Theory of Logic Circuits

Laboratory manual

Exercise 1

Combinational switching circuits

© 2008 Urszula Stańczyk, Piotr Czekalski (edt.)

1. Switching devices and circuits

The switching devices posses the property of having associated with them two distinct physical states. The actual signal values of these states differ from one device to another upon their design, but still these devices are controlled by two-valued signals and are capable of producing two-valued signals.
If a circuit is constructed of such bi-stable devices, it is capable of receiving two-valued signals at its input and producing two-valued signals on its output. Such a circuit is called a switching circuit.

Any switching circuit can be represented by the switching functions of its outputs. On the other hand, the switching circuit can implement a switching function.
If the output values of the circuit are at any time determined strictly by the inputs at that time, such circuit is called a combinational circuit.

2. Gates

Gates are the switching devices often used to implement switching functions.

The most commonly used gates are shown below in the table.

Logical symbol	Function performed	Name
A, B → Y	Logical sum $Y = A + B$	OR
A, B → Y	Logical multiplication $Y = A \cdot B$	AND
A → Y	Logical complementation $Y = \overline{A}$	NOT
A, B → Y	Complemented sum $Y = \overline{A + B}$	NOR
A, B → Y	Complemented product $Y = \overline{A \cdot B}$	NAND
A, B → Y	Exclusive sum $Y = A \oplus B = A \cdot \overline{B} + \overline{A} \cdot B$	XOR
A, B → Y	Complemented exclusive sum $Y = \overline{A \oplus B} = A \cdot B + \overline{A} \cdot \overline{B}$	XNOR

Of course, for the simplification purposes, all gates listed above have no more than two inputs, but there are multiple-input gates as well.

Gates usually are used in sets corresponding to different systems functionally complete, i.e. such sets of gates, with which it is possible to implement any switching function.
The most popular systems functionally complete are as follows:

- OR, AND, NOT
- OR, NOT
- AND, NOT
- NAND
- NOR

The switching devices used in the laboratory are of TTL technology, which means that all unconnected inputs are assumed to have the logical value "1".

3. Synthesis of a combinational switching circuit

The aim of a synthesis of a combinational circuit is to obtain the output functions and then a logical diagram of the circuit when given its algorithm of work.
The output functions of the circuit may be in different forms but they should always be optimal, taking into consideration the type and the number of switching devices used for implementation.

The stages of designing process are as follows:
1. Obtaining the algorithm of work for the circuit, basing on logical statements or a timing chart illustrating working conditions.
2. Creating truth tables or Karnaugh maps for the output functions.
3. Obtaining the canonical forms for the output functions.
4. Obtaining the minimal output functions.
5. Obtaining the output functions in forms adequate to implementation.
6. Obtaining a logical diagram.

Depending on a set of elements used for implementation, some stages of this process are not required. For example, to implement a function using multiplexer or demultiplexer the minimal form of a function is not necessary (such implementation is described in Exercise 11).
Minimisation process may base on one of several methods, such as transformations of a logical expression, using Karnaugh maps, Quine – McClusky's method, etc. For functions with a great number of variables (this "greatness" is, of course, relative), especially those that are weakly specified, we may use Kazakov method (described in Exercise 12&13).
The most commonly used are, however, Karnaugh maps (for up to 6 variables) as offering very convenient representation of a function.

3.1. Example

Design a switching circuit with four inputs $x_1x_2x_3x_4$ and one output Y, detecting two or three ones on any of its inputs and indicating that with "1" on the output.

As we have only four input variables, we may either use a truth table or a Karnaugh map to obtain a logical function for the output.
From the truth table it is possible to obtain only canonical forms of the function. The function is fully specified (i.e. there is no such a combination of inputs for which we don't care about the output – don't care conditions do not occur), which means that by writing SoP form we know instantly the PoS form.
The Karnaugh map representation, apart from giving the canonical forms of the function, offers the possibility of minimisation. Creating groups of ones and zeros we obtain the minimal SoP and PoS forms. The number of groups in the example depends on a decision whether we want to get the solution that is hazard-free. As hazards in general are described in Exercise 7, we assume that we don't consider them for all tasks given within Exercise 1.

