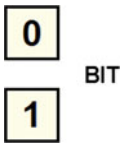
Chapter 3
Numeral Systems and Binary Arithmetic

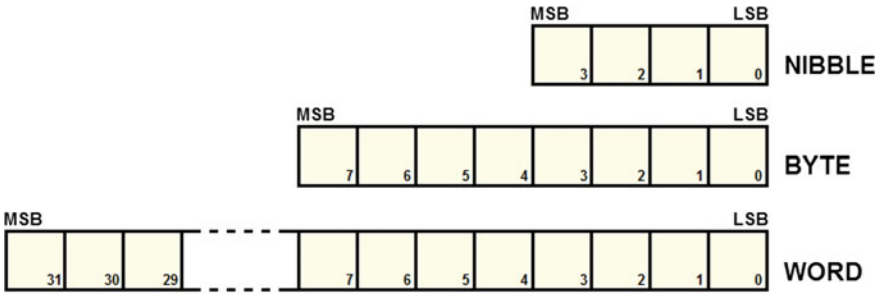
Abstract The representation of numbers is essential for the digital logic design. In this chapter, positional number systems (decimal, binary, octal, hexadecimal), BCD and Gray codes are presented together with the rules for the conversion between numbers encoded in different bases and the representations of negative numbers. Then, the rules for the arithmetic operations and the circuits that execute them are presented. The addition of binary number is examined with particular attention, since it is the operation at the basis of all computational circuits. Alphanumeric codes and the concept of parity for error detection complete the chapter.

3.1 Binary Information

The basic unit of binary information is called a *bit* (binary digit). The bit can only assume the value of 0 or 1.



Normally, bits are grouped into meaningful sets called *nibbles* (4 bits), *bytes* (8 bits), or generically *words* if they contain more bits (e.g., 16, 32, or 64 bits):



G. Donzellini et al., *Introduction to Digital Systems Design*,
https://doi.org/10.1007/978-3-319-92804-3_3

In this chapter, we will see how to use these sets of bits to represent numbers or other types of information through standard codes.

In the previous figure, the abbreviation MSB refers to the *Most Significant Bit* and LSB refers to the *Least Significant Bit*, when these sets are used to codify numbers.

3.2 Binary Numbering System (BIN)

Like the decimal system, binary numbering is *positional*. Positional notation makes it possible to express a number N as:

$$N = n_{m-1} \cdot R^{m-1} + \cdots + n_1 \cdot R^1 + n_0 \cdot R^0$$

where R is the *base* of the system; m is the number of digits the representation is composed of; the exponents denote the *position* k of each digit from 0 to $m-1$ starting from the right; the set $A = \{0, \ldots, R-1\}$ is the *alphabet* of the numbering system and is made up of *symbols* we use to represent numbers; coefficients n are the numbers that correspond to the value of the symbols in set A.

In the decimal system, the alphabet of symbols is $A = \{0, 1 \ldots, 9\}$ and, obviously, $R = 10$. When we write a number N, for example, 2017, we mean:

$$2017_{10} = 2 \cdot 10^3 + 0 \cdot 10^2 + 1 \cdot 10^1 + 7 \cdot 10^0$$

Let's refer to the *binary numbering system* as that system that has base $R = 2$ and alphabet $A = \{0, 1\}$. For example, the number 1110 in base two corresponds to the number 14 in base ten:

$$1110_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 4 + 2 + 0 = 14_{10}$$

Finally, let's denote *weight* as:

$$p = R^k$$

In the decimal system, weight refers to units, decimals, hundreds, thousands, etc., (powers of 10). In the binary numbering system, magnitudes have the power of base $R = 2$: 1, 2, 4, 8, 16, 32, 64, 128, etc.

Note: For a short guide about powers of 2, refer to Appendix A.

3.2.1 Converting from Binary System to Decimal

We obtain this conversion by directly applying the definition above. Here are examples for 4 and 8 digits:

$$0000_2 = 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 0_{10}$$

$$0011_2 = 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 3_{10}$$

$$1111_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 15_{10}$$

$$00000010_2 = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 2_{10}$$

$$10001110_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 142_{10}$$

$$11110011_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 243_{10}$$

$$11111111_2 = 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 255_{10}$$

3.2.2 Converting from Decimal System to Binary

A natural number N (with $N \geq 2$) can be expressed as:

$$N = 2 \cdot q_0 + r_0 \quad \text{with} \quad r_0 = (0, 1)$$

where q_0 is the quotient and r_0 is the remainder of the division of N by 2. We can repeat the process if $q_0 \geq 2$, by writing that:

$$q_0 = 2 \cdot q_1 + r_1 \quad \text{with} \quad r_1 = (0, 1)$$

By substituting the latter expression in the former one, we obtain:

$$N = 2 \cdot (2 \cdot q_1 + r_1) + r_0$$

Continuing, if $q_1 \geq 2$, we derive:

$$N = 2 \cdot (2 \cdot (2 \cdot q_2 + r_2) + r_1) + r_0 = 2^3 \cdot q_2 + 2^2 \cdot r_2 + 2^1 \cdot r_1 + 2^0 \cdot r_0$$

and so on as long as $q_k \geq 2$, giving us:

$$N = 2^k \cdot q_{k-1} + 2^{k-1} \cdot r_{k-1} + \ldots + 2^1 \cdot r_1 + 2^0 \cdot r_0.$$

In other words, the remainders r_i from divisions by 2 represent the binary digits of N. Consider $N_{10} = 46$, for example. Let's divide by 2 recursively, writing the remainder at the right of the line and the result below the number.

	46	0	(r_0)
(q_0)	23	1	(r_1)
(q_1)	11	1	(r_2)
(q_2)	5	1	(r_3)
(q_3)	2	0	(r_4)
(q_4)	1	1	(q_4)

We get the result of the conversion by writing in order, from the left, q_4 followed by the remainders r_4, r_3, ..., r_0:

$$N = 46_{10} = 101110_2$$

Proof

$$N = q_4 \cdot 2^5 + r_4 \cdot 2^4 + r_3 \cdot 2^3 + r_2 \cdot 2^2 + r_1 \cdot 2^1 + r_0 \cdot 2^0 =$$
$$= 1 \cdot 32 + 0 \cdot 16 + 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = 46_{10}$$

As an example, let's convert some decimal numbers into the binary system.

258_{10} :

258	0
129	1
64	0
32	0
16	0
8	0
4	0
2	0
1	$1 \rightarrow 100000010_2$

237_{10} :

237	1
118	0
59	1
29	1
14	0
7	1
3	1
1	$1 \rightarrow 11101101_2$

3.2.3 Maximum Representable Number

Given a set of m bits, the largest natural number that can be represented is:

$$N_{max} = 2^m - 1$$

For example, if we consider a byte ($m = 8$), where every bit equals 1, we will add all the magnitudes present among them.

$$11111111_2 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 =$$
$$= 255 = 256 - 1 =$$
$$= 2^8 - 1$$

3.3 Octal Number System (OCT)

This is a base $R = 8$ positional numbering system with alphabet $A = \{0, 1, 2, 3, 4, 5, 6, 7\}$. It is used to represent binary numbers but it is rarely used today, being replaced by the hexadecimal system. Nevertheless, it is still useful to be acquainted with it. Below is an example:

$$123_8 = 1 \cdot 8^2 + 2 \cdot 8^1 + 3 \cdot 8^0 = 64 + 16 + 3 = 83_{10}$$

The binary–octal conversion table, for a single octal digit:

BIN	OCT
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Since the octal number system's base is a power of 2, the conversion from binary to octal is immediate.

