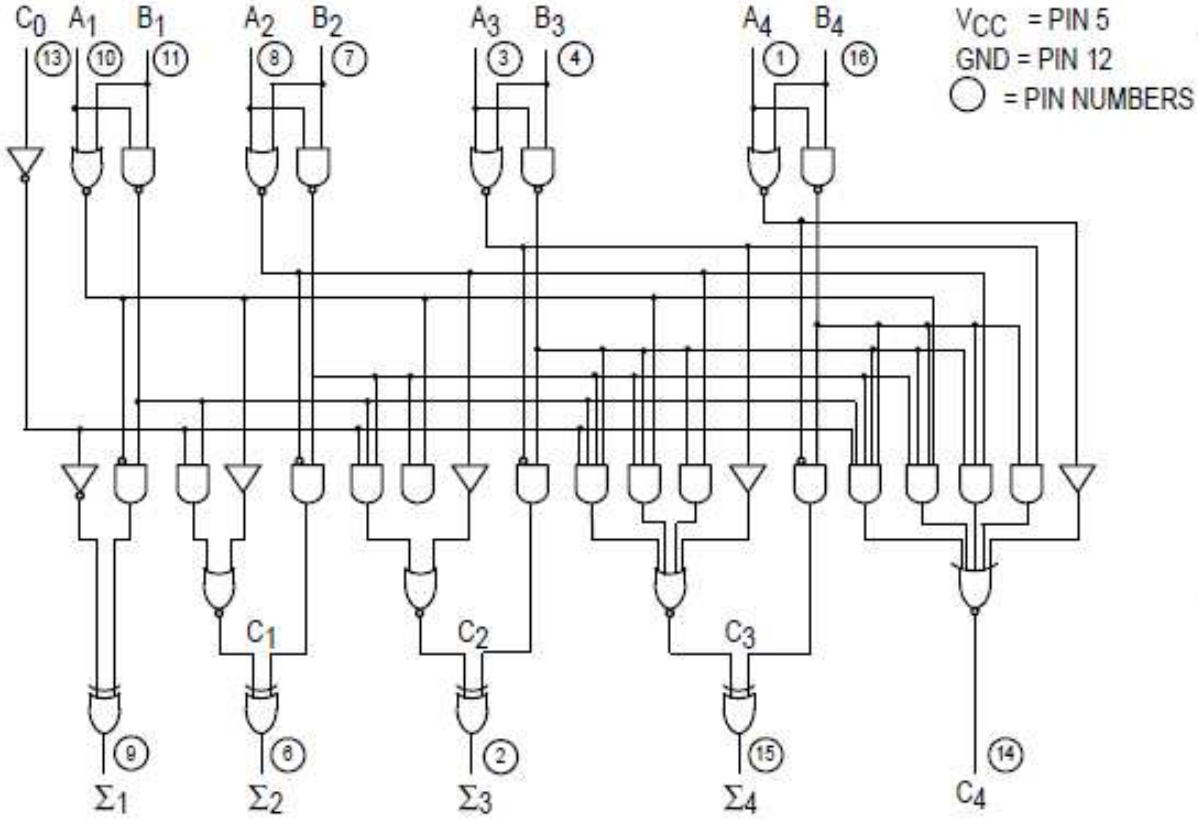


How to do carry look ahead ?

A_1-A_4	Operand A Inputs	1.0 U.L.	0.5 U.L.
B_1-B_4	Operand B Inputs	1.0 U.L.	0.5 U.L.
C_0	Carry Input	0.5 U.L.	0.25 U.L.
$\Sigma_1-\Sigma_4$	Sum Outputs (Note b)	10 U.L.	5 (2.5) U.L.
C_4	Carry Output (Note b)	10 U.L.	5 (2.5) U.L.

NOTES:
a) 1 TTL Unit Load (U.L.) = 40 µA HIGH/1.6 mA LOW.
b) The Output LOW drive factor is 2.5 U.L. for Military (54) and 5 U.L. for Commercial (74) Temperature Ranges.

