

Design of combinational circuits

- 1 Circuit price
- 2 Delay of logic devices
- 3 The importance of minimization
- 4 Analysis of circuits
- 5 Number of stages in the circuit
- 6 Use of EX-OR gates
- 7 Using multiplexers
- 8 Memory as combinational circuit
- 9 Basic combinational function blocks
 - 9.1 Decoder
 - 9.2 Multiplexer
 - 9.3 Demultiplexer
 - 9.4 Priority encoder
 - 9.5 Digital comparator
 - 9.6 Adder
 - 9.7 Subtractor
 - 9.8 Majority circuit

Design of combinational circuits

The design of a combinational circuit is based on its description - a truth table, an expression (possibly minimized), verbal description, program, etc. In this text we will assume a logical expression as a starting point.

The design methods differ depending on which components the system will be built from. We will discuss the following cases:

- Elementary logic devices
- Combinational function blocks
- Memory

Different demands can be placed on the result of the design - price, speed (delay), mechanical construction, possibility of diagnostics, reparability, fault tolerance, etc. Here we notice prices and delays as very common criteria.

1 Circuit price

Price is always an important parameter. The actual price of a manufactured component depends on many parameters, which are usually very difficult to express (if at all) mathematically. Nevertheless, it is appropriate to introduce a simple criterion that would allow comparisons of different design variants. A very simple criterion can be the number of transistors in simple components and, depending on the number of these components, the total number of transistors.

If we start from the elementary digital devices NOT-NOR-NAND in CMOS technology, it is clear that the number of transistors in one device is given by twice the number of inputs - see chapter Digital devices and technology. From this, the number of transistors corresponding to a logical expression in the form of the sum of products (which is the most common form of description of a logical function) can be easily derived. The total number of inputs of these products is given by the total number of letters in the expression (this is how we proceeded when minimizing). These products must be OR-ed. The number of inputs of an OR gate is given by the number of logical products in the expression. We must not forget the negations. The number of negated variables is counted.

So a simple guide to determining the "price" of an expression is to: count the total number of letters in the expression - if a letter occurs more than once, it is counted each time - and add the number of product terms. Next, add the number of negated variables (not the total number of negations). The resulting number multiplied by two indicates the total number of transistors.

Of course, it is also possible to start from the circuit diagram and directly calculate the number of inputs of elementary logic elements and the number of switches, and multiply the result by two. This is to be done when pricing components using a combination of transistors and CMOS switches.

Large scale combinational circuits are designed with the aid of computers and the complexity of the circuit is evaluated automatically - in this case, the number of inputs of all devices is also considered.

2 Delay of logic devices

The delay t_{pd} of logic devices was discussed in the previous chapter (Digital devices and technologies), but here we suppose that the devices are integrated in a system. The delay therefore depends on the load (especially on its capacity component) and on the number of inputs of the device - devices with a larger number of inputs (OR, NOR, AND, NAND, ...) have a longer delay. In general, the signal transition delay of the device can be calculated as follows:

$$t_{pd} = K_k + K_i \cdot N_i + K_o \cdot N_o ,$$

where K_k is a **constant** component equal to the delay of the member with 1 input and without load

K_i is the delay increment for each **input**

N_i is the number of inputs (i.e. Fan-in)

K_o is the increase in delay due to the **output** load by the inputs of other circuits (of the same technology)

N_o is the number of such loads (i.e. Fan-out)

As a result, a member with large number of inputs and outputs may be even slower than two less loaded and less branched members in series. This indicates possible adjustments. These are based on the **associative** law:

$$a + b + c + d + \dots = (a + b + c) + (d + \dots)$$

$$a \cdot b \cdot c \cdot d \cdot \dots = (a \cdot b \cdot c) \cdot (d \cdot \dots)$$

Figure 1 shows an example of the circuit it solutions in case a reduction of the number of inputs 3 to 5 in an OR - similarly to AND gates.

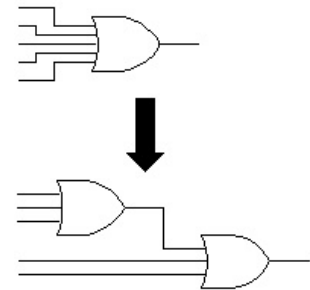


Fig. 1: Reducing the number of inputs

In case of heavy capacitive loads (by buses), the delay of ordinary devices might be unacceptable. A better choice is to boost the output power by additional bus drivers with an increased output current (tens of mA).

3 The importance of minimization

Different modifications often lead to completely different circuit structures; however, the number of logic devices may be approximately the same. Obviously, minimal SPF is not a guarantee of finding the simplest circuit, although it can be a good starting point. **Additional requirements** must be observed when designing large combinational circuits. These are the load of individual devices, the division of a large circuit into suitable structural units, the elimination of undesirable transients, circuit delays, the use of standard functional units, etc.

These requirements may be more important than the possible saving of several gates. On the other hand, the serial production must be taken into account - even small savings can be significant for large series.

4 Analysis of circuits

Sometimes it is necessary to get an expression from the circuit diagram describing the function of the circuit - that is, to analyze the circuit. When analyzing a given circuit, it is best to mark the **internal signals** first. It starts from the end and is gradually extended. For example, for the circuit in Figure 2, the result can be obtained:

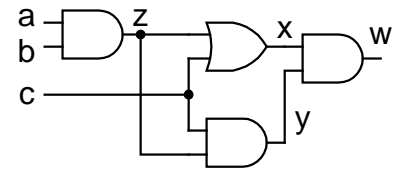


Fig. 2: Circuit analysis

$$w = (c + z). cz = (c + ab). cab = abc + abc = abc$$

The original circuit was probably unnecessarily complicated, which was not obvious at first sight.