Converting any binary number into octal requires just dividing it into groups of three digits (starting from the right) and replacing each with the corresponding octal digit.

See the examples below:

$$\begin{aligned} 100100_2 &= |\,100_2\,|\,100_2\,| = 44_8 \\ 111111_2 &= |\,111_2\,|\,111_2\,| = 77_8 \\ 11001_2 &= |\,011_2\,|\,001_2\,| = 31_8 \\ 1101_2 &= |\,001_2\,|\,101_2\,| = 15_8 \end{aligned}$$

An integer decimal number can be converted into octal by using the method of repeated division seen above for decimal to binary conversion, in this case dividing by 8.

In this example we convert the number 267_{10} into octal:

$$\begin{array}{r|ll} 267 & 3 & (267 : 8 = 33 + 3) \\ 33 & 1 & (33 : 8 = 4 + 1) \\ 4 & 4 & (4 : 8 = 0 + 4) \end{array}$$

The result is: 413_8.

3.4 Hexadecimal Number System (HEX)

This is a positional number system with base $R = 16$. Its alphabet is:

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}.$$

To represent digits higher than 9 we use the first six letters of the Latin alphabet since they are available on a typical keyboard. A DEC − BIN − HEX conversion table is as follows.

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

Since the hexadecimal number system is also based on the power of 2, the BIN–HEX conversion is immediate. We replace every group of four binary digits with the corresponding HEX digits. Here are some examples:

$$\begin{aligned} 10001111_2 &= |\,1000_2\,|\,1111_2\,| = 8\mathrm{F}_{16} = 8\mathrm{F}_H \\ 11100011_2 &= |\,1110_2\,|\,0011_2\,| = \mathrm{E}3_{16} = \mathrm{E}3_H \\ 100101_2 &= |\,0010_2\,|\,0101_2\,| = 25_{16} = 25_H \\ 11100_2 &= |\,0001_2\,|\,1100_2\,| = 1\mathrm{C}_{16} = 1\mathrm{C}_H \end{aligned}$$

In this case the conversion can also be done with repeated division, by dividing by 16. In the following example, the number 655_{10} is converted into hexadecimal.

$$\begin{array}{r|l} 655 & 15 \quad (655 : 16 = 40 + 15) \\ 40 & 8 \quad (40 : 16 = 2 + 8) \\ 2 & 2 \quad (2 : 16 = 0 + 2) \end{array}$$

The result is: $28\mathrm{F}_H$.

To get familiar with hexadecimal numbers, let's look at some examples of column addition with two numbers. Note that when a digit is over 15 (not 9 as in the decimal system), we carry it over one column.:

28_H +	22_H +	40_H +	$0F_H$ +	$3A_H$ +
33_H =	AA_H =	51_H =	$0F_H$ =	$9B_H$ =
$5B_H$	CC_H	91_H	$1E_H$	$D5_H$

3.5 Others Binary Codes

3.5.1 Binary Coded Decimal

The most natural way to represent a number in a binary format is to use the pure binary numbering system. However, the downside of this is that decimal-to-binary and binary-to-decimal conversion becomes more taxing as the numbers get bigger.

To make the conversion easier, we can represent the digits of a decimal number one by one, using the so-called Binary Coded Decimal (BCD) codes. They code the decimal numbers' digits one by one using groups of four bits. Obviously, their arithmetic properties are less than those of the pure binary number system.

In general, binary codes are said to be *weighted* if every digit has its own weight according to its position. Those in which every combination of digits is randomly associated to a certain number are called *non-weighted*.

Self-complementing codes are those where two numbers that add up to 9 are complements to one of each other (i.e., the 1s and 0s are interchanged).

Now, let's look at some of the most commonly used BCD codes:

- BCD 8421: It is a weighted code in which the digits' weights are 8, 4, 2, 1 from left to right; this is equal to pure binary terminated at 1001_2.
- BCD 5421: It is analogous to BCD 8421 but the weights are 5, 4, 2, 1.
- AIKEN 2421: It is a weighted, self-complementing BCD code with weights of 2, 4, 2, 1.

N	BCD 8421	BCD 5421	AIKEN 2421
0	0000	0000	0000
1	0001	0001	0001
2	0010	0010	0010
3	0011	0011	0011
4	0100	0100	0100
5	0101	1000	1011
6	0110	1001	1100
7	0111	1010	1101
8	1000	1011	1110
9	1001	1100	1111

- BCD XS3: It is a non-weighted, self-complementing code that is obtained by adding 3 (11 in binary) to BCD 8421; XS3 actually means "excess 3".

N	BCDXS3
0	0011
1	0100
2	0101
3	0110
4	0111
5	1000
6	1001
7	1010
8	1011
9	1100

3.5.2 GRAY Codes

The GRAY codes are non-weighted. They are characterized by the fact that each number differs by one single digit from the one that precedes it and the one that follows it. GRAY codes can be made with any number of bits. There are many types of GRAY codes. The one shown here is the "reflected" type.

N	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

3.6 Binary Arithmetic

3.6.1 Addition

Consider the addition of two bits A and B. Let's evaluate the four possible cases:

$A +$	$0 +$	$0 +$	$1 +$	$1 +$
$B =$	$0 =$	$1 =$	$0 =$	$1 =$
sum	0	1	1	10_2

In the farthest right column, the result is obviously 2_{10} but it makes sense to interpret it as "result 0, with carry over 1". So, let's denote the bit of the sum as S and introduce the bit C_o (*Carry Out)*, which represents the result. In cases where there is no carry, we will write it as 0.

$A +$	$0 +$	$0 +$	$1 +$	$1 +$
$B =$	$0 =$	$1 =$	$0 =$	$1 =$
$C_o\ S$	0 0	0 1	0 1	1 0

When we add binary numbers coded on more than one bit, the carry generated from one column must be added to the result from the column to its immediate left, as in the following examples:

$0010_2 +$	$0001_2 +$	$0011_2 +$
$0001_2 =$	$0001_2 =$	$0001_2 =$
0011_2	0010_2	0100_2

In the first example, there are no carries; in the second, there is one in the farthest right column; in the last, the two right-hand columns have carries.

Based on these observations, we define the *addition rules* for a given column by introducing the carry C_i (*Carry In)*, which comes from the adjoining column.

$C_i +$	$0 +$	$0 +$	$0 +$	$0 +$
$A +$	$0 +$	$0 +$	$1 +$	$1 +$
$B =$	$0 =$	$1 =$	$0 =$	$1 =$
$C_o\ S$	0 0	0 1	0 1	1 0

C_i +	1 +	1 +	1 +	1 +
A +	0 +	0 +	1 +	1 +
B =	0 =	1 =	0 =	1 =
$C_o\ S$	0 1	1 0	1 0	1 1

See below some examples of binary number addition.

0001_2 +	0110_2 +	0111_2 +	01011111_2 +	01011110_2 +
0111_2 =	0110_2 =	0111_2 =	00100101_2 =	00101111_2 =
1000_2	1100_2	1110_2	10000100_2	10001101_2

When calculating the sum of two numbers, the result could exceed the maximum representable number. If, for example, in our logical network we have only 4 bits available to code a number, the following sums generate a result that is too large.

