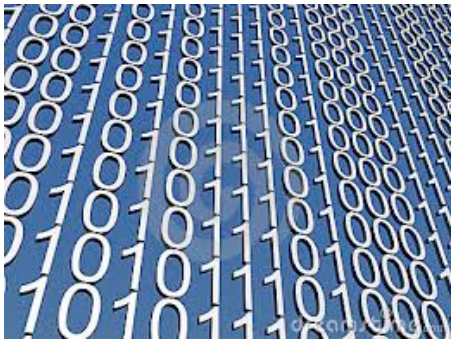
Muhammad Davut Hassan

Number Systems and Number Representation



For Your Amusement

Question: Why do computer programmers confuse Christmas and Halloween?

Answer: Because 25 Dec = 31 Oct

-- http://www.electronicsweekly.com

Goals of this Lecture

Help you learn (or refresh your memory) about:

- The binary, hexadecimal, and octal number systems
- Finite representation of unsigned integers
- Finite representation of signed integers
- Finite representation of rational numbers (if time)

Why?

- A power programmer must know number systems and data representation to fully understand C's **primitive data types**

Primitive values and
the operations on them

Agenda

Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers (if time)

The Decimal Number System

Name

- "decem" (Latin) => ten

Characteristics

- Ten symbols
  - `0 1 2 3 4 5 6 7 8 9`
- Positional
  - `2945 ≠ 2495`
  - $2945 = (2*10^3) + (9*10^2) + (4*10^1) + (5*10^0)$

(Most) people use the decimal number system

The Binary Number System

Name

- "binarius" (Latin) => two

Characteristics

- Two symbols
  - `0 1`
- Positional
  - $1010_B \neq 1100_B$

Most (digital) computers use the binary number system

Terminology

- **Bit**: a binary digit
- **Byte**: (typically) 8 bits

Decimal-Binary Equivalence

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Decimal	Binary
16	10000
17	10001
18	10010
19	10011
20	10100
21	10101
22	10110
23	10111
24	11000
25	11001
26	11010
27	11011
28	11100
29	11101
30	11110
31	11111
...	...

Decimal-Binary Conversion

Binary to decimal: expand using positional notation

$$100101_B = (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0)$$
$$= 32 + 0 + 0 + 4 + 0 + 1$$
$$= 37$$

Decimal-Binary Conversion

Decimal to binary: do the reverse

- Determine largest power of 2 ≤ number; write template

$$37 = (?*2^5)+(?*2^4)+(?*2^3)+(?*2^2)+(?*2^1)+(?*2^0)$$

- Fill in template

$$37 = (1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1)+(1*2^0)$$

$$\begin{array}{r} 37 \\ -32 \\ \hline 5 \\ -4 \\ \hline 1 \\ -1 \\ \hline 0 \end{array}$$

$$100101_B$$

Decimal-Binary Conversion

Decimal to binary shortcut

- Repeatedly divide by 2, consider remainder

```
37 / 2 = 18 R 1
18 / 2 =  9 R 0
 9 / 2 =  4 R 1
 4 / 2 =  2 R 0
 2 / 2 =  1 R 0
 1 / 2 =  0 R 1
```

Read from bottom to top: 100101_B

The Hexadecimal Number System

Name

- "hexa" (Greek) => six
- "decem" (Latin) => ten

Characteristics

- Sixteen symbols
  - **0 1 2 3 4 5 6 7 8 9 A B C D E F**
- Positional
  - $\mathbf{A13D_H \neq 3DA1_H}$

Computer programmers often use the hexadecimal number system

Decimal-Hexadecimal Equivalence

Decimal	Hex
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Decimal	Hex
16	10
17	11
18	12
19	13
20	14
21	15
22	16
23	17
24	18
25	19
26	1A
27	1B
28	1C
29	1D
30	1E
31	1F

Decimal	Hex
32	20
33	21
34	22
35	23
36	24
37	25
38	26
39	27
40	28
41	29
42	2A
43	2B
44	2C
45	2D
46	2E
47	2F
...	...

Decimal-Hexadecimal Conversion

Hexadecimal to decimal: expand using positional notation

$$25_H = (2*16^1) + (5*16^0)$$
$$= 32 + 5$$
$$= 37$$

Decimal to hexadecimal: use the shortcut

```
37 / 16 = 2 R 5
 2 / 16 = 0 R 2
```

Read from bottom to top: 25_H

Binary-Hexadecimal Conversion

Observation: $16^1 = 2^4$

- Every 1 hexadecimal digit corresponds to 4 binary digits

Binary to hexadecimal

$$1010000100111101_B$$
$$\quad A \quad 1 \quad 3 \quad D_H$$

Digit count in binary number not a multiple of 4 => pad with zeros on left

Hexadecimal to binary

$$\quad A \quad 1 \quad 3 \quad D_H$$
$$1010000100111101_B$$

Discard leading zeros from binary number if appropriate

Is it clear why programmers often use hexadecimal?

The Octal Number System

Name

- "octo" (Latin) => eight

Characteristics

- Eight symbols
  - **0 1 2 3 4 5 6 7**
- Positional
  - $1743_o \neq 7314_o$

Computer programmers often use the octal number system

Decimal-Octal Equivalence

Decimal	Octal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	11
10	12
11	13
12	14
13	15
14	16
15	17

Decimal	Octal
16	20
17	21
18	22
19	23
20	24
21	25
22	26
23	27
24	30
25	31
26	32
27	33
28	34
29	35
30	36
31	37

Decimal	Octal
32	40
33	41
34	42
35	43
36	44
37	45
38	46
39	47
40	50
41	51
42	52
43	53
44	54
45	55
46	56
47	57
...	...

Decimal-Octal Conversion

Octal to decimal: expand using positional notation

$37_O = (3*8^1) + (7*8^0)$
$= 24 + 7$
$= 31$

Decimal to octal: use the shortcut

```
31 / 8 = 3 R 7
 3 / 8 = 0 R 3
```

Read from bottom to top: 37_O

Binary-Octal Conversion

Observation: $8^1 = 2^3$

- Every 1 octal digit corresponds to 3 binary digits

Binary to octal

```
001010000100111101B
 1   2   0   4   7   5O
```

Digit count in binary number not a multiple of 3 => pad with zeros on left

Octal to binary

```
 1   2   0   4   7   5O
001010000100111101B
```

Discard leading zeros from binary number if appropriate

Is it clear why programmers sometimes use octal?

Agenda

Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers (if time)

Unsigned Data Types: Java vs. C

Java has type

- **`int`**
  - Can represent signed integers

C has type:

- **`signed int`**
  - Can represent signed integers
- **`int`**
  - Same as **`signed int`**
- **`unsigned int`**
  - Can represent only unsigned integers

To understand C, must consider representation of both unsigned and signed integers

Representing Unsigned Integers

Mathematics

- Range is 0 to ∞

Computer programming

- Range limited by computer's **word** size
- Word size is n bits => range is 0 to $2^n - 1$
- Exceed range => **overflow**

Nobel computers with gcc217

- n = 32, so range is 0 to $2^{32} - 1$ (4,294,967,295)

Pretend computer

- n = 4, so range is 0 to $2^4 - 1$ (15)

Hereafter, assume word size = 4

- All points generalize to word size = 32, word size = n

Representing Unsigned Integers

On pretend computer

Unsigned Integer	Rep
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Adding Unsigned Integers

Addition

```
          1
   3      0011B
+ 10    + 1010B
  --      ----
  13      1101B
```

Start at right column
Proceed leftward
Carry 1 when necessary

```
         11
   7      0111B
+ 10    + 1010B
  --      ----
   1     10001B
```

Beware of overflow

How would you detect overflow programmatically?

Results are mod 2^4

Subtracting Unsigned Integers

Subtraction

```
          12
        0202
  10    1010B
-  7  - 0111B
  --    ----
   3    0011B
```

Start at right column
Proceed leftward
Borrow 2 when necessary

```
          2
   3    0011B
- 10  - 1010B
  --    ----
   9    1001B
```

Beware of overflow

How would you detect overflow programmatically?

Results are mod 2^4

Shifting Unsigned Integers

Bitwise right shift (>> in C): fill on left with zeros

```
10 >> 1 => 5
```

1010_B 0101_B

```
10 >> 2 => 2
```

1010_B 0010_B

What is the effect arithmetically? (No fair looking ahead)

Bitwise left shift (<< in C): fill on right with zeros

```
5 << 1 => 10
```

0101_B 1010_B

```
3 << 2 => 12
```

0011_B 1100_B

What is the effect arithmetically? (No fair looking ahead)

Results are mod 2^4

Other Operations on Unsigned Ints

Bitwise NOT (~ in C)

- Flip each bit

```
~10 => 5
```

1010_B 0101_B

Bitwise AND (& in C)

- Logical AND corresponding bits

```
 10        1010B
& 7      & 0111B
 --        ----
  2        0010B
```

Useful for setting selected bits to 0

Other Operations on Unsigned Ints

Bitwise OR: (| in C)

- Logical OR corresponding bits

```
  10        1010B
| 1       | 0001B
  --        ----
  11        1011B
```

Useful for setting selected bits to 1

Bitwise exclusive OR (^ in C)

