Logic Circuit Design

Christian Rinderknecht

31 October 2008

Decimal numbers

Writing 123 in base 10 means $1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$. This encoding is called **positional number in base** 10 and we write 123_{10} to mean that 123 is a positional number in base 10.

More generally, $d_{n-1}d_{n-2}\ldots d_0$ represents

$$d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_0 \times 10^0$$

To way to encode **fractional numbers** is done by extending this system to negative exponents: $d_{n-1}\ldots d_0.d_{-1}d_{-2}\ldots d_{-m}$ represents

$$d_{n-1} \times 10^{n-1} + \cdots + d_0 \times 10^0 + d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2} + \cdots + d_{-m} \times 10^{-m}$$

Binary numbers

Most electronic devices are usually placed in two distinct electric states, either charged or discharged, low voltage or high voltage etc.

This leads to the interest in **binary numbers**, i.e. positional numbers in base 2. In this case, the two digits are usually 0 and 1, and are usually called **bits**.

For example, 10110101_2 is an 8-bit number. A way to get the encoding of it in base 10 consists in using the general formula for positional numbers and compute using the operations in base 10:

$$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$128 + 32 + 16 + 4 + 1$$

$$181$$

Binary numbers (cont)

A more convenient way to do the computation is to use the array

128	64	32	16	8	4	2	1
1	0	1	1	0	1	0	1

and then sum the powers of 2 indexed by 1s:

128	64	**32**	**16**	8	**4**	2	**1**
1	0	1	1	0	1	0	1

$$128 + 32 + 16 + 4 + 1 = 181$$

Binary numbers (cont)

The greatest 8-bit binary number is made of all bits 1:

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

That is to say: $128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$.

The greatest binary number of n bits equals

$$\max_n = 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 \tag{1}$$

If we multiply it by 2, we get

$$2 \times \max_n = 2^n + 2^{n-1} + \cdots + 2^2 + 2 \tag{2}$$

Binary numbers (cont)

Forming (2) − (1) we get

$$2 \times \max_n - \max_n = 2^n - 1$$
$$\max_n = 2^n - 1$$

Therefore, $\max_8 = 2^8 - 1 = 255$.

This means: we can count 2^n different values with an 8-bit number, including 0.

Given an n-bit binary number, the bit corresponding to 2^{n-1} is called **most significant bit (MSB)** and the bit associated to 2^0 is the **least significant bit (LSB)**.

For example: the MSB of 181 is 1 and its LSB is 1 also.

From decimal to binary numbers

Let us consider again the encoding of an n-bit number $d_{n-1}d_{n-2}\ldots d_0$ in base 10:

$$\begin{aligned} N &= d_{n-1} \times 10^{n-1} + d_{n-2} \times 10^{n-2} + \cdots + d_1 \times 10 + d_0 \\ &= 10 \times (d_{n-1} \times 10^{n-2} + d_{n-2} \times 10^{n-3} + \cdots + d_1) + d_0 \end{aligned}$$

This means that d_0 is the remainder of the integer division $N/10$, because, by definition, all digits are smaller than the base (here $d_i < 10$, for all i). We can proceed in the same fashion:

$$\begin{aligned} N &= 10 \times (10 \times (d_{n-1} \times 10^{n-3} + \cdots + d_2) + d_1) + d_0 \\ (N - d_0)/10 &= 10 \times (d_{n-1} \times 10^{n-3} + \cdots + d_2) + d_1 \end{aligned}$$

which means that d_1 is the remainder of the integer division $(N - d_0)/10^2$.

From decimal to binary numbers (cont)

This leads to a simple procedure to convert a number from its decimal representation to its binary one:

1. divide it by 2;
2. collect the remainder, which is a bit;
3. start again with the **quotient** if it is not 0,
4. otherwise end.

213	1	LSB
106	0	
53	1	
26	0	
13	1	
6	0	
3	1	
1	1	MSB
0		

$213_{10} = 11010101_2$

From decimal to binary numbers (cont)

In order to convert a fractional decimal number into a binary number, we convert the integer part as just described and convert separately the fractional part as follows.