1111_2 +	0111_2 +	0101_2 +	1111_2 +
0001_2 =	1011_2 =	1100_2 =	1111_2 =
10000_2	10010_2	10001_2	11110_2

To contain the result, we need 5 bits. An *overflow* error has occurred. After calculating a sum, we must always check for an overflow error, that is if anything has carried over from the column of the MSB. This rule holds for numbers without signs. Next, we will see how to check for overflow with numbers that can be positive or negative.

3.6.2 Subtraction

We define the *subtraction rules* by using criteria similar to those for addition. In this case, a column can *borrow* from the column to the left if it is necessary. Due to similarities with real circuits, we will use the same symbol used for carry. C_i is what is *borrowed* from the column on the right, while C_o is what the present column *borrows* from the left.

C_i −	0 −	0 −	0 −	0 −
A −	0 −	0 −	1 −	1 −
B =	0 =	1 =	0 =	1 =
$C_o\ S$	0 0	1 1	1 1	1 0

C_i	−	1	−	1	−	1	−	1	−
A	−	0	−	0	−	1	−	1	−
B	=	0	=	1	=	0	=	1	=
C_o	S	0	1	0	0	0	0	1	1

Here are some examples of subtraction:

1001_2 −	0111_2 −	0110_2 −	1111_2 −	01011011_2 −
0011_2 =	0101_2 =	0001_2 =	0111_2 =	00100101_2 =
0110_2	0010_2	0101_2	1000_2	00110110_2

3.6.3 Products

The *product rules* are as follows (we will not deal with division):

$$0 \times 0 = 0$$
$$0 \times 1 = 0$$
$$1 \times 0 = 0$$
$$1 \times 1 = 1$$

Here are some examples of products (they are carried out in the familiar way but with the rules seen here). Note that in the partial results of the operation, we either copy the multiplicand when it is multiplied by 1, or we write all 0s when it is multiplied by 0.

1 1 ×	1 1 ×	1 1 0 ×
1 1 =	1 0 1 =	1 0 0 =
1 1	1 1	0 0 0
1 1 −	0 0 −	0 0 0 −
1 0 0 1	1 1 − −	1 1 0 − −
	1 1 1 1	1 1 0 0 0

3.7 BCD 8421 Arithmetic

When adding in BCD 8421 arithmetic, we keep in mind that every group of 4 bits codes a decimal number so the rules of carries should be the same as with decimal

representation. In the following example, when calculating in decimal on the left, there are no carries from the units to the tens. When calculating in binary, on the right, there is no carry from the units column and the result is still made up of BCD digits (since ≤ 9), so the result is correct.

$$\begin{array}{ll} 24_{10} + & 0010\ \ 0100 + \\ 42_{10} = & 0100\ \ 0010 = \\ \hline 66_{10} & 0110\ \ 0110 \end{array}$$

However, in the second example, when we add the BCD digits, we come out with a sum that is not a BCD digit. The result is not valid (see the sum in decimal on the left), so we must make adjustments.

$$\begin{array}{ll} 27_{10} + & 0010\ \ 0111 + \\ 35_{10} = & 0011\ \ 0101 = \\ \hline 62_{10} & 0101\ \ 1100 \end{array}$$

In the result, the number 1100_2 is not in BCD 8421 (it is greater than $9_{10} = 1001_2$). The method to correct this consists in adding 0110_2 (i.e., decimal 6) to the non-BCD number.

$$\begin{array}{r} 1100\ + \\ 0110\ = \\ \hline 1\ \ 0010 \end{array}$$

The four digits to the right are now BCD. The carry is considered to be added to the BCD digit to its immediate left. In the example:

$$\begin{array}{r} 0101 + \\ 0001 = \\ \hline 0110 \end{array}$$

So, the exact result of the BCD addition, with corrections, is:

$$62_{10} = 01100010_{bcd}$$

3.8 Binary Rational Numbers

The way to deal with binary rational numbers is similar to what is used for integer numbers. Let's see two examples of conversion:

$$\begin{aligned} 0.1011_2 &= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 1 \cdot 2^{-4} = \\ &= 0.5 + 0.125 + 0.0625 = \\ &= 0.6875_{10}. \end{aligned}$$

To convert DEC into BIN, we do the following:

$$0.6875 \cdot 2 = 1.3750$$

The integer part constitutes the binary number, and the decimal part is multiplied again by 2. Continuing:

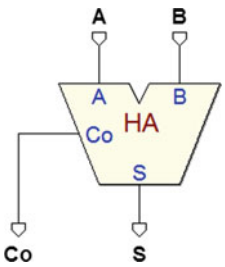
$$\begin{aligned} 0.375 \cdot 2 &= 0.750 \\ 0.750 \cdot 2 &= 1.500 \\ 0.500 \cdot 2 &= 1.000 \end{aligned}$$

we have come back to the initial number: 0.1011_2.

3.9 Arithmetic Networks

3.9.1 Half Adders

Half adders add two single-digit binary numbers generating a result and a carry. From the *Deeds* library:



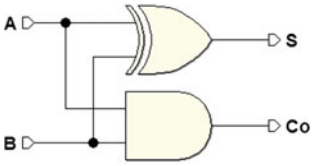
Using the addition rules seen earlier, let's complete the truth table. The addends are A and B, the sum S, and the carry C_o:

A	B	C_o	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Without the use of a map, we immediately recognize the functions in the table:

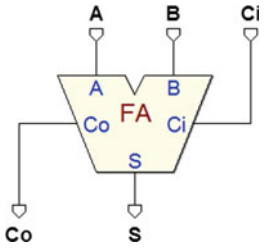
$$C_o = A \cdot B \qquad \text{e} \qquad S = A \oplus B$$

So the logical network of the half adder will look like this:



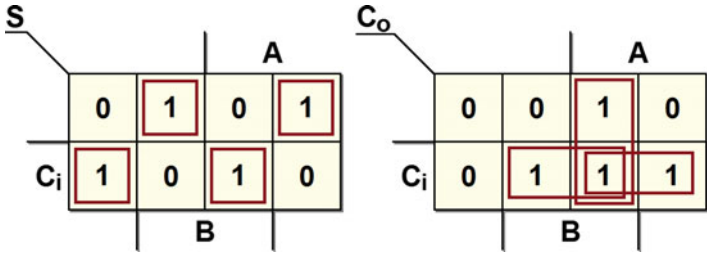
3.9.2 *Full Adders*

Full adders extend the possibilities of half adders by adding the carry from the preceding sum to the input.