- Logical exclusive OR corresponding bits

```
  10        1010B
^ 10      ^ 1010B
  --        ----
   0        0000B
```

x ^ x sets all bits to 0

Aside: Using Bitwise Ops for Arith

Can use <<, >>, and & to do some arithmetic efficiently

$x * 2^y == x << y$

- $3*4 = 3*2^2 = 3<<2 \Rightarrow 12$

$x / 2^y == x >> y$

- $13/4 = 13/2^2 = 13>>2 \Rightarrow 3$

$x \% 2^y == x \& (2^y-1)$

- $13\%4 = 13\%2^2 = 13\&(2^2-1)$
  $= 13\&3 \Rightarrow 1$

13	1101_B
& 3	& 0011_B
--	----
1	0001_B

Fast way to **multiply** by a power of 2

Fast way to **divide** by a power of 2

Fast way to **mod** by a power of 2

Aside: Example C Program

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{  unsigned int n;
   unsigned int count;
   printf("Enter an unsigned integer: ");
   if (scanf("%u", &n) != 1)
   {  fprintf(stderr, "Error: Expect unsigned int.\n");
      exit(EXIT_FAILURE);
   }
   for (count = 0; n > 0; n = n >> 1)
       count += (n & 1);
   printf("%u\n", count);
   return 0;
}
```

What does it write?

How could this be expressed more succinctly?

Agenda

Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers (if time)

Signed Magnitude

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

Definition

High-order bit indicates sign

0 => positive

1 => negative

Remaining bits indicate magnitude

$1101_B = -101_B = -5$

$0101_B = 101_B = 5$

Signed Magnitude (cont.)

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

Computing negative

neg(x) = flip high order bit of x

$\mathbf{neg}(0101_B) = 1101_B$

$\mathbf{neg}(1101_B) = 0101_B$

Pros and cons

+ easy for people to understand
+ symmetric
- two reps of zero

Ones' Complement

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

Definition

High-order bit has weight -7

1010_B = (1*-7)+(0*4)+(1*2)+(0*1)
= -5

0010_B = (0*-7)+(0*4)+(1*2)+(0*1)
= 2

Ones' Complement (cont.)

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

Computing negative

neg(x) = ~x

$\mathbf{neg}(0101_B) = 1010_B$

$\mathbf{neg}(1010_B) = 0101_B$

Computing negative (alternative)

neg(x) = 1111_B - x

$\mathbf{neg}(0101_B) = 1111_B - 0101_B$

$= 1010_B$

$\mathbf{neg}(1010_B) = 1111_B - 1010_B$

$= 0101_B$

Pros and cons

+ symmetric

- two reps of zero

Two's Complement

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

Definition

High-order bit has weight -8

1010_B = (1*-8)+(0*4)+(1*2)+(0*1)
= -6

0010_B = (0*-8)+(0*4)+(1*2)+(0*1)
= 2

Two's Complement (cont.)

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

Computing negative

neg(x) = ~x + 1

neg(x) = onescomp(x) + 1

$\mathbf{neg}(0101_B) = 1010_B + 1 = 1011_B$

$\mathbf{neg}(1011_B) = 0100_B + 1 = 0101_B$

Pros and cons

- not symmetric

+ one rep of zero

Two's Complement (cont.)

Almost all computers use two's complement to represent signed integers

Why?

- Arithmetic is easy
  - Will become clear soon

Hereafter, assume two's complement representation of signed integers

Adding Signed Integers

pos + pos

```
          11
   3      0011B
 + 3    + 0011B
  --      ----
   6      0110B
```

pos + pos (overflow)

```
          111
   7      0111B
 + 1    + 0001B
  --      ----
  -8      1000B
```

pos + neg

```
         1111
   3      0011B
+ -1    + 1111B
  --      ----
   2     10010B
```

How would you detect overflow programmatically?

neg + neg

```
          11
  -3      1101B
+ -2    + 1110B
  --      ----
  -5     11011B
```

neg + neg (overflow)

```
         1 1
  -6      1010B
+ -5    + 1011B
  --      ----
   5     10101B
```

Subtracting Signed Integers

Perform subtraction with borrows

or

Compute two's comp and add

```
          1
          22
  3       0011B
- 4     - 0100B
 --       ----
 -1       1111B
```

```
   3       0011B
+ -4     + 1100B
  --       ----
  -1       1111B
```

```
 -5       1011B
- 2     - 0010B
 --       ----
 -7       1001B
```

```
          111
  -5       1011
+ -2     + 1110
  --       ----
  -7      11001
```

Negating Signed Ints: Math

Question: Why does two's comp arithmetic work?

Answer: $[-b] \bmod 2^4 = [\text{twoscomp}(b)] \bmod 2^4$

$$
\begin{aligned}
&[-b] \bmod 2^4 \\
&= [2^4 - b] \bmod 2^4 \\
&= [2^4 - 1 - b + 1] \bmod 2^4 \\
&= [(2^4 - 1 - b) + 1] \bmod 2^4 \\
&= [\text{onescomp}(b) + 1] \bmod 2^4 \\
&= [\text{twoscomp}(b)] \bmod 2^4
\end{aligned}
$$

See Bryant & O'Hallaron book for much more info

Subtracting Signed Ints: Math

And so:

$$[a - b] \bmod 2^4 = [a + \text{twoscomp}(b)] \bmod 2^4$$

$$
\begin{aligned}
&[a - b] \bmod 2^4 \\
&= [a + 2^4 - b] \bmod 2^4 \\
&= [a + 2^4 - 1 - b + 1] \bmod 2^4 \\
&= [a + (2^4 - 1 - b) + 1] \bmod 2^4 \\
&= [a + \text{onescomp}(b) + 1] \bmod 2^4 \\
&= [a + \text{twoscomp}(b)] \bmod 2^4
\end{aligned}
$$

See Bryant & O'Hallaron book for much more info

Shifting Signed Integers

Bitwise left shift (<< in C): fill on right with zeros

```
3 << 1 => 6
```

0011_B 0110_B

```
-3 << 1 => -6
```

1101_B -1010_B

What is the effect arithmetically?

Bitwise **arithmetic** right shift: fill on left **with sign bit**

```
6 >> 1 => 3
```

0110_B 0011_B

```
-6 >> 1 => -3
```

1010_B 1101_B

What is the effect arithmetically?

Results are mod 2^4

Shifting Signed Integers (cont.)

Bitwise **logical** right shift: fill on left **with zeros**

