2. Number systems and codes

Numerical systems can be symbolic, like the Roman one. In the Roman system we have the following coding:

symbol	I	V	X	L	C	D	M
value	1	5	10	50	100	500	1000

These symbols allow to code numbers by symbol sequences, like I, II, III, IV, V, VI, VII, VIII, IX, X, XI, etc. A repetition of symbols is allowed for most symbols: XX=20, XXX=30, CC=200, CCC=300, but not for V, L and D. A lower symbol in front of a higher one means a subtraction: CM=900, XC=90.

Example: 1996 = MCMXCVI.

For us, the decimal system is more readable, simply because we are used to think in powers of 10 (we have 10 fingers; note that for example the French also utilize 20, which is the number of fingers and toes: 90 = quatre vingt dix). In general, a number is represented by a sequence of digits separated by a comma (or point in the English system):

$$N_r = d_j d_{j-1} \ldots d_3 d_2 d_1 d_0 . d_{-1} d_{-2} d_{-3} \ldots d_{-k+1} d_{-k}$$

where:
N = the number representation
r = the number base (or radix)
$d_j \ldots d_0$ = the integer part and
$d_{-1} \ldots d_{-k}$ = the fraction.

The parameter r can have *any* integer value ≥ 2, but only the values 2, 8, 10 and 16 are normally used. In the expression above we have the following digit weighting:

$$N_r = d_j \cdot r^j + \ldots + d_2 \cdot r^2 + d_1 \cdot r^1 + d_0 \cdot r^0 + d_{-1} \cdot r^{-1} + d_{-2} \cdot r^{-2} + \ldots + d_{-k} \cdot r^{-k}$$

In the case of the **decimal** system, $r = 10$ and we use the 10 symbols 0,1,2,3,...,9. *Example:*

$$1996 = 1 \cdot 10^3 + 9 \cdot 10^2 + 9 \cdot 10^1 + 6 \cdot 10^0 = 1 \cdot 1000 + 9 \cdot 100 + 9 \cdot 10 + 6 \cdot 1.$$

The **binary** system, with $r = 2$ and only the two symbols 0 and 1 is, since the appearance of computers, normal because it is very easy to make electronic circuits with only two stable states (voltages)[1]. The digits are called **bits** (from BInary digiTS). Sometimes we use the **octal** system (0,1,2,3,...,7; $r = 8$) or the **hexadecimal** system (0,1,2,...,8,9,A,B,C,D,E,F; $r = 16$), mainly for representing long series of bits. The most important representations are given in table 1.

[1] We note here that circuits with 3 or more stable states (voltages) can be made, but these require much more components like resistors and transitors. Binary circuits allow a much higher integration density, e.g. VLSI (Very Large Scale Integration).

decimal	binary	octal	hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Table 1: Common number codes.

2.1 Number conversions

Converting binary to/from octal and hexadecimal is fairly easy if we know table 1.

Examples:
$10011100101.01010_2 = 0100\ 1110\ 0101\ .\ 0101\ 0000_2 = 4E5.50_{16}$
$10011100101.01010_2 = 010\ 011\ 100\ 101\ .\ 010\ 100_2 = 2345.24_8$

The reason is that these systems are all based on powers of 2. It is different for other bases, like 10. The conversion is then based on successive divisions. If

$$N = d_3 \cdot r^3 + d_2 \cdot r^2 + d_1 \cdot r^1 + d_0 \cdot r^0$$

we have

$$N/r = d_3 \cdot r^2 + d_2 \cdot r^1 + d_1 + d_0/r.$$

The last term is a fraction, the **rest** d_0. Dividing successively all quotients, we obtain the other digits (rests) d_1, d_2 and d_3.

Example: Convert 430_{10} to binary; successive division by 2.

```
         430 215 107  53  26  13   6   3   1
quotient 215 107  53  26  13   6   3   1   0
    rest   0   1   1   1   0   1   0   1   1
```

Reading the rests from right to left, the result is 110101110_2. The division process stops when the quotient is 0.

If we have a fraction, we use successive multiplications instead of divisions. If

$$N = d_{-1} \cdot r^{-1} + d_{-2} \cdot r^{-2} + d_{-3} \cdot r^{-3} + d_{-4} \cdot r^{-4}$$

we have

$$rN = d_{-1} + d_{-2} \cdot r^{-1} + d_{-3} \cdot r^{-2} + d_{-4} \cdot r^{-3}.$$

The first term is the integer digit d_{-1} and successive multiplication of the fraction gives the other digits d_{-2}, d_{-3} and d_{-4}.

Example: Convert $.523_{10}$ to binary; successive multiplication by 2.

```
          .523  .046  .092  .184  .368  .736  .472  .944 etc
product  1.046 0.092 0.184 0.368 0.736 1.472 0.944 1.888
  digit  1     0     0     0     0     1     0     1
```

The 8-bit approximation result is $.10000101_2$ and the multiplication can continue until the product becomes 1.000. The more digits, the more accurate the conversion will be.