Let's derive the truth table from the addition rules. A and B are the addends, C_i is the carry from the previous addition, S is the sum, and C_o is the carry:

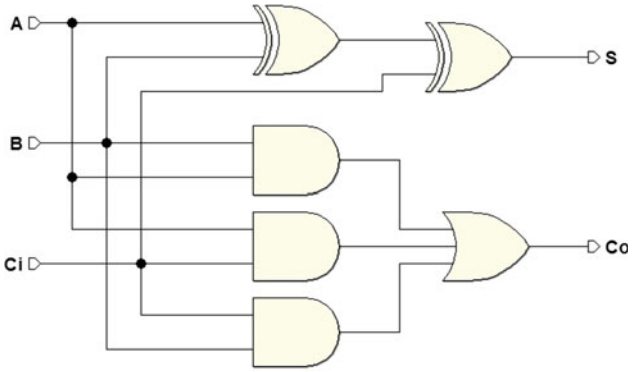
C_i	A	B	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1



From the maps, we derive the synthesis:

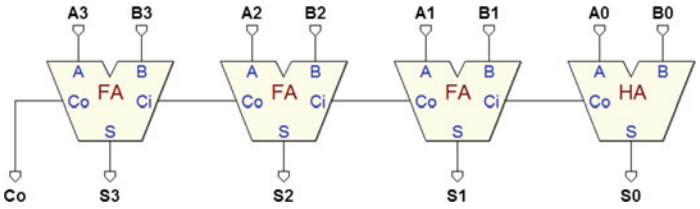
$$\begin{aligned} S &= \overline{A}\,\overline{B}\,C_i + \overline{A}\,B\,\overline{C}_i + A\,\overline{B}\,\overline{C}_i + A\,B\,C_i = \\ &= \overline{A}\,(\overline{B}\,C_i + B\,\overline{C}_i) + A\,(\overline{B}\,\overline{C}_i + BC_i) = \\ &= \overline{A}\,(B \oplus C_i) + A\,(\overline{B \oplus C_i}) = \\ &= A \oplus (B \oplus C_{IN}) \\ C_o &= A\,B + A\,C_i + C_i\,B \end{aligned}$$

and then the logical schematic:



3.9.3 *Ripple Carry Adder*

The network in the figure below (*ripple carry adder*) allows us to calculate the sum of two 4-bit numbers, and it is extendable to any number with m bits:



Because the carries are propagated from one stage to another, the execution time of a sum T_s is proportional to m.

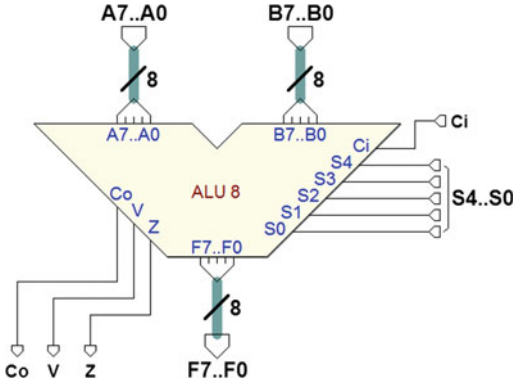
$$T_s = (m-1) \cdot t_{fa} + t_{ha}$$

where t_{fa} is the propagation delay of the full adders (FA), and t_{ha} that of the half adders (HA). The C_o, and therefore the sum, is valid only after all the intermediate carries are propagated along the network.

This is why, when we need high-speed calculation on a large number of bits, we rely on more efficient architectures where the carries are calculated, not in ripple fashion but in parallel, as in the *Carry Look Ahead* networks (not dealt with in this book).

3.9.4 Arithmetic Logic Unit (ALU)

The ALU is a combinational network that makes it possible to do different operations on two binary numbers (A and B). The figure below shows an example of an 8-bit ALU (from the *Deeds* library):



The operands are $A7..A0$ and $B7..B0$, while the result is taken from the outputs $F7..F0$. For addition and subtraction, there is one input Ci and one carry output Co.

Outputs V and Z are also available: $V = 1$ shows that the arithmetic operation gave rise to an *overflow*, while $Z = 1$ is activated when the result of the operation is zero. The operations are selected by the group of inputs $S4..S0$, according to the table below:

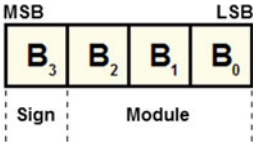
$S4..S0$	$Function$	$Notes$
00000	$F = 0$	
00001	$F = +1$	
00010	$F = -1$	
00011	$F = -128$	Minimum negative value
00100	$F = A$	
00101	$F = B$	
00110	$F = \overline{A}$	Ones' complement of A
00111	$F = \overline{B}$	Ones' complement of B
01000	$F = A \; and \; B$	(bitwise)
01001	$F = A \; and \; \overline{B}$	(bitwise)
01010	$F = \overline{A} \; and \; B$	(bitwise)
01011	$F = \overline{A \; or \; B}$	(bitwise)
01100	$F = A \; or \; B$	(bitwise)
01101	$F = A \; or \; \overline{B}$	(bitwise)
01110	$F = \overline{A} \; or \; B$	(bitwise)
01111	$F = \overline{A \; and \; B}$	(bitwise)
10000	$F = A \oplus B$	(bitwise)
10001	$F = \overline{A \oplus B}$	(bitwise)
10010	$F = \overline{A} + 1$	Two's complement of A
10011	$F = \overline{B} + 1$	Two's complement of B
10100	$F = A + 1$	Increment of A
10101	$F = B + 1$	Increment of B
10110	$F = A - 1$	Decrement of A
10111	$F = B - 1$	Decrement of B
11000	$F = A + B$	Addition
11001	$F = A + B + Ci$	Addition (with input carry)
11010	$F = sat\,(A + B)$	Saturating Addition
11011	$F = A - B$	Subtraction
11100	$F = A - B - Ci$	Subtraction (with input borrow)
11101	$F = sat\,(A - B)$	Saturating Subtraction
11110	$F = B - A$	Reversed Subtraction
11111	$F = B - A - Ci$	Reversed Subtraction (with input borrow)

3.10 Relative Numbers in Binary

There are many methods to represent relative numbers (signed numbers) in binary. In general, positive numbers are represented as if they had no sign whereas methods differ as to how negative numbers are coded. When the codifications we use change, so change the rules to manage the operations involving negative numbers.

3.10.1 Representation in "Module and Sign" Code

Let's consider a 4-bit packet: we use the MSB for the sign bit and the rest for the module:



We code the sign *minus* with 1 and the sign *plus* with 0. We get the following table:

B_3	B_2	B_1	B_0	Dec	
0	1	1	1	+7	
0	1	1	0	+6	
0	1	0	1	+5	
0	1	0	0	+4	
0	0	1	1	+3	
0	0	1	0	+2	
0	0	0	1	+1	
0	0	0	0	+0	
1	0	0	0	−0	(!)
1	0	0	1	−1	
1	0	1	0	−2	
1	0	1	1	−3	
1	1	0	0	−4	
1	1	0	1	−5	
1	1	1	0	−6	
1	1	1	1	−7	

There are some downsides to this representation:

1. The zero has two different codes: 0000 and 1000.
2. The code has bad arithmetic properties.

As shown in the following examples, we cannot use a normal adder.

$$\begin{array}{rl} 3_{10} + & 0011_2 + \\ 4_{10} = & 0100_2 = \\ \hline 7_{10} & 0111_2 \quad \text{(correct)} \end{array} \qquad \begin{array}{rl} 4_{10} + & 0100_2 + \\ 4_{10} = & 0100_2 = \\ \hline -0_{10} & 1000_2 \quad \text{(overflow)} \end{array}$$

$$\begin{array}{rl} 7_{10} + & 0111_2 + \\ -2_{10} = & 1010_2 = \\ \hline 5_{10} & 10001_2 \quad \text{(carry)} \end{array}$$

So, in a *module and sign* representation, we would need to use a more complex, tailor-made adder.

