Chapter 2
Combinational Network Design

Abstract This chapter deals with the transition from Boolean algebra to the implementation of combinational networks. Karnaugh maps provide a simple and intuitive method to represent and minimize functions with a few variables. The Variable-Entered Maps extend their usefulness and overcome some of their limitation. Standard networks such as decoders, multiplexers, and demultiplexers provide a wider view of combinational circuits, where the random approach of classical synthesis is enriched with an architectural one that introduces the concepts of programmable logic. Finally, this chapter deals with time behavior of non-ideal components and its implications on the synthesis.

2.1 Karnaugh Maps

In Chap. 1, we used two *"languages"* to describe a logical network: *Boolean expressions* and *truth tables*. Here, we will see a third language: *maps*.

Describing a function through its Boolean expression or truth table makes simplifying them logically an arduous task. Maps offer a way to write truth tables in a format that makes simplification easier. As we will see, maps order and highlight minterms because they have a geometric structure that makes applying the *absorption* and *logic adjacency* properties easy.

Absorption:

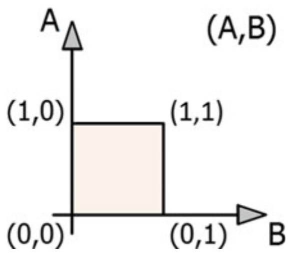
$$\begin{aligned} A + A\,B &= A & \qquad A + \overline{A}\,B &= A + B \\ A \cdot (A + B) &= A & \qquad A \cdot (\overline{A} + B) &= A\,B \end{aligned}$$

Logic Adjacency:

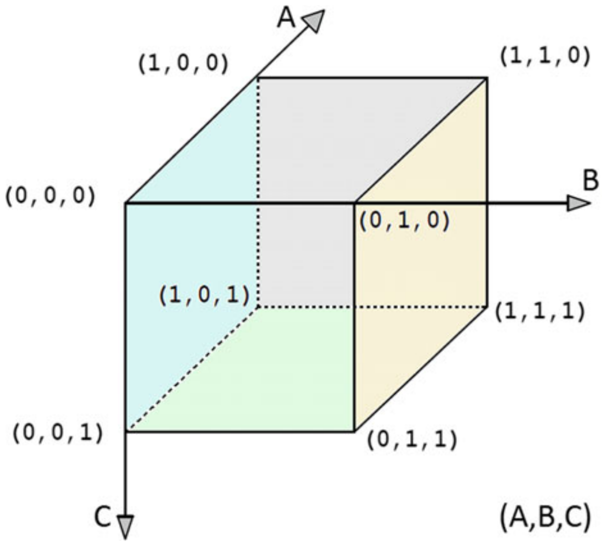
$$\begin{aligned} A\,B + A\,\overline{B} &= A \\ (A + B) \cdot (A + \overline{B}) &= A \end{aligned}$$

Given a two-variable function $f(A, B)$, we represent in a two-dimensional space (A, B) its set definition, which consists in the four combinations (A, B):

G. Donzellini et al., *Introduction to Digital Systems Design*,
https://doi.org/10.1007/978-3-319-92804-3_2



For a three-variable function $f(A, B, C)$, the set of definition can be represented in a three-dimensional space by placing each of the eight possible combinations of A, B, and C at the vertexes of a cube:



By moving along one edge of the cube from one vertex to another, *geometrically adjacent* one, we see that *only one variable* changes value.

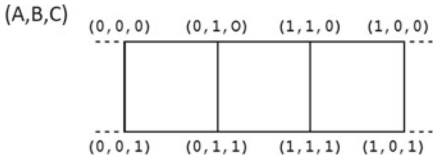
There is *logic adjacency* between two combinations of variables when they are at *distance 1*; that is, they differ by only one variable.

The cube above is an order-3 cube. Within its structure, various cubes of inferior order, or subcubes, can be identified. The order is the number of geometrical dimensions: a square is an "order-2" (two-dimensional) cube; an n-order cube will have 2^n vertexes. In the cube above, we see:

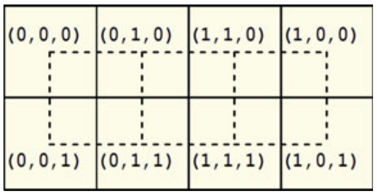
- Eight vertexes (*order-0 cubes*): Every vertex has one single combination of all the variables; for example, in the vertex (1,1,1) all the input variables have the value 1.

- Twelve edges (*order-1 cubes*): Each edge has two combinations of variables at distance 1. In other words, each edge is univocally located by one single pair of two-out-of-three variables; e.g., the upper left-hand edge of the figure is defined by B = 0 and C = 1.
- Six faces (*order-2 cubes*): Each face has four different combinations of variables, and it is univocally located by the value of a single variable; e.g., the farthest right face in the figure is defined by B = 1.

Now that, we have made these preliminary points, and let's draw the *map* corresponding to a three-dimensional cube by "cutting it out" and arranging its eight vertexes on a plane, maintaining as much as possible the same position they had in space.



The map is a lattice of cells, each containing one of the vertexes:

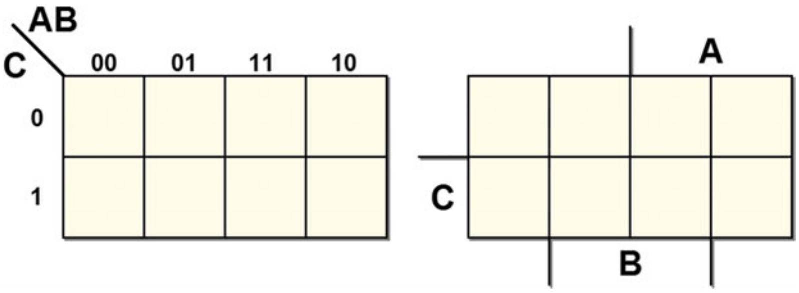


We see that:

- In the four lower cells, $C = 1$.
- In the four higher cells, $C = 0$.
- In the four cells to the right, $A = 1$.
- In the four cells to the left, $A = 0$.
- In the four central cells, $B = 1$.
- In the four lateral cells, $B = 0$.

So, we can represent the areas of the map by using the variables as the "coordinates" of the cells. One method is shown in the figure below (left), where the values of the variables identify the rows and columns.

The figure below (right) divides the map into geometrical areas corresponding to the values of the variables. This is the method we will use in this book. The name of each variable is drawn in correspondence with the area where it equals 1.



Since a map's cell corresponds univocally to one combination of variable values, it is possible to copy into the cell itself the corresponding function value $f(C, B, A)$ for that specific combination.

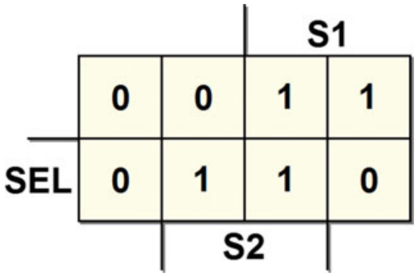
These types of maps are called *Karnaugh maps* (or *K-maps*).

2.2 Using Maps for AND-OR Synthesis

Here, we will look at the truth table of the multiplexer we examined in Chap. 1:

SEL	$S1$	$S2$	U
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

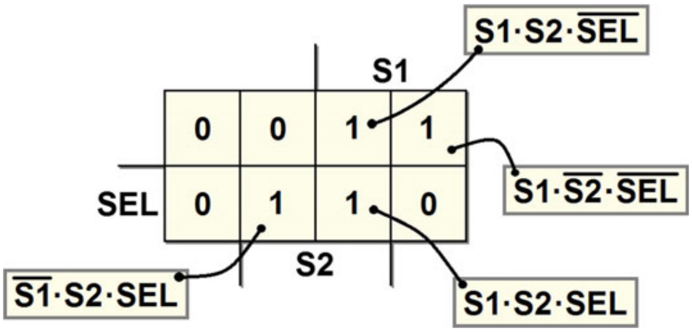
Let's copy the function values into a Karnaugh map:



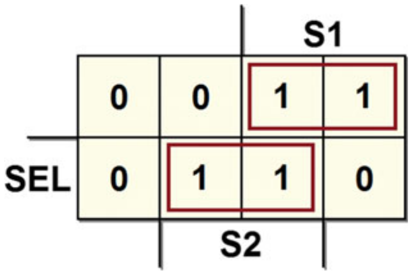
The expression of the AND-OR canonical form that we found was:

$$U = \overline{S1} \cdot S2 \cdot SEL + S1 \cdot \overline{S2} \cdot \overline{SEL} + S1 \cdot S2 \cdot \overline{SEL} + S1 \cdot S2 \cdot SEL$$

In the following figure, we see the minterms and their position on the map.


S1·S2·$\overline{\text{SEL}}$
S1
0
0
1
1
S1·$\overline{\text{S2}}$·$\overline{\text{SEL}}$
SEL
0
1
1
0
S2
$\overline{\text{S1}}$·S2·SEL
S1·S2·SEL

Remember that cells are adjacent if they are at distance 1, that is if they differ by one variable only. On the map, we can group together the 1s that are at distance 1 in two unidimensional subcubes (or "groupings"):



In each grouping, we have a variable whose value changes within the group, while the values of the other variables do not. For each grouping, let's write a term that contains only the variables whose value remains the same and ignores the others.