In the case of the analysis of circuits with NAND and NOR gates, it may be necessary to apply several times **DeMorganova** rules. The result of the circuit analysis according to Fig. 3 again indicates an unnecessarily complex circuit.

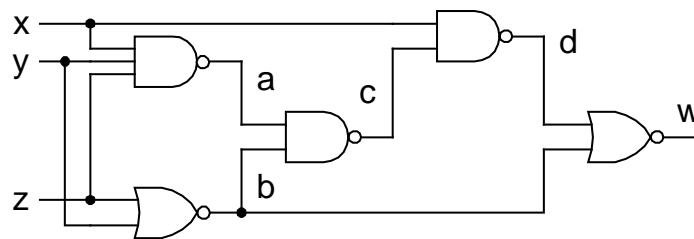


Fig. 3: Circuit analysis using DeMorgan rules

5 Number of stages in the circuit

The design of the circuit can be based on the sum of products. The circuit is then always a maximum of **three stages**. The first stage implements the inversion of (some) variables, the second stage the products (implicants), the third stage their sum. If the input signals from the previous circuits are supplied in both direct and negated form (e.g. from flip-flops), invertors at the input can be eliminated and a two-stage circuit will suffice. Preliminary knowledge of the number of stages of the circuit through which the signal passes makes it possible to **predict** its delay time, which is obviously very important information. The individual stages are shown in Fig. 4.

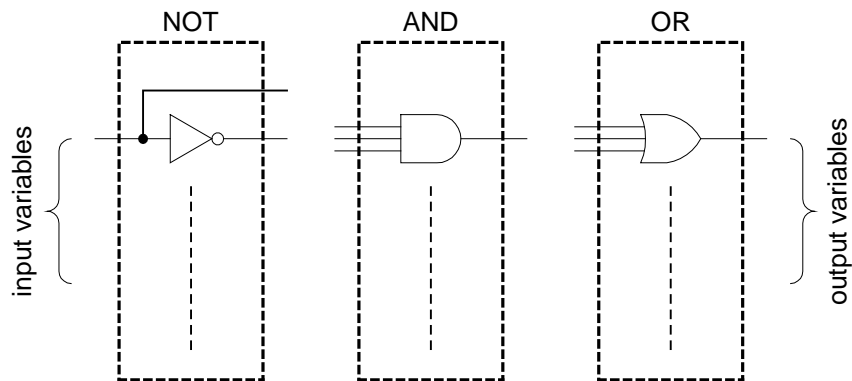


Fig. 4: Circuit corresponding to SPF of the function

When designing circuits with **inverted** gates (NOR, NAND), the procedure is the same as for circuits with AND-OR gates, only DeMorgan rules are applied. At the beginning of the arrangement is the introduction of double inversion of the expression (an even number of inversions does not change the result). For example, the expression corresponds to the circuit in Fig. 5 on the left. We can edit the expression (the circuit in the middle):

$$y = \bar{a}\bar{b}c + \bar{a}d + bd = \overline{\overline{\bar{a}\bar{b}c + \bar{a}d + bd}} = \overline{\overline{\bar{a}\bar{b}c} \cdot \overline{\bar{a}d} \cdot \overline{bd}}$$

Another possible arrangement (the circuit on the right):

$$y = \bar{a}\bar{b}c + \bar{a}d + bd = \overline{\overline{\bar{a}\bar{b}c} + \overline{\bar{a}d} + \overline{bd}} = \overline{\overline{\bar{a} + b + c} + \overline{a + d} + \overline{b + d}}$$

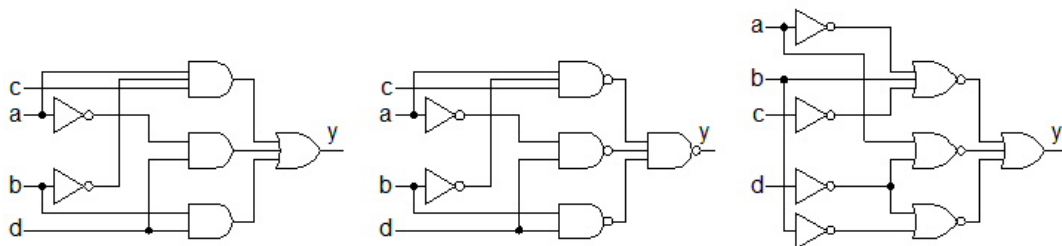


Fig. 5: Different circuits with the same function

All circuits will have three stages with the same topology, only the type of the gates changes. This way it is possible to tailor the type of gates to the available technology. The overall delay might be different due to different types of the gates.

6 Use of EX-OR gates

In some cases, the use of EX-OR gates (exclusive OR, also XOR for short) leads to a simpler circuit implementation. Unlike functions OR, XOR "excludes" the case of two ones at the inputs and gives outputs state '0', hence the name **exclusive** OR. The symbol \oplus is introduced as the operator:

$$0 \oplus 0 = 1 \oplus 1 = 0$$

$$0 \oplus 1 = 1 \oplus 0 = 1$$

The symbol indicates the sum and has the meaning of an **arithmetic** sum, but in arithmetic **mod 2**. Indeed, neglecting transfer to a higher order, the value is equal to the arithmetic sum of two one-bit numbers A and B . Therefore, the XOR function is also called the "modulo 2 sum".