The fractional part of a decimal number is of the shape:

$$\begin{aligned} F &= d_{-1} \times 10^{-1} + d_{-2} \times 10^{-2} + \cdots + d_{-m} \times 10^{-m} \\ 10 \times F &= d_{-1} + d_{-2} \times 10^{-1} + \cdots + d_{-m} \times 10^{-m+1} \\ 10(10F - d_{-1}) &= d_{-2} + \cdots + d_{-m} \times 10^{-m+2} \end{aligned}$$

So d_{-1} is the integer part of the multiplication by 10, i.e., $d_{-1} = \lfloor 10F \rfloor$, d_{-2} is the integer part of the fractional part of $10F$ multiplied by 10 etc.

From decimal to binary numbers (cont)

Therefore, let us multiply the fractional part of the decimal number successively by 2 and retain the integer part as the bit, until the fractional part becomes 0.

For example

$$
\begin{array}{r}
0.625 \\
\times 2 \\
\hline
\mathbf{1}.250 \\
\times 2 \\
\hline
\mathbf{0}.500 \\
\times 2 \\
\hline
\mathbf{1}.000
\end{array}
$$

$$0.625_{10} = 0.101_2$$

Octal and hexadecimal numbers

Another common base for numbers used in digital systems is 8. The corresponding encoding of numbers are called **octal numbers**. The digits used in octal are $0, 1, 2, 3, 4, 5, 6, 7$.

Another common base is 16; the corresponding representation of numbers is named **hexadecimal numbers**. The digits are from 0 to 9, followed by A to F because we need more digits than the base 10 allows, i.e., the Arabic numbers. This means that $\mathrm{A}_{16} = 10_{10}$, $\mathrm{B}_{16} = 11_{10}$ etc. until $\mathrm{F}_{16} = 15_{10}$.

For example

$$174_8 = 1 \times 8^2 + 7 \times 8^1 + 4 \times 8^0 = 124_{10}$$
$$\mathrm{B78}_{16} = 11 \times 16^2 + 7 \times 16^1 + 8 \times 16^0 = 2936_{10}$$

Octal numbers and binary numbers

Consider the n-bit binary number general form:

$$b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2 + b_0$$

We can group the bits by groups of three, from right to left:

$$\cdots + (b_8 2^8 + b_7 2^7 + b_6 2^6) + (b_5 2^5 + b_4 2^4 + b_3 2^3) + (b_2 2^2 + b_1 2 + b_0)$$

We can factorise 2^0, 2^3, 2^6 etc. and get

$$\cdots + (b_8 2^2 + b_7 2 + b_6) \times 2^6 + (b_5 2^2 + b_4 2 + b_3) \times 2^3 + (b_2 2^2 + b_1 2 + b_0)$$

That is, since $8 = 2^3$ and $x^{pq} = (x^p)^q$, then $2^{3q} = (2^3)^q = 8^q$ and

$$\cdots + (b_8 2^2 + b_7 2 + b_6) \times 8^2 + (b_5 2^2 + b_4 2 + b_3) \times 8^1 + (b_2 2^2 + b_1 2 + b_0) \times 8^0$$

Octal numbers and binary numbers (cont)

Therefore, in order to convert an octal number to its equivalent binary representation, we convert separately its digits into binary and simply catenate them.

If we want to convert from binary representation to octal

1. group the bits three-by-three from right to left,
2. convert each group to octal digits,
3. catenate the partial results.

For example

$$\begin{aligned} 7351_8 &= (7_8)(3_8)(5_8)(1_8) = (111_2)(011_2)(101_2)(001_2) \\ &= 111011101001_2 \end{aligned}$$

Octal numbers and decimal numbers

There are two way to convert from decimal to octal:

1. the *direct way* consists in dividing successively the quotients by 8 and saving the remainders;
2. the *indirect way* consists in converting from decimal to binary (by successive divisions by 2) and then from binary to octal (easy).

The indirect way is usually easier because less error-prone.

Hexadecimal numbers

All the previous discussion about octal numbers is meaningful in a similar way for hexadecimal numbers, except that the bit groupings have to be four bits long.

For example

$$\begin{aligned} \mathrm{B396}_{16} &= (\mathrm{B}_{16})(3_{16})(9_{16})(6_{16}) \\ &= (1011_2)(0011_2)(1001_2)(0110_2) \\ &= 1011001110010110_2 \end{aligned}$$

Fractional octal and hexadecimal numbers

The same rules apply for the fractional part of octal and hexadecimal numbers.