From the upper grouping, we get $S1 \cdot \overline{SEL}$: this is the area where $S1 = 1$ and $SEL = 0$, while $S2$ assumes different values.

Likewise, we have $S2 \cdot SEL$ in the lower grouping: this is the area where $S2 = 1$ and $SEL = 1$, while $S1$ varies.

In the end, by adding the terms obtained, we quickly and directly derive the minimized expression:

$$U = S1 \cdot \overline{SEL} + S2 \cdot SEL$$

In Chap. 1, we obtained the same expression by minimizing the canonical form using the properties of Boolean algebra.

Proof

In the above right grouping, the two minterms are:

$$S1 \cdot \overline{S2} \cdot \overline{SEL} \qquad \text{and} \qquad S1 \cdot S2 \cdot \overline{SEL}$$

In OR, these two terms can be simplified:

$$\begin{aligned}
&S1 \cdot \overline{S2} \cdot \overline{SEL} + S1 \cdot S2 \cdot \overline{SEL} = \\
&S1 \cdot \overline{SEL} \cdot (S2 + \overline{S2}) = \\
&S1 \cdot \overline{SEL}
\end{aligned}$$

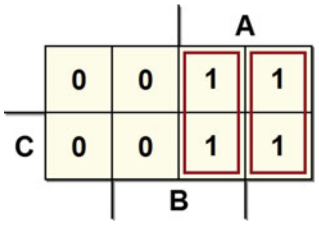
The eliminated variable, $S2$, is the one that assumes different values within the grouping. The other term is treated likewise: here, the variable that changes is $S1$, thus showing:

$$\begin{aligned}
&S1 \cdot S2 \cdot SEL \; + \; \overline{S1} \cdot S2 \cdot SEL = \\
&S2 \cdot SEL \cdot (S1 + \overline{S1}) = \\
&S2 \cdot SEL
\end{aligned}$$

The function is $U = S1 \cdot \overline{SEL} + S2 \cdot SEL$, as defined above.

2.2.1 Implicants and Prime Implicants in Karnaugh Maps

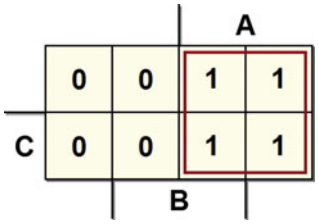
Consider the following map.



From the two unidimensional subcubes, we derive:

$$U = A\,B \; + \; A\,\overline{B}$$

Each one is an implicant because if $A\,\overline{B}$ or $A\,B$ equals 1, the function U also equals 1. Let's now consider the two-dimensional subcube that comprises all the 1s as in the following figure.

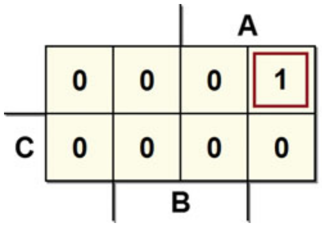


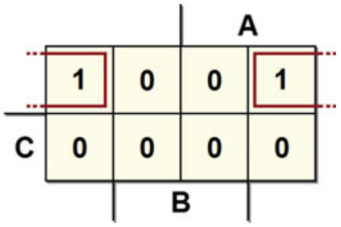
In this subcube, the only variable that does not change is A: thus, the function is reduced to the expression $U = A$. We could reach the same conclusion this way:

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

The simplification is possible because $\overline{A}\,B$ and $A\,B$ are not prime implicants while A is. To obtain the best grouping, we must locate the largest possible subcubes. If *there is no* largest subcube that completely covers what we are considering, it is called a prime implicant.

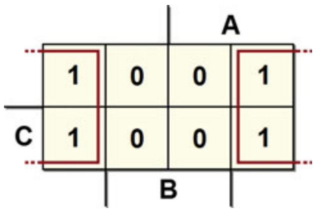
Let's see some examples of how to derive the expression of the function from the maps. From the map on the left, we derive $U = A\ \overline{B}\ \overline{C}$, while we get $U = \overline{B}\ \overline{C}$ from the one on the right.





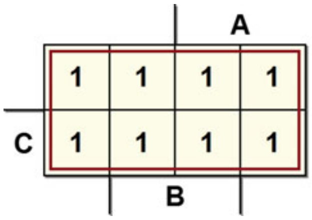
Despite appearances, the two cells are *logically adjacent*, in the map on the right. As we have seen, the map is a representation on a two-dimensional space of a multi-dimensional cube. So, we need to imagine the map as a piece of paper that can be folded over itself vertically and horizontally; the upper and lower edges and the right and left edges are adjacent to each other. As proof, note that only variable A changes between each two pair of cells.

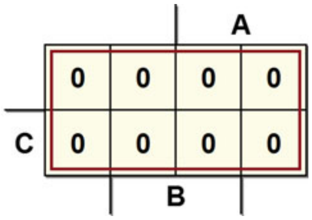
Now, consider the following map:



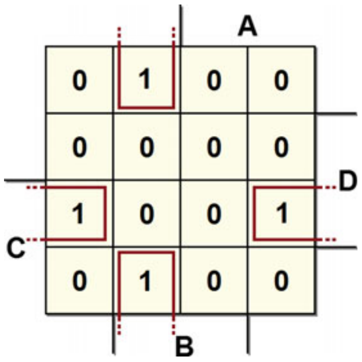
The largest subcube containing the 1s is highlighted: thus $U = \overline{B}$.

Now let's look two borderline cases. The map on the left has a 1 in every cell and represents the constant function $U = 1$; likewise, the map on the right only contains $0s$, represented by the constant $U = 0$:



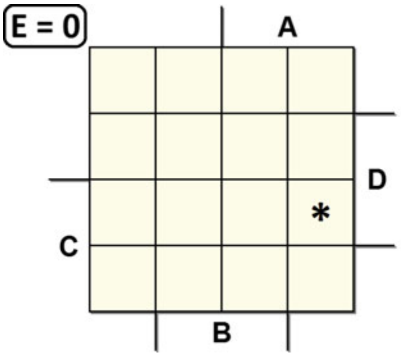


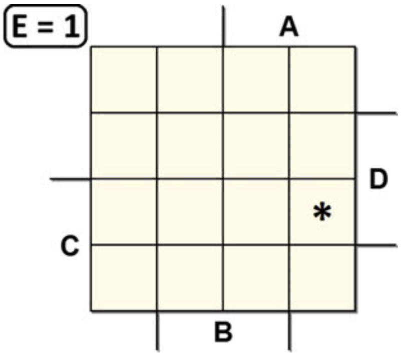
In a three-variable map, we have seen that all the minterms that differ by one variable are represented on the map by adjacent cells. This also holds for four-variable maps, as in the following example:



From this map, we get: $U = \overline{A}\,B\,\overline{D} + \overline{B}\,C\,D$.

Five-variable maps (order-5) can be represented by two order-4 maps, in which cells in the same position in the two maps are adjacent (imagine one map on the top of the other):



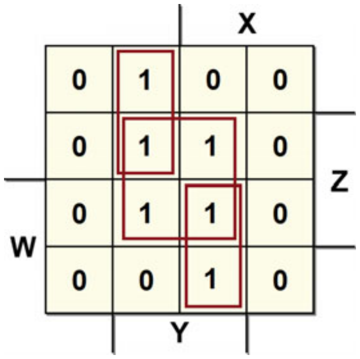


The cells with asterisks are adjacent: it is therefore possible to group them and, therefore, eliminate the variable E.

Six-variable maps are made with four order-4 maps. Higher order maps are too complicated to represent.

2.2.2 Using Maps for Minimization

Let's define the *essential prime implicant* as the only prime implicant that contains a certain 1 on the map. On the following map, for example, there are two order-1 essential prime implants and one order-2 essential prime implicant.



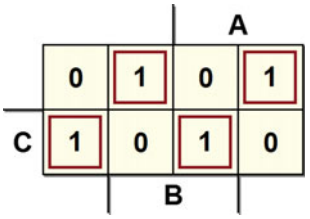
The cells with the 1*s* can be considered more than once. This is useful for finding higher order subcubes. We obtain the synthesis:

$$U = Y\,Z + \overline{X}\,Y\,\overline{W} + X\,Y\,W$$

An implicant completely covered by essential prime implicants is called a *redundant implicant*. Redundant implicants, as such, must be eliminated from the synthesis. Therefore, by using the maps, we obtain the minimum synthesis by taking only the essential prime implicants.

2.2.3 "Checkerboard" Maps

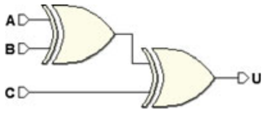
A *checkerboard map*, as seen in the figure below, cannot be minimized in terms of two-level AND-OR or OR-AND networks.