The XOR function can also be written as the sum of products:

$$y = a \oplus b = \bar{a}b + a\bar{b}$$

An interesting property of the XOR gate can be seen from this expression. If we look at one of its input (e.g. a) as the input of signals to the output y , and a second input (e.g. b) as the control input, then when $b = 0$ is $y = a$, when $b = 1$ is $y = \bar{a}$. Thus, a **controlled inversion** is available - a function very often needed - see Fig. 6.

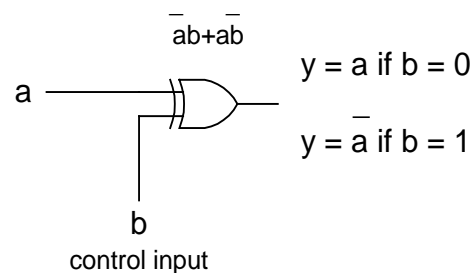


Fig. 6: XOR gate as controlled inversion

Another advantageous use of the XOR gate is in **parity bit generators**. Parity is based on the current number of input signals with a value of '1'. The **odd parity** generator outputs a value of '1' if there is an **odd** number of ones at the inputs. Similarly, the **even parity** generator gives a value of '1' at the output if there is an **even** number of ones at the inputs. For two inputs, the XOR member would suffice - from the truth table above, it is clear that it implements odd parity. The associative properties of the XOR function can be used for multiple inputs, similarly to the OR and AND functions:

$$a \oplus b \oplus c \oplus \dots = (a \oplus b) \oplus (c \oplus d) \oplus \dots$$

For example, the eight-input odd parity generator is shown in Fig. 7. The even parity generator would have an inverted output.

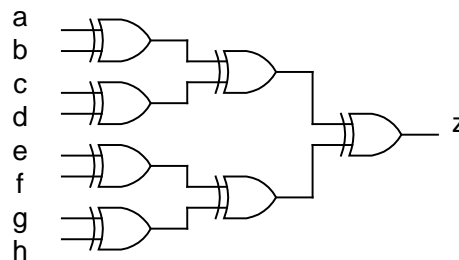


Fig. 7: Odd parity generator

The XOR function gives a value of '1' if the values on both inputs are opposite. That is why it is also called "**non-equivalence**". Conversely, the inversion of the XOR function is the "**equivalence**" function with the " \equiv " symbol. It applies:

$$a \equiv b = \overline{a \oplus b}$$

Just as the XOR function could be expressed by the sum of products, equivalence can also be written as:

$$a \equiv b = \overline{a} \overline{b} + ab$$

XOR gates can also be useful when editing expressions, when the expression $\overline{a}b + a\overline{b}$ sometimes needs to be implemented. For example, the function:

$$\begin{aligned} \overline{a}b\overline{c} + \overline{a}b\overline{c} + \overline{a}b\overline{c} + abc &= \overline{a}(b\overline{c} + \overline{b}c) + a(\overline{b}c + bc) = \overline{a}(b \oplus c) + a(b \equiv c) = \\ &= \overline{a}(b \oplus c) + a\overline{(b \oplus c)} \end{aligned}$$

Substituting for $b \oplus c = x$, then we can continue up to a simple result:

$$\overline{a}(b \oplus c) + \overline{a\overline{(b \oplus c)}} = \overline{a}x + a\overline{x} = a \oplus x = a \oplus b \oplus c$$

7 Using multiplexers

Multiplexers, especially two-input multiplexers, can be considered as an elementary logic component. Their internal wiring was described in the chapter Digital Components and Technologies.

Two-input multiplexers can be used to decompose a circuit. Instead of a single complex combinational circuit with n input variables, we obtain two simpler circuits, each with $n - 1$ input variables.

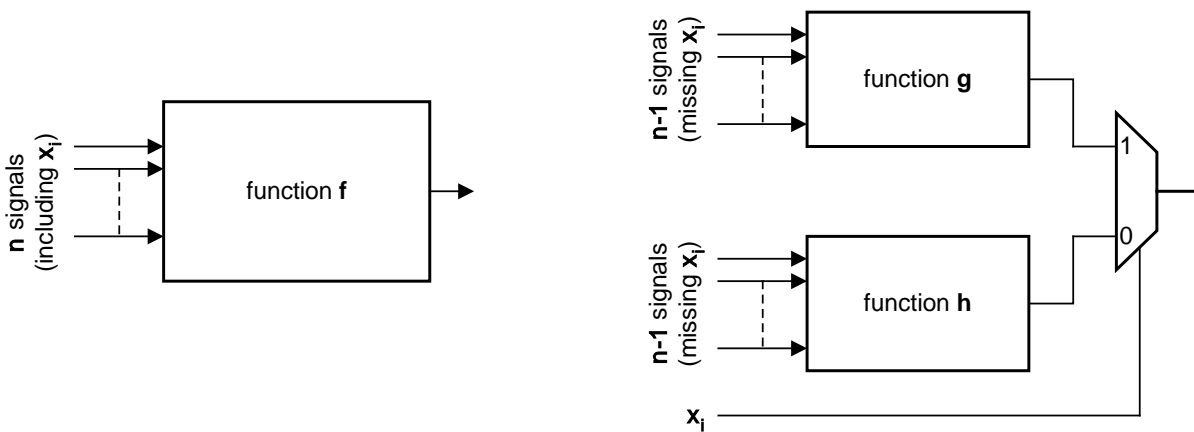


Fig. 8: Circuit implementation of Shannon's decomposition theorem

One (selected) input signal is used to control the multiplexer - this signal is then no longer fed into the decomposed circuits. This procedure can be repeated to select another variable and insert another multiplexer. Now we will have 4 even simpler circuits. By repeating this procedure it is possible to reach the final state, when the whole system will consist only of two-input multiplexers. The decomposition can of course be terminated at any stage.