Consider

$$515.6_8 = 110\ \ 001\ \ 101.110_2$$
$$14D.C_{16} = 1\ \ 0100\ \ 1101.1100_2$$

Addition

Consider two decimal numbers

$$N_1 = \cdots + a_2 10^2 + a_1 10 + a_0$$
$$N_2 = \cdots + b_2 10^2 + b_1 10 + b_0$$

and their addition in this way:

$$N_1 + N_2 = \cdots + (a_2 + b_2)10^2 + (a_1 + b_1)10 + (a_0 + b_0)$$

The problem is that it is possible that $10 \leqslant a_i + b_i$ for some i. For instance

$$\begin{aligned} 23 + 8 &= (2 \times 10 + 3) + 8 = 2 \times 10 + (3 + 8) = 2 \times 10 + (\mathbf{1} \times 10 + 1) \\ &= (2 + \mathbf{1}) \times 10 + 1 = 31 \end{aligned}$$

Addition

In this case, a **carry**, i.e., an overflowing digit 1, has to be added to the digits of the next power. That is why addition works from right to left.

Another example:

$$\begin{array}{r} \mathbf{1} \\ 2\,3\,6\,7 \\ +\,1\,3\,2\,6 \\ \hline 3\,6\,9\,3 \end{array}$$

The same phenomenon happens with binary, octal or hexadecimal numbers. With binary:

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	**1**0

Subtraction

During addition, the addition of two digits may cause an **overflow**, i.e., the result is greater or equal than the base.

The risk of subtraction is to have an **underflow** while subtracting two digits, i.e., the result may be lower than 0. In this case, we "borrow" a carry, i.e., a digit of 1 to the next power (subtraction works from right to left, as addition).

For example, consider

$$\begin{array}{r} \mathbf{3} \\ 4\,7\,2 \\ -\,3\,9\,1 \\ \hline 8\,1 \end{array}$$

Subtraction on binary numbers

The principle is the same for binary numbers. The bit subtractions are

$$\begin{array}{r} 0 \\ -\,0 \\ \hline 0 \end{array} \qquad \begin{array}{r} \mathbf{1}0 \\ -\;1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ -\,0 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ -\,1 \\ \hline 0 \end{array}$$

where **1** is the **borrow**.

$$\begin{array}{r} \mathbf{0\,1} \\ 1\,0\,1\,1\,1\,0\,0\,1 \\ -\,0\,0\,1\,1\,0\,0\,1\,1 \\ \hline 1\,0\,0\,0\,0\,1\,1\,0 \end{array}$$

Here we had to borrow two times, until a 1 is found.

Multiplication

Consider the familiar decimal multiplication on an example:

$$
\begin{array}{rr}
 & 1\,4\,7 \\
 & \times\, 2\,6\,5 \\
\hline
 & 7\,3\,5 \\
+ & 8\,8\,2 \\
+ & 2\,9\,4 \\
\hline
 & 3\,8\,9\,5\,5
\end{array}
$$

Notice that when two digits are multiplied, a carry can be produced, which can be 8 at most (from $9 \times 9 = \mathbf{8} \times 10 + 1$).

Multiplication (cont)

Instead, the multiplication of bits never generates a carry. The rules are simply:

$$\begin{array}{r} 0 \\ \times 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ \times 1 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ \times 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 1 \\ \times 1 \\ \hline 1 \end{array}$$

For example

$$\begin{array}{r} 1\,0\,1\,1 \\ \times\ 1\,0\,0\,1 \\ \hline 1\,0\,1\,1 \\ 0\,0\,0\,0 \\ 0\,0\,0\,0 \\ 1\,0\,1\,1 \\ \hline 1\,1\,0\,0\,0\,1\,1 \end{array}$$

2-complement binary

Until now, we said nothing about what happens if we subtract a number from a number which is strictly smaller. In practice, we found ourselves with a borrow from nowhere. Or what happens if we add two numbers and we get a carry we cannot use.

What do we do if we have to compute, for example $3 - 7$ in decimal? In this case, we use an unary negative operator and write -4, but we cannot do the same with binary numbers because the purpose of binary numbers is to be handled at the hardware level, where there is no minus sign.