```
6 >> 1 => 3
```

0110_B 0011_B

```
-6 >> 1 => 5
```

1010_B 0101_B

What is the effect arithmetically**???**

In C, right shift (>>) could be logical or arithmetic

- Not specified by C90 standard
- Compiler designer decides

Best to avoid shifting signed integers

Other Operations on Signed Ints

Bitwise NOT (~ in C)

- Same as with unsigned ints

Bitwise AND (& in C)

- Same as with unsigned ints

Bitwise OR: (| in C)

- Same as with unsigned ints

Bitwise exclusive OR (^ in C)

- Same as with unsigned ints

Best to avoid with signed integers

Agenda

Number Systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers (if time)

Rational Numbers

Mathematics

- A **rational** number is one that can be expressed as the **ratio** of two integers
- Infinite range and precision

Compute science

- Finite range and precision
- Approximate using **floating point** number
  - Binary point “floats” across bits

IEEE Floating Point Representation

Common finite representation: **IEEE floating point**

- More precisely: ISO/IEEE 754 standard

Using 32 bits (type float in C):

- 1 bit: sign (0=>positive, 1=>negative)
- 8 bits: exponent + 127
- 23 bits: binary fraction of the form 1.*ddddddddddddddddddddddd*

Using 64 bits (type double in C):

- 1 bit: sign (0=>positive, 1=>negative)
- 11 bits: exponent + 1023
- 52 bits: binary fraction of the form 1.*dddddddddddddddddddddddddddddddddddddddddddddddddddddd*

Floating Point Example

11000001110110110000000000000000

32-bit representation

Sign (1 bit):

- 1 => negative

Exponent (8 bits):

- $10000011_B = 131$
- $131 - 127 = 4$

Fraction (23 bits):

- $1.10110110000000000000000_B$
- $1 + (1*2^{-1})+(0*2^{-2})+(1*2^{-3})+(1*2^{-4})+(0*2^{-5})+(1*2^{-6})+(1*2^{-7}) = 1.7109375$

Number:

- $-1.7109375 * 2^4 = -27.375$

Floating Point Warning

Decimal number system can represent only some rational numbers with finite digit count

- Example: 1/3

Binary number system can represent only some rational numbers with finite digit count

- Example: 1/5

Beware of **roundoff error**

- Error resulting from inexact representation
- Can accumulate

Decimal Approx	Rational Value
.3	3/10
.33	33/100
.333	333/1000
...	

Binary Approx	Rational Value
0.0	0/2
0.01	1/4
0.010	2/8
0.0011	3/16
0.00110	6/32
0.001101	13/64
0.0011010	26/128
0.00110011	51/256
...	

Summary

The binary, hexadecimal, and octal number systems

Finite representation of unsigned integers

Finite representation of signed integers

Finite representation of rational numbers

Essential for proper understanding of

- C primitive data types
- Assembly language
- Machine language

Number Systems- Binary System

Muhammad Davut Hassan

Number System

- Number
  - It is a symbol representing a unit or quantity.
- Number System
  - Defines a set of symbols used to represent quantity
- Radix
  - The base or radix of number system determines how many numerical digits the number system uses.

Types of Number System

- Decimal System
- Binary Number System
- Octal Number System
- Hexadecimal Number System

Decimal Number System

- Ingenious method of expressing all numbers by means of tens symbols originated from India. It is widely used and is based on the ten fingers of a human being.
- It makes use of ten numeric symbols
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Inherent Value and Positional Value

- The inherent value of a symbol is the value of that symbol standing alone.
  - Example 6 in number 256, 165, 698
    - The symbol is related to the quantity six, even if it is used in different number positions
- The positional value of a numeric symbol is directly related to the base of a system.
  - In the case of decimal system, each position has a value of 10 times greater that the position to its right. Example: 423, the symbol 3 represents the ones (units), the symbol 2 represents the tens position (10 x 1), and the symbol 4 represents the hundreds position (10 x 10). In other words, each symbol move to the left represents an increase in the value of the position by a factor of ten.

Inherent and Positional Value cont.

$2539 = 2X1000 + 5X100 + 3X10 + 9X1$

$= 2X10^3 + 5X10^2 + 3X10^1 + 9 x10^0$

This means that positional value of symbol 2 is 1000 or using the base 10 it is 10^3

Binary Number System

- Uses only two numeric symbols 1 and 0
  - Under the binary system, each position has a value 2 times greater than the position to the right.

Octal Number System

- Octal number system is using 8 digits to represent numbers. The highest value = 7. Each column represents a power of 8. Octal numbers are represented with the suffix 8.

Hexadecimal Number System

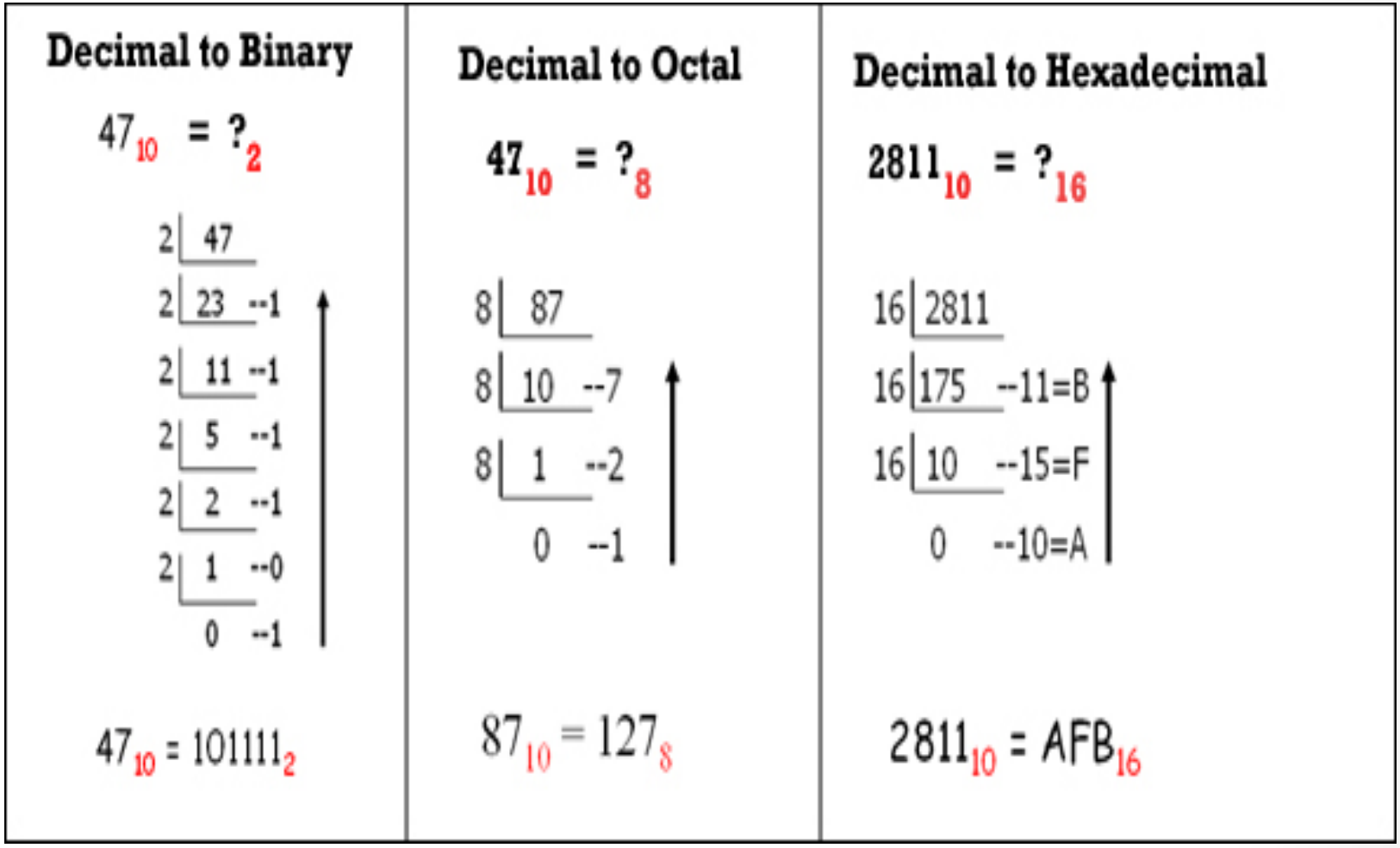
- Provides another convenient and simple method for expressing values represented by binary numerals.
- It uses a base, or radix, of 16 and the place values are the powers of 16.

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

Radix Conversion

- The process of converting a base to another.
- To convert a decimal number to any other number system, divide the decimal number by the base of the destination number system. Repeat the process until the quotient becomes zero. And note down the remainders in the reverse order.
- To convert from any other number system to decimal, take the positional value, multiply by the digit and add.

Radix Conversion



Radix Conversion

Binary to Decimal	Octal to Decimal	Hexa to Decimal
$1011_2 = ?_{10}$	$342_8 = ?_{10}$	$B2_{16} = ?_{10}$
$1x2^3 + 0x2^2 + 1x2^1 + 1x2^0$	$3x8^2 + 4x8^1 + 2x8^0$	$Bx16^1 + 2x16^0$
$=1x8 + 0x4 + 1x2 + 1x1$	$=3x64 + 4x8 + 2x1$	$=11x16 + 2x1$
$=8 + 0 + 2 + 1$	$=192 + 32 + 2$	$=176 + 2$
$=11$ (base 10)	$=226$ (base 10)	$=178$ (base 10)
$1011_2 = 11_{10}$	$342_8 = 226_{10}$	$B2_{16} = 178_{10}$

Decimal to Binary Conversion of Fractions

- Division – Multiplication Method
  - Steps to be followed
    - Multiply the decimal fraction by 2 and noting the integral part of the product
    - Continue to multiply by 2 as long as the resulting product is not equal to zero.
    - When the obtained product is equal to zero, the binary of the number consists of the integral part listed from top to bottom in the order they were recorded.

- Example 1: Convert 0.375 to its binary equivalent

Multiplication	Product	Integral part
0.375 x 2	0.75	0
0.75 x 2	1.5	1
0.5 x 2	1.0	1

0.375_{10} is equivalent to 0.011_2

Exercises

- Convert the following *decimal* numbers into *binary* and *hexadecimal* numbers:
  1. 128
  2. 207
- Convert the following *binary* numbers into *decimal* and *hexadecimal* numbers:
  1. 11111000
  2. 1110110

Exercises

- Convert the number in binary (110110) into octal and hex format.
  - In octal (base 8)
  - In Hexadecimal (base 16)
- Convert the number in binary (1110110) into octal and hex format.
  - In octal (base 8)
  - In Hexadecimal (base 16)

Exercises

- Convert decimal 12.75 to binary representation

- Convert binary number 1010.0011 into decimal representation

Fast Conversion
Binary to Power of 2 Base

- If you have a binary number to be converted into base which is power of 2,
  - Split the number in a group beginning from the right by the factor of power «n» (2^n)
  - Then convert the binary group directly to the power of 2 base

- Example
- $(100110010)_2 = (\ldots\ldots)_8$
- $(1100110)_2 = (\ldots\ldots)_8$

Fast Conversion
Binary to Power of 2 Base

- Examples

- $(10110010)_2 = (\ldots\ldots)_{16}$
- $(1100110)_2 = (\ldots\ldots)_{16}$

Fast Conversion
Power of 2 Base to Binary

- If you have a number, which is a power of 2, to be converted into base two,
  - Split each digit of the number,
  - Then convert eah digit directly to binary number with n digits
    - Where n is the power factor

- Examples
- $(53227)_8 = (\ldots\ldots)_2$
- $(125)_8 = (\ldots\ldots)_2$
- $(AD2)_{16} = (\ldots\ldots)_2$
- $(C3)_{16} = (\ldots\ldots)_2$
-

What about Octal to Hex Conversion

- Examples
  - $(125)_8 = (\ldots\ldots)_{16}$
  - $(125)_{16} = (\ldots\ldots)_8$

THANKS FOR YOUR ATTENTION!

Muhammad Davut Hassan

Muhammad Davut Hassan

CHAPTER 3

Numbers and Numeral Systems

Numbers play an important role in almost all areas of mathematics, not least in calculus. Virtually all calculus books contain a thorough description of the natural, rational, real and complex numbers, so we will not repeat this here. An important concern for us, however, is to understand the basic principles behind how a computer handles numbers and performs arithmetic, and for this we need to consider some facts about numbers that are usually not found in traditional calculus texts.

More specifically, we are going to review the basics of the decimal numeral system, where the base is 10, and see how numbers may be represented equally well in other numeral systems where the base is not 10. We will study representation of real numbers as well as arithmetic in different bases. Throughout the chapter we will pay special attention to the binary number system (base 2) as this is what is used in most computers. This will be studied in more detail in the next chapter.

3.1 Terminology and Notation

We will usually introduce terminology as it is needed, but certain terms need to be agreed upon straightaway. In your calculus book you will have learnt about natural, rational and real numbers. The natural numbers $\mathbb{N}_0 = \{0,1,2,3,4,\ldots\}$[1] are the most basic numbers in that both rational and real numbers can be constructed from them. Any positive natural number n has an opposite number

[1] In most books the natural numbers start with 1, but for our purposes it is convenient to include 0 as a natural number as well. To avoid confusion we have therefore added 0 as a subscript.

$-n$, and we denote by $\mathbb{Z}$ the set of natural numbers augmented with all these negative numbers,

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}.$$

We will refer to $\mathbb{Z}$ as the set of integer numbers or just the integers.

Intuitively it is convenient to think of a real number x as a decimal number with (possibly) infinitely many digits to the right of the decimal point. We then refer to the number obtained by setting all the digits to the right of the decimal point to 0 as the *integer part* of x. If we replace the integer part by 0 we obtain the *fractional part* of x. If for example $x = 3.14$, its integer part is 3 and its fractional part is 0.14. A number that has no integer part will often be referred to as a fractional number.

Definition 3.1. *Let $x = d_n d_{n-1} \cdots d_2 d_1 . d_0 d_{-1} d_{-2} \cdots$ be a decimal number. Then the number $d_n d_{n-1} \cdots d_2 d_1$ is called the integer part of x while the number $0.d_{-1} d_{-2} \cdots$ is called the fractional part of x.*