It can be verified that it represents an *XOR tree*:

$$\begin{aligned} U &= \overline{A}\ B\ \overline{C} + A\ \overline{B}\ \overline{C} + \overline{A}\ \overline{B}\ C + A\ B\ C = \\ &= (\overline{A}\ B + A\ \overline{B}) \cdot \overline{C} + (\overline{A}\ \overline{B} + A\ B) \cdot C = \\ &= (A \oplus B) \cdot \overline{C} + \overline{(A \oplus B)} \cdot C = \\ &= A \oplus B \oplus C = (A \oplus B) \oplus C \end{aligned} \quad \Rightarrow$$

$\Rightarrow$

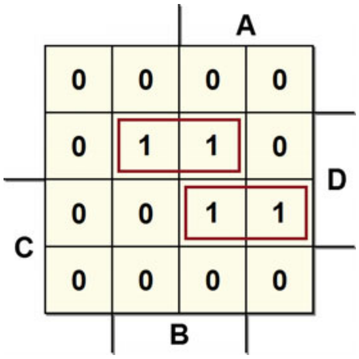


2.2.4 Examples of AND-OR Synthesis

1. Considering the truth table below, let's complete the map at the right.

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
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	0
1	1	1	1	1

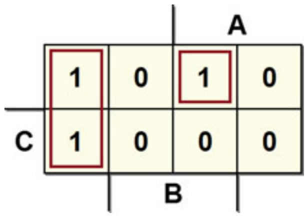
$\Rightarrow$



The grouping in the third row gives us the term $A\,C\,D$; the grouping in the second gives us $B\,\overline{C}\,D$, thus: $F = A\,C\,D + B\,\overline{C}\,D$. Let's verify:

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

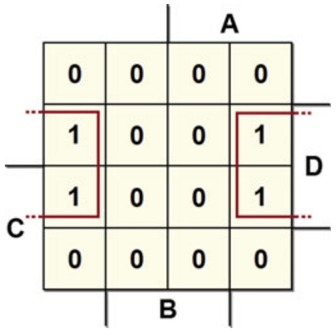
2. Let's simplify $F = \overline{A}\,\overline{B}\,\overline{C} + A\,B\,\overline{C} + \overline{A}\,\overline{B}\,C$. We get this map:



By grouping the two 1s at the left we get: $\overline{A}\,\overline{B}$. The overall function:

$$F = \overline{A}\,\overline{B} + A\,B\,\overline{C}.$$

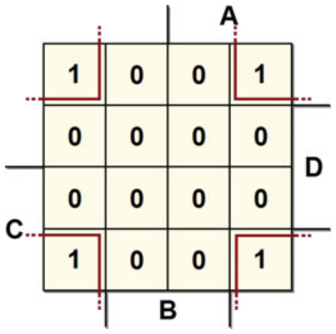
3. Let's synthesize the following map:



The four 1s that are adjacent even though on opposide edges, can be grouped into a single product. The resulting function is:

$$F = \overline{B}\,D.$$

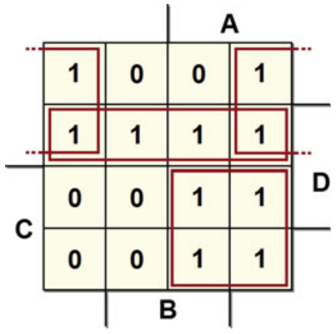
4. Likewise, we synthesize the other map with the 1s at the angles.



As above, we group one single term. The function is:

$$F = \overline{B}\,\overline{D}.$$

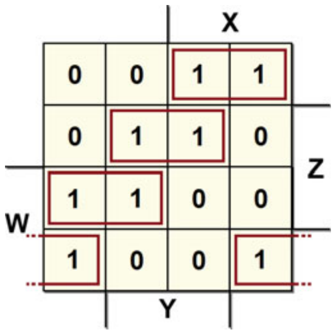
5. Let's derive the function from the map below:



The groupings all have four bits. The upper grouping provides the term $\overline{B}\,\overline{C}$; in the second row, we have $\overline{C}\,D$; at the bottom left, we get $A\,C$. So, the desired function is:

$$F = \overline{B}\,\overline{C} + \overline{C}\,D + A\,C.$$

6. Here is another example (*cyclical map*):



There are four order-1 essential implicants, so we get:

$$U = X\,\overline{Z}\,\overline{W} + Y\,\overline{W}\,Z + \overline{X}\,Z\,W + \overline{Y}\,\overline{Z}\,W$$

2.3 OR-AND Synthesis

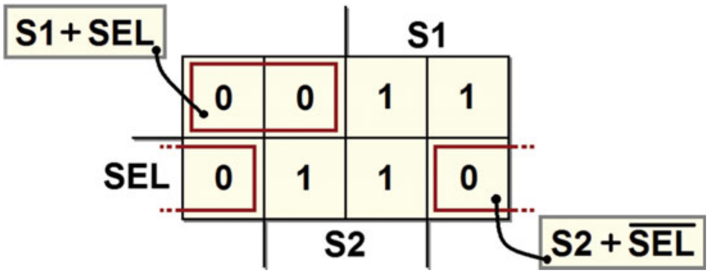
In an AND-OR synthesis, we have seen that we synthesize the 1s of the function through the AND of the input variables. They are *direct* if the variable equals 1 and *negated* if it equals 0. Then, all outputs of the AND gates are OR-ed together.

In the OR-AND synthesis, according to the second form of Shannon's theorem, we synthesize the 0 of the function through the OR of the input variables (*direct* if 0 and *negated* if 1) and all the ORs end up in an AND.

To minimize the OR-AND synthesis using Karnaugh maps, the property of logic adjacency is still valid. Let's take the two-input channel multiplexer as an example, but use the OR-AND technique. Here is the truth table:

SEL	$S1$	$S2$	U
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

After completing the map with the corresponding values from the table, we identify the subcubes that group the 0*s* (the rules are the same as for the subcubes with 1*s*). In the map below, we see the best grouping possible for our example:



From this, we directly derive the minimized OR-AND expression:

$$U = (S1 + SEL)\,(S2 + \overline{SEL}).$$

2.3.1 Synthesis of the Negated Function

For an OR-AND synthesis, we can work differently, by first doing the AND-OR synthesis from the 0s as if they were 1s and as if the function were negated.

If we apply this method to the previous example, we get the following AND-OR synthesis of $\overline{U}$:

$$\overline{U} = \overline{S1} \cdot \overline{SEL} + \overline{S2} \cdot SEL$$

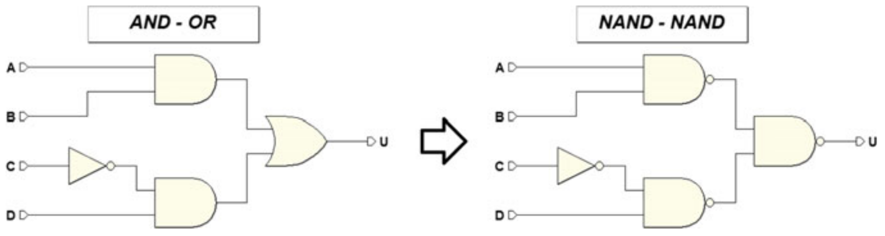
By applying the De Morgan Theorem, we obtain the OR-AND expression:

$$U = (S1 + SEL)(S2 + \overline{SEL})$$

Note that it is not possible to obtain the OR-AND synthesis by applying the principle of duality directly to the AND-OR expression.

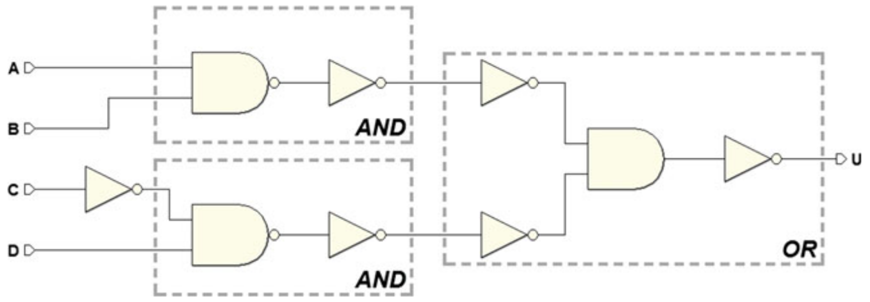
2.4 NAND-NAND Synthesis

Once we have the AND-OR synthesis, we can implement the function with a network composed exclusively of NANDs (NAND-NAND synthesis) by substituting every OR or AND with a NAND, as in the figure below:

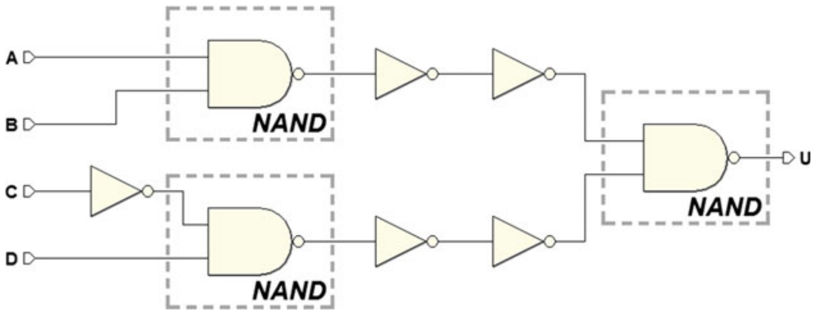


Proof:

Let's substitute the ANDs on the left with NANDs followed by NOT. Then, let's apply De Morgan's Theorem to the OR gate, transforming it into AND, whose inputs and output are negated:



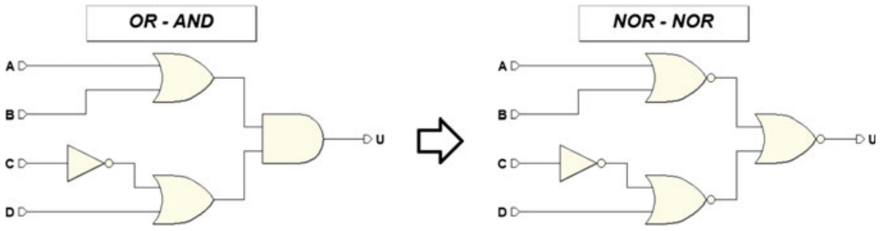
Let's merge the AND gate with the NOT that follows and obtains a NAND, and then separates the two NANDS on the left from the NOTs.