What we need is an encoding of binary numbers that copes both with positive and negative numbers in a uniform way, i.e., the negativity or positivity is coded in the bits of the number *themselves*. One of these encodings is called 2-**complement binary numbers**.

2-complement binary (cont)

The idea consists in interpreting the leftmost bit as a negative positional value, that is to say

$$N = b_{n-1}\, b_{n-2} \ldots b_0$$

means

$$N = -b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \cdots + b_0$$

Thus, in an 8-bit 2-complement binary number, the leftmost bit has a positional value of -128 rather than $+128$ (it is **not** the MSB). For example, $N = 01110101$ is interpreted as

-128	**64**	**32**	**16**	8	**4**	2	**1**
0	1	1	1	0	1	0	1

$$64 + 32 + 16 + 4 + 1 = 117$$

2-complement binary (cont)

Another example:

$-\mathbf{128}$	64	$\mathbf{32}$	$\mathbf{16}$	8	$\mathbf{4}$	2	1
1	0	1	1	0	1	0	0

$$-128 + 32 + 16 + 4 = -76$$

In other words, the interpretation of 2-complement numbers is the usual one, except that the leftmost bit has a negative positional value.

If the leftmost bit is 1 then the number is negative. Why?

Assume that in an n-bit 2-complement number, the leftmost bit is 1. Then the highest positive value that can be formed with the remaining $n - 1$ bits is having only 1s. In other words, the n bits are all 1s.

2-complement binary (cont)

So this number is

$$N = -2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 1$$

We showed page 6 that

$$2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1$$

So

$$N = -2^{n-1} + (2^{n-1} - 1) = -1 < 0$$

If the leftmost bit is 0 then the number is positive. Why? Because it is simply of the form

$$0 \times 2^{n-1} + b_{n-2}2^{n-2} + b_{n-3}2^{n-3} + \cdots + b_1 2 + b_0 \geqslant 0$$

2-complement binary (cont)

Because the leftmost bit of a 2-complement number tells the sign, it is sometimes called the *sign bit*, but this is incorrect because, in n-bit numbers, the leftmost bit b_{n-1} is not used to encode the sign itself, but $-b_{n-1} \times 2^{n-1}$.

When interpreting a 2-complement number, it is necessary to say how many bits are involved in order to determine the bit with negative positional value.

With an 8-bit 2-complement binary number, the lowest value that can be represented is 10000000_2, i.e., -128_{10}, and the highest is 01111111_2, i.e., $+127_{10}$.

With n-bits, it ranges from -2^{n-1} to $2^{n-1} - 1$: **the interval is asymmetric**, since 2^{n-1} is out of range.

2-complement binary/Addition

The addition of two 2-complement numbers is carried out as usual, without regard for the sign of each one.

Of course, there can be an overflow (i.e., the result is too big to fit the bit length) or an underflow (i.e., the result is too low to fit the bit length). For example

$$\begin{array}{r} \mathbf{1} \\ 0\,1\,0\,1\,1\,0\,1\,1 \\ +\,0\,1\,0\,0\,0\,1\,0\,0 \\ \hline 1\,0\,0\,1\,1\,1\,1\,1 \end{array}$$

is wrong because we added two positive numbers and the result is negative.

2-complement binary/Addition (cont)

An overflow can occur if the two numbers are positive and an underflow can occur if the two numbers are negative. But why and when are we sure that a problem occurred during addition? Let us consider all the cases with n bits. In the following, X, Y and Z are bits at position $n-2$.

$$\begin{array}{r} ? \\ 0\,\mathrm{X}\ldots \\ +\,0\,\mathrm{Y}\ldots \\ \hline 0\,\mathrm{Z}\ldots \end{array} \qquad \begin{array}{r} \mathbf{1}\ ? \\ 0\,\mathrm{X}\ldots \\ +\,0\,\mathrm{Y}\ldots \\ \hline 1\,\mathrm{Z}\ldots \end{array}$$

In the first case, we add two positive numbers with no carry out of the position $n-2$: there is no overflow. It is **correct**.
In the second case, there is a carry out of the position $n-2$: there is an overflow because the result is interpreted as a negative number. It is **incorrect**.