For rational numbers there are standard operations we can perform to find the integer and fractional parts. When two positive natural numbers a and b are divided, the result will usually not be an integer, or equivalently, there will be a remainder. The notation $a \mathbin{//} b$ denotes the result of division when the remainder is ignored and is often referred to as integer division. For example $3 \mathbin{//} 2 = 1$, $9 \mathbin{//} 4 = 2$ and $24 \mathbin{/\!/} 6 = 4$. We also need notation for the remainder in the division, for this we write $a \,\%\, b$. This means that $3 \,\%\, 2 = 1$, while $23 \,\%\, 5 = 3$.

We will use standard notation for intervals of real numbers. Two real numbers a and b with $a < b$ define four intervals that only differ in whether the end points a and b are included or not. The closed interval $[a, b]$ contains all real numbers between a and b, including the end points. Formally we can express this by $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$. The other intervals can be defined similarly,

$$(a, b) = \{x \in \mathbb{R} \mid a < x < b\} \text{ (open)}; \qquad (a, b] = \{x \in \mathbb{R} \mid a < x \leq b\} \text{ half open};$$
$$[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\} \text{ (closed)}; \qquad [a, b) = \{x \in \mathbb{R} \mid a \leq x < b\} \text{ half open}.$$

With this notation we can say that a fractional number is a real number in the interval $[0, 1)$.

3.2 Natural Numbers in Different Numeral Systems

We usually represent natural numbers in the decimal numeral system, but in this section we are going to see that this is just one of infinitely many numeral systems. We will also give a simple method for converting a number from its decimal representation to its representation in a different base.

3.2.1 Alternative Numeral Systems

In the decimal system we use a positional convention and express numbers in terms of the ten digits 0, 1, ..., 8, 9, and let the position of a digit determine how much it is worth. For example the string of digits 3761 is interpreted as

$$3761 = 3 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 1 \times 10^0.$$

Numbers that have a simple representation in the decimal numeral system are often thought of as special. For example it is common to celebrate a 50th birthday in a special way or mark the centenary anniversary of an important event like a country's independence. However, the numbers 50 and 100 are only special when they are written in the decimal number system.

Any natural number can be used as the base for a number system. Consider for example the *septenary* numeral system which has 7 as the base and uses the digits 0-6. In this system the numbers 3761, 50 and 100 become

$$\begin{aligned} 3761 &= 13652_7 = 1 \times 7^4 + 3 \times 7^3 + 6 \times 7^2 + 5 \times 7^1 + 2 \times 7^0, \\ 50 &= 101_7 = 1 \times 7^2 + 0 \times 7^1 + 1 \times 7^0, \\ 100 &= 202_7 = 2 \times 7^2 + 0 \times 7^1 + 2 \times 7^0, \end{aligned}$$

so 50 and 100 are not so special anymore.

These examples make it quite obvious that we can define numeral systems with almost any natural number as a base. The only restriction is that the base must be greater than one. To use 0 as base is quite obviously meaningless, and if we try to use 1 as base we only have the digit 0 which means that we can only represent the number 0.

We record the general construction in a formal definition. In this way the construction receives a name (Definition 3.2) which we can refer to later, and it becomes more visible as you flip through the pages.

Definition 3.2. *Let β be a natural number greater than 1 and let n_0, n_1, ..., $n_{\beta-1}$ be β distinct numerals (also called digits) such that n_i denotes the number $n_i = i$. A natural number representation in base β is an ordered collection of digits $(d_n d_{n-1} \ldots d_1 d_0)_\beta$ which is interpreted as the natural number*

$$d_n\beta^n + d_{n-1}\,\beta^{n-1} + d_{n-2}\,\beta^{n-2} + \cdots + d_1\beta^1 + d_0\beta^0$$

where each digit d_i is one of the β numerals $\{n_i\}_{i=0}^{\beta-1}$.

Formal definitions in mathematics often appear complicated until one gets under the surface, so let us consider the details of the definition. The base β is not so mysterious. In the decimal system $\beta = 10$ while in the septenary system $\beta = 7$. The beginning of the definition simply states that any natural number greater than 1 can be used as a base.

In the decimal system we use the digits 0–9 to write down numbers, and in any numeral system we need digits that can play a similar role. If the base is 10 or less it is natural to use the obvious subset of the decimal digits as numerals. If the base is 2 we use the two digits $n_0 = 0$ and $n_1 = 1$; if the base is 5 we use the five digits $n_0 = 0$, $n_1 = 1$, $n_2 = 2$, $n_3 = 3$ and $n_4 = 4$. However, if the base is greater than 10 we have a challenge in how to choose numerals for the numbers $10, 11, \ldots, \beta - 1$. If the base is less than 40 it is common to use the decimal digits together with the initial characters of the latin alphabet as numerals. In base $\beta = 16$ for example, it is common to use the digits 0–9 augmented with $n_{10} = a$, $n_{11} = b$, $n_{12} = c$, $n_{13} = d$, $n_{14} = e$ and $n_{15} = f$. This is called the *hexadecimal* numeral system and in this system the number 3761 becomes

$$eb1_{16} = e \times 16^2 + b \times 16^1 + 1 \times 16^1 = 14 \times 256 + 11 \times 16 + 1 = 3761.$$

Definition 3.2 defines how a number can be expressed in the numeral system with base β. However, it does not say anything about how to find the digits of a fixed number. And even more importantly, it does not guarantee that a number can be written in the base-β numeral system in only one way. This is settled in our first lemma below.

Lemma 3.3. *Any natural number can be represented uniquely in the base-β numeral system.*

Proof. To keep the argument as transparent as possible, we give the proof for a specific example, namely $a = 3761$ and $\beta = 8$ (the octal numeral system). Since $8^4 = 4096 > a$, we know that the base-8 representation cannot contain more than 4 digits. Suppose that $3761 = (d_3 d_2 d_1 d_0)_8$; our job is to find the value of the four digits and show that each of them only has one possible value. We start by determining d_0. By definition of base-8 representation of numbers we have the relation

$$3761 = (d_3 d_2 d_1 d_0)_8 = d_3 8^3 + d_2 8^2 + d_1 8 + d_0. \tag{3.1}$$

We note that only the last term in the sum on the right is not divisible by 8, so the digit d_0 must therefore be the remainder when 3761 is divided by 8. If we perform the division we find that

$$d_0 = 3761 \,\%\, 8 = 1, \qquad 3761 \,//\, 8 = 470.$$

We observe that when the right-hand side of (3.1) is divided by 8, the result is $d_3 8^2 + d_2 8 + d_1$. In other words be must have

$$470 = d_3 8^2 + d_2 8 + d_1.$$

But then we see that d_1 must be the remainder when 470 is divided by 8. If we perform this division we find

$$d_1 = 470 \,\%\, 8 = 6, \qquad 470 \,//\, 8 = 58.$$

Using the same argument as before we see that the relation

$$58 = d_3 8 + d_2 \tag{3.2}$$

must hold. In other words d_2 is the remainder when 58 is divided by 8,

$$d_2 = 58 \,\%\, 8 = 2, \qquad 58 \,//\, 8 = 7.$$

If we divide both sides of (3.2) by 8 and drop the remainder we are left with $7 = d_3$. The net result is that $3761 = (d_3 d_2 d_1 d_0)_8 = 7261_8$.

We note that during the computations we never had any choice in how to determine the four digits, they were determined uniquely. We therefore conclude that the only possible way to represent the decimal number 3761 in the base-8 numeral system is as 7261_8. ∎

The proof is clearly not complete since we have only verified Lemma 3.3 in a special case. However, the same argument can be used for any a and β and we leave it to the reader to write down the details in the general case.

Lemma 3.3 says that any natural number can be expressed in a unique way in any numeral system with base greater than 1. We can therefore use any such numeral system to represent numbers. Although we may feel that we always use the decimal system, we all use a second system every day, the base-60 system. An hour is split into 60 minutes and a minute into 60 seconds. The great advantage of using 60 as a base is that it is divisible by 2, 3, 4, 5, 6, 12, 20 and 30 which means that an hour can easily be divided into many smaller parts without resorting to fractions of minutes. Most of us also use other numeral systems without knowing. Virtually all electronic computers use the base-2 (binary) system and we will study this in the next chapter.

We are really only considering natural numbers in this section, but let us add a comment about how to represent negative numbers in the base-β numeral system. This is not so difficult. There is nothing particularly decimal about the minus sign, so the number $-a$ may be represented like a, but preceded with $-$. Therefore, we represent for example the decimal number -3761 as -7261_8 in the octal numeral system.

3.2.2 Conversion to the Base-β Numeral System

The method used in the proof of Lemma 3.3 for converting a number to base β is important, so we record it as an *algorithm*. An algorithm is just a detailed recipe for accomplishing a specific task, and in these notes we will come across many algorithms. An algorithm may be performed manually as in the proof above, but for most of the algorithms we encounter the intention is that they should be implemented in a computer program. Our algorithms are therefore formulated in a syntax that is reminiscent of a programming language.

Algorithm 3.4. *Let a be a natural number that in base β has the $n+1$ digits $(d_n d_{n-1} \cdots d_0)_\beta$. These digits may be computed by performing the operations:*

$a_0 := a$;
for $i := 0, 1, \ldots, n$
    $d_i := a_i \,\%\, \beta$;
    $a_{i+1} := a_i \mathbin{//} \beta$;

Let us add a little explanation since this is our first algorithm. We start by setting the variable a_0 equal to a. We then let i take on the values 0, 1, 2, ..., n. For each value of i we perform the operations that are indented, i.e., we compute the numbers $a_i \,\%\, \beta$ and $a_i \mathbin{//} \beta$ and store the results in the variables d_i and a_{i+1}. Note the use of the *assignment* operator :=. This simply means 'compute the value of the right-hand side and store the result in the variable on the left'. This is different from the relation = which tests whether the two sides are equal.