LOGIC DIAGRAM

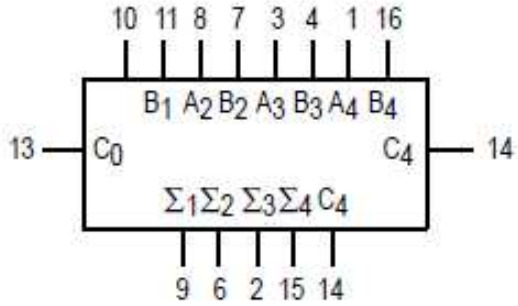


D SUFFIX
SOIC
CASE 751B-03

ORDERING INFORMATION

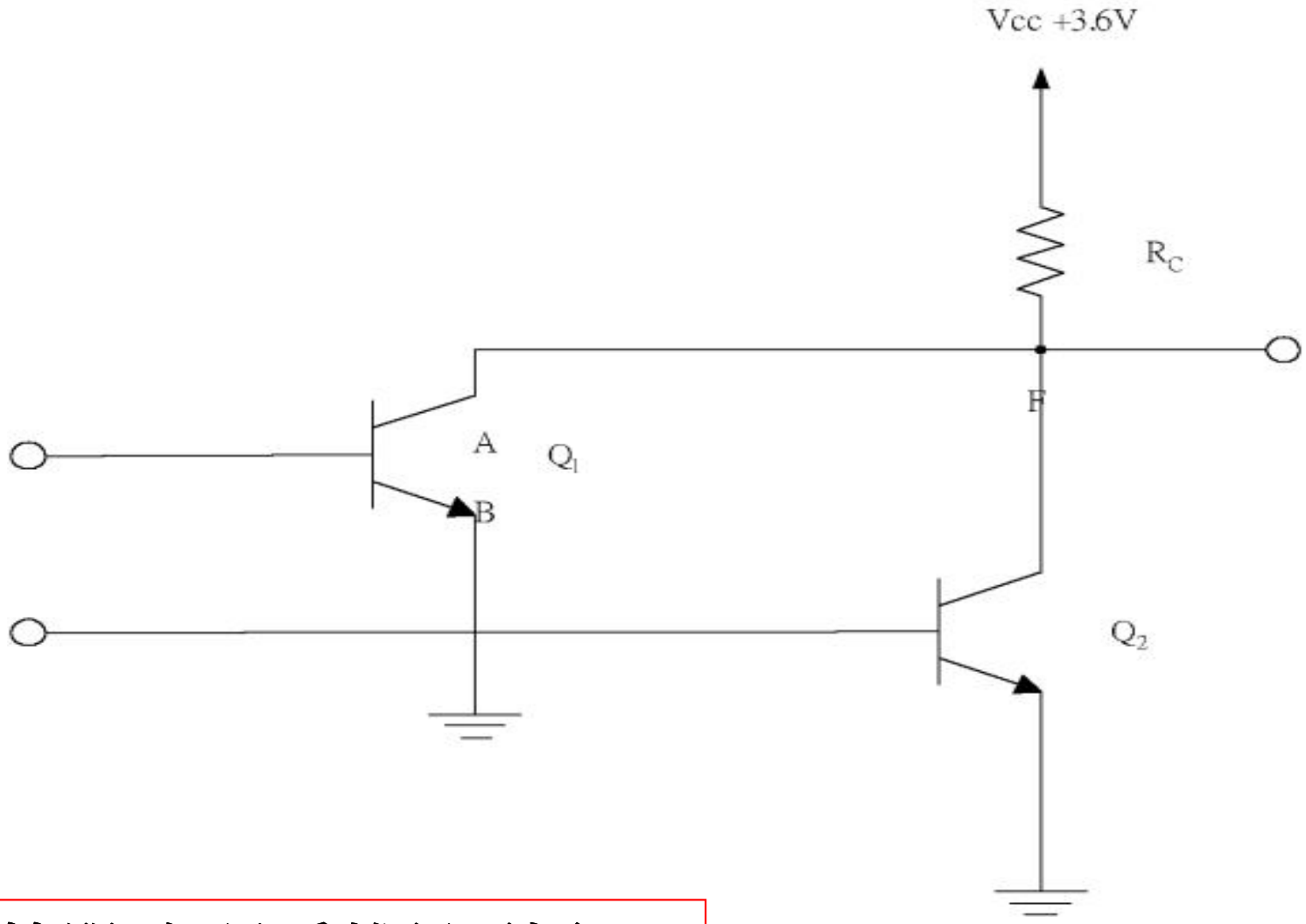
SN54LSXXJ	Ceramic
SN74LSXXN	Plastic
SN74LSXXD	SOIC