2-complement binary/Addition (cont)

$$\begin{array}{r} ? \\ 1\,\mathrm{X}\ldots \\ +\,0\,\mathrm{Y}\ldots \\ \hline 1\,\mathrm{Z}\ldots \end{array} \qquad \begin{array}{r} \mathbf{1\,1}\,? \\ 1\,\mathrm{X}\ldots \\ +\quad 0\,\mathrm{Y}\ldots \\ \hline 0\,\mathrm{Z}\ldots \end{array}$$

In the first case, we add a negative number to an absolutely smaller positive one with no carry out of the position $n-2$: it is **correct** and the result is negative.

In the second case, there is a carry out of the position $n-2$, which represents the addition of 2^{n-1}. This is why it cancels out the bit 1 with negative positional value, $-2^{n-1}+2^{n-1}=0$, so it is **correct** (and *we ignore the leftmost carry*).

2-complement binary/Addition (cont)

$$\begin{array}{rr} \mathbf{1}\ ? & \mathbf{1}\,\mathbf{1}\ ? \\ 1\,\mathrm{X}\ldots & 1\,\mathrm{X}\ldots \\ +\ 1\,\mathrm{Y}\ldots & +\ 1\,\mathrm{Y}\ldots \\ \hline 0\,\mathrm{Z}\ldots & 1\,\mathrm{Z}\ldots \end{array}$$

In the first case, there is no carry out of the position $n-2$ and the result is positive, $(-2^{n-1})+(-2^{n-1})=-2^n$, instead of negative, so it is **incorrect** (underflow with n bits).

In the second case, there are two carries and the result is **correct**, i.e.,

$$(+2^{n-1})+(-2^{n-1})+(-2^{n-1})=-2^{n-1}$$

(*We ignore the leftmost carry.*)

2-complement binary/Addition (cont)

Summary

In order to perform the 2-complement addition, consider the two numbers as normal binary numbers and then perform the usual addition.

Considering the possible carries out of the positions $n-1$ and $n-2$:

- if there is only **one carry in total**, then the sum is **incorrect**:
  - overflow if the result is negative,
  - underflow if the result is positive;
- otherwise the sum is correct and the carries, if any, are discarded.

2-complement binary/Complement and negation

The **complement** of a normal binary number consists simply in turning all its bits 1 into 0 and all its bits 0 into 1. For example, the complement of 10010_2 is 01101_2, or simply 1101_2. The complement of the complement is the number itself.

This operation is sometimes called the 1-**complement**.

How do we find the **negation** of a 2-complement number? In other words, given the n-bit, 2-complement number $N = b_{n-1}b_{n-2}\ldots b_0$, what are the bits of $-N$?

What we want is to solve the following puzzle:

$$\begin{array}{rccccc} & 0 & 0 & 0 & \ldots & 0 \\ - & b_{n-1} & b_{n-2} & b_{n-3} & \ldots & b_0 \\ \hline & \mathbf{?} & \mathbf{?} & \mathbf{?} & \ldots & \mathbf{?} \end{array}$$

2-complement binary/Complement and negation (cont)

Notice that, with n-bit, 2-complement binary numbers, the following holds:

$$0 = -1 + 1 \iff \underbrace{000\ldots00}_{n\text{ bits}} = \underbrace{111\ldots11}_{n\text{ bits}} + \underbrace{000\ldots01}_{n\text{ bits}}$$

So our initial question is equivalent to

$$\begin{array}{rcccccc} & 1 & 1 & 1 & \ldots & 1 & + \quad 00\ldots01 \\ - & b_{n-1} & b_{n-2} & b_{n-3} & \ldots & b_0 & \\ \hline & \mathbf{?} & \mathbf{?} & \mathbf{?} & \ldots & \mathbf{?} & \end{array}$$

2-complement binary/Complement and negation (cont)

Notice also that subtracting from 1 is the same as complementing:

$$1 - 1 = 0 \iff 1 - 0 = 1$$

If we write $\overline{0} = 1$ and $\overline{1} = 0$ for the complement, we simply have $1 - b = \overline{b}$ and

$$\begin{array}{rccccccc}
 & 1 & 1 & 1 & \dots & 1 & + & 00\dots01 \\
- & b_{n-1} & b_{n-2} & b_{n-3} & \dots & b_0 & & \\
\hline
 & \overline{b_{n-1}} & \overline{b_{n-2}} & \overline{b_{n-3}} & \dots & \overline{b_0} & & \\
+ & 0 & 0 & 0 & \dots & 1 & & \\
\hline
 & \mathbf{?} & \mathbf{?} & \mathbf{?} & \dots & \mathbf{?} & &
\end{array}$$

2-complement binary/Complement and negation (cont)

The last step is the addition of 1, which is easy. In summary:

$$A - B = A + (-B) = A + (\overline{B} + 1) = (A + \overline{B}) + 1$$

where $\overline{B}$ is the bitwise 1-complement of B (i.e., bit by bit 1-complement).