Example: Convert $.523_{10}$ to hexadecimal; successive multiplication by 16.

```
           .523     .368     .888  etc
          x  16    x  16    x  16
          -----    -----    -----
           3138     2208     5328
           523      368      888
         +-----   +-----   +-----
 product  8.368    5.888   14.208
   digit  8        5        E
```

This gives $.85E_{16}$, which equals $.100001011110_2$ (series 1000 0101 1110; see table 1).

Check: $.85E_{16}$ = ? in decimal.

2.2 Negative numbers

The numbers that we have seen above are all assumed to be positive. This means that in front of all numbers we can think a + sign, irrespective of the number basis (radix). If we want negative numbers, we put a – sign of course. This is quite natural for us, but not for computers! For binary numbers we can use three different forms.

2.2.1 Sign–magnitude

The unsigned number, called magnitude, is complemented with a sign bit s to the left of the most significant number bit. This sign bit is 0 for positive numbers and 1 for negative ones. Assuming 4-bit numbers sxxx with xxx=101 (the absolute value 5), we get 0101=+5 and 1101=–5.

2.2.2 One's and two's complements

In order to simplify the electronic circuits for addition and subtraction, the so-called 1's and 2's complement codes are used. Reason: a 2's complement subtraction needs only two circuits: an adder and an inverter, and an adder is required for additions (see Chapter 10).

The 1's complement is obtained by inverting all bits of the corresponding positive number:

$$-5 = -0101 = \bar{0}\bar{1}\bar{0}\bar{1} = 1010.$$

The 2's complement (the name comes from subtracting a fraction from 2) is obtained by inverting all bits and then adding a 1:

$$-5 = -0101 = \bar{0}\bar{1}\bar{0}\bar{1} + 1 = 1010 + 1 = 1011.$$

This means that the 2's complement equals the 1's complement plus 1. We therefore get table 2. From table 2 we can see:

decimal	sign-magnitude	1's complement	2's complement
+127	0111 1111	0111 1111	0111 1111
+126	0111 1110	0111 1110	0111 1110
+25	0001 1001	0001 1001	0001 1001
+2	0000 0010	0000 0010	0000 0010
+1	0000 0001	0000 0001	0000 0001
+0	0000 0000	0000 0000	0000 0000
–0	1000 0000	1111 1111	0000 0000
–1	1000 0001	1111 1110	1111 1111
–2	1000 0010	1111 1101	1111 1110
–25	1001 1001	1110 0110	1110 0111
–127	1111 1111	1000 0000	1000 0001
–128	–	–	1000 0000

Table 2: Common 8-bit number codes.

- The positive numbers are the same in all representations.
- The most significant bit (sign bit) is always 0 for positive and 1 for negative numbers.
- Only the 2's complement has the same representation for ± 0.
- The 2's complement allows to represent numbers that the other ones don't: –128.
- All this means that the 2's complement is to be preferred.
- Another way to compute the 2's complement is to scan the bits from right to left (from least significant to most significant), copying the bits until we have the first 1 and then inverting all bits. *Check this!*

2.3 Arithmetic

2.3.1 Addition

A binary addition is done in the same way as a decimal one: adding the digits going from right to left and adding a carry (a 1) to the digits to the left if required.

Examples:

```
    5+3 =  0000 0101      5+(-3) =  0000 0101
           0000 0011                1111 1101  2's complement
         +----------              +----------
           0000 1000 = 8           10000 0010 = 2 (ignore carry bit!)
                                   ^carry
```

2.3.2 Subtraction

The subtraction is done by using addition and the 2's complement:

$$N_1 - N_2 = N_1 + (-N_2) = N_1 + (\bar{N}_2 + 1).$$

Example:

```
   15-8 =  0000 1111
           1111 1000
         +----------
          10000 0111 = 7
```

NOTE: the carry bit is ignored if we use the 2's complement. By contrast, if we use the 1's complement, the carry bit needs to be added to the least significant bit. This is called an end-around carry. *Check this!*

2.3.3 Multiplication

The binary multiplication in the sign-magnitude form is done by determining the sign from the two signs of the operands (+x+=+, +x–=–, –x–=+; this an exclusive-OR operation, see Chapter 4) and putting this sign in front of the product of the magnitudes.

Example:

```
     5x107 =   110 1011
               x    101
               --------
               110 1011
              0000 000
            1 1010 11
           +-----------
 Sign ->  S 10 0001 0111 = 535
```

We see that, like in the decimal system, it is a repetition of shifts and additions (store intermediate results). If we have 2's complement numbers, one way to do the multiplication is to convert the number(s) to sign-magnitude, do the s-m multiplication, and then convert the result to the 2's complement. However, this is stupid because much more efficient ways are available (this will be treated in *Arquitectura de Computadores*).