Truth table

$x_1x_2x_3x_4$	Y
0 0 0 0	0
0 0 0 1	0
0 0 1 0	0
0 0 1 1	1
0 1 0 0	0
0 1 0 1	1
0 1 1 0	1
0 1 1 1	1
1 0 0 0	0
1 0 0 1	1
1 0 1 0	1
1 0 1 1	1
1 1 0 0	1
1 1 0 1	1
1 1 1 0	1
1 1 1 1	0

Karnaugh map

x_1x_2 \ x_3x_4	00	01	11	10
00	0	0	1	0
01	0	1	1	1
11	1	1	0	1
10	0	1	1	1

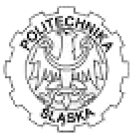
Y

$Y=\Sigma(3,5,6,7,9,10,11,12,13,14)x_1x_2x_3x_4$
$Y=\Pi(0,1,2,4,8,15)x_1x_2x_3x_4$

SoP Y = $\overline{x_1}\cdot x_2\cdot x_4 + x_1\cdot x_2\cdot \overline{x_3} + x_1\cdot \overline{x_2}\cdot x_3 + x_1\cdot \overline{x_3}\cdot x_4 +$
$+ \overline{x_1}\cdot x_3\cdot x_4 + x_2\cdot x_3\cdot \overline{x_4}$

PoS Y = $(x_1+x_2+x_3)\cdot(x_1+x_2+x_4)\cdot(x_1+x_3+x_4)$
$\cdot(x_2+x_3+x_4)\cdot(\overline{x_1}+\overline{x_2}+\overline{x_3}+\overline{x_4})$

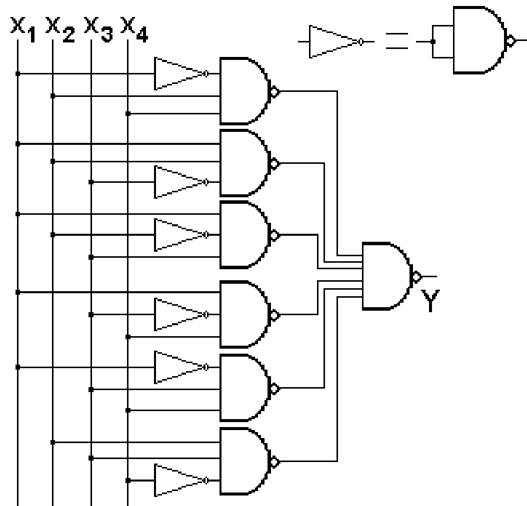
Which one of these forms above should be chosen for implementation depends on the logical elements we are to use.
The most commonly used systems functionally complete are two: one using only NAND gates and the other using NOR gates.
Usually with NANDs the least number of transformations and elements used for implementation is achieved from the SoP form, while for NOR gates it is from the PoS form.
It is not, however, the general rule.
The implementation with NANDs requires adding double complementation, applying DeMorgan's theorem to SoP form and then in one step we get the function containing only complemented products.



$$Y=\overline{\overline{\bar{x}_1 \cdot x_2 \cdot x_4 + x_1 \cdot x_2 \cdot \bar{x}_3 + x_1 \cdot \bar{x}_2 \cdot x_3 + x_1 \cdot \bar{x}_3 \cdot x_4 + \bar{x}_1 \cdot x_3 \cdot x_4 + x_2 \cdot x_3 \cdot \bar{x}_4}}$$

$$Y=\overline{\overline{\bar{x}_1 \cdot x_2 \cdot x_4} \cdot \overline{x_1 \cdot x_2 \cdot \bar{x}_3} \cdot \overline{x_1 \cdot \bar{x}_2 \cdot x_3} \cdot \overline{x_1 \cdot \bar{x}_3 \cdot x_4} \cdot \overline{\bar{x}_1 \cdot x_3 \cdot x_4} \cdot \overline{x_2 \cdot x_3 \cdot \bar{x}_4}}$$

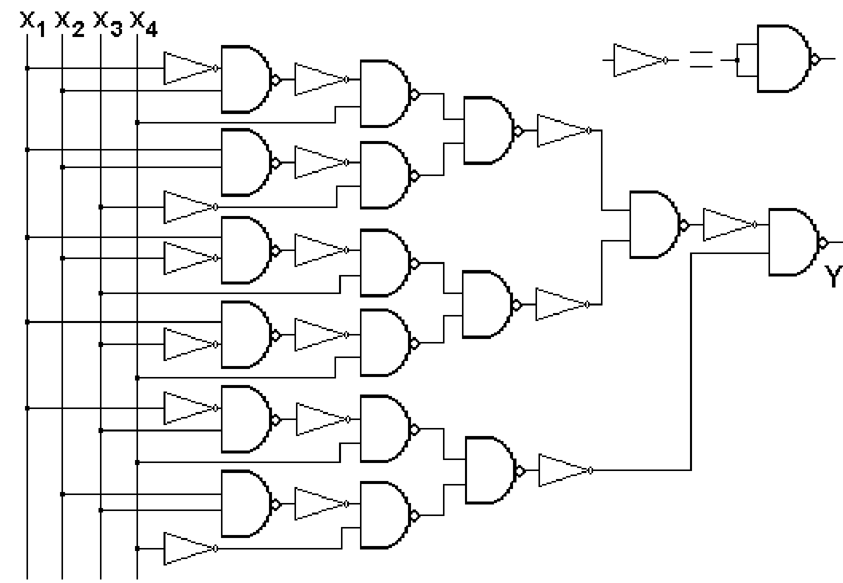
The number of multiple-input NAND gates is 7 (not counting 6 NOT gates, which also have to be, however, implemented using NANDs) and a logical diagram looks as follows.