3.10.2 Complementation

The complement to the base

Let the number N in base R be represented in a positional notation with m digits:

$$N = n_{m-1} \cdot R^{m-1} + n_{m-2} \cdot R^{m-2} + \cdots + n_1 \cdot R^1 + n_0 \cdot R^0$$

We define the complement to the base R of the number N:

$$C_R(N) = R^m - N$$

Let's now consider another number Q, represented with the same base and the same number of digits. By adding the number Q to $C_R(N)$, we obtain the difference between the two plus the term R^m, which represents a carry outside the number's format.

$$Q + C_R(N) = Q + (R^m - N) = (Q - N) + R^m$$

Bringing the difference $Q - N$ to the left:

$$Q - N = Q + C_R(N) - R^m$$

Let's now look at an example with decimal numbers. $R = 10$, so the complement is "to ten"; we use only two digits ($m = 2$), and assume $Q = 48_{10}$ e $N = 12_{10}$.

$$48_{10} - 12_{10} = 48_{10} + C_{10}(12_{10}) - 10^2$$

By applying the definition of the complement of the base to our example:

$$C_{10}(12_{10}) = 10^2 - 12 = 88$$

We get:

$$48_{10} - 12_{10} = 48_{10} + 88_{10} - 10^2 = 136_{10} - 10^2 = 36_{10}$$

Despite the apparent complication, we have a great advantage: the negative number has been substituted by its complement to ten, which is positive. Then taking away 10^2 from the result is very simple, as we will soon see, since it does not require subtraction.

In an arithmetic network, we can do without subtractors and use only adders as long as we know an easy way to calculate the complement to the base of a number. Note that the complement operation is, from the point of view of the calculation, the same as changing the sign of a number.

Let's go back to our example and do the addition in column:

	48 +
	88 =
carry 1	36

At this point, taking away 10^2 is simple: we only need to ignore the carry. The result is what we expect: $36_{10} = 48_{10} - 12_{10}$.

Note that it is necessary to represent all the numbers with the same m. For example, let's use the same quantities as before but represented on eight digits ($m = 8$):

$$C_{10}(12_{10}) = 10^8 - 12_{10} = 99999988_{10}$$

	00000048 +
	99999988 =
carry 1	00000036

The carry exceeds the m digits: taking away R^m from the result means just ignoring it.

Now let's look at binary numbers with $R = 2$. We'll use the 4-bit format ($m = 4$), and we'll evaluate the complement *to two* of $N = 0101_2 = 5_{10}$ from the definition:

$$C_2(0101_2) = 2^m - N = 10000_2 - 0101_2 = 1011_2$$

As we'll see below, the $C_2(N)$ can be calculated more quickly using the complement of "base minus one".

The complement to "base minus one"

We define *the complement to base minus one (complement to one)* of the number N:

$$C_{R-1}(N) = (R^m - 1) - N$$

Let's compare this definition with that of the complement of the base. The result is:

$$C_R(N) = C_{R-1}(N) + 1$$

Calculating the complement to the base minus one is simpler than calculating the complement to the base. Therefore, to obtain the complement to the base, we prefer to calculate the complement to the base minus one and then add 1.

Let's look at an example in decimal with $N = 12_{10}$. Let's represent the number with eight digits and calculate the complement to nine (base ten minus one), by first evaluating the term:

$$(R^m - 1) = 10^8 - 1 = 99999999$$

This number is composed only of the digit "9" repeated m times. The complement to nine of 12 is:

$$C_9(12) = 99999999 - 00000012 = 99999987$$

It is simpler to calculate the complement to nine because when we subtract the number we never need to borrow. Then, by adding 1 to the result, we get the complement to ten.

Le's take the example of the binary number $N = 0101_2 = 5_{10}$ seen above ($M = 4$). To evaluate ones' complement (base two minus one), we first calculate the term:

$$(R^m - 1) = 2^m - 1 = 10000_2 - 1 = 1111_2$$

The situation is similar to the one before: this number is composed of the digit "1" repeated m times. So, ones' complement of 0101_2 is:

$$C_1(0101_2) = 1111_2 - 0101_2$$

To make this clearer, let's do the subtraction in column.

$$\begin{array}{rr} & 1111_2 - \\ & 0101_2 = \\ \hline C_1(0101_2) & 1010_2 \end{array}$$

We see that when we subtract the number from a number made only of "1", there is no need to borrow. To calculate the result, all we need to do is replace all the "1" with "0" and vice versa. From a circuital perspective, this means negating all the bits the number N is composed of.

$$C_1(N) = \overline{N}$$

From what we have seen, by adding $+1$ to the number obtained, we get the two's complement:

$$C_2(N) = C_1(N) + 1 = \overline{N} + 1$$

3.10.3 Representation in "Ones' Complement" Code

We can represent negative binary numbers through ones' complement (C_1). In the following table, we see an example for 4-bit numbers. The positive numbers are coded in pure binary, leaving the most significant digit at 0. The negatives are the C_1 of the corresponding positive number calculated according to the criteria outlined above.

To generate the code of a negative number, as above, we just invert all the bits of the corresponding positive number, for example:

B_3	B_2	B_1	B_0	Dec	
0	1	1	1	+7	
0	1	1	0	+6	
0	1	0	1	+5	
0	1	0	0	+4	
0	0	1	1	+3	
0	0	1	0	+2	
0	0	0	1	+1	
0	0	0	0	+0	
1	1	1	1	−0	(!)
1	1	1	0	−1	
1	1	0	1	−2	
1	1	0	0	−3	
1	0	1	1	−4	
1	0	1	0	−5	
1	0	0	1	−6	
1	0	0	0	−7	

$$-3_{10} = C_1(3_{10}) = C_1(0011_2) = 1100_2$$

Arithmetic Properties

- The sum of two positives:

$$\begin{array}{ll} +2 + & 0010 + \\ +5 = & 0101 = \\ \hline +7 & 0111 \end{array}$$

- The sum of a positive and a negative with a positive result:

$$\begin{array}{ll} +5 + & 0101 + \\ -2 = & 1101 = \\ \hline +3 & 10010 \quad (\text{“End Around Carry”}) \rightarrow 0010 + 1 = 0011 \end{array}$$

- The sum of a positive and a negative with a negative result:

$$\begin{array}{ll} -5 + & 1010 + \\ +2 = & 0010 = \\ \hline -3 & 1100 \end{array}$$

- The sum of two negatives:

$$\begin{array}{ll} -3 + & 1100 + \\ -3 = & 1100 = \\ \hline -6 & 11000 \quad (\text{“End Around Carry”}) \rightarrow 1000 + 1 = 1001 \end{array}$$

The carry “outside” the number format should be added to the LSB of the result. C_P denotes the carry to the MSB and C_S the carry outside the MSB.

Let’s consider the sum of two numbers $Q + N$; the table below shows us the values of C_P and C_S for all the combinations of the signs of addends and result, when the sum is correct, i.e., when the number’s format is able to contain the result:

Q	N	Sum	C_P	C_S
+	+	+	0	0
+	−	+	1	1
+	−	−	0	0
−	−	−	1	1

If C_P and C_S are different, there is an *overflow* error, and we can detect it as $OVF = C_P \oplus C_S$. Here is an example of an overflow error:

$$\begin{array}{ll} +7 + & 0111 + \\ +1 = & 0001 = \\ \hline +8 & 1000 \quad C_P = 1, C_S = 0, \rightarrow OVF \end{array}$$

3.10.4 Representation in "Two's Complement" Code

The most commonly used method for representing negative binary numbers relies on two's complement (C_2). In the table on the right, we find an example of code for four-bit numbers.