Finally, when we eliminate the double negation, we get the NAND-NAND network. If the AND-OR function has one or more inputs that go directly to the OR gate, they must be negated.

2.5 NOR-NOR Synthesis

Starting from an OR-AND network, we substitute each OR or AND with a NOR gate and get a NOR-NOR synthesis as in the example below. The proof is analogous to the NAND-NAND network.

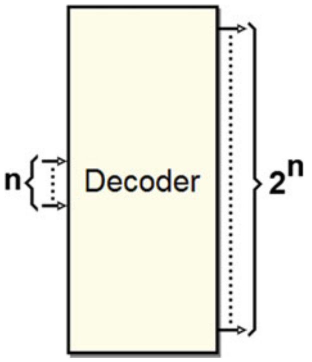


2.6 Standard Combinational Networks

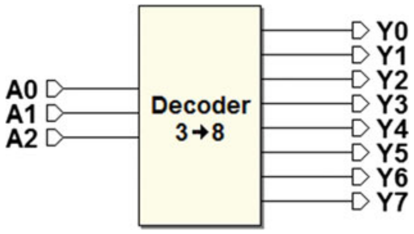
These are general-use combinational networks with a regular circuit structure that are available to the user as functional blocks.

2.6.1 Decoders

Decoders have n inputs and 2^n outputs: every combination of inputs activates one and only one output.



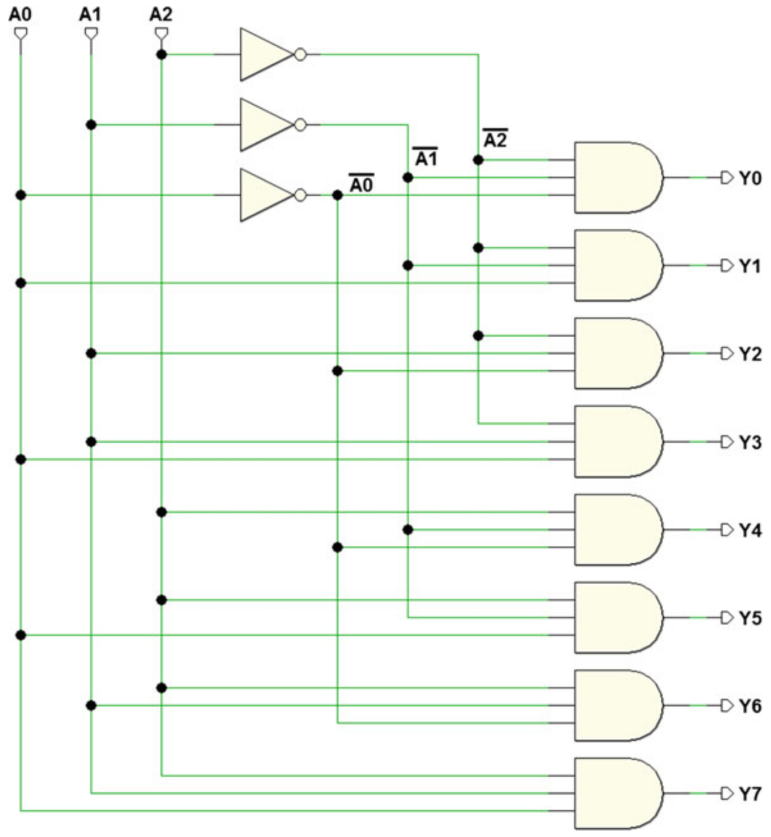
A $3 \rightarrow 8$ decoder has 3 inputs and 8 outputs:



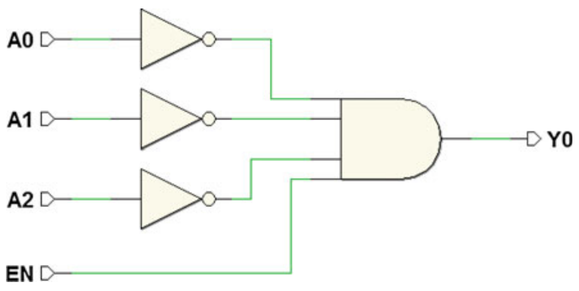
As we can see in the truth table below, output activation occurs in order. The combination $A2A1A0 =$ “000” activates the output $Y0$ and so on until $A2A1A0 =$“111” activates output $Y7$:

$A2$	$A1$	$A0$	$Y0$	$Y1$	$Y2$	$Y3$	$Y4$	$Y5$	$Y6$	$Y7$
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Synthesizing a truth table with eight outputs requires using a combinational network for each output. Here, each output is activated by a single combination of inputs, so maps are not useful since each output calls for only one minterm. The figure below shows the synthesis of the $3 \rightarrow 8$ decoder:

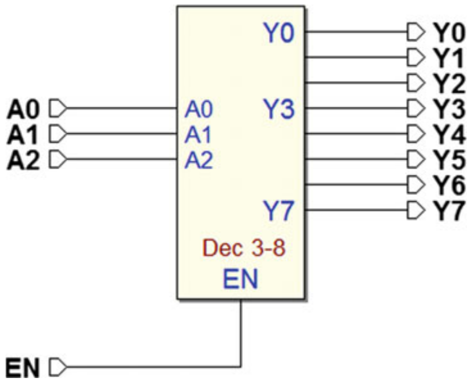


Decoders generally have one *enable input*. When the decoder is enabled, it behaves as described above; otherwise, no output is activated. In the figure below, we see the synthesis of the output $Y0$, with the enable input EN.



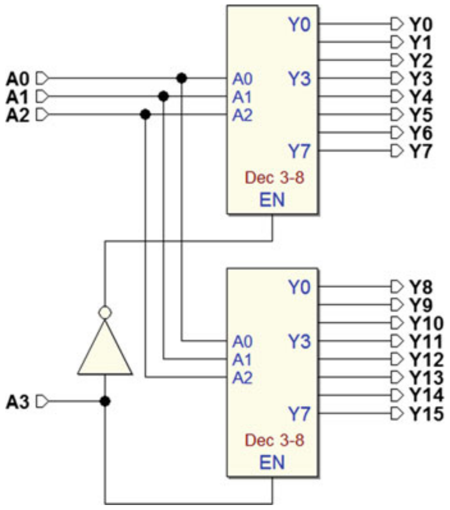
$$Y0 = \overline{A0}\,\overline{A1}\,\overline{A2}\,EN$$

The figure below shows the "Dec 3–8" component from the *Deeds* library, along with its truth table (notice the functionality of the EN input):



EN	$A2$	$A1$	$A0$	$Y0$	$Y1$	$Y2$	$Y3$	$Y4$	$Y5$	$Y6$	$Y7$
0	–	–	–	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

More than one decoder can be connected in order to form a larger one. For example, with two "Dec 3–8s" we get a $4 \rightarrow 16$ decoder:



The truth table below can be divided into two parts: the upper part where the input $A3 = 0$ and the lower part where $A3 = 1$. The "Dec 3–8" that generates the first eight outputs must be enabled ($EN = 1$) when $A3 = 0$ and the other when $A3 = 1$. A simple NOT, as we see in the logic diagram, does the job.

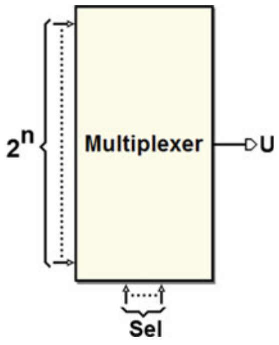
$A3$	$A2$	$A1$	$A0$	$Y0$	$Y1$	$Y2$	$Y3$	$Y4$	$Y5$	$Y6$	$Y7$	$Y8$	$Y9$	$Y10$	$Y11$	$Y12$	$Y13$	$Y14$	$Y15$
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

We can extend this technique at will to obtain larger decoders than the ones available.

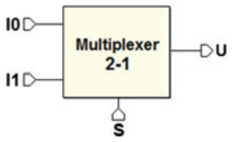
Decoders are very important because they are very commonly used. For example, they represent the fundamental component to implement "addressing" of digital system devices. In other words, they allow us to use an identification number to identify and/or select the elements of a system and operate on them, as with memories and microcomputer systems.

2.6.2 Multiplexer

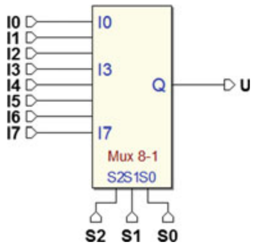
The "multiplexer" (short form "mux," or "channel selector" or simply "selector") is a combinational network with 2^n data inputs, one single output, and n selection inputs. The output assumes the input value identified by the selection inputs.