LOGIC SYMBOL



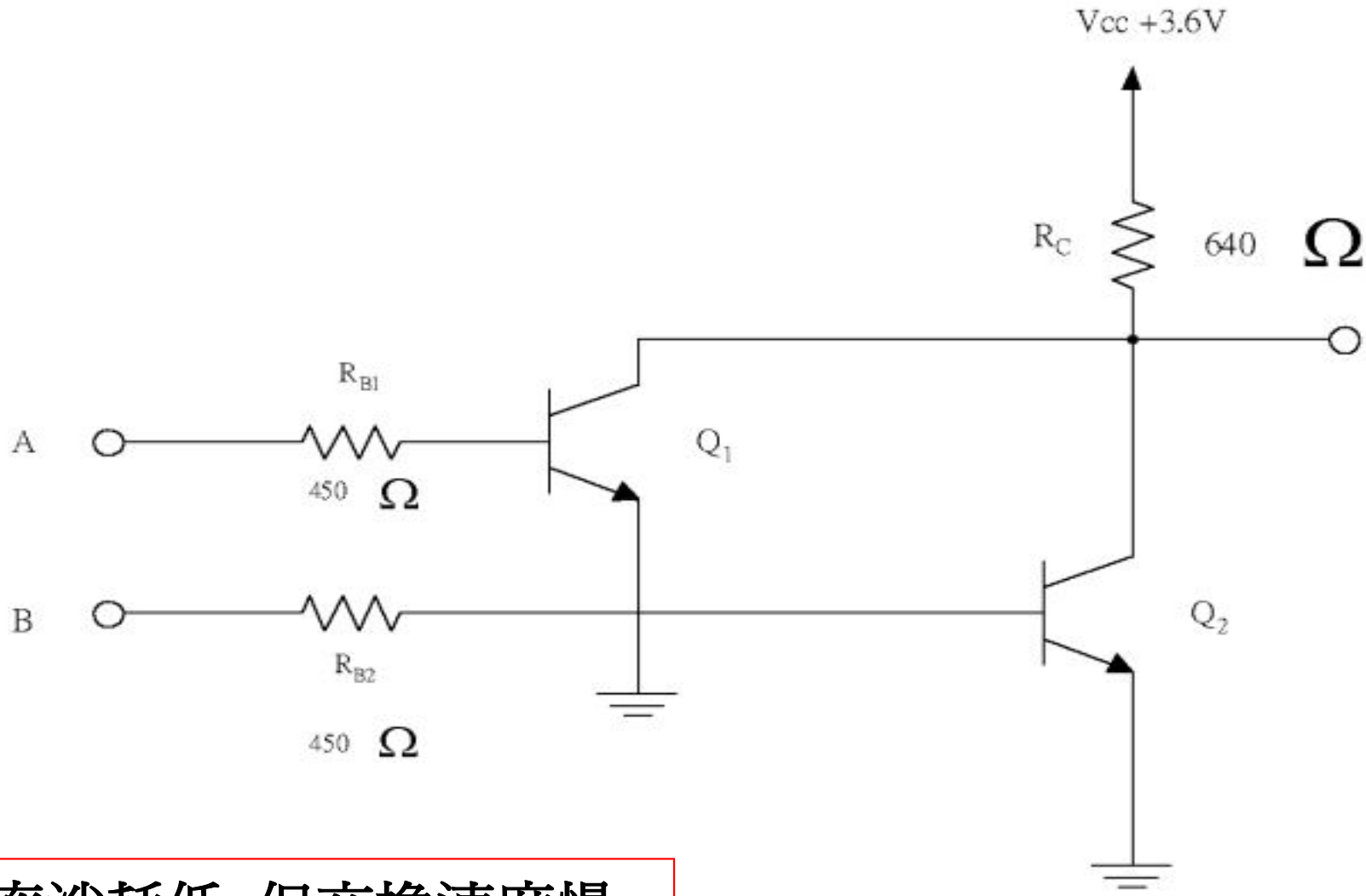
74LS83A datasheet

DCTL族NOR gate 基本電路



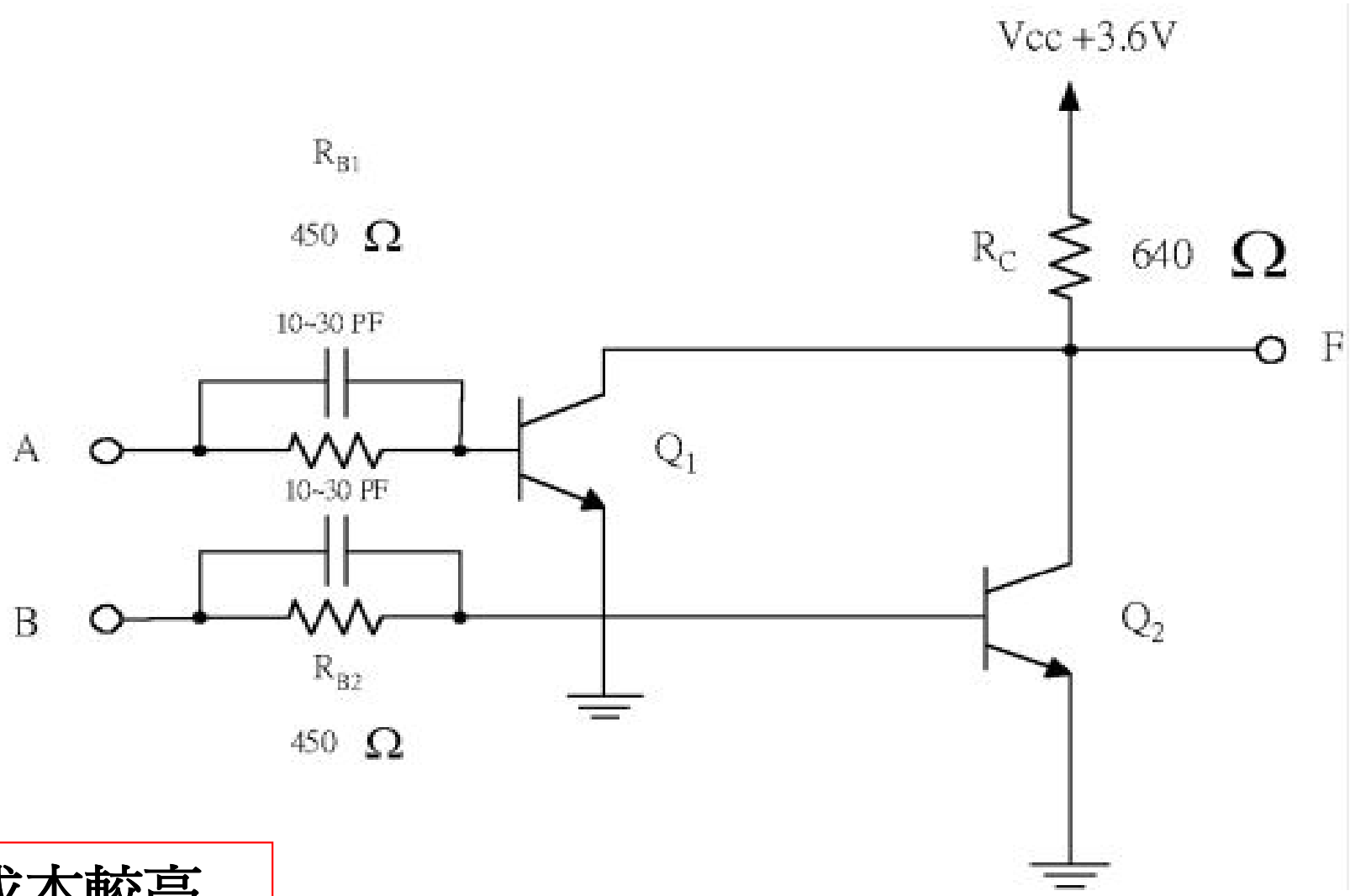
簡單, 但易受雜訊干擾

RTL族 NOR Gate 基本電路