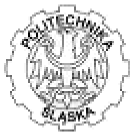


As for implementation in reality we rarely have access to multiple-input gates and most commonly used are 2-input gates, we have to be able to do it.
We need to transform the function in such a way that it is adequate to such implementation. It is done by adding double complementation for all cases of more than two inputs needed for a gate as shown below.

$$Y=\overline{\overline{\overline{\overline{\overline{\overline{\overline{\overline{\bar{x}_1 \cdot x_2}} \cdot x_4} \cdot \overline{\overline{\overline{x_1 \cdot x_2}} \cdot \bar{x}_3}}} \cdot \overline{\overline{\overline{\overline{\overline{x_1 \cdot \bar{x}_2}} \cdot x_3} \cdot \overline{\overline{\overline{x_1 \cdot \bar{x}_3}} \cdot x_4}}}}} \cdot \overline{\overline{\overline{\overline{\overline{\bar{x}_1 \cdot x_3}} \cdot x_4} \cdot \overline{\overline{\overline{x_2 \cdot x_3}} \cdot \bar{x}_4}}}}$$

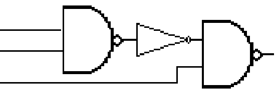
Then a logical diagram looks as follows.



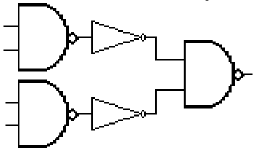


When we compare these two diagrams we may easily extract from them some fragments showing how to implement multiple-input NANDs with 2-input gates.

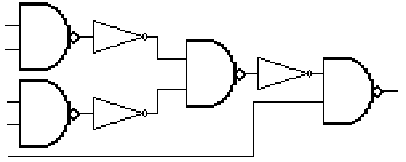
3-input gate



4-input gate



5-input gate



For implementation with NORs the process is analogous.

Sometimes, due to simpler logical expressions it is recommended to use SoP form for implementation with NORs or PoS form for implementation with NANDs. Then the transformations are as follows.

$$Y=\overline{\overline{\overline{\overline{\bar{x}_1 \cdot x_2 \cdot x_4}} + \overline{\overline{x_1 \cdot x_2 \cdot \bar{x}_3}} + \overline{\overline{x_1 \cdot \bar{x}_2 \cdot x_3}} + \overline{\overline{x_1 \cdot \bar{x}_3 \cdot x_4}} + \overline{\overline{\bar{x}_1 \cdot x_3 \cdot x_4}} + \overline{\overline{x_2 \cdot x_3 \cdot \bar{x}_4}}}} =$$

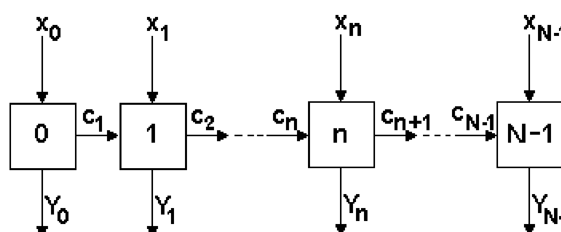
$$= \overline{\overline{\overline{x_1 + \bar{x}_2 + \bar{x}_4} + \overline{\bar{x}_1 + \bar{x}_2 + x_3} + \overline{\bar{x}_1 + x_2 + \bar{x}_3} + \overline{\bar{x}_1 + x_3 + \bar{x}_4} + \overline{x_1 + \bar{x}_3 + \bar{x}_4} + \overline{\bar{x}_2 + \bar{x}_3 + x_4}}}$$