For example, let us negate the 8-bit, 2-complement binary number 10110100:

$$\begin{array}{rr} & 1\,0\,1\,1\,0\,1\,0\,0 \\ \hline & 0\,1\,0\,0\,1\,0\,1\,1 \\ + & 0\,0\,0\,0\,0\,0\,0\,1 \\ \hline & 0\,1\,0\,0\,1\,1\,0\,0 \end{array}$$

Exercise. Negate the 8-bit, 2-complement binary number 10000000.

2-complement binary/Complement and negation (cont)

Negation gives a simple way to convert from 2-complement binary to decimal when the number is negative: first, negate it and then convert to decimal as usual, and finally add a minus sign to the result.

For example,

$$10001001 = -01110111 = -119_{10}$$

Conversely, in order to convert a negative decimal number to 2-complement binary notation, we convert the negated decimal (i.e., we forget the minus sign) and then negate the result.

Gray codes

There are several ways to code numbers using a binary alphabet, i.e., 0 and 1. One is the usual binary code and another is the 2-complement binary code.

There is another one, quite useful, called **unit distance code**, or **Gray code**. It is designed so that only one bit changes when a number is incremented or decremented. For example, a Gray code for numbers from 0 to 15 is

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

Binary-Coded Decimal

There is another interesting binary coding for numbers, where each decimal digit is separately encoded in binary: **Binary-Coded Decimal**, or, in short, **BCD**.

For example, to code the number 37_{10} in BCD, we do

$$37_{10} = (3_{10})(7_{10}) = (0011_2)(0111_2) = 00110111_2$$

Every decimal digit can be encoded with four bits.

Conversely, to convert a BCD number into decimal, we separate the bits in groups of four and convert them separately with the usual manner.

Binary-Coded Decimal (cont)

Note that, since four bits are used for coding each decimal digit, some combinations of bits are useless because there is no digit corresponding to numbers 10 to 15, hence the coding is not as compact as for the usual binary coding.

So, for instance, 110001010111_2 is not a BCD.

Binary-Coded Decimal/Addition

If we add BCD numbers as basic binary numbers we do not always get the correct result. For example

$$\begin{array}{rcc} & 0\,0\,1\,1 & 1\,0\,0\,0 \\ + & 0\,1\,0\,0 & 0\,0\,0\,1 \\ \hline & 0\,1\,1\,1 & 1\,0\,0\,1 \end{array}$$

is correct, but

$$\begin{array}{rcc} & 1\,0\,0\,0 & 0\,1\,1\,1 \\ + & 1\,0\,0\,1 & 0\,1\,0\,0 \\ \hline 1 & 0\,0\,0\,1 & 1\,0\,1\,1 \end{array}$$

is not, because $1011_2 = 11_{10}$ is not a decimal digit.

Binary-Coded Decimal/Addition (cont)

Actually, the problem happens wherever the number corresponding to a digit ranges from $9 + 1 = 10$ to $9 + 9 = 18$.

To be more precise, there are two sub-cases:

1. from 10_{10} to 15_{10}: the four bits range from 1010 to 1111 and there is no carry;
2. from 16_{10} to 18_{10}: the four bits range from 0000 to 0010 and there is a carry.