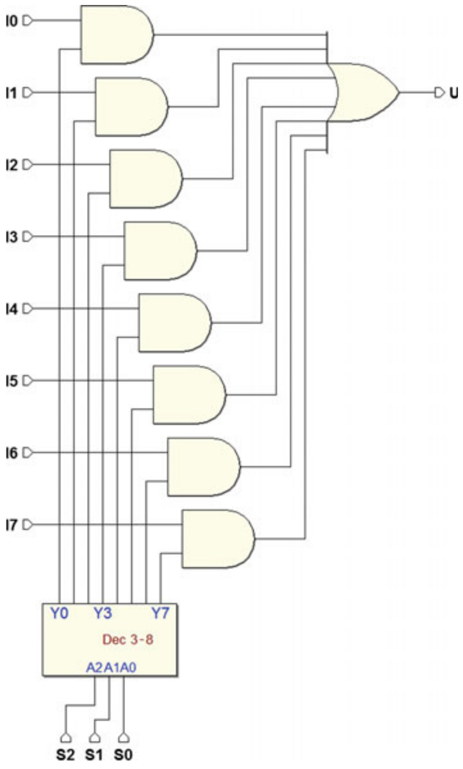
We have already seen the two-input channel selector and how it is created through logical gates. In this case, input S selects which of the two I0 and I1 inputs will be reproduced as the output.



It is easy to extend this concept to multiplexers with a higher number of inputs. In the figure below, three inputs S2, S1, and S0 control which of the eight inputs (I0..I7) will be brought over to the output.

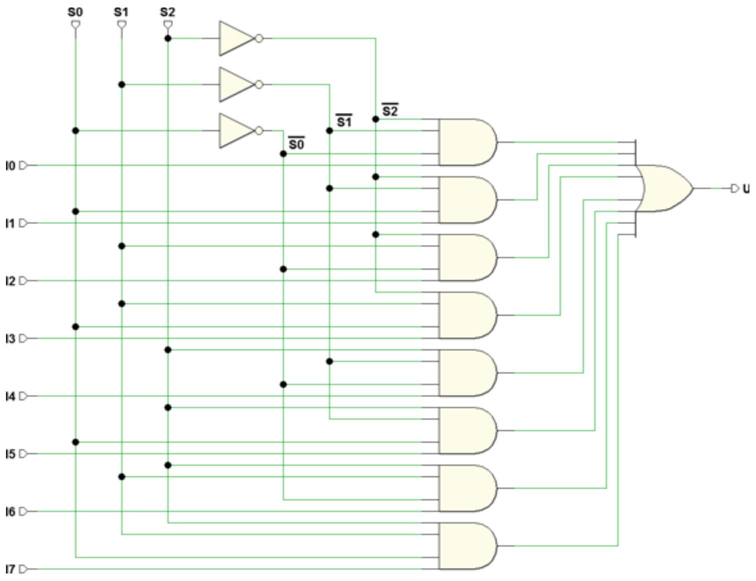


The network has eight data inputs and three selection inputs. It is impractical to do the synthesis through the truth table of a network with 11 inputs (providing 2048 rows in the table). The concept of decoding (see below) can help here.



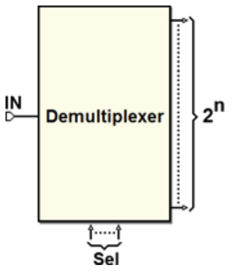
Sending the selection inputs to a decoder, we enable only one of the eight AND gates that transmit each of the eight inputs I0..I7 through the OR gate.

Considering the circuital structure of the decoder already discussed, by just adding an input to each of the AND gates that generate the decoder's outputs, we can get rid of the eight external ANDs. The result is that the network is reduced to just two levels, as in the next figure.

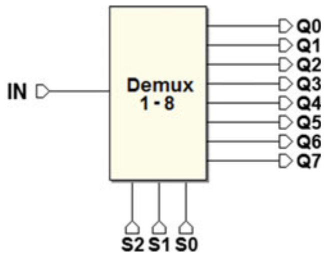


2.6.3 Demultiplexers

The demultiplexer ("deselector" or "DEMUX") is a combinational network with only one data input IN, n selection inputs, and 2^n outputs. It is a combinational network that mirrors the multiplexer. The value of the single input is transferred to one of the many outputs, which is chosen by the logical value of the selection inputs.



The next figure shows an example of an eight-output demultiplexer.

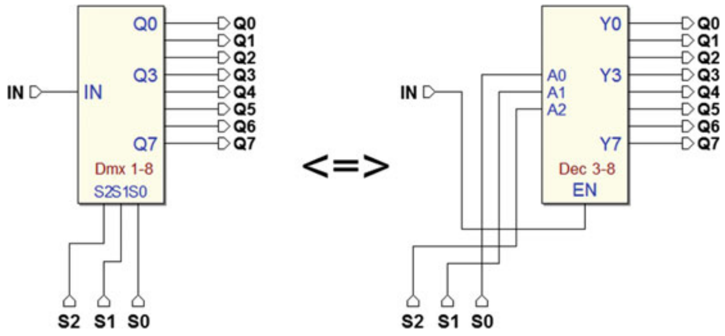


The truth table is represented below. The non-selected outputs always generate 0, while the selected one (from $S2$, $S1$, and $S0$) copies the value of input IN. Therefore, in the first row of the table, (for any $S2$, $S1$, and $S0$ with $IN = 0$), the selected and non-selected outputs all equal 0. However, on the other half of the table (for $IN = 1$), the selected output equals 1, since it copies input IN:

IN	$S2$	$S1$	$S0$	$Q0$	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$	$Q7$
0	–	–	–	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	1	0	0	0	0	0	0
1	0	1	0	0	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	0	1	0	0
1	1	1	0	0	0	0	0	0	0	1	0
1	1	1	1	0	0	0	0	0	0	0	1

This truth table could be examined from another perspective: all the outputs are 0 when input IN equals 0, while the only output at 1 is the selected output if IN equals 1. This behavior matches that of a decoder with an enable input. It follows that a decoder with an enable input can be used as a demultiplexer, using the enable input as a data input.

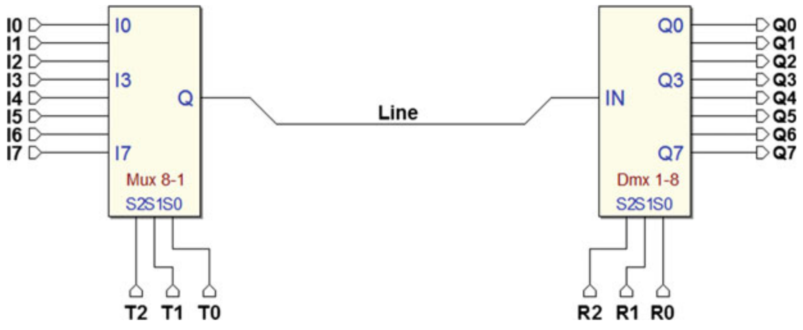
The figure below shows that the "Demux 1–8" and "Dec 3–8" (*Deeds* components) are equivalent:



Example

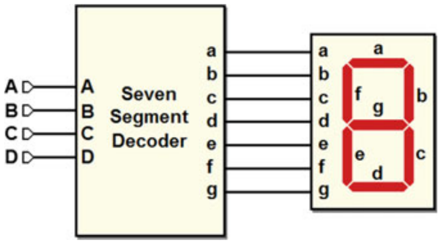
When a multiplexer and a demultiplexer are connected as in the following figure, they allow data transfer from more than one source through a single line to more than one destination. This is especially useful when the transmitting system is very far from the receiving system: it can make radio or cable transmission practical.

The multiplexer's $T2$, $T1$, and $T0$ selection inputs allow us to choose the source to transmit, while $R2$, $R1$, and $R0$ select the data destination.



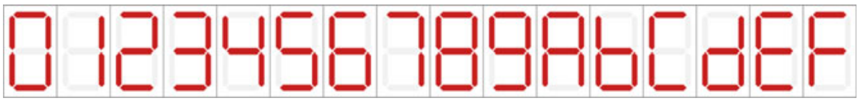
2.6.4 *Seven-Segment Display Decoder*

A seven-segment display is a device for visualizing numbers and letters. It has seven inputs, one for every luminous segment. Let's assume that a 1 turns on the corresponding segment, while a 0 keeps it off. The segments are denoted by the lower-case letters "a, b, c, d, e, f, g" as in the figure below.



The "hexadecimal" (Hex) code will be examined in Chap. 3 but for now suffice it to say that it encodes, using 4 bits, the decimal digits from 0 to 9 followed by the first six letters of the alphabet (for a total of 16 symbols).

According to the four-bit input binary number $DCBA$, the decoder must activate the segments that make up the corresponding hexadecimal symbol, as seen in the figure below.

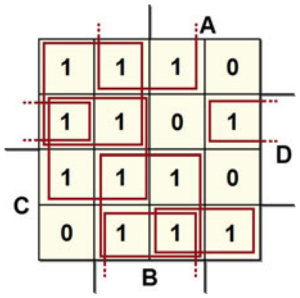


The decoder will have seven outputs: a, b, c, d, e, f, and g, one for each segment. Each of the outputs is controlled by an independent combinational network that switches that particular segment on or off.