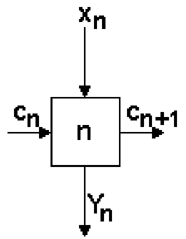
$$Y=\overline{\overline{\overline{\overline{(x_1 + x_2 + x_3)}} \cdot \overline{\overline{(x_1 + x_2 + x_4)}} \cdot \overline{\overline{(x_1 + x_3 + x_4)}} \cdot \overline{\overline{(x_2 + x_3 + x_4)}} \cdot \overline{\overline{(\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4)}}}} =$$

$$= \overline{\overline{\overline{\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3} \cdot \overline{\bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_4} \cdot \overline{\bar{x}_1 \cdot \bar{x}_3 \cdot \bar{x}_4} \cdot \overline{\bar{x}_2 \cdot \bar{x}_3 \cdot \bar{x}_4} \cdot \overline{x_1 \cdot x_2 \cdot x_3 \cdot x_4}}}$$

4. Iterative switching circuits

Iterative circuits are such combinational circuits that consist of a set of identical cells connected in a cascade, as shown in the picture below.





In designing iterative networks we design a typical cell such as *n*-cell and than we are able to use it as many times as necessary for a circuit. What’s more, if there is a need for changing the number of cells (inputs) in the circuit, we can do it without actually changing the design or the structure of the whole circuit.

A description of the *n*-cell is given in the form of logical expressions using as variables all inputs to the cell. These inputs are inputs of the circuit (x_n) and a carry signal. On the other hand, carry

signal passes on information gathered in some previous cells. This makes the whole process and equations recursive. As it is with every recursive procedure, to work, it requires to be provided with some assumptions for a start. That is why, to make a solution for an iterative circuit complete, apart from specifying the *n*-cell in a form of equations or logical diagrams for its output and carry signal, we also have to give assumptions for incoming carry signal to the first cell in the circuit. Its index “1” or “0” depends on a convention. We have N cells either from 0 to N-1 or from 1 to N.

Among different iterative circuits we recognise the group connected with arithmetical operations – there could be an adder, a subtractor or a comparator of binary numbers. Both an adder and a subtractor should perform their operations starting with the least significant bits of binary numbers and proceed towards the most significant bits. With a comparator, however, there is a different case. The binary numbers may be compared either starting from the least or the most significant bits.

These circuits work, of course, in parallel, unlike serial arithmetical circuits, which then must be sequential switching circuits instead of combinational ones.

5. Codes and code converters

Coding is an action of assigning some symbols to different information. The set of symbols is called a code and a code using all possible combinations of symbols is called a complete code. Codes that represent decimal numbers as binary are called binary-decimal.
The most often used codes are listed below.

Decimal digit	Natural binary code	Gray+3 code	Watts' code	Excess 3 (Plus 3) code	1 out of 10 code
0	0000	0010	0000	0011	0000000001
1	0001	0110	0001	0100	0000000010
2	0010	0111	0011	0101	0000000100
3	0011	0101	0010	0110	0000001000
4	0100	0100	0110	0111	0000010000
5	0101	1100	1110	1000	0000100000
6	0110	1101	1010	1001	0001000000
7	0111	1111	1011	1010	0010000000
8	1000	1110	1001	1011	0100000000
9	1001	1010	1000	1100	1000000000

There is a group of codes called cyclic, which have such property that two consecutive numbers differ in only one bit (are logically adjacent).
It is very convenient to represent or create such codes with the help of Karnaugh map.

	00	01	11	10
00	0	1	2	3
01	7	6	5	4
11	8	9	10	11
10	15	14	13	12

Gray code

	00	01	11	10
00	0	1	2	3
01				4
11				5
10	9	8	7	6

Watts' code

Code converters are such switching circuits that operate on codes and they are divided into three categories:

- Encoders that translate a number from a 1 out of n code into any other code,
- Decoders, which transform a number in a code different to a 1 out of n code into this code,
- Translators that transform a number form one code to another but neither of them is a 1 out of n code.

As designing an encoder or decoder seems to be a task easy enough, in the example given we will show the process of designing a translator.

5.1. Example

Design a translator from the Gray code into the cyclic code given by the map below. Present a solution in a form of the minimal output functions.

	00	01	11	10
00	6	7	8	9
01	5	14	15	10
11	4	13	12	11
10	3	2	1	0

The first step, not mandatory but recommended, is to create a truth table where on the left we place the numbers in the input code (usually in increasing order) and on the right their equivalents in the output code.