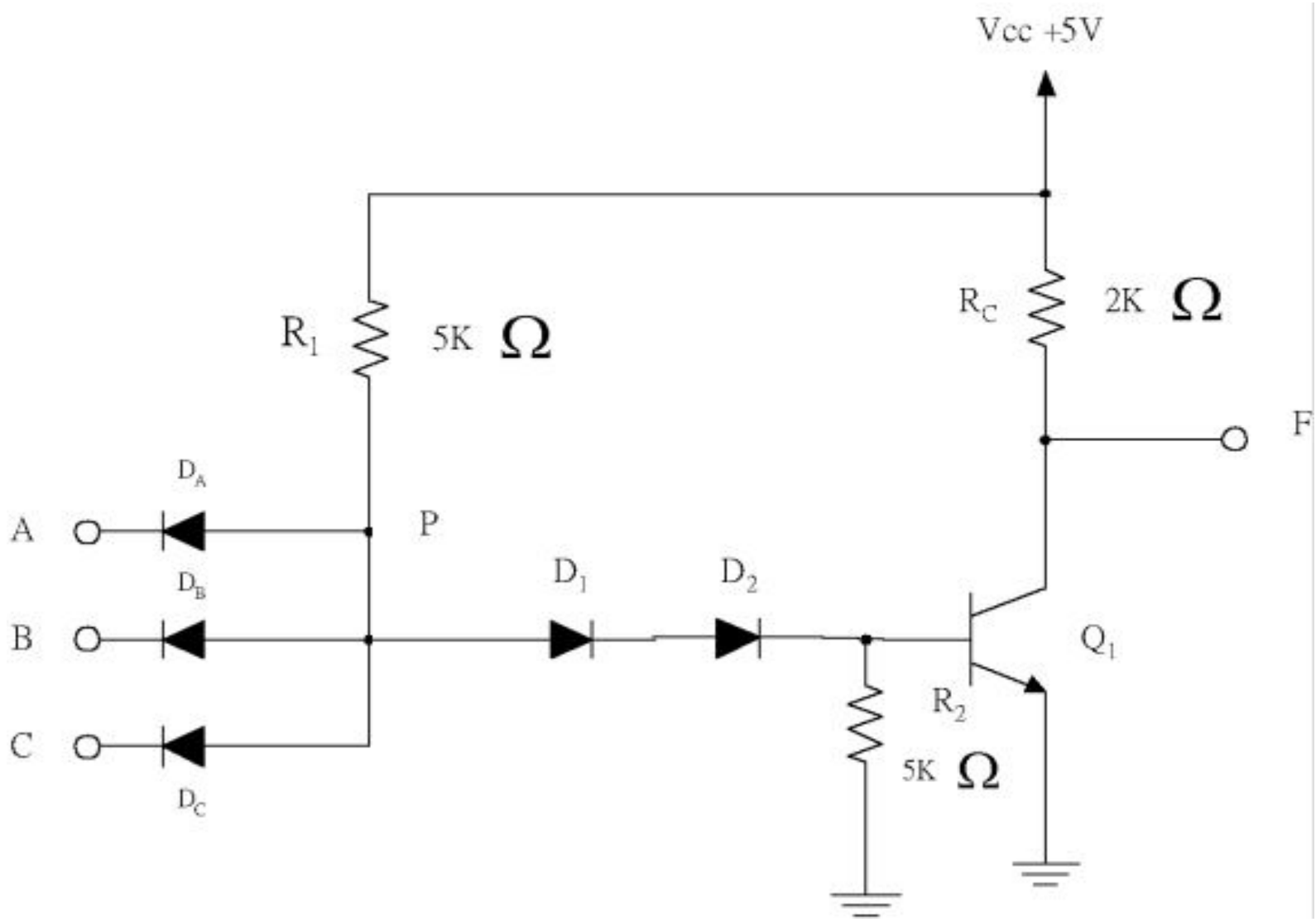
功率消耗低, 但交換速度慢

RCTL族 NOR Gate 基本電路

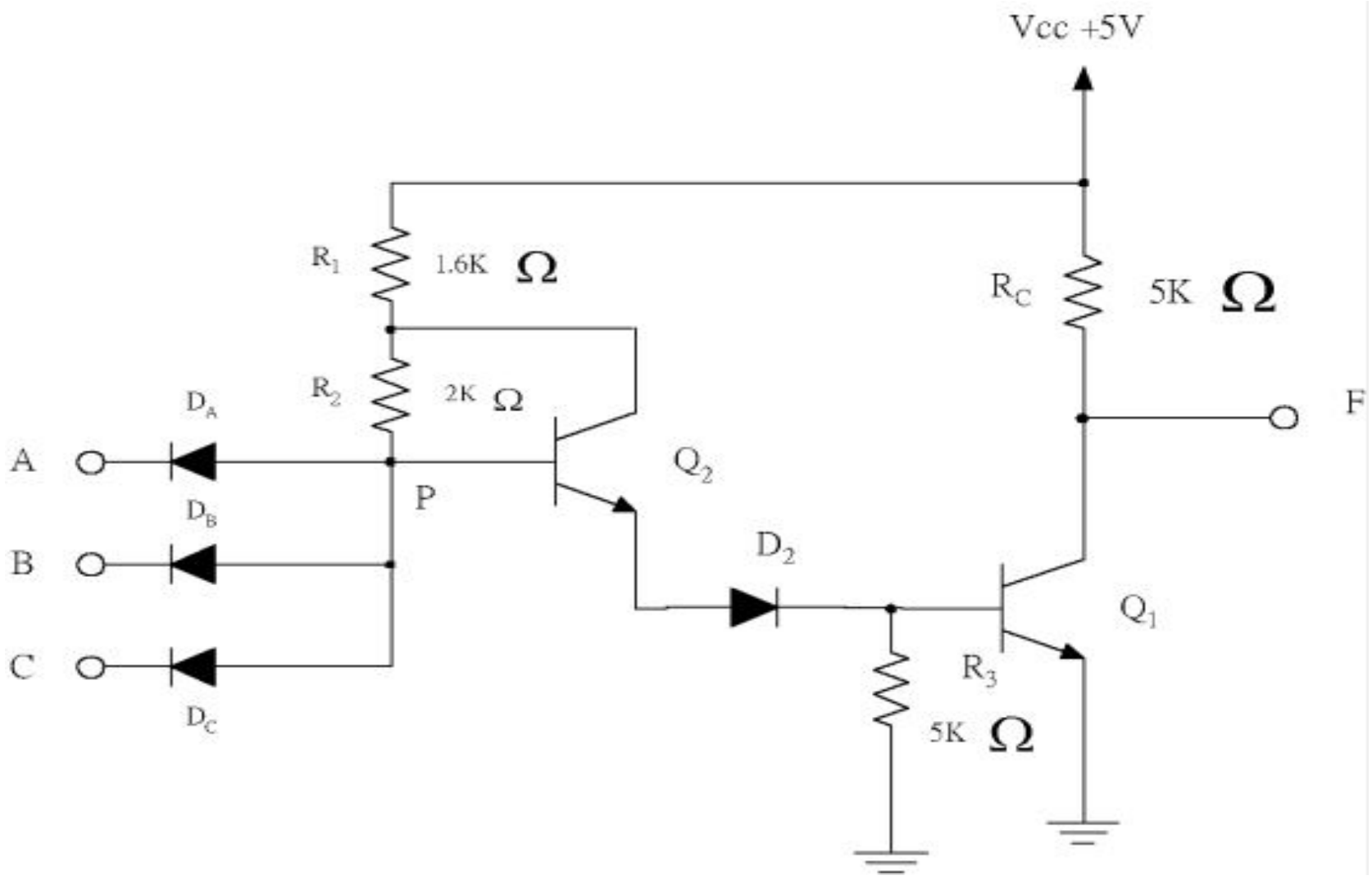


成本較高

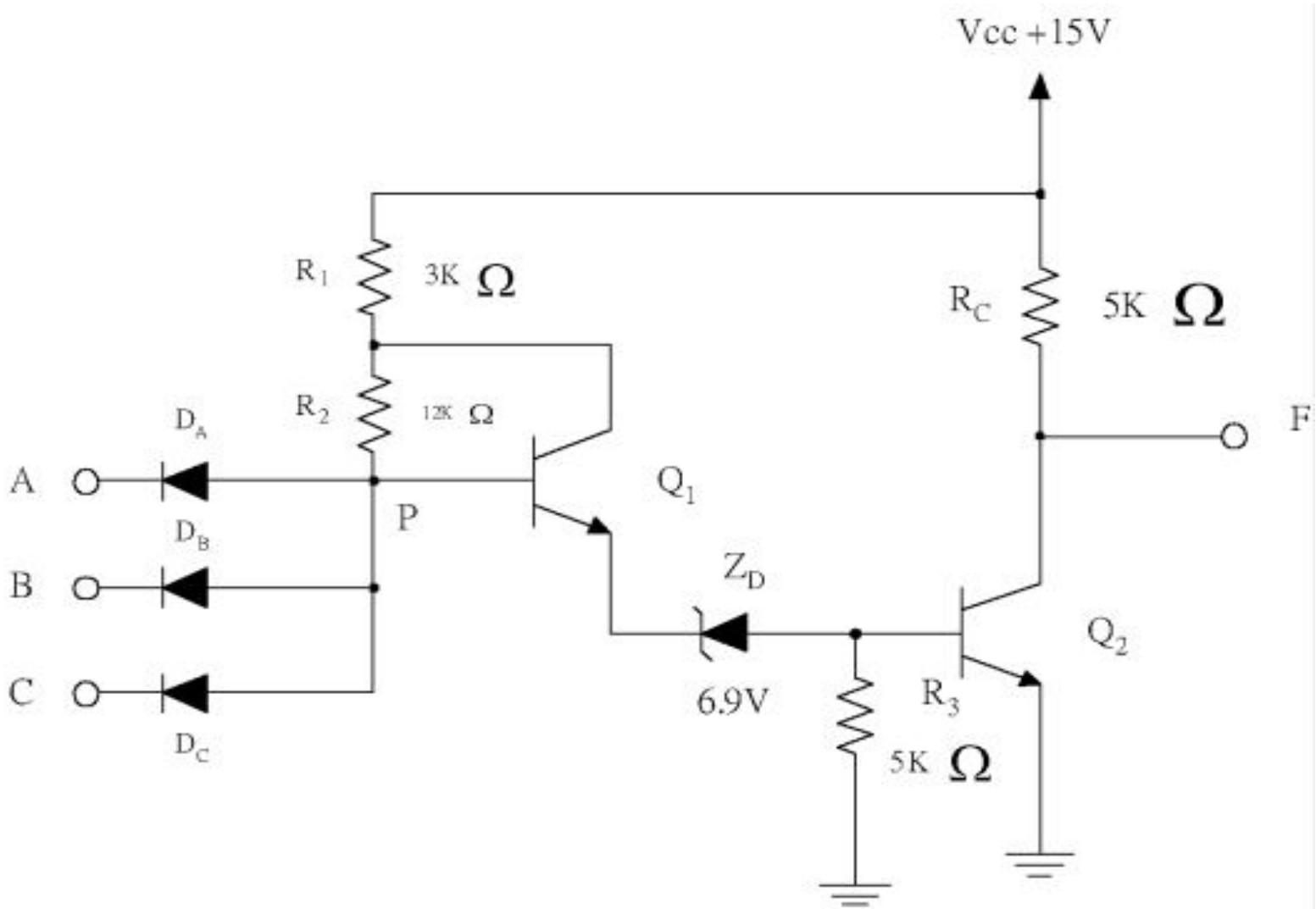
DTL族 NAND Gate 基本電路