The positive numbers here are coded in pure binary as before, leaving the most significant digit at 0, while the negative numbers are calculated as two's complement of the corresponding positive number.

B_3	B_2	B_1	B_0	Dec
0	1	1	1	+7
0	1	1	0	+6
0	1	0	1	+5
0	1	0	0	+4
0	0	1	1	+3
0	0	1	0	+2
0	0	0	1	+1
0	0	0	0	+0
1	1	1	1	−1
1	1	1	0	−2
1	1	0	1	−3
1	1	0	0	−4
1	0	1	1	−5
1	0	1	0	−6
1	0	0	1	−7
1	0	0	0	−8

For practice, let's calculate the number -1_{10} starting from $+1_{10}$:

$$C_2(0001) = C_1(0001) + 1 = 1110 + 1 = 1111$$

If we calculate C_2 of zero, we get zero again (the zero in C_2 has a *univocal representation*). Note that the operation C_2 changes the number's sign: if N is positive, $C_2(N)$ is negative and vice versa.

Also, if we perform $C_2(C_2(N))$, we get the number N again:

$$C_2(C_2(N)) = 2^m - (C_2(N)) = 2^m - (2^m - N) = N$$

Arithmetic Properties

Let's look again at the carry into the MSB (C_P) and the carry outside the number (C_S). The same considerations for representation in C_1 also hold for C_2. When adding two numbers Q and N, the table used for checking the carries for *overflow* errors still holds:

Q	N	Sum	C_P	C_S
+	+	+	0	0
+	−	+	1	1
+	−	−	0	0
−	−	−	1	1

As before, there is overflow when C_P and C_S are different: $OVF = C_P \oplus C_S$.

C_2 code allows us to execute additions regardless of the sign of the addends by using a normal binary adder with no need to make corrections to the result (unlike with C_1).

3.10.5 Sign Extension

A signed binary number represented in C_2 code (or in C_1), over m bits, can be extended to a larger number of bits $v > m$, provided that the sign and value are preserved. Let's consider a positive number, for example 6_{10}, represented on four bits and its corresponding negative in C_2:

$$6_{10} = 0110_2 \qquad -6_{10} = 1010_2$$

Let's examine the positive number and consider the sign as an integral part of the number. In positional notation, it is:

$$0110_2 = 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

If we represent it with 8 bits, the positional notation will be:

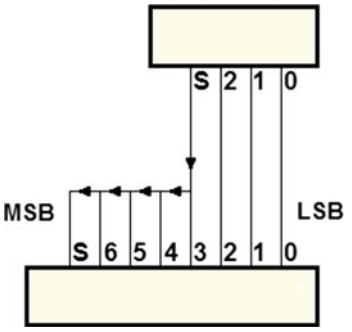
$$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 00000110_2$$

We have actually added non-significant zeroes to the left of the number, as you might expect. Still, this method cannot work for negative numbers, first of all because we will change the sign to positive but also because we will also change the value.

So, let's evaluate $C_2(6_{10})$, represented with 8 bits.

$$C_2(00000110_2) = C_1(00000110_2) + 1 = 11111001_2 + 1 = 11111010_2$$

As we can see, to extend a negative number to a larger number of bits, we only need to add 1s (rather than 0s) to the left. In other words, in either case, we must add digits with a value equal to the sign on the left. From a circuital perspective, the extension of the sign is translated into a very simple network.

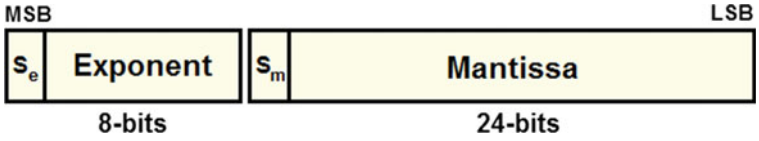


3.11 Representation of Real Numbers

There are essentially two methods to deal with real numbers in binary arithmetic.

- *Fixed point.* To assign a certain number of bits for the integer part of the number and the others for the fractional part. For example:

- *Floating point.* With this method, the bits available (32, in the example) are divided as follows:



In the figure, S_e is the sign of the exponent, and S_m is the sign of the mantissa. There is a "normalized" representation with a $0, \ldots$-type mantissa and a 2^{Exp}-type exponent, for example: $-0, 10110001011011_2 * 2^{00101111_2}$.

3.12 Alphanumeric Codes

Alphanumeric codes allow us to represent uppercase and lowercase letters, the ten decimal numbers, punctuation marks, and the so-called special symbols. There are codes that allow the writer to use the characters of almost all the written languages in the world, including Chinese and Japanese. The most common among these is Unicode.[1] This type of code is very complex and is not organized only on simple correspondence tables but on libraries of software supported by modern environments for applications development. An explanation of Unicode goes beyond the scope of this book.

One code that is still rather commonly used and relatively simple is *ASCII*,[2] which has 7 significant bits in the standard version. It codes the 26 uppercase and lowercase letters of the English language, the 10 decimal numbers, punctuation, and the symbols used in that language.

It also includes a certain number of communication codes, the so-called non-printable characters, which are used only sparingly in modern systems. Below are two tables of ASCII code characters: "non-printable" and printable.

ASCII: Not Printable Characters					
Dec	Code	Description	Dec	Code	Description
0	NULL	(Null character)	16	DLE	(Data link escape)
1	SOH	(Start of Header)	17	DC1	(Device control 1)
2	STX	(Start of Text)	18	DC2	(Device control 2)
3	ETX	(End of Text)	19	DC3	(Device control 3)
4	EOT	(End of Transmiss	20	DC4	(Device control 4)ion)
5	ENQ	(Enquiry)	21	NAK	(Negative acknowledgement)
6	ACK	(Acknowledgemen	22	SYN	(Synchronous idle)t)
7	BEL	(Bell)	23	ETB	(End of transmission block)
8	BS	(Backspace)	24	CAN	(Cancel)
9	HT	(Horizontal Tab)	25	EM	(End of medium)
10	LF	(Line feed)	26	SUB	(Substitute)
11	VT	(Vertical Tab)	27	ESC	(Escape)
12	FF	(Form feed)	28	FS	(File separator)
13	CR	(Carriage return)	29	GS	(Group separator)
14	SO	(Shift Out)	30	RS	(Record separator)
15	SI	(Shift In)	31	US	(Unit separator)

[1] http://www.unicode.org

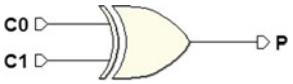
[2] American Standard Code for Information Interchange