The basis decomposition is **Shannon's decomposition theorems**:

$$f(x_n, x_{n-1}, \dots, x_i, \dots, x_1) = x_i \cdot f(x_n, x_{n-1}, \dots, 1, \dots, x_1) + \bar{x}_i \cdot f(x_n, x_{n-1}, \dots, 0, \dots, x_1)$$

Symbolically it is possible to write:

$$f(n \text{ variables}) = x_i \cdot g(n-1 \text{ variables}) + \bar{x}_i \cdot h(n-1 \text{ variables})$$

The relationship can easily be deduced: if the variable $x_i = 1$ (and therefore $\bar{x}_i = 0$), then the first term on the right hand side is valid and in the place of the variable x_i will be 1; the second term will be zero. Similarly, if the variable $x_i = 0$ (and therefore $\bar{x}_i = 1$), then the first term on the right hand side will be zero and the second term will be valid; in the place of the variable x_i will be 1. Both the functions on the right will have only $n-1$ variables.

When performing decomposition, it is actually a matter of **factoring out** the variables x_i and \bar{x}_i before the parentheses.

Example:

Let's decompose the function

$$y = dc\bar{b} + d\bar{c}ba + \bar{d}a \quad \text{according to } \mathbf{d}: \quad y = d(c\bar{b} + \bar{c}ba) + \bar{d}(a)$$

We get a decomposition on the inputs MX1 (see Fig. 9). Next, we will decompose the functions on the MX1 inputs according to **c** :

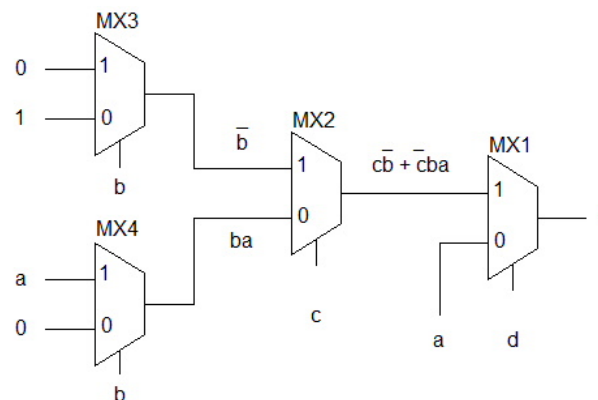
At input "1": $c(\bar{b}) + \bar{c}(ba)$

At input "0": the variable a can no longer be decomposed, so it remains.

We get a decomposition on MX2 inputs. Next, we will decompose the functions on the MX3 inputs according to **b** :

At input "1": $\bar{b} = b(0) + \bar{b}(1)$

At input "0": $ba = b(a) + \bar{b}(0)$



Giant. 9 : Implementation of the multiplexer function

Nothing can be decomposed further. The whole circuit is now implemented only by two-input multiplexers.

The MX3 connection shows that the multiplexer and the constants 0 and 1 can be used to implement an inversion, the MX4 connection shows that the multiplexer can be used to implement an AND gate.

The analysis of the implementation price according to Fig. 9 and the implementation of the original expression by AND-OR (NAND-NOR) components shows that the solution with multiplexers is more economical. An interesting conclusion is that the multiplexers alone can serve as **elementary components** for the implementation of **any** logic functions.

Another case of multiplexers is a **multi-input multiplexer**, which is essentially a signal switcher that serves many purposes. One of them is a very easy implementation of combinational circuits, but only simple ones. First, we derive the expression for its function. Assume a multiplexer with a group of k signals for input selection. This is a vector with the meaning of **address**. With the k components, 2^k combinations can be distinguished and thus one of the 2^k possible input channels can be selected - see Fig. 10 on the left for the case of a multiplexer with eight inputs.

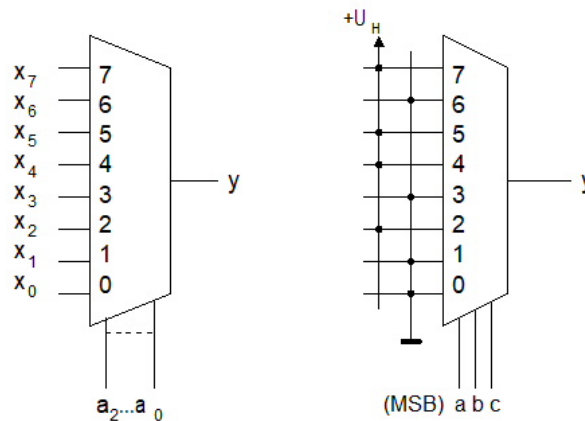


Fig. 10: Eight-input multiplexer

At address 000, input x_0 will be selected, at address 001 (a_2 is MSB) input x_1 will be selected, etc. In general, the following will apply to output y :

$$y = \bar{a}_2 \bar{a}_1 \bar{a}_0 \cdot x_0 + \bar{a}_2 \bar{a}_1 a_0 \cdot x_1 + \dots + a_2 a_1 a_0 \cdot x_7$$

The expression describes the function of the signal switch x_i and the possible application of the multiplexer in this connection is clear. However, if we apply the constants '0' or '1' to the inputs x_7 to x_0 , only some products $\tilde{a}_2 \tilde{a}_1 \tilde{a}_0$ will be left in the expression for y (the symbol "~" means possible but not necessary inversion). Obviously, we get an expression for y formally identical to the **sum of the minterms** created from the variables a_2, a_1, a_0 . It is thus possible to implement any function of 3 variables. Similar considerations can be applied to higher multiplexer dimensions and numbers of variables. In Fig. 10 on the right, the function $y = \sum m(2,4,5,7)$ is implemented.