Algorithm 3.4 demands that the number of digits in the representation to be computed is known in advance. If we look back on the proof of Lemma 3.3, we note that we do not first check how many digits we are going to compute, since when we are finished the number that we divide (the number a_i in Algorithm 3.4) has become 0. We can therefore just repeat the two indented statements in the algorithm until the result of the division becomes 0. The following version of the algorithm incorporates this. We also note that we do not need to keep the results of the divisions; we can omit the subscript and store the result of the division $a \mathbin{//} \beta$ back in a.

Algorithm 3.5. *Let a be a natural number that in base β has the $n+1$ digits $(d_n d_{n-1} \cdots d_0)_\beta$. These digits may be computed by performing the operations:*

$i := 0$;
while $a > 0$
    $d_i := a \,\%\, \beta$;
    $a := a \,//\, \beta$;
    $i := i + 1$;

The 'while $a > 0$' statement means that all the indented statements will be repeated until a becomes 0. The variable i is needed so that we can number the digits correctly, starting with d_0, then d_1 and so on. The statement $i := i + 1$ may look a bit strange, but if we remember the meaning of the := operator, it is not so strange. Each time we reach this statement, the variable already has a value; even the first time since i is initialized with the value 0 before we reach the while statement. Then the statement $i := i + 1$ leads to the following: Extract the value of i, add 1, and store the result back in i. Suppose for example that i has the value 3 when we reach the statement, then i will be 4 when the statement has been completed. The statement $a := a \,//\, \beta$ should be interpreted in the same way.

It is also important to realize that the order of the indented statements is not arbitrary. When we do not keep all the results of the divisions, it is important that d_i is computed before a is updated with its new value. And when i is initialized with 0, we must update i at the end, since otherwise the subscript in d_i will be wrong.

We will not usually discuss algorithms in this much detail, but will comment when we introduce important new concepts and constructs.

The results produced by Algorithm 3.5 can be conveniently organized in a table. The example in the proof of Lemma 3.3 can be displayed as

3761	1
470	6
58	2
7	7

The left column shows the successive integral parts resulting from repeated division by 8, whereas the right column shows the remainder in these divisions. Let us consider one more example.

Example 3.6. Instead of converting 3761 to base 8 let us convert it to base 16. We find that 3761 // 16 = 235 with remainder 1. In the next step we find 235 // 16 = 14

with remainder 11. Finally we have $14 // 16 = 0$ with remainder 14. Displayed in a table this becomes

3761	1
235	11
14	14

Recall that in the hexadecimal system the letters a–f usually denote the values 10–15. We have therefore found that the number 3761 is written $\mathrm{eb1}_{16}$ in the hexadecimal numeral system.

Since we are particularly interested in how computers manipulate numbers, we should also consider an example of conversion to the binary numeral system, as this is the numeral system used in most computers. Instead of dividing by 16 we are now going to repeatedly divide by 2 and record the remainder. A nice thing about the binary numeral system is that the only possible remainders are 0 and 1: it is 0 if the number we divide is an even integer and 1 if the number is an odd integer.

Example 3.7. Let us continue to use the decimal number 3761 as an example, but now we want to convert it to binary form. If we perform the divisions and record the results as before we find

3761	1
1880	0
940	0
470	0
235	1
117	1
58	0
29	1
14	0
7	1
3	1
1	1

In other words we have $3761 = 111010110001_2$. This example illustrates an important property of the binary numeral system: Computations are simple, but long and tedious. This means that this numeral system is not so good for humans as we tend to get bored and make sloppy mistakes. For computers, however, this is perfect as computers do not make mistakes and work extremely fast.

3.3 Representation of Fractional Numbers

We have seen how integers can be represented in numeral systems other than decimal, but what about fractions and irrational numbers? In the decimal system such numbers are characterized by the fact that they have two parts, one to the left of the decimal point, and one to the right, like the number 21.828. The part to the left of the decimal point — the integer part — can be represented in base-β as outlined above. If we can represent the part to the right of the decimal point — the fractional part — in base-β as well, we can follow the convention from the decimal system and use a point to separate the two parts. Negative rational and irrational numbers are as easy to handle as negative integers, so we focus on how to represent positive numbers without an integer part, in other words numbers in the open interval $(0,1)$.

3.3.1 Rational and Irrational Numbers in Base-β

Let a be a real number in the interval $(0,1)$. In the decimal system we can write such a number as 0, followed by a point, followed by a finite or infinite number of decimal digits, as in

$$0.45928\ldots$$

This is interpreted as the number

$$4\times10^{-1}+5\times10^{-2}+9\times10^{-3}+2\times10^{-4}+8\times10^{-5}+\cdots.$$

From this it is not so difficult to see what a base-β representation of a similar number must look like.

Definition 3.8. *Let β be a natural number greater than 1 and let n_0, n_1, …, $n_{\beta-1}$ be β distinct numerals (also called digits) such that n_i denotes the number $n_i = i$. A fractional representation in base β is a, finite or infinite, ordered collection of digits $(0.d_{-1}d_{-2}d_{-3}\ldots)_\beta$ which is interpreted as the real number*

$$d_{-1}\beta^{-1}+d_{-2}\,\beta^{-2}+d_{-3}\,\beta^{-3}+\cdots \tag{3.3}$$

where each digit d_i is one of the β numerals $\{n_i\}_{i=0}^{\beta-1}$.

This definition is considerably more complicated than Definition 3.2 since we may have an infinite number of digits. This becomes apparent if try to check the size of numbers on the form given by (3.3). Since none of the terms in the sum are negative, the smallest number is the one where all the digits are 0, i.e., where $d_i = 0$ for $i = -1, -2, \ldots$. But this can be nothing but the number 0.

The largest possible number occurs when all the digits are as large as possible, i.e. when $d_i = \beta - 1$ for all i. If we call this number x, we find

$$\begin{aligned} x &= (\beta-1)\beta^{-1} + (\beta-1)\beta^{-2} + (\beta-1)\beta^{-3} + \cdots \\ &= (\beta-1)\beta^{-1}(1 + \beta^{-1} + \beta^{-2} + \cdots \\ &= \frac{\beta-1}{\beta}\sum_{i=0}^{\infty}(\beta^{-1})^i. \end{aligned}$$

In other words x is given by a sum of an infinite geometric series with factor $\beta^{-1} = 1/\beta < 1$. This series converges to $1/(1-\beta^{-1})$ so x has the value

$$x = \frac{\beta-1}{\beta}\frac{1}{1-\beta^{-1}} = \frac{\beta-1}{\beta}\frac{\beta}{\beta-1} = 1.$$

Let us record our findings so far.

Lemma 3.9. *Any number on the form* (3.3) *lies in the interval* $[0,1]$.

The fact that the base-β fractional number with all digits equal to $\beta-1$ is the number 1 is a bit disturbing since it means that real numbers cannot be represented uniquely in base β. In the decimal system this corresponds to the fact that $0.99999999999999\ldots$ (infinitely many 9s) is in fact the number 1. And this is not the only number that has two representations. Any number that ends with an infinite number of digits equal to $\beta-1$ has a simpler representation. Consider for example the decimal number $0.12999999999999\ldots$. Using the same technique as above we find that this number is 0.13. However, it turns out that these are the only numbers that have a double representation, see Theorem 3.10 below.

Let us now see how we can determine the digits of a fractional number in a numeral system other than the decimal one. As for natural numbers, it is easiest to understand the procedure through an example, so we try to determine the digits of $1/5$ in the octal system. According to Definition 3.8 we seek digits $d_{-1}d_{-2}d_{-3}\ldots$ (possibly infinitely many) such that the relation

$$\frac{1}{5} = d_{-1}8^{-1} + d_{-2}8^{-2} + d_{-3}8^{-3} + \cdots \tag{3.4}$$

becomes true. If we multiply both sides by 8 we obtain

$$\frac{8}{5} = d_{-1} + d_{-2}8^{-1} + d_{-3}8^{-2} + \cdots. \tag{3.5}$$

The number 8/5 lies between 1 and 2 and we know from Lemma 3.9 that the sum $d_{-2}8^{-1} + d_{-3}8^{-2} + \cdots$ can be at most 1. Therefore we must have $d_{-1} = 1$. Since d_{-1} has been determined we can subtract this from both sides of (3.5)

$$\frac{3}{5} = d_{-2}8^{-1} + d_{-3}8^{-2} + d_{-4}8^{-3} + \cdots \tag{3.6}$$

(to be precise, we subtract 1 on the left and d_{-1} on the right). This equation is on the same form as (3.4) and can be used to determine d_{-2}. We multiply both sides of (3.6) by 8,

$$\frac{24}{5} = d_{-2} + d_{-3}8^{-1} + d_{-4}8^{-3} + \cdots . \tag{3.7}$$

The fraction 24/5 lies in the interval $(4,5)$ and since the terms on the right that involve negative powers of 8 must be a number in the interval $[0,1]$, we must have $d_{-2} = 4$. We subtract this from both sides of (3.7) and obtain

$$\frac{4}{5} = d_{-3}8^{-1} + d_{-4}8^{-2} + d_{-5}8^{-3} + \cdots . \tag{3.8}$$

Multiplication by 8 now gives

$$\frac{32}{5} = d_{-3} + d_{-4}8^{-1} + d_{-5}8^{-2} + \cdots .$$

from which we conclude that $d_{-3} = 6$. Subtracting 6 and multiplying by 8 we obtain

$$\frac{16}{5} = d_{-4} + d_{-5}8^{-1} + d_{-6}8^{-2} + \cdots .$$

from which we conclude that $d_{-4} = 3$. If we subtract 3 from both sides we find

$$\frac{1}{5} = d_{-5}8^{-1} + d_{-6}8^{-2} + d_{-7}8^{-3} + \cdots .$$

But this relation is essentially the same as (3.4), so if we continue we must generate the same digits again. In other words, the sequence $d_{-5}d_{-6}d_{-7}d_{-8}$ must be the same as $d_{-1}d_{-2}d_{-3}d_{-4} = 1463$. But once d_{-8} has been determined we must again come back to a relation with 1/5 on the left, so the same digits must also repeat in $d_{-9}d_{-10}d_{-11}d_{-12}$ and so on. The result is that

$$\frac{1}{5} = 0.1463146314631463\cdots_8 .$$

Based on this procedure we can prove an important theorem.

Theorem 3.10. *Any real number in the interval* $(0,1)$ *can be represented in a unique way as a fractional base-*β *number provided representations with infinitely many trailing digits equal to* $\beta-1$ *are prohibited.*