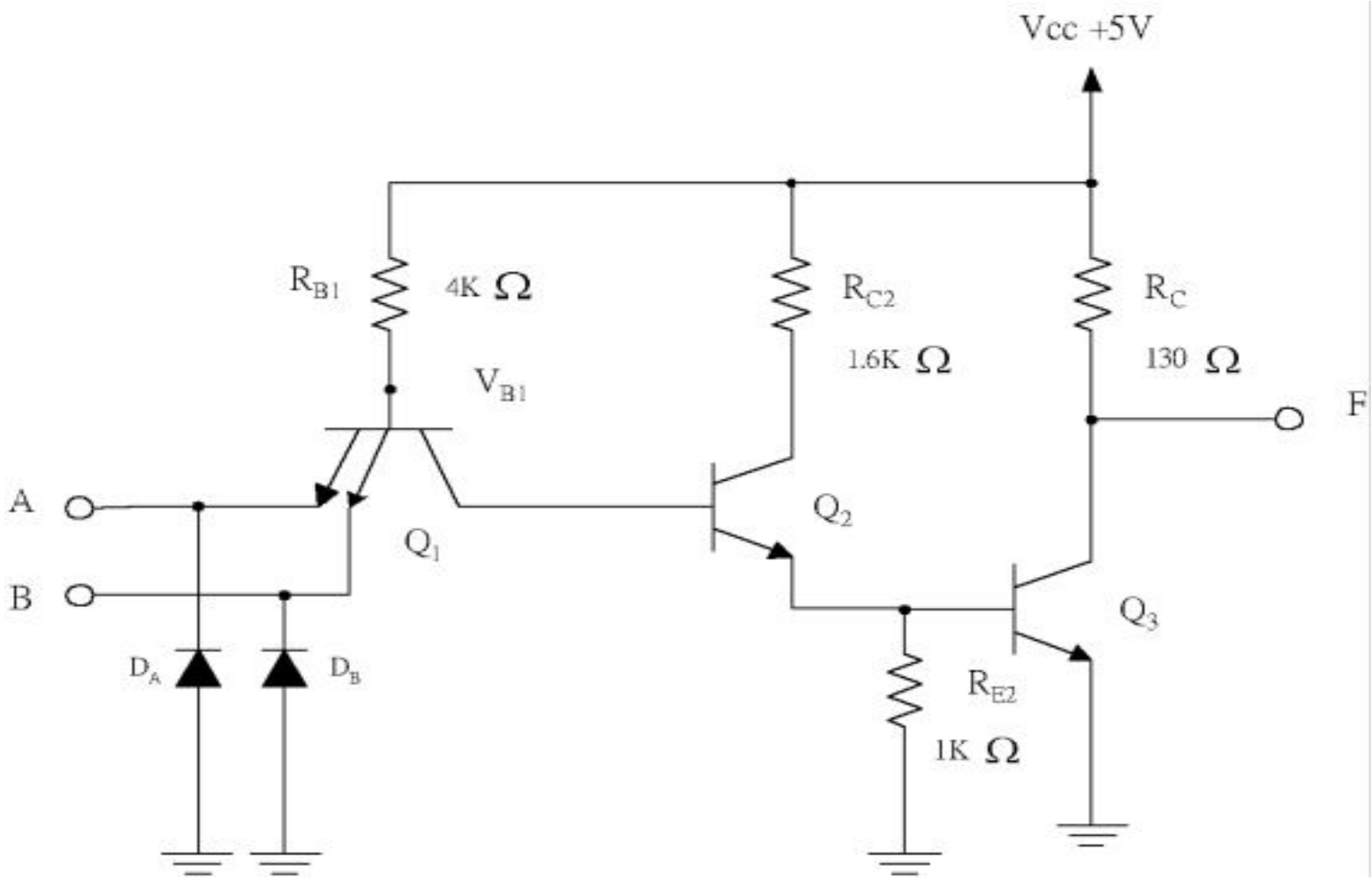
改良型 DTL族 NAND Gate 基本電路



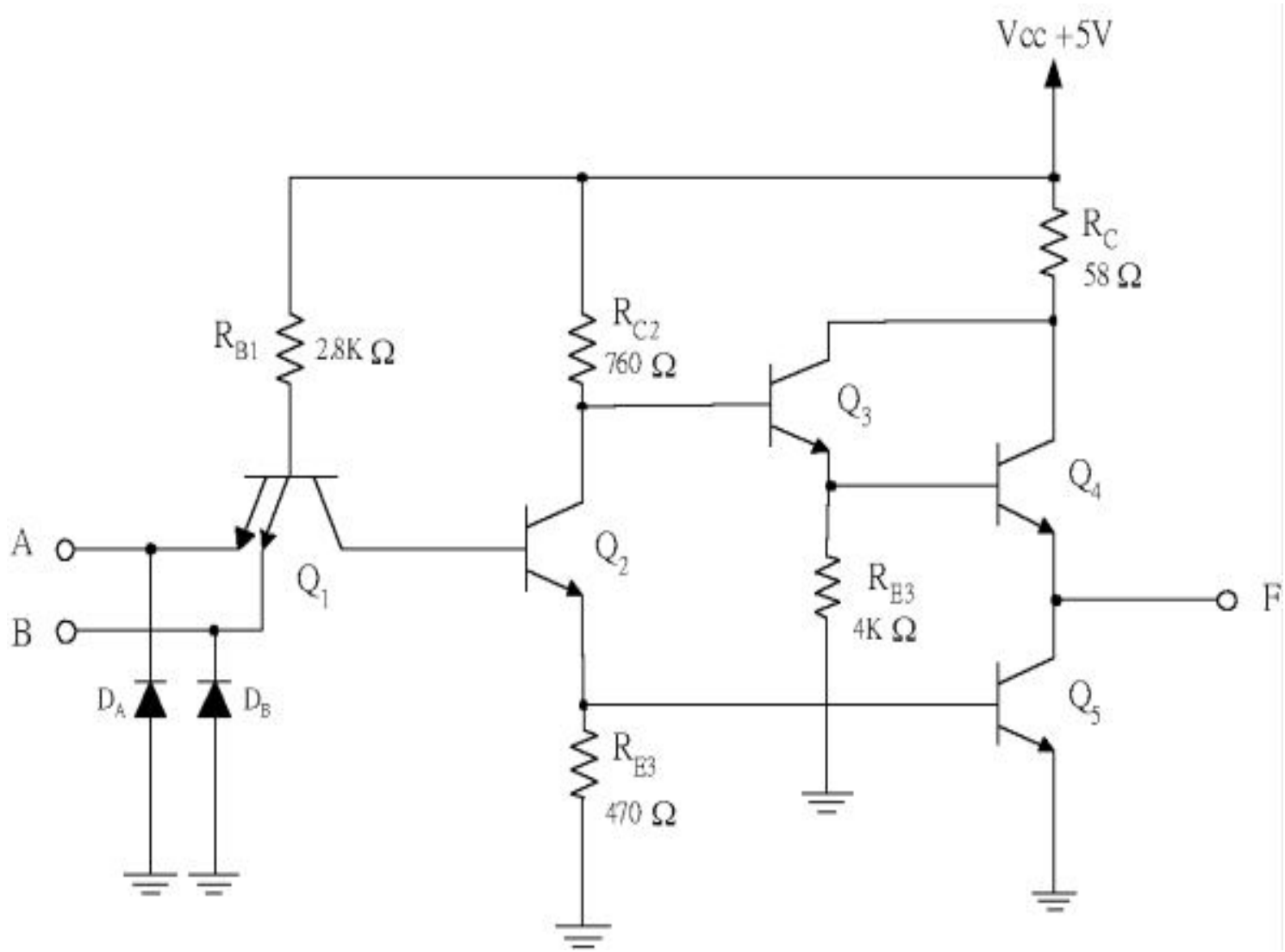
HTL族 NAND Gate 的基本電路



TTL族 NAND Gate基本電路



高速度 TTL NAND Gate 基本電路



TTL IC輸出端與輸入端邏輯電壓準位

	輸出端（簡稱 O）	輸入端（簡稱 I）
高電位（簡稱 H）	V_{OH} > 2.4 V（大於 2.4 伏特）	V_{IH} > 2.0 V（大於 2.0 伏特）
低電位（簡稱 L）	V_{OL} < 0.4 V（小於 0.4 伏特）	V_{IL} < 0.8 V（小於 0.8 伏特）