Obviously, this implementation does not need any minimization, because the complexity of the circuit is the same for all functions of 3 variables. Only the connection of the inputs with U_L or U_H changes. This connection can be easily re-arranged so that it is programmable - through drilled holes or interrupted paths on the printed circuit board, in the integrated version by programmable switches, etc.

8 Realization of memory combinational circuits

In this case, only address inputs and data outputs of the memory are used. The content of the memory is assumed to be constant and defined before the start of its operation as a combinational circuit. The memory has n address inputs and k data outputs. The input variables of the combinational functions correspond to the individual bits of the address and the output variables correspond to the individual bits of the data - see Fig. 11. By means of a memory of a suitable size it is possible to implement a group of k arbitrary functions of n variables. The proof is simple - the table of contents of the memory including all addresses and data stored on them is formally completely identical to the truth table of the function.

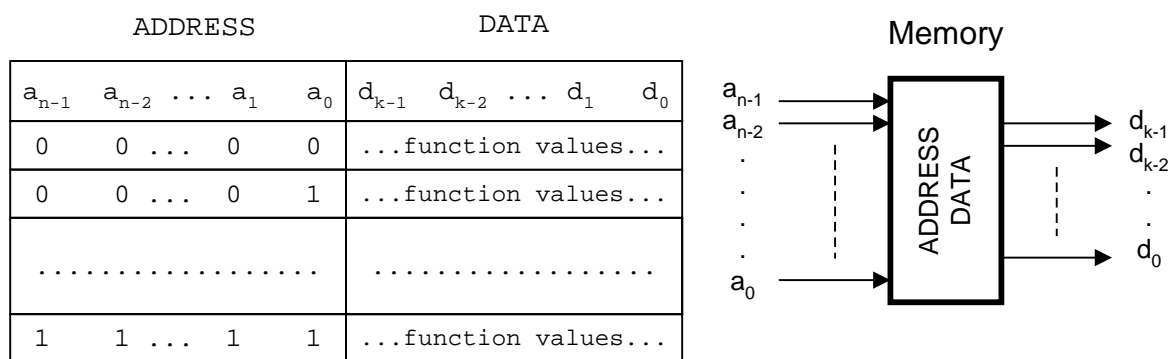


Fig. 11: Memory as a combinational circuit

The memory must first be programmed. The content that will be transferred to memory is stored in the **programmer** or computer as a **memory image**. The entire area is reset first. Then data will be written to the appropriate addresses according to the required function. After creating the whole memory image, the whole area is programmed (transferred) to the given memory, which will then serve as a combinational circuit. The whole process can be performed by a computer.

The contents of the image memory can be generated by various methods. The obvious procedure is to copy the truth table of the function to the image memory. In the language of logic functions, this means entering by individual minterms. However, this procedure becomes too laborious with larger truth tables.

A logic function can also be expressed in the form of the sum of implicants (SPF). Line numbers in the image memory on which ones are written are derived from the individual implicants - see the example below.

The important fact is that it does not matter whether the expression is minimized or **not** - all methods eventually lead to the same size and the same contents of the memory. However,

there can be a significant difference in the laboriousness of determining the contents of the image memory.

The implementation of logic functions by memories is suitable for various code converters, tables of function values, etc. The advantage is lost for very simple functions of a **large number** of variables. Consider, for example, the AND function of 16 variables. A single logic device would suffice, but the memory would have 2^{16} address bits (!). However, memory is used very often to implement functions of a small number of variables (e.g. 4 to 6), especially in programmable logic devices - **PLD**. Electrically programmable and erasable **EEPROM** and **FLASH** memories are used, also **SRAM** memories (static version, SRAM). In case of SRAM, it is always necessary to ensure that it is loaded with the required content after the power supply has started.

Example:

Design a combinational circuit implemented by means of a memory:

$$f(a, b, c) = \bar{b} + \bar{a}\bar{c} + ac$$

The implicants of the function from the example are expressed numerically; we sort the variables, for example, as follows:

(MSB) a, b, c (LSB)

Substitute zero instead of the inverted variable, one instead of the non-inverted variable. Absent variables are marked with a hyphen.

Implicant \bar{b} :	- 0 -
Implicant $\bar{a}\bar{c}$:	0 - 0
Implicant ac :	1 - 1

The implicants \bar{b} misses two variables, thus it covers four minterms. After adding the absent variables to 0 or 1 in all possible combinations, the following minterms are created :

0 0 0 (decimal 0)
 0 0 1 (decimal 1)
 1 0 0 (decimal 4)
 1 0 1 (decimal 5)

The implicant $\bar{a}\bar{c}$ covers the following minterms:

0 0 0 (decimal 0) - is already covered once
 0 1 0 (decimal 2)

The implicant ac covers the following minterms:

1 0 1 (decimal 5) - is already covered once, but it does not matter
 1 1 1 (decimal 7)

The function therefore can be written as:

$$y = \sum m(0, 1, 2, 4, 5, 7)$$

A condensed notation of the function as the sum of the minterms is described in the chapter "Combinational circuits".