ASCII: Printable Characters					
Dec	Code	Description	Dec	Code	Description
32		Space	80	P	Capital P
33	!	Exclamation mark	81	Q	Capital Q
34	”	Quotation mark	82	R	Capital R
35	#	Number sign	83	S	Capital S
36	$	Dollar sign	84	T	Capital T
37	%	Percent sign	85	U	Capital U
38	&	Ampersand	86	V	Capital V
39	’	Apostrophe	87	W	Capital W
40	(	round brackets	88	X	Capital X
41	)	round brackets	89	Y	Capital Y
42	*	Asterisk	90	Z	Capital Z
43	+	Plus sign	91	[	square brackets
44	,	Comma	92	\	Backslash
45	-	Hyphen	93	]	square brackets
46	.	Dot	94	⌒	Circumflex accent
47	/	Slash	95	_	underscore
48	0	number zero	96	‘	Grave accent
49	1	number one	97	a	Lowercase a
50	2	number two	98	b	Lowercase b
51	3	number three	99	c	Lowercase c
52	4	number four	100	d	Lowercase d
53	5	number five	101	e	Lowercase e
54	6	number six	102	f	Lowercase f
55	7	number seven	103	g	Lowercase g
56	8	number eight	104	h	Lowercase h
57	9	number nine	105	i	Lowercase i
58	:	Colon	106	j	Lowercase j
59	;	Semicolon	107	k	Lowercase k
60	<	Less-than sign	108	l	Lowercase l
61	=	Equals sign	109	m	Lowercase m
62	>	Greater-than sign	110	n	Lowercase n
63	?	Question mark	111	o	Lowercase o
64	@	At sign	112	p	Lowercase p
65	A	Capital A	113	q	Lowercase q
66	B	Capital B	114	r	Lowercase r
67	C	Capital C	115	s	Lowercase s
68	D	Capital D	116	t	Lowercase t
69	E	Capital E	117	u	Lowercase u
70	F	Capital F	118	v	Lowercase v
71	G	Capital G	119	w	Lowercase w
72	H	Capital H	120	x	Lowercase x
73	I	Capital I	121	y	Lowercase y
74	J	Capital J	122	z	Lowercase z
75	K	Capital K	123	{	curly brackets
76	L	Capital L	124	|	vertical-bar
77	M	Capital M	125	}	curly brackets
78	N	Capital N	126	~	Tilde ; swung dash
79	O	Capital O	127	DEL	Delete

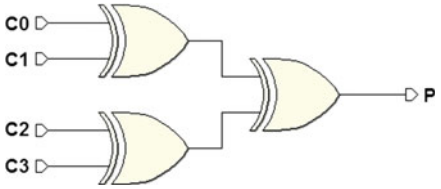
3.13 Error Detection Codes: Parity Generator and Checker

Assume that in an *8-bit* system, we are using *ASCII standard code* (that only has *7 significant bits*, denoted here as $C6 \ldots C0$). So, we can use the eighth bit to create an *"error detection"* code by using it as a *"parity bit"*. In this way, we add information which is redundant for coding ASCII, but useful to control *the integrity of the data*.

With $P = 1$ at the output, an XOR gate signals the presence of an odd number of ones in the input.



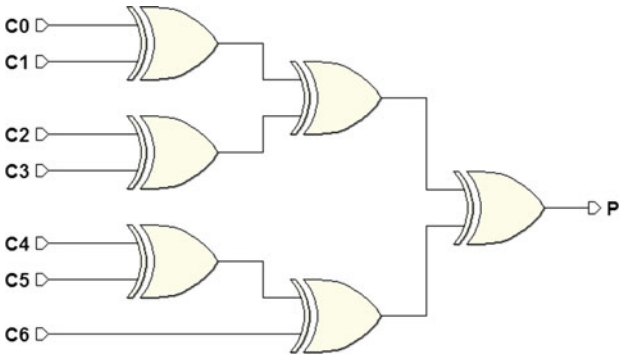
Thus, operation is called *"parity check"*. Through a tree structure, we can extend the parity check to any number of inputs.



The truth table of the 4-input XOR tree will show a 1 for each input combination with an odd number of ones:

C3	C2	C1	C0	P
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

In the figure below, a 7-bit parity check processes bits $C6 \ldots C0$ of the input data word, producing a 1 in output P if the set has an odd number of ones (*"odd parity"*). This structure is called a *"parity generator"*:

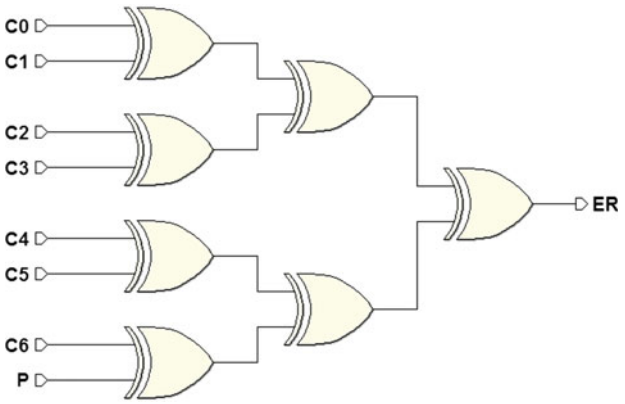


The 8-bit data word, composed of P and $C6 \ldots C0$ has an even number of ones (*"even parity"*). In the set of bits of the data word, P is called the *"parity bit"*.

P	C6	C5	C4	C3	C2	C1	C0

Associating a *redundancy* (in this case, a *parity bit*) to an original data word allows us to perform a quality check on it when data is moved from one system to another. The distance between systems, the characteristics of the communication channel, and the presence of noise all degrade the quality of the signal along its path, to the point that might damage its contents. Some bits can be received wrong. In real systems, errors are always present. They cannot be completely avoided but, with the right techniques, the probability of their happening can be greatly reduced. The error rate can be studied in statistical terms.

We introduce an analogous structure on the receiver side to verify that data parity has been preserved. We recalculate the parity of the received bits $C0 \ldots C7$ and also include bit P (see the figure below).



If no error has been detected, output ER (*Error*) will be zero. If ER is 1, it means that parity has not been preserved due to an error. Note that the check is accurate only if the error involves *one bit*: if there are *two* affected bits, for example, ER does not show it.

Thus, this is only useful if the probability of error is low. If we assume the probability of one bit error out of 10^4 transmitted, the probability of an error on two bits will be 10^8 bits (product of the two).

3.14 Exercises

3.14.1 Binary Numbers

1. Write 168_{10} and 143_{10} as binary numbers.
2. Calculate the decimal value of the following binary numbers:
   (a) 1110111_2
   (b) 101011101001011_2
3. Do the following calculations:
   (a) $11001101_2 + 1010101_2$
   (b) $1011011_2 - 101110_2$
   (c) $1001001_2 - 10101_2$
4. Multiply these binary numbers:
   (a) 1110_2 e 1011_2
   (b) 1011_2 e 101_2
   (c) 11100_2 e 011_2
   (d) 110_2 e 1111_2

3.14.2 Signed Binary Numbers

1. Do these calculations in two's complement with 7 bits:
   (a) $15 + 11$
   (b) $15 + (-11)$
   (c) $15 - 11$
   (d) $4 - 11$
   (e) $29 + 17$
   (f) $29 - 17$
   (g) $29 + (-17)$
   (h) $(-11) - 29$
   (i) $8 + (-18)$
   (j) $7 - 17$
2. Do these calculations in two's complement using the minimum number of bits to avoid overflow.
   (a) $3 + (-7)$

(b) $-3+7$
(c) $11+(-11)$
(d) $18-(-3)$

3.14.3 Octal and Hexadecimal Numbers

1. Convert the following numbers from octal to hexadecimal.
   (a) 276534_8
   (b) 22017555724_8
2. Convert these decimals into binary and hexadecimals.
   (a) 7722_{10}
   (b) 1435_{10}
3. Convert these decimals into octals and hexadecimals.
   (a) 36625_{10}
   (b) 124_{10}
4. Calculate the decimal values of these hexadecimal numbers.
   (a) $1A2B07_{16}$
   (b) 11047_{16}
5. Calculate the decimal values of these octal numbers.
   (a) 3111_8
   (b) 276534_8
6. Solve the equations of these hexadecimal numbers.
   (a) $41AB7_{16}+C2D6F_{16}$
   (b) $A23CE_{16}+363E6_{16}$

3.15 Solutions

3.15.1 Binary Numbers

1.