Proof. We have already seen how the digits of $1/5$ in the octal system can be determined, and it is easy to generalize the procedure. However, there are two additional questions that must be settled before the claims in the theorem are completely settled.

We first prove that the representation is unique. If we look back at the conversion procedure in the example we considered, we had no freedom in the choice of any of the digits. The digit d_{-2} was for example determined by equation 3.7, where the left-hand side is 4.8 in the decimal system. Then our only hope of satisfying the equation is to choose $d_{-2}=4$ since the remaining terms can only sum up to a number in the interval $[0,1]$.

How can the procedure fail to determine the digits uniquely? In our example, any digit is determined by an equation on the form (3.7), and as long as the left-hand side is not an integer, the corresponding digit is uniquely determined. If the left-hand side should happen to be an integer, as in

$$5 = d_{-2} + d_{-3}8^{-1} + d_{-4}8^{-3} + \cdots,$$

the natural solution is to choose $d_{-2}=5$ and choose all the remaining digits as 0. However, since we know that 1 may be represented as a fractional number with all digits equal to 7, we could also choose $d_{-2}=4$ and $d_i=7$ for all $i<-2$. The natural solution is to choose $d_{-2}=5$ and prohibit the second solution, which secures the uniqueness of the representation.

The second point that needs to be settled is more subtle; do we really compute the correct digits? It may seem strange to think that we may not compute the right digits since the digits are forced upon us by the equations. But if we look carefully, the equations are not quite standard since the sums on the right may contain infinitely many terms. In general it is therefore impossible to achieve equality in the equations, all we can hope for is that we can make the sum on the right in (3.4) come as close to $1/5$ as we wish by including sufficiently many terms.

Set $a=1/5$. Then equation (3.6) can be written

$$8(a - d_{-1}8^{-1}) = d_{-2}8^{-1} + d_{-3}8^{-2} + d_{-4}8^{-3} + \cdots$$

while (3.8) can be written

$$8^2(a - d_{-1}8^{-1} - d_{-2}8^{-2}) = d_{-3}8^{-1} + d_{-4}8^{-2} + d_{-5}8^{-3} + \cdots.$$

After i steps the equation becomes

$$8^i(a - d_{-1}8^{-1} - d_{-2}8^{-2} - \cdots - d_{-i}8^{-i}) = \\ d_{-i-1}8^{-1} + d_{-i-2}8^{-2} + d_{-i-3}8^{-3} + \cdots .$$

The expression in the bracket on the left we recognize as the error e_i in approximating a by the first i numerals in the octal representation. We can rewrite this slightly and obtain

$$e_i = 8^{-i}(d_{-i-1}8^{-1} + d_{-i-2}8^{-2} + d_{-i-3}8^{-3} + \cdots).$$

From Lemma 3.9 we know that the number in the bracket on the right lies in the interval $[0,1]$ so we have $0 \le e_i \le 8^{-i}$. But this means that by including sufficiently many digits (choosing i sufficiently big), we can get e_i to be as small as we wish. In other words, by including sufficiently many digits, we can get the octal representation of $a = 1/5$ to be as close to a as we wish. Therefore our method for computing numerals does indeed generate the digits of a. ∎

Some simple properties of fractional numbers in base β will be useful for understanding how computers handle numbers, but before we discuss this we will formulate precisely the algorithm for computing the base β representation.

3.3.2 An Algorithm for Converting Fractional Numbers

The basis for the proof of Theorem 3.10 is the procedure for computing the digits of a fractional number in base-β. We only considered the case $\beta = 8$, but it is simple to generalize the algorithm to arbitrary β.

Algorithm 3.11. *Let a be a fractional number whose first n digits in base β are $(0.d_{-1}d_{-2}\cdots d_{-n})_\beta$. These digits may be computed by performing the operations:*

for $i := -1, -2, \ldots, -n$
  $d_i := \lfloor a * \beta \rfloor$;
  $a := a * \beta - d_i$;

Compared with the description on pages 34 to 35 there should be nothing mysterious in this algorithm except for perhaps the notation $\lfloor x \rfloor$. This is a fairly standard way of writing the floor function which is equal to the largest integer that is less than or equal to x. We have for example $\lfloor 3.4 \rfloor = 3$ and $\lfloor 5 \rfloor = 5$.

When converting natural numbers to base-β representation there is no need to know or compute the number of digits beforehand, as is evident in Algorithm 3.5. For fractional numbers we do need to know how many digits to compute as there may often be infinitely many. A for-loop is therefore a natural construction in Algorithm 3.11.

It is convenient to have a standard way of writing down the computations involved in converting a fractional number to base-β, and it turns out that we can use the same format as for converting natural numbers. Let us take as an example the computations in the proof of Theorem 3.10 where the fraction $1/5$ was converted to base-8. We start by writing the number to be converted to the left of the vertical line. We then multiply the number by β (which is 8 in this case) and write the integer part of the result, which is the first digit, to the right of the line. The result itself we write in brackets to the right. We then start with the fractional part of the result one line down and continue until the result becomes 0 or we have all the digits we want,

$1/5$	1	$(8/5)$
$3/5$	4	$(24/5)$
$4/5$	6	$(32/5)$
$2/5$	3	$(16/5)$
$1/5$	1	$(8/5)$

3.3.3 Properties of Fractional Numbers in Base-β

Real numbers in the interval $(0,1)$ have some interesting properties related to their representation. In the decimal numeral system we know that fractions with a denominator that only contains the factors 2 and 5 can be written as a decimal number with a finite number of digits. In general, the decimal representation of a rational number will contain a finite sequence of digits that are repeated infinitely many times, while for an irrational number there will be no such structure. In this section we shall see that similar properties are valid when fractional numbers are represented in any numeral system.

For rational numbers Algorithm 3.11 can be expressed in a different form which makes it easier to deduce properties of the digits. So let us consider what happens when a rational number is converted to base-β representation. A rational number in the interval $(0,1)$ has the form $a = b/c$ where b and c are nonzero natural numbers with $b < c$. If we look at the computations in Algorithm 3.11, we note that d_i is the integer part of $(b * \beta)/c$ which can be computed as $(b * \beta) // c$. The right-hand side of the second statement is $a * \beta - d_1$, i.e., the fractional part of $a * \beta$. But if $a = b/c$, the fractional part of $a * \beta$ is given by the remainder in

the division $(b * \beta)/c$, divided by c, so the new value of a is given by

$$a = \frac{(b * \beta) \,\%\, c}{c}.$$

This is a new fraction with the same denominator c as before. But since the denominator does not change, it is sufficient to keep track of the numerator. This can be done by the statement

$$b := (b * \beta) \,\%\, c. \tag{3.9}$$

The result of this is a new version of Algorithm 3.11 for rational numbers.

Algorithm 3.12. *Let $a = b/c$ be a rational number in $(0,1)$ whose first n digits in base β are $(0.d_{-1}d_{-2}\cdots d_{-n})_\beta$. These digits may be computed by performing the operations:*

for $i := -1, -2, \ldots, -n$
    $d_i := (b * \beta) \,//\, c$;
    $b := (b * \beta) \,\%\, c$;

This version of the conversion algorithm is more convenient for deducing properties of the numerals of a rational number. The clue is to consider more carefully the different values of b that are computed by the algorithm. Since b is the remainder when integers are divided by c, the only possible values of b are $0, 1, 2, \ldots, c-1$. Sooner or later, the value of b must therefore become equal to an earlier value. But once b gets back to an earlier value, it must cycle through exactly the same values again until it returns to the same value a third time. And then the same values must repeat again, and again, and again, ... And since the numerals d_i are computed from b, they must repeat with the same frequency. This proves part of the following lemma.

Lemma 3.13. *Let a be a fractional number. Then the digits of a written in base β will eventually repeat, i.e.,*

$$a = 0.d_{-1}\cdots d_{-i}d_{-(i+1)}\cdots d_{-(i+n)}d_{-(i+1)}\cdots d_{-(i+n)}\cdots_\beta$$

if and only if a is a rational number.

As an example, consider the fraction 1/7 written in different numeral systems. If we run Algorithm 3.12 we find

$$1/7 = 0.001001001001001\cdots_2,$$
$$1/7 = 0.010212010212010212\cdots_3,$$
$$1/7 = 0.1_7.$$

In the binary numeral system, there is no initial sequence of digits; the sequence 001 repeats from the start. In the trinary system, there is no intial seqeunce either and the repeating sequence is 010212, whereas in the septenary system the initial seqeunce is 1 and the repeating sequence 0 (which we do not write according to the conventions of the decimal system).

An example with an initial sequence is the fraction 87/98 which in base 7 becomes $0.6133333\cdots_7$. Another example is 503/1100 which is $0.457272727272\cdots$ in the decimal system.

The argument preceding Lemma 3.13 proves the fact that if a is a rational number, then the digits must eventually repeat. But this statement leaves the possibility open that there may be nonrational (i.e., irrational) numbers that may also have digits that eventually repeat. However, this is not possible and this is the reason for the 'only if'-part of the lemma. In less formal language the complete statement is: *The digits of a will eventually repeat if a is a rational number, and only if a is a rational number.* This means that there are two statements to prove: (i) The digits repeat if a is a rational number and (ii) if the digits do repeat then a must be a rational number. The proof of this latter statement is left to excercise 14.

Although all rational numbers have repeating digits, for some numbers the repeating sequence is '0', like 1/7 in base 7, see above. Or equivalently, some fractional numbers can in some numeral systems be represented exactly by a finite number of digits. It is possible to characterize exactly which numbers have this property.

Lemma 3.14. *The representation of a fractional number a in base-β consists of a finite number of digits,*

$$a = (0.d_{-1}d_{-2}\cdots d_{-n})_\beta,$$

if and only if a is a rational number b/c with the property that all the prime factors of c divide β.