9 Basic combinational function blocks

When designing digital systems, larger function blocks are used if possible. They can be implemented as integrated circuits, usually medium integration (MSI), or as standard blocks in computer design programs. Here the basic and most common blocks will be gradually discussed.

9.1 Decoder

The decoder converts the binary code to a *1 of N* code. With this code, there is always an active state on one and only one output (here we assume that the active state is '0'). The position of this active output corresponds to the binary number of k bits delivered to the input. It holds that $N = 2^k$. In Fig. 12 is a truth table of a decoder from a three-bit binary code to a *1 of 8* code and also a schematic representation of the decoder.

input			output							
MSB		LSB								
x_2	x_1	x_0	y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	1	1	1	1	1	1	1	0
0	0	1	1	1	1	1	1	1	0	1
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	1	1	0	1	1	1	1
1	0	1	1	1	0	1	1	1	1	1
1	1	0	1	0	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1

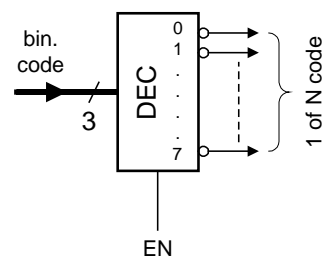


Fig. 12: Truth table of the decoder and its schematic symbol

Decoders are very often equipped with a **blocking** (or selection) input marked EN (enable), by which all outputs can be put into an inactive state, i.e. '1'. With a non-inverted EN in the figure, the decoder would be blocked at $EN = 0$ and unlocked at $EN = 1$. It is necessary to distinguish the blocking of **three-state devices** from the blocking of the **decoder** - in the first case the blocked output is forced to a high-impedance state, in the second case to an inactive state (i.e. state 1). The EN input can also be **inverted**.

The decoder is typically used for selecting one of several circuits. Very often these are three-state circuits connected to a common bus - see Fig. 13. Their selection inputs are usually inversed (\overline{CS}) and the inversed outputs of the decoder correspond to this. When the decoder is blocked, no three-state circuit from the group is selected.

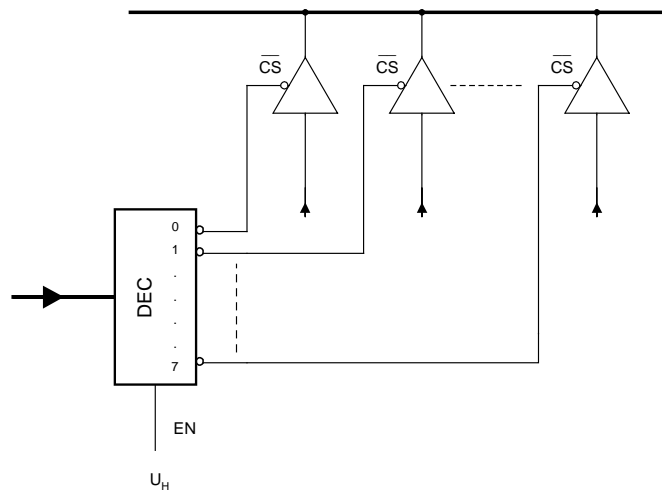


Fig. 13: Controlling the circuits on the bus by the decoder

The internal connection of the decoder is simple. Each output signal (usually inverted) is formed by the logical product of all input signals.

9.2 Multiplexer

The multiplexers and their use have been discussed in the previous text. Multiplexer has address inputs (binary) and inputs of switched signals. These are numbered and the numbers indicate at which value of the address the given input signal is converted to output. Fig. 14. left shows the internal connection of the four-input multiplexer using elementary logic elements. Another possibility is in the same picture on the right, where a part of the multiplexer (here 8-input) is a decoder, which controls the CMOS switches (shown symbolically). This connection is also used in analog circuits.

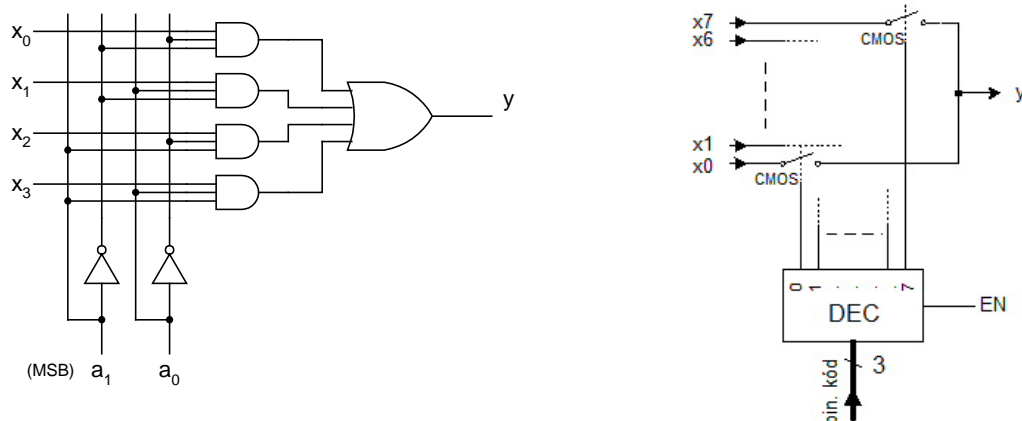


Fig. 14: Two variants of internal connection of a multiplexer

9.3 Demultiplexer

A demultiplexer is the opposite of a multiplexer. It has one input and several outputs. The address in the binary code selects the output to which the signal from the input is transferred. The status of all other outputs is '0'.