2.3.4 Division

If a multiplication is based on a repetition of additions, it should be clear that a division is based on a repetition of subtractions. The easiest and trivial case is the sign-magnitude representation. Special hardware for multiplications and divisions is normally present in a computer, although in many systems like RISC (Reduced Instruction Set Computers) there are two circuits: a fixed-point multiplier and a circuit to compute the reciprocal (1/N), apart from the much more important floating-point unit (*Arquitectura de Computadores*).

2.3.5 Over/Underflow

Adding two N-bit numbers, the result may not be representable with the N bits available; in many cases N+1 bits are required. In this case an **overflow** occurs. This will be studied in Chapter 10. In general, this condition is detected by a special electronic circuit, and normally in an adder unit or ALU (Arithmetic and Logical Unit) of the CPU (Central Processing Unit) of a computer.

2.4 Codes

Not only numbers can be represented by sequences of 1's and 0's, also letters, characters and other symbols. Combinations of bits are called digital codes. Codes can be classified as follows:

- Numerical codes for representing numbers.
- Alphanumerical codes for representing numbers and characters.
- Special codes for error detection and correction.

We will look here at two numerical codes, i.e. the **8421** or **BCD** code and the **GRAY** code, as well as at an alphanumerical code, the **ASCII** code.

2.4.1 BCD

In the 8421 or BCD code (Binary Coded Decimal) each decimal digit is represented by a binary code of 4 bits. The weights of the bits are 8, 4, 2 and 1. The advantage of the BCD code is that it allows an easy conversion from binary to decimal formats and this code is often used in computer peripherals for entering or printing data. The BCD code is shown in table 3.

The other bit combinations (e.g. 1011) are not valid. Representing a decimal number in BCD requires a substitution of each decimal digit by its binary code.

Example: $39.5_{10} = 0011\ 1001\ .\ 0101_{\mathrm{BCD}}$

The reverse conversion requires grouping the bits in sets of 4, starting from the period or comma.

Example: $010100110010._{\mathrm{BCD}} = 0101\ 0011\ 0010. = 532_{10}$

decimal	8421
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Table 3: The BCD or 8421 code.

The **addition** in BCD code is done as follows:

1. Add the two numbers as is done in the binary addition.
2. If the sum of 4 bits is equal or smaller than 9, this is a valid BCD number.
3. If the sum is greater than 9 or if there is a carry, the result is not valid. In this case, we need to add 6 in order to skip the 6 invalid codes 1010 ... 1111 (see table 3). A carry must be added to the bit group to the left.

Example: 36+84=120

```
          0011 0110  =  36
          1000 0100  =  84
        +----------
          1011 1010  both invalid because > 9
          0110 0110  add 6 to each group
        +----------
     0001 0010 0000  = 120
```

2.4.2 GRAY

The Gray code allows to detect errors going from one position to the next/previous, because these differ in only one bit. It is used in systems which need to detect positions, for example in a computer mouse. The Gray code is also called mirror code, because of the way it is constructed (vertical mirroring around the –, changing 0's to 1's at the left):

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

Table 4 shows the complete 4-bit Gray code.

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

Table 4: The 4-bit Gray or mirror code.

In order to convert from binary to Gray, we need to:
1. Keep the MSB (most significant bit)
2. Going from left to right, take the sum of two adjacent bits, forgetting carries.

Example:

```
10010111  =  binary
11011100  =  Gray
```

In order to convert from Gray to binary, we need to:
1. Keep the MSB (most significant bit)
2. Sum the next Gray bit with the previous binary bit.

Example:

```
11011100  =  Gray
10010111  =  binary
 ^
 0=1(left binary)+1(above Gray)
```

2.4.3 ASCII

The ASCII code (American Standard Code for Information Interchange) is an alphanumerical code for communication, for example for connecting a keyboard to a computer. An alphanumerical code has at least representations for the 10 decimal numbers and the 26 characters of the alphabet. Together this is already 36 codes and requires 6 bits (6 bits corresponds to a maximum of 64 different bit combinations). It normally contains also special codes for manipulating the information stream. Examples of special codes are LF (line feed) and CR (carriage return), which serve to start a new line in the first column of a computer screen. Normally, the ASCII code takes 7 or 8 bits (8 bits is 1 byte), see table 5.

LSBs	MSBs							
	000	001	010	011	100	101	110	111
0000	NUL	DLE	SP	0	@	P	‘	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	”	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	’	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	–	=	M	]	m	}
1110	SO	RS	.	>	N	↑	n	˜
1111	SI	US	/	?	O	_	o	DEL

Table 5: The 7-bit ASCII code.

Most important abbreviations used in table 5:
MSBs: most significant bits, LSBs: least significant bits

ACK: acknowledge
BEL: bell
BS: back space
CR: carriage return
EOT: end of transmission
ESC: escape
ETB: end of transmission block
FF: form feed
LF: line feed
SP: space

Example: a small text string could be

```
hexadecimal: 4D 75 69 74 6F 20 70 72 61 7A 65 72 0A 0D
             M  u  i  t  o     p  r  a  z  e  r
                            SP                   LF CR
```