Gray code g_3 g_2 g_1 g_0	Output code o_3 o_2 o_1 o_0
0 0 0 0	1 0 1 0
0 0 0 1	1 0 1 1
0 0 1 1	1 0 0 1
0 0 1 0	1 0 0 0
0 1 1 0	1 1 0 0
0 1 1 1	0 1 0 0
0 1 0 1	0 0 0 0
0 1 0 0	0 0 0 1
1 1 0 0	0 0 1 1
1 1 0 1	0 0 1 0
1 1 1 1	0 1 1 0
1 1 1 0	1 1 1 0
1 0 1 0	1 1 1 1
1 0 1 1	1 1 0 1
1 0 0 1	0 1 0 1
1 0 0 0	0 1 1 1

Then we need to create Karnaugh map for each of the output bits $o_3o_2o_1o_0$. The left side of the table informs us where we should put the appropriate value in the Karnaugh map. However, such detailed procedure of filling maps is very time-consuming.

To make the process faster, we should keep in mind that the order of values of the output code corresponds to the way of proceeding through the map with the input code. If we follow this order, we will get complete maps for all outputs much faster.

o_3:

g_3g_2 \ g_1g_0	00	01	11	10
00	1	1	1	1
01	0	0	0	1
11	0	0	0	1
10	0	0	1	1

o_2:

g_3g_2 \ g_1g_0	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	0	1	1
10	1	1	1	1

o_1:

g_3g_2 \ g_1g_0	00	01	11	10
00	1	1	0	0
01	0	0	0	0
11	1	1	1	1
10	1	0	0	1

o_0:

g_3g_2 \ g_1g_0	00	01	11	10
00	0	1	1	0
01	1	0	0	0
11	1	0	0	0
10	1	1	1	1

With the assumption that we do not consider the risk of hazards, the output functions in their minimal SoP form look as follows.

$$o_3 = \overline{g_3} \cdot \overline{g_2} + g_1 \cdot \overline{g_0} + \overline{g_2} \cdot g_1$$

$$o_2 = g_3 \cdot \overline{g_2} + g_2 \cdot g_1$$

$$o_1 = g_3 \cdot g_2 + g_3 \cdot \overline{g_0} + \overline{g_3} \cdot \overline{g_2} \cdot \overline{g_1}$$

$$o_0 = g_3 \cdot \overline{g_2} + \overline{g_2} \cdot g_0 + g_2 \cdot \overline{g_1} \cdot \overline{g_0}$$

6. Tasks to be performed during laboratory

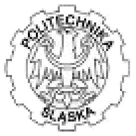
1. Design a combinational circuit controlling conditions of work for a water mixer. The mixer can be filled with cold, warm or hot water. The temperature of water t_w is indicated by two temperature sensors X_4 and X_5 (X_4=1 when $t_w \geq t_4$ and X_5=1 when $t_w \geq t_5$, where $t_4 < t_5$ and t_4, t_5 correspond to the level of temperature switching on the sensors).
   Cold water is poured by Z_1 valve, warm water by Z_4 valve and hot water by Z_2 valve. The are two outlets from a water container: Z_3 and Z_5. The level of water is indicated by three level sensors X_1, X_2, X_3, which switch on when water exceeds their level.

   The circuit should meet the following requirements:
   - Outflow of water through Z_3 when water temperature is $t_5 > t_w \geq t_4$ and the level is above X_3,
   - Outflow of water through Z_5 when water temperature is either $t_w > t_5$ or $t_4 > t_w$ and the level exceeds X_1,
   - Filling with warm and hot or cold water (depending on the water temperature t_w) when water is below X_3,
   - Filling the container with either cold and hot water, or only hot or only cold water (depending on a water temperature t_w) when water is above X_3,
   - Filling the container with either only cold or only hot water (depending on a water temperature t_w) when water is above X_2,
   - No filling (no matter what t_w is) when water exceeds X_1.

2. Design an iterative switching circuit detecting an odd number of ones on its N inputs and indicating that with “1” on the output from the last cell in the circuit.

3. Design an iterative switching circuit detecting a sequence 111 on any three consecutive bits and indicating that with “0” on the output from the cell corresponding to the last bit of the sequence.

4. Design a circuit comparing two 2-bit binary numbers and indicating their equality with “0” on the output.



5. Design a circuit with two programming inputs *p* and *q* and two data inputs *a* and *b*. When the programming signals have the same values (i.e. $p = q$) the circuit should perform a logical sum of its data inputs, otherwise it should produce their logical multiplication.
6. Design a translator from Gray code into natural binary code.
7. Design a translator from the cyclic code presented in the map below into natural binary code.

	00	01	11	10
00	6	7	8	9
01	5	0	1	10
11	4	3	2	11
10	15	14	13	12

7. Instructions to follow

1. Solve all tasks before the exercise.
2. Implement the circuits specified by your supervisor (using given elements).
3. Present working circuits to your supervisor for acceptance.