9.4 Priority encoder

The opposite of a decoder is an **encoder**. It converts *1 of N* code to binary code. However, it is necessary to ensure that exactly one signal is always active at its inputs and the others are inactive - as it would be at the decoder outputs. This is usually not possible, so a **priority encoder** is used more often.

For a priority encoder, it is permissible for **more than one** input signal to be active (assume the active state as '1'). Inputs are given **priority**. The simplest is the priority graded according to the connection. It is fixed for each input and cannot be changed. We can, for example, set the priority of the input with the number 0 as the lowest, the input with the number 1 as the higher, etc. until the priority of the input with the highest number as the highest.

Fig. 15 shows an example of a priority encoder with 8 inputs (x) in the code *1 of 8* and three outputs (y_2, y_1, y_0) in the binary code. A separate output z is used to inform that at least one input signal is active - if the values '0' are on all inputs, the binary code y_2, y_1, y_0 on the output u does not make sense and the values can be considered unspecified. The number 000 on the output is intended for active x_0 .

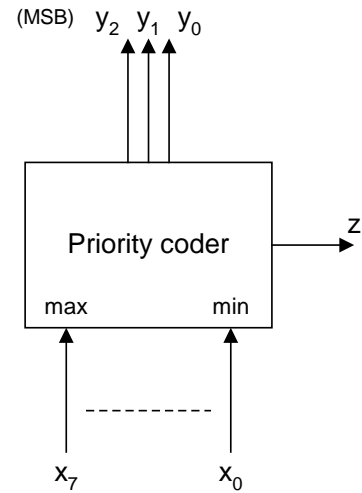


Fig. 15: Inputs and outputs of the 8- input priority encoder

When solving the internal circuits of the priority encoder, we will face a problem of a large number of its inputs. For example with 8 input variables, the truth table should have a full 256 rows (!). However, it can be drastically simplified by applying the properties of unspecified states - see chapter Combinational circuits. If the input with the highest priority is active (here x_7), the outputs must have state 111 (= 7) regardless of the inputs with the lower priority - i.e. unspecified states can be filled in the remaining entries of the first line. If the signal x_6 is active and x_7 is inactive, the outputs must have state 110 (= 6) and signals $x_5 \dots x_0$ are unspecified, etc. ... up to x_0 . The table was thus shortened to only nine rows - see Fig. 16. The circuit diagram will not be complicated and will be further simplified by applying group minimization. The following relations apply to individual output variables:

inputs								outputs (MSB)			
x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	y_2	y_1	y_0	z
1	-	-	-	-	-	-	-	1	1	1	1
0	1	-	-	-	-	-	-	1	1	0	1
0	0	1	-	-	-	-	-	1	0	1	1
0	0	0	1	-	-	-	-	1	0	0	1
0	0	0	0	1	-	-	-	0	1	1	1
0	0	0	0	0	1	-	-	0	1	0	1
0	0	0	0	0	0	1	-	0	0	1	1
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	-	-	-	0

$$y_2 = x_7 + \bar{x}_7 x_6 + \bar{x}_7 \bar{x}_6 x_5 + \bar{x}_7 \bar{x}_6 \bar{x}_5 x_4$$

$$y_1 = x_7 + \bar{x}_7 x_6 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 x_3 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 x_2$$

$$y_0 = x_7 + \bar{x}_7 \bar{x}_6 x_5 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 x_3 + \bar{x}_7 \bar{x}_6 \bar{x}_5 \bar{x}_4 \bar{x}_3 \bar{x}_2 x_1$$

$$z = x_7 + x_6 + x_5 + x_4 + x_3 + x_2 + x_1 + x_0$$

Fig. 16: Truth table of the priority encoder and expressions for the outputs

The above priority encoder is "centralized" in the sense that all input signals are routed to the inputs of one circuit and all output signals are generated in the same circuit. However, the priority encoder can be designed in a different way, as **decentralized** - see Fig. 17.

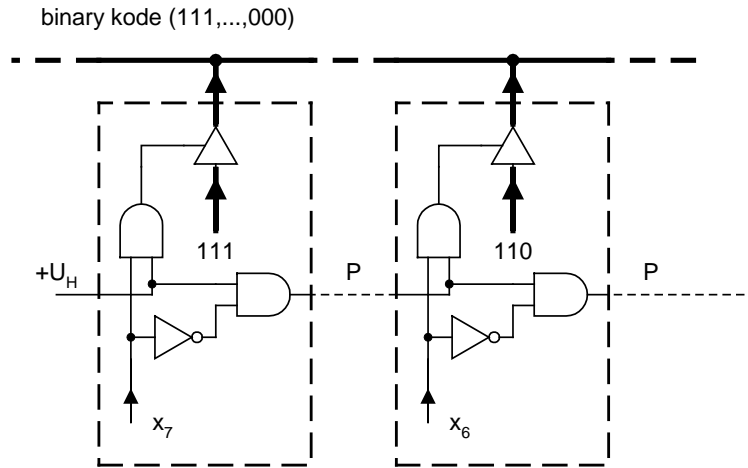


Fig. 17: Priority chain

The **priority chain** selects the **active** input according to its priority. Its repeating identical parts (framed by dashed lines) are placed in individual modules of the system. They are connected by transmission signals P . If a higher priority input signal (e.g. x_7) is active, all transmission signals to the right of it are zero and the passage of all signals x_6 , x_5 , etc. is blocked. Therefore, only one module with active input signals enable the tri-state device through which is the module connected to the bus. Then an appropriate combination of bits appears on the bus which identifies the module.