Proof. Since the statement is of the 'if and only if' type, there are two claims to

be proved. The fact that a fractional number in base-β with a finite number of digits is a rational number is quite straightforward, see exercise 15.

What remains is to prove that if $a = b/c$ and all the prime factors of c divide β, then the representation of a in base-β will have a finite number of digits. We give the proof in a special case and leave it to the reader to write down the proof in general. Let us consider the representation of the number $a = 8/9$ in base-6. The idea of the proof is to rewrite a as a fraction with a power of 6 in the denominator. The simplest way to do this is to observe that $8/9 = 32/36$. We next express 32 in base 6. For this we can use Algorithm 3.5, but in this simple situation we see directly that

$$32 = 5 \times 6 + 2 = 52_6.$$

We therefore have

$$\frac{8}{9} = \frac{32}{36} = \frac{5 \times 6 + 2}{6^2} = 5 \times 6^{-1} + 2 \times 6^{-2} = 0.52_6. \quad \blacksquare$$

In the decimal system, fractions with a denominator that only has 2 and 5 as prime factors have finitely many digits, for example $3/8 = 0.375$, $4/25 = 0.16$ and $7/50 = 0.14$. These numbers will *not* have finitely many digits in most other numeral systems. In base-3, the only fractions with finitely many digits are the ones that can be written as fractions with powers of 3 in the denominator,

$$\begin{aligned}
\frac{8}{9} &= 0.22_3, \\
\frac{7}{27} &= 0.021_3, \\
\frac{1}{2} &= 0.111111111111\cdots_3, \\
\frac{3}{10} &= 0.02200220022\cdots_3.
\end{aligned}$$

In base-2, the only fractions that have an exact representation are the ones with denominators that are powers of 2,

$$\begin{aligned}
\frac{1}{2} &= 0.5 = 0.1_2, \\
\frac{3}{16} &= 0.1875 = 0.0011_2, \\
\frac{1}{10} &= 0.1 = 0.00011001100110011\cdots_2.
\end{aligned}$$

3.4 Arithmetic in Base β

The methods we learn in school for performing arithemetic are closely tied to properties of the decimal numeral system, but the methods can easily be generalized to any numeral system. We are not going to do this in detail, but some examples should illustrate the general ideas. All the methods should be familiar from school, but if you never quite understood the arithmetic methods, you may have to think twice to understand why it all works. Although the methods themselves are the same across the world, it should be mentioned that there are many variations in how the methods are expressed on paper. You may therefore find the description given here unfamiliar at first.

3.4.1 Addition

Addition of two one-digit numbers is just like in the decimal system as long as the result has only one digit. For example, we have $4_8 + 3_8 = 4 + 3 = 7 = 7_8$. If the result requires two digits, we must remember that the carry is β in base-β, and not 10. So if the result becomes β or greater, the result will have two digits, where the left-most digit is 1 and the second has the value of the sum, reduced by β. This means that

$$5_8 + 6_8 = 5 + 6 = 11 = 8 + 11 - 8 = 8 + 3 = 13_8.$$

This can be written exactly the same way as you would write a sum in the decimal numeral system, you must just remember that the value of the carry is β.

Let us now try the larger sum $457_8 + 325_8$. This just requires successive one digit additions, just like in the decimal system. One way to write this is

$$\begin{array}{r} {\scriptstyle 1\,1} \\ 457_8 \\ +325_8 \\ \hline = 1004_8 \end{array}$$

This corresponds to the decimal sum $303 + 213 = 516$.

3.4.2 Subtraction

One-digit subtractions are simple, for example $7_8 - 3_8 = 4_8$. A subtraction like $14_8 - 7_8$ is a bit more difficult, but we can 'borrow' from the '1' in 14 just like in the decimal system. The only difference is that in base-8, the '1' represents 8 and not 10, so we borrow 8. We then see that we must perform the subtraction $12 - 7$ so the answer is 5 (both in decimal and base 8). Subtraction of larger numbers can be done by repeating this. Consider for example $321_8 - 177_8$. This can be written

$$\begin{array}{r} {\scriptstyle 8\,8} \\ 3\not{2}1_8 \\ -177_8 \\ \hline = 122_8 \end{array}$$

By converting everything to decimal, it is easy to see that this is correct.

3.4.3 Multiplication

Let us just consider one example of a multiplication, namely $312_4 \times 12_4$. As in the decimal system, the basis for performing multiplication of numbers with multiple digits is the multiplication table for one digit numbers. In base 4 the multiplication table is

	1	2	3
1	1	2	3
2	2	10	12
3	3	12	21

We can then perform the multiplication as we are used to in the decimal system

$$\begin{array}{r} 312_4 \times 12_4 \\ \hline 1230_4 \\ 312_4 \\ \hline 11010_4 \end{array}$$

The number 1230_4 in the second line is the result of the multiplication $312_4 \times 2_4$, i.e., the first factor 312_4 multiplied by the second digit of the right-most factor 12_4. The number on the line below, 312_4, is the first factor multiplied by the first digit of the second factor. This second product is shifted one place to the left since multiplying with the first digit in 12_4 corresponds multiplication by 1×4. The number on the last line is the sum of the two numbers above, with a zero added at the right end of 312_4, i.e., the sum is $1230_4 + 3120_4$. This sum is calculated as indicated in Section 3.4.1 above.

Exercises

3.1 Convert the following natural numbers:

a) 40 to base-4

b) 17 to base-5

c) 17 to base-2

d) 123456 to base-7

e) 22875 to base-7

f) 126 to base 16

3.2 Convert the following rational numbers:

a) 1/4 to base-2

b) 3/7 to base-3

c) 1/9 to base-3

d) 1/18 to base-3

e) 7/8 to base-8

f) 7/8 to base-7

g) 7/8 to base-16

h) 5/16 to base-8

i) 5/8 to base-6

3.3 Convert π to base-9.

3.4 Convert to base-8:

a) 1011001_2

b) 110111_2

c) 10101010_2

3.5 Convert to base-2:

a) 44_8

b) 100_8

c) 327_8

3.6 Convert to base-16:

a) 1001101_2

b) 1100_2

c) 10100111100100_2

3.7 Convert to base-2:

a) $3c_{16}$

b) 100_{16}

c) $e51_{16}$

3.8 a) Convert 7 to base-7, 37 to base-37, and 4 to base-4 and formulate a generalisation of what you observe.

b) Determine β such that $13 = 10_\beta$. Also determine β such that $100 = 10_\beta$ For which numbers $a \in \mathbb{N}$ is there a β such that $a = 10_\beta$?

3.9 a) Convert 400 to base-20, 4 to base-2, 64 to base-8, 289 to base-17 and formulate a generalisation of what you observe.

b) Determine β such that $25 = 100_\beta$. Also determine β such that $841 = 100_\beta$. For which numbers $a \in \mathbb{N}$ is there a number β such that $a = 100_\beta$?

c) For which numbers $a \in \mathbb{N}$ is there a number β such that $a = 1000_\beta$?

3.10 a) For which value of β is $a = b/c = 0.b_\beta$? Does such a β exist for all $a < 1$? And for $a \geq 1$?

b) For which rational number $a = b/c$ does there exist a β such that $a = b/c = 0.01_\beta$?

c) For which rational number $a = b/c$ is there a β such that $a = b/c = 0.0b_\beta$? If β exists, what will it be?

3.11 If $a = b/c$, what is the maximum length of the repeating sequence?

3.12 Show that if the digits of the fractional number a eventually repeat, then a must be a rational number.

3.13 Show that a fractional number in base-β with a finite number of digits is a rational number.

3.14 Show that if the digits of the fractional number a eventually repeat, then a must be a rational number.

3.15 Show that a fractional numbers in base-β with a finite number of digits is a rational number.

3.16 If $a = b/c$, what is the maximum length of the repeating sequence?

3.17 Perform the following additions:

a) $3_7 + 1_7$

b) $5_6 + 4_6$

c) $110_2 + 1011_2$

d) $122_3 + 201_3$

e) $43_5 + 10_5$

f) $3_5 + 1_7$

3.18 Perform the following subtractions:

a) $5_8 - 2_8$

b) $100_2 - 1_2$

c) $527_8 - 333_8$

d) $210_3 - 21_3$

e) $43_5 - 14_5$

f) $3_7 - 11_7$

3.19 Perform the following multiplications:

a) $110_2 \cdot 10_2$

b) $110_2 \cdot 11_2$

c) $110_3 \cdot 11_3$

d) $43_5 \cdot 2_5$

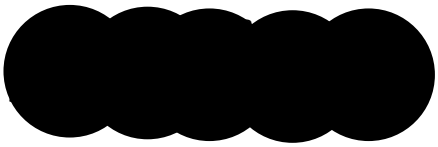
e) $720_8 \cdot 15_8$

f) $210_3 \cdot 12_3$

g) $101_2 \cdot 11_2$

Muhammad Davut Hassan

Karnaugh Maps



Simplifying Boolean Expressions

We can use the sum-of-products method to generate any digital logic circuit for which we can write the truth table. While the sum-of-products method will always produce a correct circuit, it usually does not produce the optimal circuit. We want the simplest possible circuit because fewer gates or simpler gates mean reduced cost, improved reliability, and often increased speed. We can simplify a circuit by simplifying the Boolean expression for it, then using the techniques already learned to produce the circuit that is equivalent to the simplified expression. The need to simplify Boolean expressions occurs in programming as well as hardware design, and the techniques discussed here are equally applicable to programming.

One way to simplify a Boolean expression is to apply the laws of Boolean algebra, some of which are summarized in the table on p. 144 of Tanenbaum. We will apply the laws of Boolean algebra to simplify $AB + A\overline{B}$. We choose this expression because it is key to how Karnaugh maps work; other expressions can also be simplified using Boolean algebra.

$AB + A\overline{B}$	*original expression*
$A(B + \overline{B})$	*after applying distributive law*
$A1$	*after applying inverse law*
$1A$	*after applying commutative law*
A	*after applying identity law*

Therefore, $AB + A\overline{B} = A$.

Karnaugh Maps

Karnaugh maps are a graphical way of using the relationship $AB + A\overline{B} = A$ to simplify a Boolean expression and thus simplify the resulting circuit. A Karnaugh map is a completely mechanical method of performing this simplification, and so has an advantage over manipulation of expressions using Boolean algebra. Karnaugh maps are effective for expressions of up to about six variables. For more complex expressions the Quine-McKluskey method, not discussed here, may be appropriate.

The Karnaugh map uses a rectangle divided into rows and columns in such a way that any product term in the expression to be simplified can be represented as the intersection of a row and a column. The rows and columns are labeled with each term in the expression and its complement. The labels must be arranged so that each horizontal or vertical move changes the state of one and only one variable.

To use a Karnaugh map to simplify an expression:

1. Draw a "map" with one box for each possible product term in the expression. The boxes must be arranged so that a one-box movement either horizontal or vertical changes one and only one variable. See Figure 1.