Binary-Coded Decimal/Addition (cont)

In the first case, we would like to map $1010_2 = 10_{10}$ to $0001\ 0000_2 = 10_{\text{BCD}}$, i.e., to 0000 plus a carry, $1011_2 = 11_{10}$ to $0001\ 0001_2 = 11_{\text{BCD}}$ etc. until $1111_2 = 15_{10}$ to $0001\ 0101_2 = 15_{\text{BCD}}$. This can be simply achieved by adding $6_{10} = 0110_2$ to the digit, e.g.

$$10_{10} + 6_{10} = 1010_2 + 0110_2 = 0001\ 0000_2$$

In the second case, we would like to map $0001\ 0000_2 = 16_{10}$ to $0001\ 0110_2 = 16_{\text{BCD}}$ etc. until $0001\ 0010_2 = 18_{10}$ to $0001\ 1000_2 = 18_{\text{BCD}}$. This can be also achieved by simply adding $6 = 0110_2$ to the rightmost four bits, e.g.

$$0000 + 0110 = 0110$$

Binary-Coded Decimal/Addition (cont)

Therefore the rule for addition is:

1. add the two numbers as usual;
2. to each block of 4 bits in the result, add 0110 if a carry was issued from it or if it is strictly greater than 1001, else add 0000.

For instance

$$
\begin{array}{rcccc}
 & & 1 & & \\
 & 0\,1\,1\,0 & 1\,0\,0\,0 & 0\,1\,1\,0 \\
+ & 0\,0\,1\,1 & 1\,0\,0\,0 & 0\,0\,0\,1 \\
\hline
1 & 1\,1 & & & \\
 & 1\,0\,1\,0 & 0\,0\,0\,0 & 0\,1\,1\,1 \\
+ & 0\,1\,1\,0 & 0\,1\,1\,0 & 0\,0\,0\,0 \\
\hline
1 & 0\,0\,0\,0 & 0\,1\,1\,0 & 0\,1\,1\,1
\end{array}
$$

Binary-Coded Decimal/Addition (cont)

Actually, the additions of 0110 must be carried over again if any block is strictly greater than 1001, due to the carries of step (2).

For instance $36 + 65 = 101$ in BCD is given in the facing column.

$$
\begin{array}{cccccccc}
 & & 1 & 1 & & & 1 & \\
 & & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
+ & & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\
\hline
 & & & & 1 & 1 & 1 & 1 & & \\
 & & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
+ & & & & & & 0 & 1 & 1 & 0 \\
\hline
 & 1 & 1 & 1 & & & & & & \\
 & & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
+ & & 0 & 1 & 1 & 0 & & & & \\
\hline
 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Binary-Coded Decimal/Subtraction

BCD subtraction consists in

1. a normal binary subtraction;
2. to each block of four bits that borrowed to the next group, subtract 0110, otherwise 0000.

The reason of the latter is that when we borrow, we actually borrow 16_{10} instead of 10_{BCD}.

$$
\begin{array}{rcccc}
 & & 0 & 1\,0 & 0\,1 \\
 & 0\,0\,1\,1 & 0\,1\,0\,1 & 1\,0\,0\,0 \\
- & 0\,0\,1\,0 & 0\,1\,1\,0 & 0\,0\,1\,0 \\
\hline
 & 0\,0\,0\,0 & 1\,1\,1\,1 & 0\,1\,1\,0 \\
- & 0\,0\,0\,0 & 0\,1\,1\,0 & 0\,0\,0\,0 \\
\hline
 & 0\,0\,0\,0 & 1\,0\,0\,1 & 0\,1\,1\,0
\end{array}
$$

Boolean algebra

A **boolean algebra** models logical statements based on two values and relationships "and" (**conjunction**), "or" (**disjunction**) and "not" (**negation**). The values are usually called "true" and "false", or "1" and "0" in circuit design.
If A is a boolean variable, i.e. a name whose interpretation can only either be "true" or "false", then

$$A = 1$$

reads "A is true", and

$$A = 0$$

reads "A is false."

Boolean algebra/Operators

The three operators of a boolean algebra are as follows.

The binary operator **and**, symbolised by a dot: $A \cdot B$ reads "A and B." If the meaning is clear, the dot can be omitted, as in AB. This operator is defined by the equations

$$0 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$1 \cdot 0 = 0$$
$$1 \cdot 1 = 1$$

The binary operator **or**, symbolised by '+': $A + B$ reads "A or B." This operator is defined by the equations

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 1$$

Boolean algebra/Operators (cont)

The last operator is the unary operator **negation**, symbolised by a bar over its argument. It is also called **complement**, or **one-complement** but, be careful: in the context of 2-complement binary numbers (page 23), for example, negation is **not** the complement.
This operator is simply defined as

$$\overline{1} = 0$$
$$\overline{0} = 1$$

Warning: Page 36, $\overline{B}$ means the 1-complement of an **integer**, not one bit. In other words, B was *not* a boolean variable.