※ 說明：

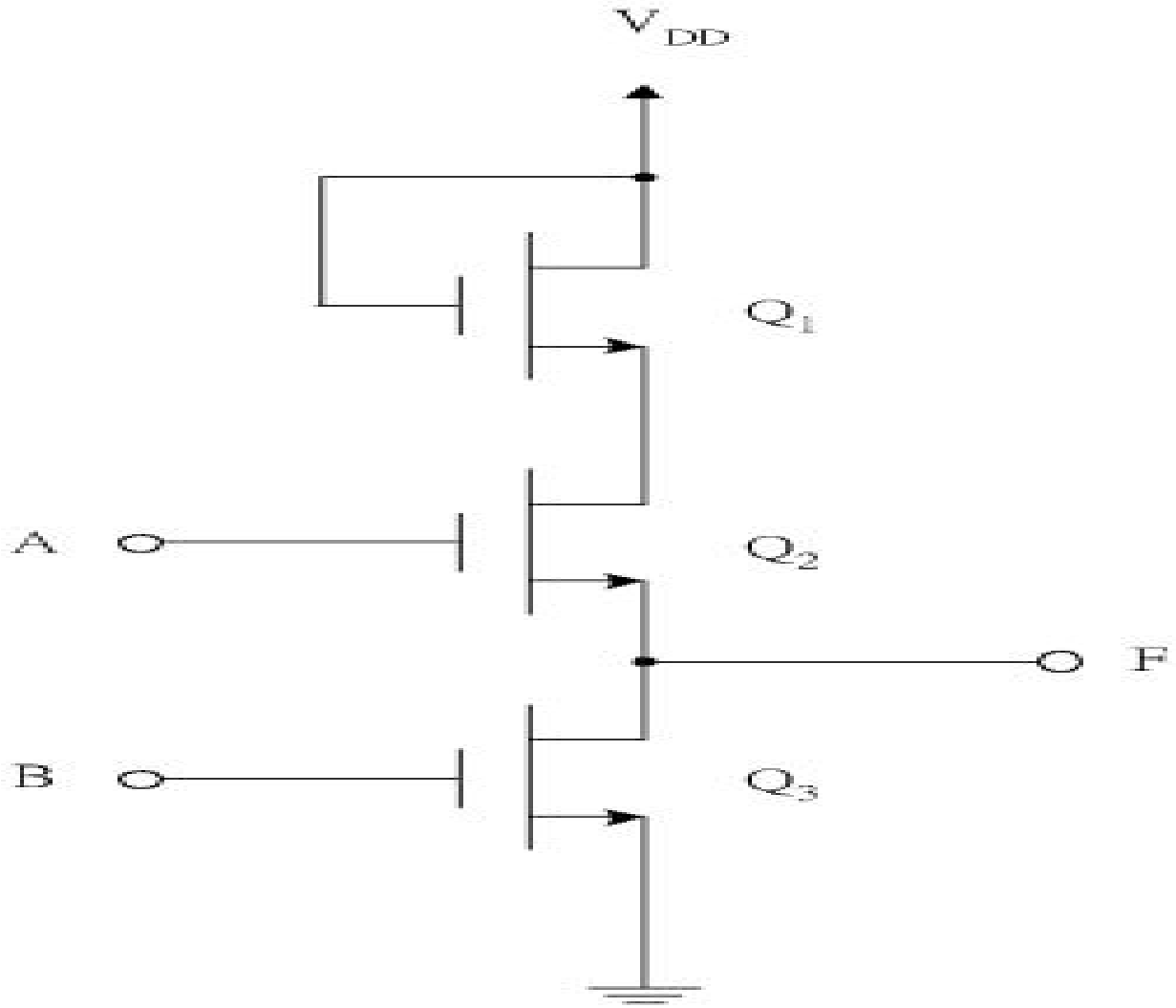
V_{OH} > 2.4 V：表輸出端爲高電位時的電壓需大於 2.4 伏特。

V_{OL} < 0.4 V：表輸出端爲低電位時的電壓需小於 0.4 伏特。

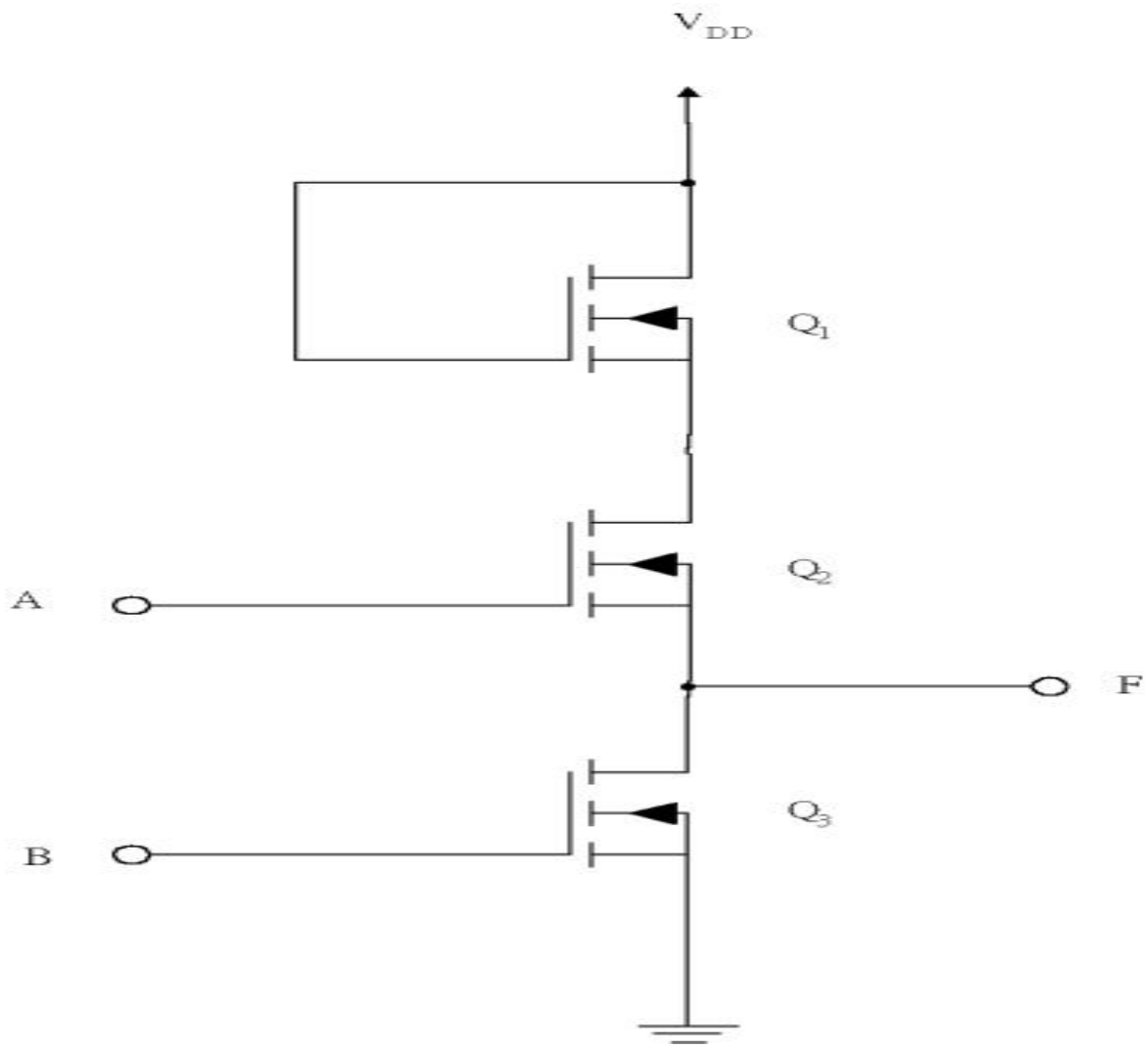
V_{IH} > 2.0 V：表輸入端爲高電位時的電壓需大於 2.0 伏特。