168	0		143	1	
84	0		71	1	
42	0		35	1	
21	1		17	1	
10	0		8	0	
5	1		4	0	
2	0		2	0	
1	1	$\rightarrow 10101000_2$	1	1	$\rightarrow 10001111_2$

2. (a) $1110111_2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + = 119_2$

   (b) $1010111010010111_2 = 22347_{10}$

3. (a)
$$\begin{array}{r} 11001101 + \\ 1010101 = \\ \hline 100100010 \end{array}$$

   (b)
$$\begin{array}{r} 1011011 - \\ 101110 = \\ \hline 0101101 \end{array}$$

   (c)
$$\begin{array}{r} 1001001 - \\ 10101 = \\ \hline 110100 \end{array}$$

4. (a)
$$\begin{array}{r} 1110 \times \\ 1011 = \\ \hline 1110 \\ 11100 \\ 000000 \\ 1110000 \\ \hline 10011010 \end{array}$$

   (b)
$$\begin{array}{r} 1011 \times \\ 101 = \\ \hline 1011 \\ 00000 \\ 101100 \\ \hline 110111 \end{array}$$

   (c)
$$\begin{array}{r} 11100 \times \\ 00011 = \\ \hline 11100 \\ 111000 \\ 0000000 \\ 00000000 \\ 000000000 \\ \hline 1010100 \end{array}$$

   (d)
$$\begin{array}{r} 110 \times \\ 1111 = \\ \hline 110 \\ 1100 \\ 11000 \\ 110000 \\ \hline 1011010 \end{array}$$

3.15.2 Signed Binary Numbers

1. (a)
$$\begin{array}{r} 15 + \\ 11 = \\ \hline 26 \end{array} \qquad \begin{array}{r} 0001111 + \\ 0001011 = \\ \hline 0011010 \end{array}$$

(b)
$$\begin{array}{r} 15 + \\ -11 = \\ \hline 4 \end{array} \qquad \begin{array}{r} 0001111 + \\ 1110101 = \\ \hline 0000100 \end{array}$$

(c)
$$\begin{array}{r} 15 - \\ 11 = \\ \hline 4 \end{array} \qquad \begin{array}{r} 0001111 - \\ 0001011 = \\ \hline 0000100 \end{array}$$

(d)
$$\begin{array}{r} 4 - \\ 11 = \\ \hline -7 \end{array} \qquad \begin{array}{r} 0000100 - \\ 0001011 = \\ \hline 1111001 \end{array}$$

(e)
$$\begin{array}{r} 29 + \\ 17 = \\ \hline 46 \end{array} \qquad \begin{array}{r} 0011101 + \\ 0010001 = \\ \hline 0101110 \end{array}$$

(f)
$$\begin{array}{r} 29 - \\ 17 = \\ \hline 12 \end{array} \qquad \begin{array}{r} 0011101 - \\ 0010001 = \\ \hline 0001100 \end{array}$$

(g)
$$\begin{array}{r} 29 + \\ -17 = \\ \hline 12 \end{array} \qquad \begin{array}{r} 0011101 + \\ 1101111 = \\ \hline 0001100 \end{array}$$

(h)
$$\begin{array}{r} -11 - \\ 29 = \\ \hline -40 \end{array} \qquad \begin{array}{r} 1110101 - \\ 0011101 = \\ \hline 1011000 \end{array}$$

(i)
$$\begin{array}{r} 8 + \\ -18 = \\ \hline -10 \end{array} \qquad \begin{array}{r} 0001000 + \\ 1101110 = \\ \hline 1110110 \end{array}$$

(j)
$$\begin{array}{r} 7 - \\ 17 = \\ \hline -10 \end{array} \qquad \begin{array}{r} 0000111 - \\ 0010001 = \\ \hline 1110110 \end{array}$$

2. (a) 4 bits are needed:
$$\begin{array}{r} 3 + \\ -7 = \\ \hline -4 \end{array} \qquad \begin{array}{r} 0011 + \\ 1001 = \\ \hline 1100 \end{array}$$

(b) 4 bits are needed:

$-3+$	$1101+$
$7=$	$0111=$
4	0100

(c) 5 bits are needed:

$11+$	$01011+$
$-11=$	$10101=$
0	00000

(d) 6 bits are needed:

$18-$	$010010-$
$-3=$	$111101=$
21	010101

3.15.3 Octal and Hexadecimal Numbers

1. The quickest solution is to first write the number in binary:
   (a) 276534_8 in binary is 010111110101011110_2.
   The hexadecimal representation is: $17D5C_H$.
   (b) 22017555724_8 is $010010000001111101101101111010100_2$ in binary.
   Hexadecimal: $903EDBD4_H$.

2. (a) Converting 7722_{10}:

7722	0
3861	1
1930	0
965	1
482	0
241	1
120	0
60	0
30	0
15	1
7	1
3	1
1	1

7722	10	(A)	(ex: $7722 : 16 = 482 + 10$)
482	2		
30	14	(E)	
1	1		

In binary: 1111000101010_2, in hexadecimal: $1E2A_H$.

(b) Converting 1435_{10}:

1435	1
717	1
358	0
179	1
89	1
44	0
22	0
11	1
5	1
2	0
1	1

1435	11
89	9
5	5

(B) (ex: $1435 : 16 = 89 + 11$)

In binary: 10110011011_2, in hexadecimal: $59B_H$.

3. (a) Converting 36625_{10}:

36625	1	$(36625 : 8 = 4578 + 1)$
4578	2	$(4578 : 8 = 572 + 2)$
572	4	$(572 : 8 = 71 + 4)$
71	7	$(71 : 8 = 8 + 7)$
8	0	$(8 : 8 = 1 + 0)$
1	1	

36625	1	$(36625 : 16 = 2289 + 1)$
2289	1	$(2289 : 16 = 143 + 1)$
143	F	$(143 : 16 = 8 + 15)$
8	8	

In octal: 107421_8, in hexadecimal: $8F11_H$.

(b) Converting 124_{10}:

124	4	$(124 : 8 = 15 + 4)$
15	7	$(15 : 8 = 1 + 7)$
1	1	

124	12	(C) $(124 : 16 = 7 + 12)$
7	7	

In octal: 174_8, in hexadecimal: $7C_H$.

4. First, we convert the number into binary and then into decimal:

(a) $11010001010110000111_2 = 1714951_{10}$
(b) $10001000001000111_2 = 69703_{10}$

5. From octal to binary and then to decimal:

(a) $11001001001_2 = 1609_{10}$
(b) $10111110101011100_2 = 97628_{10}$

6. (a)
$$\begin{array}{r} 4\ 1\ A\ B\ 7\ + \\ C\ 2\ D\ 6\ F\ = \\ \hline 1\ 0\ 4\ 8\ 2\ 6 \end{array}$$

(b)
$$\begin{array}{r} A\ 2\ 3\ C\ E\ + \\ 3\ 6\ 3\ E\ 6\ = \\ \hline D\ 8\ 7\ B\ 4 \end{array}$$