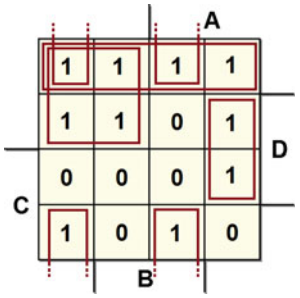
The following table describes the networks that generate a, b, c, d, e, f, and g (the Hex column was inserted to facilitate understanding).

D	C	B	A	a	b	c	d	e	f	g	Hex
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	0	1	1	1	0	1	1	1	A
1	0	1	1	0	0	1	1	1	1	1	B
1	1	0	0	1	0	0	1	1	1	0	C
1	1	0	1	0	1	1	1	1	0	1	D
1	1	1	0	1	0	0	1	1	1	1	E
1	1	1	1	1	0	0	0	1	1	1	F

Below is the synthesis of the networks that drive segments "a" and "b".



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



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

Likewise, we find the other outputs.

2.6.5 *Seven-Segment BCD Decoder (using "don't-cares")*

We want to design a similar device to visualize only the decimal digits from 0 to 9. Since there are only ten symbols to represent, we need only ten input combinations. On the other hand, ten combinations still require four inputs (if there were three inputs, there would only be $2^3 = 8$ combinations available), so there are six unused combinations.

When the input is a combination that does not correspond to any decimal digit, the problem of what to visualize on the screen can be resolved by two different approaches:

- Leave all the segments of the display off.
- Ignore which segments are on or off.

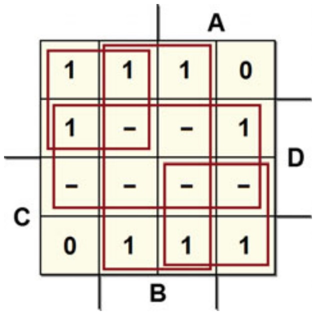
The purpose of the device is actually to decode a decimal number, not to operate on combinations that fall outside the expected ones. In practice, we hypothesize the inputs $DCBA$ always remain within the constraints, in this case, meaning combinations from "0000" to "1001" (from 0 to 9 in decimal figures).

If we are not interested in what we will visualize if we have unexpected combinations (the second approach), we place the symbol "-" meaning "don't-care") on the table and the maps corresponding to the outputs related to nonsignificant inputs. This symbol will then be treated in the synthesis randomly as either 0 or 1, in order to get the most economical implementation of the device. This is, therefore, the decoder's truth table:

D	C	B	A	a	b	c	d	e	f	g	Dec
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
1	0	1	0	–	–	–	–	–	–	–	–
1	0	1	1	–	–	–	–	–	–	–	–
1	1	0	0	–	–	–	–	–	–	–	–
1	1	0	1	–	–	–	–	–	–	–	–
1	1	1	0	–	–	–	–	–	–	–	–
1	1	1	1	–	–	–	–	–	–	–	–

The "don't-cares" allow us to minimize the network more efficiently since we can make them into 0s or 1s in order to choose the largest subcubes.

For example, we can choose among four subcubes for output “a”, as below:



We get: $a = B + D + \overline{A}\,\overline{C} + A\,C$.

2.6.6 *Using Multiplexers to Synthesize Combinational Networks*

An interesting way to create combinational networks without going to the synthesis procedure is to use a multiplexer. This method has the advantage of allowing the network to be re-configured, even while it is working, when it is matched with memorization systems (which we have not seen yet).

We use multiplexers like those we studied earlier that allow us to generate the desired function by selecting the values of the function itself as if we were *reading* them from the truth table. We need to have the truth table of the function. If, however, we start from the Boolean expression, we must obtain the truth table from it.

Example 1:

Synthesize with a multiplexer a function $f(A, B, C)$, which is expressed as:

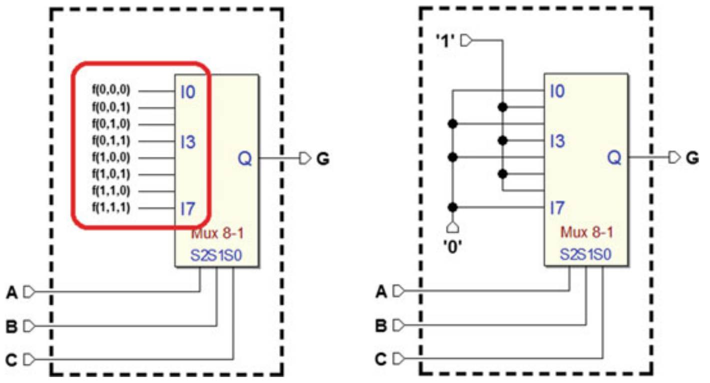
$$G = \overline{B}\,C + A\,B\,\overline{C} + \overline{A}\,B\,C.$$

Let’s get the truth table:

A	B	C	G
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

As shown in the next figure (left), we prepare a network based on a multiplexer $8 \rightarrow 1$ connecting input variables A, B, and C to the selection inputs.

Thus (same figure, right), we connect the eight inputs $I0..I7$ of the multiplexer to constants 0 and 1 in the same order as they appear in the truth table.



The network is ready: for each combination of inputs A, B, and C, the multiplexer will transfer the value we have set as input over to the output.

Example 2:

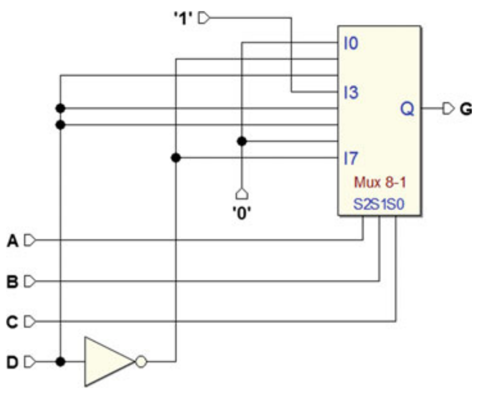
The function is: $H = A\,\overline{B}\,D + \overline{A}\,B\,D + B\,C\,\overline{D} + \overline{A}\,C\,\overline{D}$.

There are four inputs (A, B, C, and D), but we want to use a multiplexer with just three selection lines. The trick is to condition the eight inputs $I0..I7$ to the value of input D.

Let's derive the truth table from the expression: its farthest right column reports the value of H as a function of D taken from the same table.

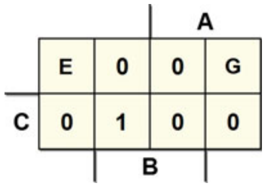
The figure on the right shows the resulting network when the constants and the inputs D and $\overline{D}$ are connected:

A B C D	H	H
0 0 0 0	0	
0 0 0 1	0	0
0 0 1 0	1	
0 0 1 1	0	$\overline{D}$
0 1 0 0	0	
0 1 0 1	1	D
0 1 1 0	1	
0 1 1 1	1	1 ⇒
1 0 0 0	0	
1 0 0 1	1	D
1 0 1 0	0	
1 0 1 1	1	D
1 1 0 0	0	
1 1 0 1	0	0
1 1 1 0	1	
1 1 1 1	0	$\overline{D}$



2.7 Variable-Entered Maps

Now, let's discuss a method of synthesis using Karnaugh maps that contain variables. This method will be used (in Chap. 7) to synthesize Finite State Machines (FSMs) with the ASM method, since applying the rules of synthesis will directly produce this type of map. The figure below shows an example of a map with entered variables.



Aside from variables A, B, and C, shown around the map, let's look at variables E and G shown inside two of the cells. This map shows a five-variable function $f(A, B, C, E, G)$. Writing a certain variable (or an entire expression) inside a cell means conditioning the result of the function to that variable or expression.

In the example here, the upper left-hand cell corresponds to the minterm $\overline{A}\,\overline{B}\,\overline{C}$; however, since the variable is in the cell, the term must be conditioned to E, giving: $\overline{A}\,\overline{B}\,\overline{C}\,E$.

In the upper right-hand cell, we condition the corresponding minterm to the variable G, giving: $A\,\overline{B}\,\overline{C}\,G$. The resulting function is:

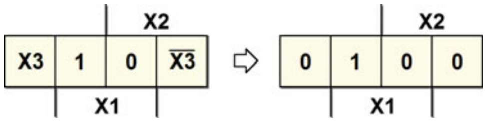
$$f = \overline{A}\,\overline{B}\,\overline{C}\,E + A\,\overline{B}\,\overline{C}\,G + \overline{A}\,B\,C$$

In this example, the function's expression cannot be minimized, but in general the problem of minimization exists. Now, let's look at one of the methods to obtain a minimum synthesis (or at least minimized as far as possible).

2.7.1 *Synthesizing Maps with Entered Variables*

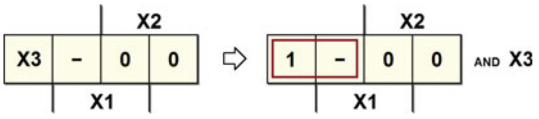
Here, we examine a method for synthesizing maps with entered variables. In the example, we find variable $X3$ in a cell and its negated form in another. We take the following steps:

1. We reduce all the variables on the map to zero and synthesize the 1s.



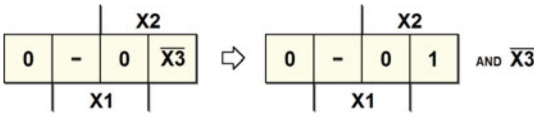
We obtain the first partial result. $f' = X_1\,\overline{X_2}$.