V_{IL} < 0.8 V：表輸入端爲低電位時的電壓需小於 0.8 伏特。

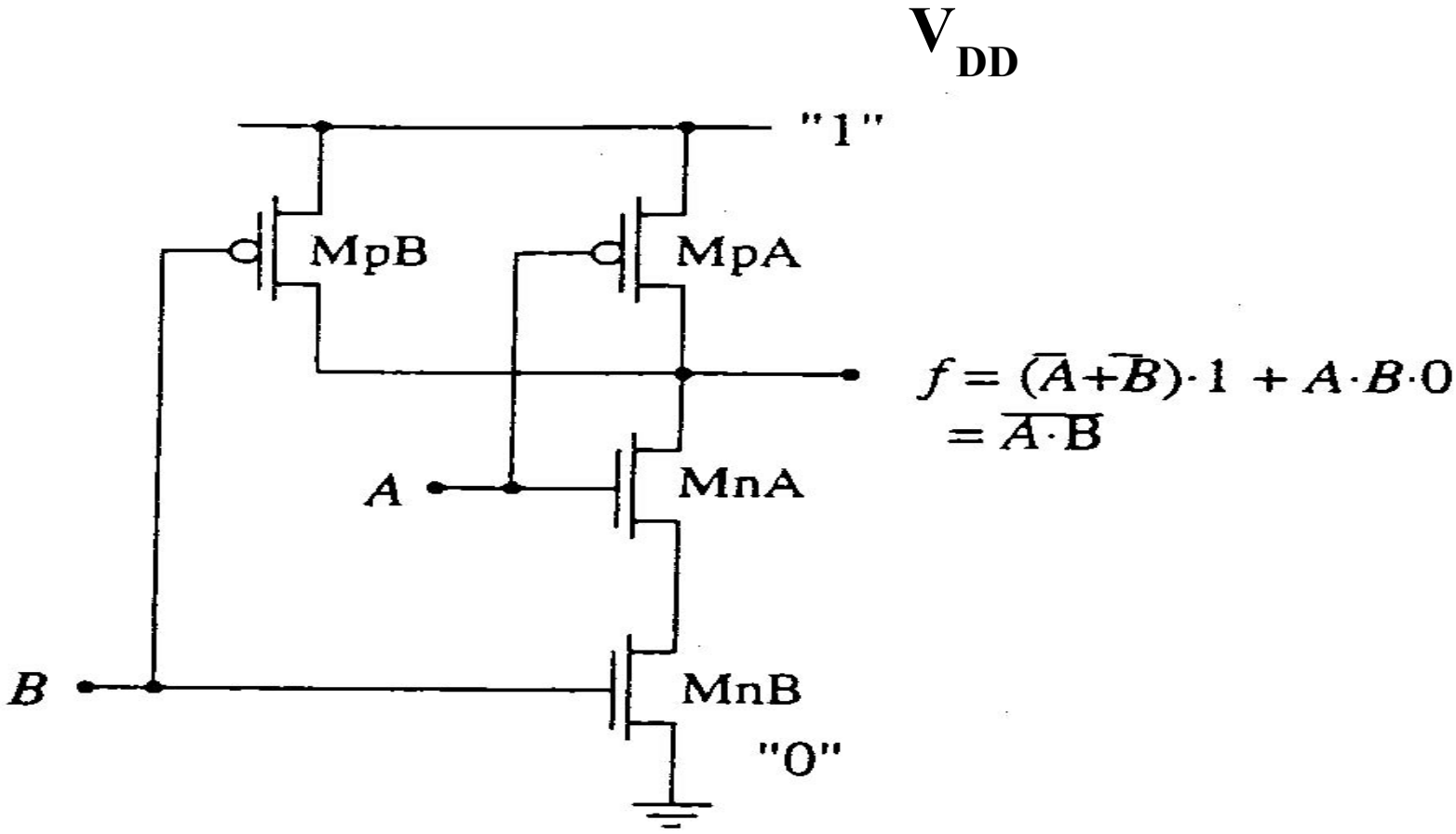
空乏型 NMOS NAND Gate (N通道)



增強型 NMOS NAND Gate



CMOS NAND Gate



Chapter 1 Data Storage

Q&A