2. For each product term in the expression to be simplified, place a checkmark in the box whose labels are the product's variables and their complements.
3. Draw loops around adjacent pairs of checkmarks. Blocks are "adjacent" horizontally and vertically only, not diagonally. A block may be "in" more than one loop, and a single loop may span multiple rows, multiple columns, or both, so long as the number of checkmarks enclosed is a power of two.
4. For each loop, write an unduplicated list of the terms which appear; *i.e.* no matter how many times A appears, write down only one A.
5. If a term and its complement both appear in the list, *e.g.* both A and $\overline{A}$, delete both from the list.
6. For each list, write the Boolean product of the remaining terms.
7. Write the Boolean sum of the products from Step 5; this is the simplified expression.

Karnaugh Maps for Expressions of Two Variables

Start with the expression $AB + A\overline{B}$. This is an expression of two variables. We draw a rectangle and divide it so that there is a row or column for each variable and its complement.

Next, we place checks in the boxes that represent each of the product terms of the expression. The first product term is AB, so we place a check in the upper left block of the diagram, the conjunction of A and B. The second is $A\overline{B}$, so we place a check in the lower left block. Finally, we draw a loop around adjacent pairs of checks.

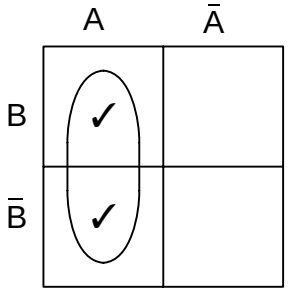


Figure 1. Karnaugh map for $AB + A\overline{B}$.

The loop contains A, B, A, and $\overline{B}$. We remove one A so that the list is unduplicated. The B and $\overline{B}$ "cancel," leaving only A, which is the expected result: $AB + A\overline{B} = A$.

Let us try a slightly more interesting example: simplify $A\overline{B} + \overline{A}B + \overline{A}\,\overline{B}$. There are two variables, so the rectangle is the same as in the first example. We perform the following steps:

- Place a check in the $A\overline{B}$ area.
- Place a check in the $\overline{A}B$ area.
- Place a check in the $\overline{A}\,\overline{B}$ area.
- Draw loops around pairs of adjacent checks.

The Karnaugh map appears in Figure 2. Because there are two loops, there will be two terms in the simplified expression. The vertical loop contains $\overline{A}$, B, $\overline{A}$, and $\overline{B}$. We remove one $\overline{A}$ to make an unduplicated list. The B and $\overline{B}$ cancel, leaving the remaining $\overline{A}$. From the horizontal loop we remove the duplicate $\overline{B}$, then remove A and $\overline{A}$ leaving only $\overline{B}$ in the second term. We write the Boolean sum of these, and the result is $\overline{A} + \overline{B}$, so:

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

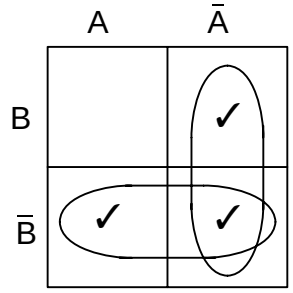


Figure 2. Karnaugh map for $A\overline{B} + \overline{A}B + \overline{A}\,\overline{B}$

Expressions of Three Variables

Recall that an essential characteristic of a Karnaugh map is that moving one position horizontally or vertically changes one and only one variable to its complement. For expression of three variables, the basic Karnaugh diagram is shown in Figure 3.

As with the diagram for two variables, adjacent squares differ by precisely one literal. The left and right edges are considered to be adjacent, as though the map were wrapped around to form a cylinder.

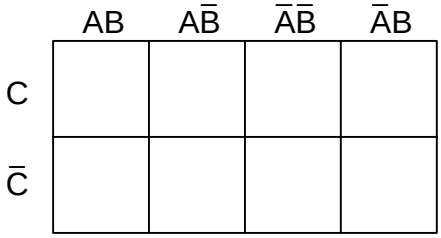


Figure 3. Form of a Karnaugh map for expressions of three variables.

Now we'll work through a complete example, starting with deriving a circuit from a truth table using the sum of products method, simplifying the sum of products expression, and drawing the new, simpler circuit.

Truth Table

A	B	C	X	Product Terms
0	0	0	1	$\overline{A}\,\overline{B}\,\overline{C}$
0	0	1	0	
0	1	0	0	
0	1	1	1	$\overline{A}BC$
1	0	0	0	
1	0	1	0	
1	1	0	0	
1	1	1	1	ABC

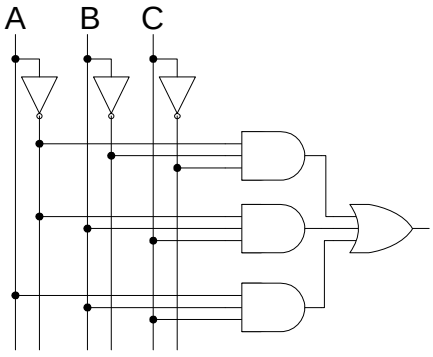
a)

Sum-of-Products Expression

$$\overline{A}\,\overline{B}\,\overline{C} + \overline{A}BC + ABC$$

b)

Digital Logic Circuit



c)

Figure 4. a) A truth table with product terms, b) the resulting sum-of-products expression, and c) the equivalent digital logic circuit.

The truth table in Figure 4a generates an expression with three product terms, as shown in Figure 4b. A measure of the complexity of a digital logic circuit is the number of gate inputs. The circuit in Figure 4c has 15 gate inputs. The Karnaugh map for the expression in Figure 4b is shown at the right. In this Karnaugh map, the large loop surrounds $A\,B\,C$ and $\overline{A}\,B\,C$; note that it "wraps around" from the left edge of the map to the right edge. The A and $\overline{A}$ cancel, so these two terms simplify to BC.

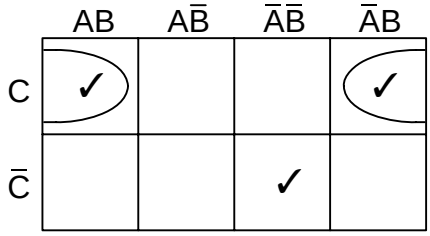


Figure 5. Simplifying $\overline{A}\,\overline{B}\,\overline{C} + \overline{A}\,B\,C + A\,B\,C$ with a Karnaugh map.

$\overline{A}\,\overline{B}\,\overline{C}$ is in a cell all by itself, and so contributes all three of its terms to the final expression. The simplified expression is $B\,C + \overline{A}\,\overline{B}\,\overline{C}$ and the simplified circuit is shown in Figure 6. In the simplified circuit, one three-input AND gate was removed, a remaining AND gate was changed to two inputs, and the OR gate was changed to two inputs, resulting in a circuit with ten gate inputs.

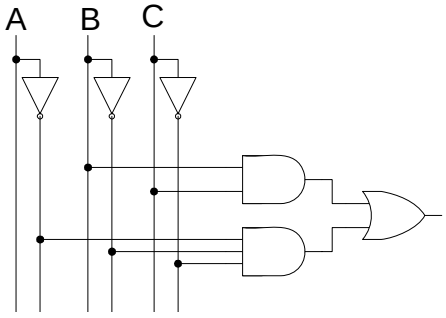


Figure 6. Simplified circuit for the truth table of Figure 4a.

Let's consider another example. The truth table in Figure 7a generates a sum-of-products expression with five product terms of three variables each. The sum-of-products expression is shown in Figure 7b. The digital logic circuit for this expression, shown in Figure 7c, has nine gates and 23 gate inputs. The Karnaugh map for this expression is shown in Figure 8.

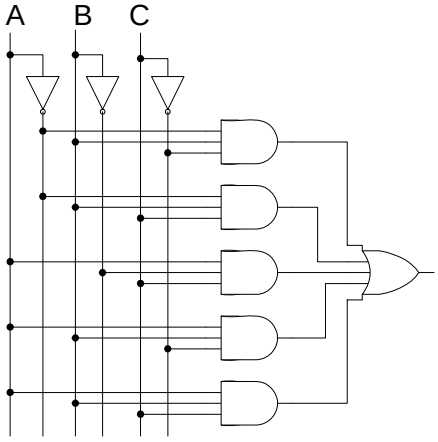
Truth Table

A	B	C	X	Product Terms
0	0	0	0	
0	0	1	0	
0	1	0	1	$\overline{A}\,B\,\overline{C}$
0	1	1	1	$\overline{A}\,B\,C$
1	0	0	0	
1	0	1	1	$A\,\overline{B}\,C$
1	1	0	1	$A\,B\,\overline{C}$
1	1	1	1	$A\,B\,C$

Sum-of-Products Expression

$$\overline{A}\,B\,\overline{C} + \overline{A}\,B\,C + A\,\overline{B}\,C + A\,B\,\overline{C} + A\,B\,C$$

Digital Logic Circuit



a) **b)** **c)**

Figure 7. a) A truth table with product terms, b) the resulting sum-of-products expression, and c) the equivalent digital logic circuit.

After removing duplicates, the large loop contains A and $\overline{A}$ *and also* C and $\overline{C}$; these cancel. All that's left after removing the two complement pairs is B. The small loop contains B and $\overline{B}$, which are removed, so it yields AC. We have simplified the expression in Figure 7b to B+AC.

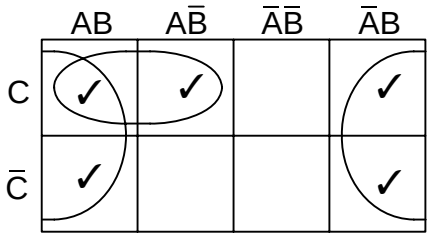


Figure 8. Karnaugh map for the expression of figure 7.

The circuit for B+AC is shown in Figure 9. We have simplified the circuit from nine gates and 23 inputs to two two-input gates. This is a substantial reduction in complexity.

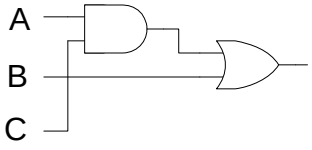


Figure 9. Simplified circuit equivalent to Figure 7c.

Getting the Best Results

For maximum simplification, you want to make the loops in a Karnaugh map as big as possible. If you have a choice of making one big loop or two small ones, choose the big loop. The restriction is that the loop must be rectangular and enclose a number of checkmarks that is a power of two.

When a map is more than two rows deep, *i.e.* when it represents more than three variables, the top and bottom edges can be considered to be adjacent in the same way that the right and left edges are adjacent in the two-by-four maps above.

If all checkmarks in a loop are enclosed within other loops as well, that loop can be ignored because all its terms are accounted for. In the Karnaugh map in Figure 10, the vertical loop is redundant and can be ignored.

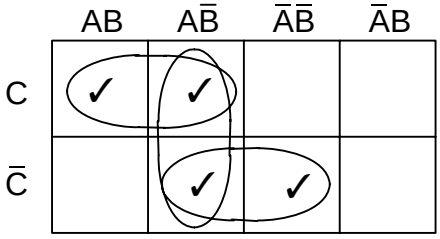


Figure 10. Karnaugh map showing a redundant loop.

Sometimes not all possible combinations of bits represented in a truth table can occur. For example, if four bits are used to encode a decimal digit, combinations greater than 1001 cannot occur. In that case, you can place a "D" (for "don't care") in the result column of the truth table. These D's may be treated as *either* ones or zeroes, and you can place check marks on the map in the D's positions if doing so allows you to make larger loops.

One General Form for a Karnaugh Map

There are several possible forms for a Karnaugh map, including some three-dimensional versions. All that is required is that a movement of one position changes the value of one and only one variable. We have shown a form for maps of two and three variables. Below are maps for four and five variables.

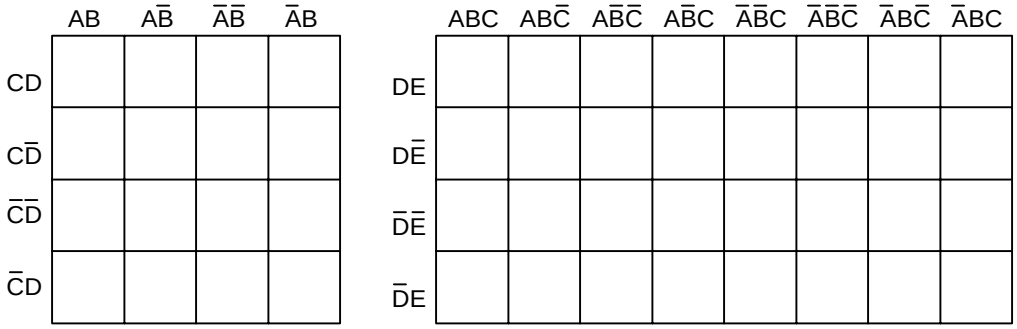


Figure 11. One form of the Karnaugh map for expressions of four and five variables.

A Notation Reminder

The *Boolean product* of two variables is written as AB, A∧B or A·B; the variables are combined using the AND function.

The *Boolean sum* of two variables is written as A+B or A∨B; the variables are combined using the OR function.

The *complement* of a Boolean variable is written as $\overline{A}$; it is evaluated using the NOT function.

The product, sum, and complement can be applied to expressions as well as single variables. Parentheses can be used to show precedence when needed.

Bibliography

Mendelson, Elliott, *Schaum's Outline of Theory and Problems of Boolean Algebra,* McGraw-Hill, 1970.

Stallings, William, *Computer Organization and Architecture: Designing for performance,* Prentice-Hall, 1996.

Tanenbaum, Andrew S., *Structured Computer Organization,* Prentice-Hall, 2006.

Exercises

1. Verify that the circuit in Figure 9 is equivalent to the circuit in Figure 7c by deriving the truth table for the circuit in Figure 9 and comparing it to Figure 7a.

2. Sketch a Karnaugh map for expressions of six variables. *Hint:* See Figure 11.

3. The truth table for binary addition has three inputs: the addend, the augend, and the carry in. The output has two parts, the sum and the carry out. Write the truth table for the sum part of binary addition. Use a Karnaugh map to simplify the expression represented by this truth table. *Hint:* This is a sneaky question, but you will learn a lot about the power of Karnaugh maps.

4. Write the truth table for the carry part of binary addition. Use a Karnaugh map to simplify the sum-of-products expression which this truth table produces.

5. Use a Karnaugh map to simplify $A\,B\,C + A\,\overline{B}\,\overline{C} + \overline{A}\,\overline{B}\,\overline{C}$.