2. We turn the 1s synthesized in step 1 into don't-care terms. We then choose the cells containing the same variable, making them equal to 1, leaving the remaining ones at 0. We consider the variable and its negated value to be two independent variables to be considered one by one in two different steps. We then synthesize this map and place the result in AND with the chosen variable.



We write the second partial result as: $f'' = \overline{X_2}\, X_3$.

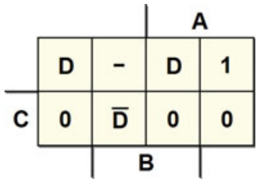
We repeat step 2 for all the other variables on the map. When this cycle of steps is finished, the equivalent expression is given by the OR of all the AND expressions found. In the example, step 2 is repeated once more:



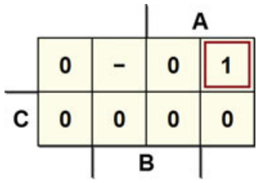
From which we derive the last partial result: $f''' = \overline{X_1}\, X_2\, \overline{X_3}$, so the final expression is:

$$f = f' + f'' + f''' = X_1\, \overline{X_2} + \overline{X_2}\, X_3 + \overline{X_1}\, X_2\, \overline{X_3}$$

Example 1: Given the following map, derive the corresponding function.

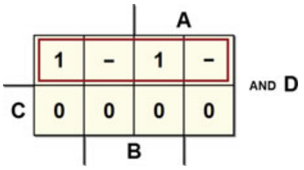


1. Set the cells containing D and $\overline{D}$ to 0 and synthesize the remaining 1s. Note that the don't-care terms remain the same and could be used to minimize the map, although it would not help in this example.

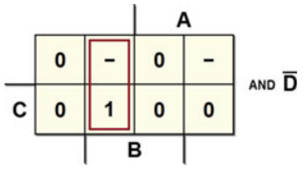


We obtain the expression: $f' = A\, \overline{B}\, \overline{C}$.

2. We substitute the only 1 on the map with a don't-care term, set the cells containing D at 1, and set the cells with $\overline{D}$ at 0.

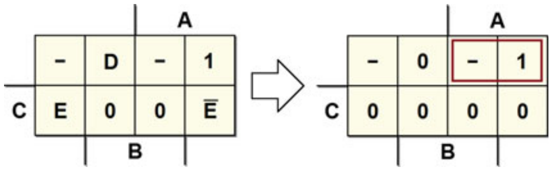


   This way, we obtain the synthesis $\overline{C}$, which is placed in AND with D, so the second term of the desired expression is: $f'' = \overline{C}\,D$.
3. We repeat step 2 for the cells with $\overline{D}$, as if this were an independent variable from D, by setting the cells with $\overline{D}$ to 1 and those with D to 0.

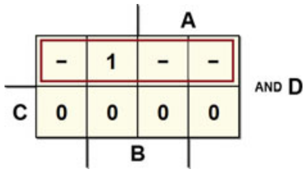


   From this map, we derive the term $\overline{A}\,B$. When this is placed in AND with $\overline{D}$, it provides the last term of the desired expression: $f''' = \overline{A}\,B\,\overline{D}$.
4. The final expression is: $f = f' + f'' + f''' = A\,\overline{B}\,\overline{C} + \overline{C}\,D + \overline{A}\,B\,\overline{D}$.

Example 2: Deriving the function from a map with two entered variables:

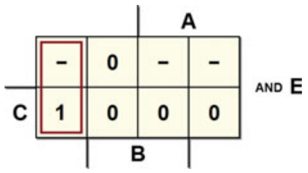


After the first step, we get $A\,\overline{C}$. Note that an equally valid synthesis of step 1 could be $\overline{B}\,\overline{C}$. Let's start to do step 2, by inserting 1 in the cells with D, and 0 in those with E e $\overline{E}$:

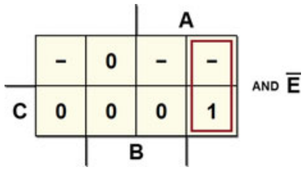


The synthesis of the map gives $\overline{C}$, so the desired term is $\overline{C}\,D$. Notice how only the considered variable enters into the term, while the others appear neither in direct nor in negated form.

Let's do step 2 again, this time for E: we reduce cells D and $\overline{E}$ to zero:



Synthesizing the map provides the term $\overline{A}\,\overline{B}$, hence $\overline{A}\,\overline{B}\,E$. Finally, we do step 1 once again for cells containing $\overline{E}$, reducing those with D and E to zero:



We obtain the term $A\,\overline{B}$, which gives us $A\,\overline{B}\,\overline{E}$. The final expression is:

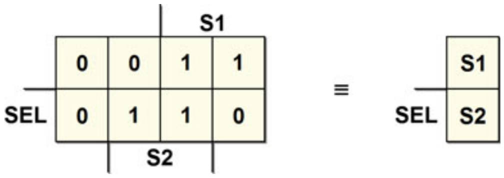
$$f = A\,\overline{C} + \overline{C}\,D + \overline{A}\,\overline{B}\,E + A\,\overline{B}\,\overline{E}$$

2.7.2 Entered Variables and Theorems of Expansion

A map with entered variables could be considered a hybrid, a middle ground between representing logical networks through their Boolean expression and doing it through their Karnaugh map.

The "classic" K-map that contains only 0s and 1s is none other than a map with entered variables that happens to lack entered variables. If we enter all the variables of a map, what we obtain in the only remaining cell is the Boolean expression of the network. It would be possible to demonstrate, by using Shannon's expansion theorems, how maps with entered variables can result from the "compression" of ordinary Karnaugh maps and vice versa.

It is interesting to look again at the map of the two-input multiplexer and compare it with the map of the same network with entered variables, obtained by compressing the K-map by entering the variables $S1$ and $S2$.

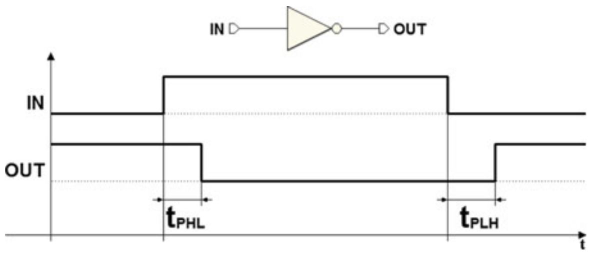


It would have been much simpler to create the map with entered variables by means of the multiplexer's specifications: if $SEL = 0$, the output is $S1$, and if $SEL = 1$ the output is $S2$. In fact, the variable-entered map can be seen as a synthetic tool for describing combinational networks.

2.8 Time Behavior of Combinational Networks

2.8.1 Definitions and Timing Models

A logical gate introduces a delay between the change in value of the input and that of the output. In the figure below, the input and output signals of a NOT are represented in idealized form, since the transitions between logical values are instantaneous.



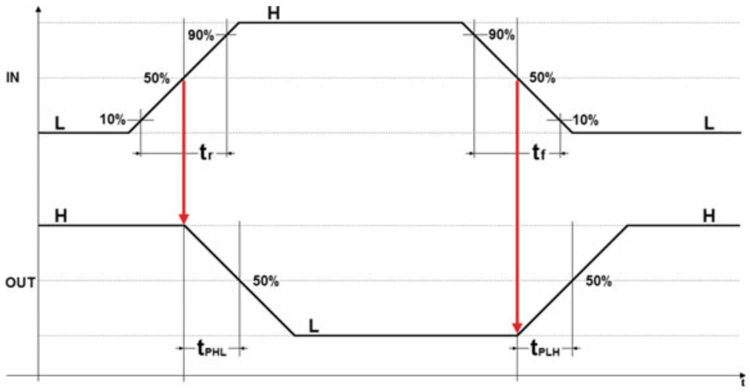
t_{PHL} : propagation time (high to low)
t_{PLH} : propagation time (low to high)

The delays in this form are called ***transport delays***. This idealized model makes them quick and easy to read.

For simplicity's sake, we could denote propagation times generically as:

$$t_p = \max(t_{PLH}, t_{PLH})$$

When necessary, it is useful to adopt a more realistic signal model where level transitions (edges) do not occur instantly (in 0 time) but in finite times with linear progression. We use this model to describe ***inertial delays***:



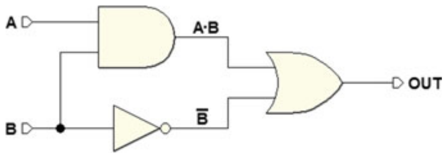
With this model, we assume that the logical level transitions occur when the signal reaches (rises to or lowers to) 50% of its excursion.

Propagation times t_{PHL} and t_{PLH} are the times between the signal-level transition at input and that at output.