The function of the encoder with the priority chain is therefore identical to the function of the centralized decoder. The difference is in the construction - the second variant is suitable for a **modular** arrangement of the system. The priorities of individual modules are determined by the order of their interconnection in the chain.

9.5 Digital comparator

The digital comparator compares the size of two numbers in binary code. Most often is used The information about the **equivalence** of two numbers. This is obtained very simply by comparing pairs of bits at the same position. If we denote one number A and its bits A_i , and the other B and its bits B_i , then the following holds for the output K :

$$K = (a_{n-1} \equiv b_{n-1})(a_{n-2} \equiv b_{n-2}) \dots (a_0 \equiv b_0)$$

The symbol " \equiv " indicates equivalence (opposite to XOR). Equivalences at the position of all bits of both numbers must apply simultaneously. There are also comparators evaluating the inequalities $A > B$ and $A < B$ on the other two outputs. From these outputs, along with the output for $A = B$, more information can be derived: $A \geq B$ or $A \leq B$ - see Fig. 18.

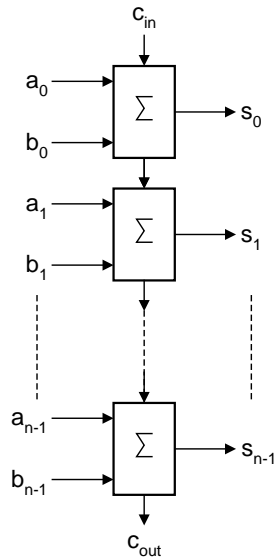


Fig. 20: Long path of carry

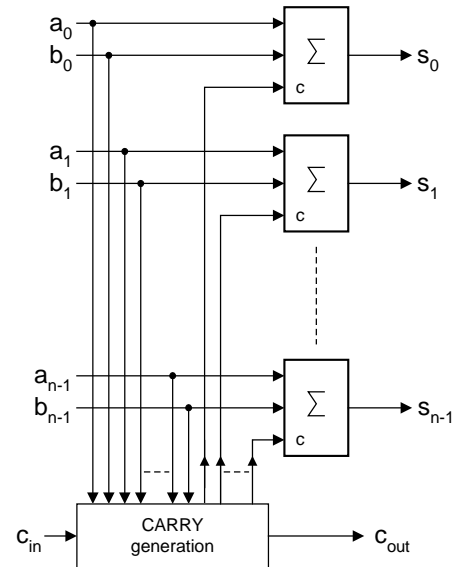


Fig. 21: Carry look-ahead adder

Fast adders are therefore designed differently. The carry inputs and outputs are not connected in a chain, but carry signals are generated **simultaneously** for all stages by a circuit which processes all input bits a_i and b_i at once. Since each logic combinational function can be implemented by a circuit with a length of at most three stages (NOT - AND - OR), the delay of all circuits for the carry generation is the same and minimal. However, their complexity increases from the lowest to the highest order and becomes unmanageable for large numbers (e.g. 32 bit). Adders of this size must be formed from smaller modules (8 bit).

The adder is then called a **look-ahead adder** and is significantly faster, but also more complex (Fig. 21).

Input c_{in} is in normal operation grounded, but in the state '1' increases the value of a number S by one - it can be used in miscellaneous algorithms. The c_{out} output can inform about a number **overflow**, which is a situation where the sum of two n - bit numbers is so large that it cannot be expressed by an n - bit result. The resulting number is incorrect - it lacks the most significant bit $n+1$.

9.7 Subtractor

Subtracting a number is solved as adding a number with the opposite sign. If negative numbers are expressed as **twos complement**, the inversion of the sign is simple - it means the inversion of all bits and the subsequent addition of number one. The following Fig. 22 shows a circuit solution enabling the addition and subtraction of two numbers.

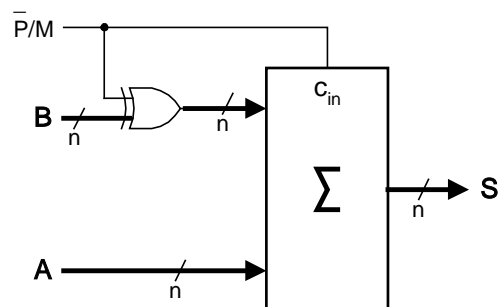


Fig. 22: Adder and subtractor of numbers in twos complement

The input \bar{P}/M selects operation $A+B$ (when $\bar{P}/M = '0'$) or $A-B$ (when $\bar{P}/M = '1'$). The group of n XOR gates works as a **controlled inversion**. With $\bar{P}/M = '1'$, B is inverted and at the same time $c_{in} = '1'$. This adds one and inverts B . When $\bar{P}/M = '0'$, B is not inverted and $c_{in} = '0'$.

9.8 Majority circuit

The majority circuit M_n , where n is an **odd number**, has n inputs and 1 output. The output is the value that **most** of the n inputs have. The most common circuit is M_3 . For this circuit with inputs a, b, c and output y it is easy to derive:

$$y = ab + ac + bc + abc$$

The majority function is very often used in telecommunication systems to **correct** signal **errors** distorted by noise and interference. Another use is in control systems with increased **reliability**. The system is manufactured in several (M) identical copies, the outputs of which are processed by the majority circuits. Any single failure is corrected.