Boolean algebra/Identities

For any boolean variable A, the following identities hold:

$$1 + A = 1 \quad (3)$$
$$0 + A = A \quad (4)$$
$$A + A = A \quad (5)$$
$$A + \overline{A} = 1 \quad (6)$$
$$0 \cdot A = 0 \quad (7)$$
$$1 \cdot A = A \quad (8)$$
$$A \cdot A = A \quad (9)$$
$$A \cdot \overline{A} = 0 \quad (10)$$
$$\overline{\overline{A}} = A \quad (11)$$

Boolean algebra/Identities

These identities can be proved by replacing A by 0 and 1, and use the definitions of "and", "or" and "not" pages 48 and 49. In the same way, we can prove:

Commutative laws

$$A + B = B + A \tag{12}$$
$$A \cdot B = B \cdot A \tag{13}$$

Distributive laws

$$A + (B \cdot C) = (A + B)(A + C) \tag{14}$$
$$A(B + C) = (A \cdot B) + (A \cdot C) \tag{15}$$

Boolean algebra/Identities (cont)

It is common to give precedence to negation over conjunction, which in turn has precedence over disjunction. This allows to omit some parentheses and, for instance, rewrite the distributive laws as

$$A + BC = (A + B)(A + C)$$
$$A(B + C) = AB + AC$$

Boolean algebra/Identities (cont)

Then, we have more identities:

Absorption laws

$$A + AB = A \tag{16}$$
$$A(A + B) = A \tag{17}$$

Logic adjacency

$$AB + A\overline{B} = A \tag{18}$$
$$(A + B)(A + \overline{B}) = A \tag{19}$$

Boolean algebra/Identities (cont)

De Morgan

$$\overline{A + B} = \overline{A} \cdot \overline{B} \tag{20}$$

$$\overline{AB} = \overline{A} + \overline{B} \tag{21}$$

For example

$$\overline{A + BC} = \overline{A}(\overline{BC}) = \overline{A}(\overline{B} + \overline{C})$$

$$\overline{(A + B)(C + D)} = \overline{A + B} + \overline{C + D} = \overline{A}\,\overline{B} + \overline{C}\,\overline{D}$$

Boolean algebra/Truth tables

One way to define a boolean function is to write its **truth table**, i.e. a table whose columns are the arguments of the function and its result, and the rows provide all the cases of 0 and 1 for the arguments. For example, function $L(A, B, S)$ is defined by

A	B	S	L
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Boolean algebra/Min-terms

Consider a boolean function $F(A, B)$, defined by the truth table

A	B	AB	$A\overline{B}$	$\overline{A}B$	$\overline{A}\,\overline{B}$	$F(A, B)$
0	0	0	0	0	1	1
0	1	0	0	1	0	0
1	0	0	1	0	0	0
1	1	1	0	0	0	1

We inserted in the truth table the values of the boolean expressions AB, $A\overline{B}$, $\overline{A}B$ and $\overline{A}\,\overline{B}$, which are called **min-terms**. The interesting point is that each min-term is true only for **one** combination of values of A and B.

This leads to a way of finding a definition of F based on boolean expressions instead of a truth table.

Boolean algebra/Min-terms (cont)

Indeed, for each combination of values of A and B such that $F(A, B)$ is 1, consider the corresponding min-term and make a disjunction of them. We get here

$$F(A, B) = \overline{A}\,\overline{B} + AB$$

Why does it work? For each combination (A, B) for which $F(A, B) = 1$, one of the min-terms is 1, i.e., $\overline{A}\,\overline{B} = 1$ or $AB = 1$, so the disjunction is also 1. For each combination (A, B) for which $F(A, B) = 0$, both min-terms will be 0, so is their disjunction.

Boolean algebra/Min-terms (cont)

Therefore, given a truth table of a boolean function, form the expression made of the disjunction of the min-terms for which the result is 1.
This form is called, in general, **normal disjunctive form**, or **sum of product** (**SOP**).
Counter-example: $\overline{A + B}$ is **not** a SOP: the negations must apply to variables or 0 or 1.
Any boolean expression can be transformed, by means of the identities we gave, into a SOP.