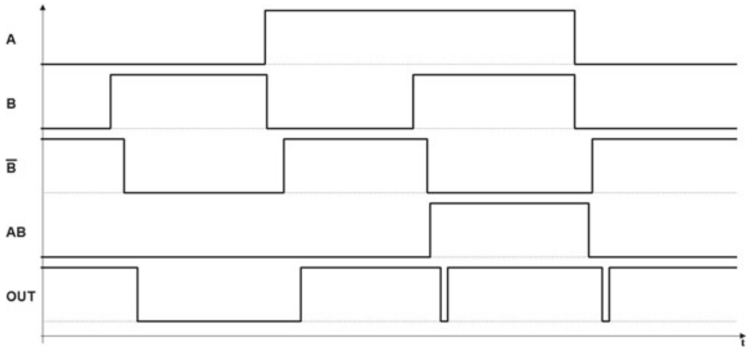
The rise time t_r is the time interval between 10 and 90% of the signal excursion from low to high. In a similar way, the fall time t_f is defined as the time interval between 10 and 90% of the signal excursion from high to low. Make sure not to confuse rise and fall times with propagation times.

Representation through *transport delays* requires only the translation of the output signal over the input signal over time and does not explain the fact that in reality, signals whose duration falls under a certain threshold are not propagated.

The inertial model, however, does allow us to represent this behavior. Let's look at the following logical circuit.

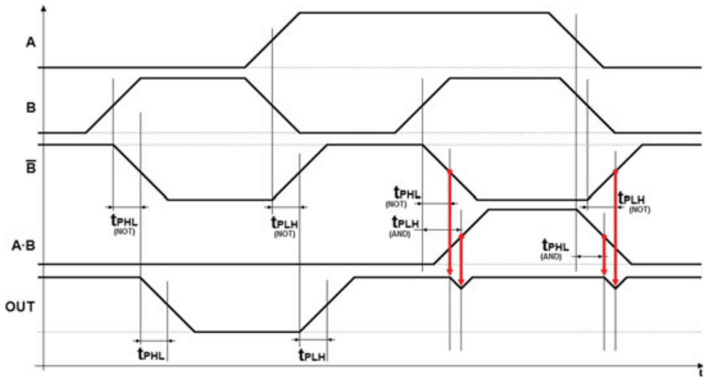


If we analyze the circuit with a digital circuit simulator set to calculate only the *transport delays*, we get:



Thus, based on this model, there should be two short pulses on the OUT output due to the delay differences along the signal paths and the difference between times t_{PHL} and t_{PLH}.

If we repeat the simulation using the *inertial delay* model, we get:

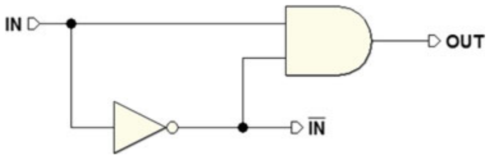


In this new, more realistic simulation, we see that the two pulses are present but will not be propagated since their amplitude fails to reach the threshold due to the finite slope of the transitions.

2.8.2 Hazards

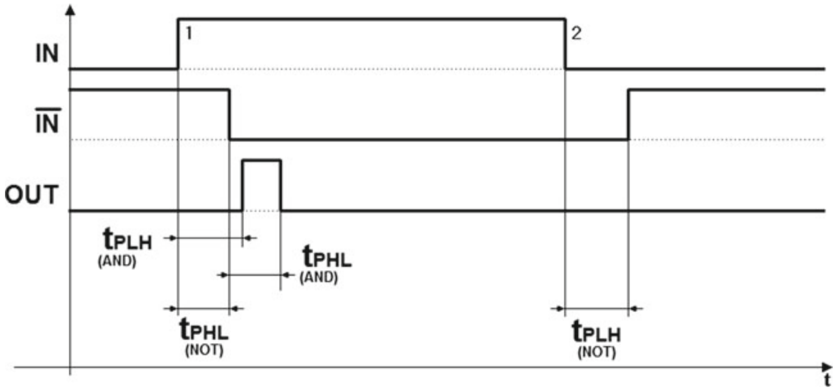
As we have seen, combinational networks can give rise to impulsive behavior due to differences in delays. These behaviors are called ***hazards***. Depending on the physical component's technology, these behaviors might be mitigated or removed by the inertial behavior of the circuits. In any case, these phenomena are potentially damaging due to their effect on the circuits that receive them and they should be avoided when they can cause errors.

The hazards that arise due to asymmetrical delay paths, due in turn to the presence of inverters or other gates, are called ***static hazards***. They can generally be eliminated (or *masked*) through algebraic methods like the one explained later.



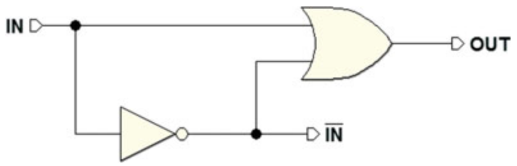
In the figure above, a simple network made up of an AND gate and a NOT gate allows us to observe the effect of different paths.

This network should generate a constant ($OUT = IN \cdot \overline{IN} = 0$). To perform a simplified time analysis, let us only consider *transport delays* so that the hazard will not be masked by the inertial delay.

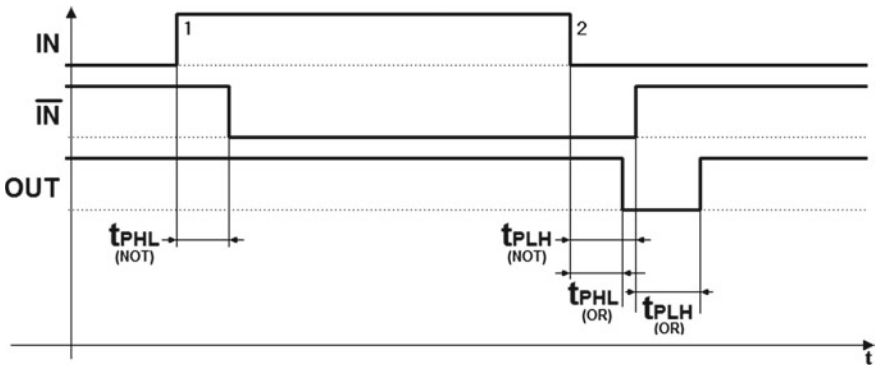


The figure above shows that the output is not always 0 but produces a pulse at 1 (the hazard). Due to the delay of the NOT, the AND inputs are together at 1 for a certain, small period of time.

To be thorough, let's examine another simple circuit based on an OR gate. The output should be constant here as well (given that $OUT = IN + \overline{IN} = 1$):



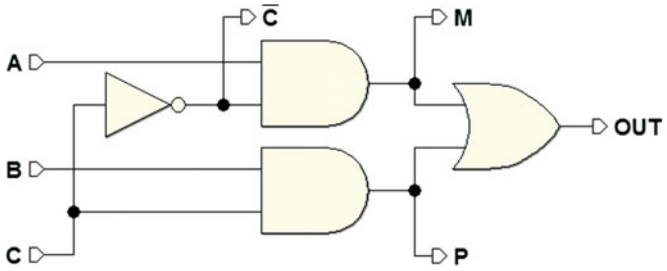
The time analysis using the same criteria as above also shows a hazard (in the opposite transition) in this case as well. Timing analysis, executed as in the case before, shows the presence of a hazard, this time on the falling edge of the input transition.



In AND-OR networks, hazards generally appear as brief transitions to 0 of signals that should be stable at 1. In OR-AND networks, we see signals that transition to 1 when they should be at 0.

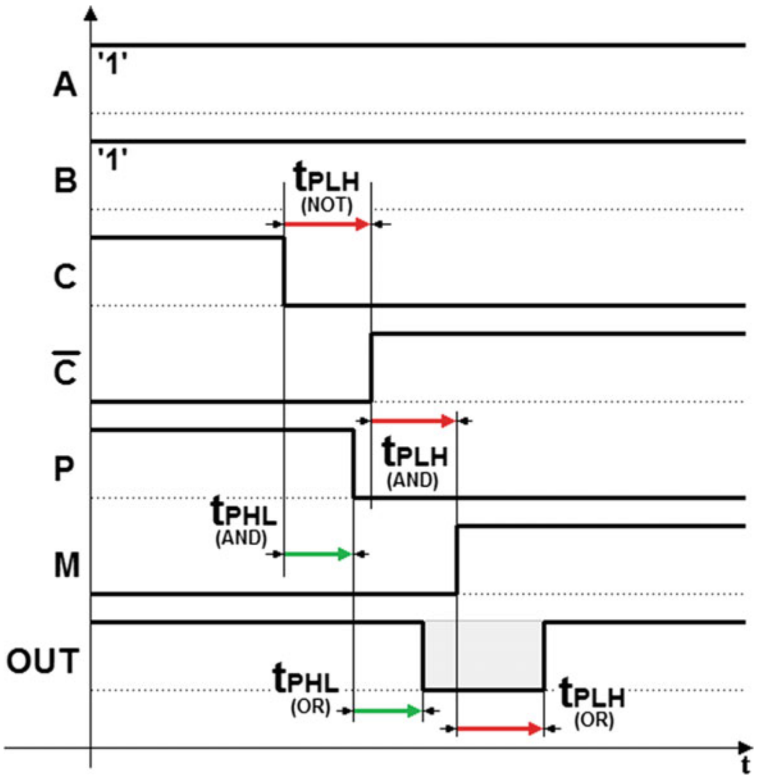
2.8.3 *Elimination of Static Hazards*

In the following two-level AND-OR circuit (the 2 $\rightarrow$ 1 multiplexer), there is a hazard in the transition 1 $\rightarrow$ 0 of C when inputs A and B equal 1:

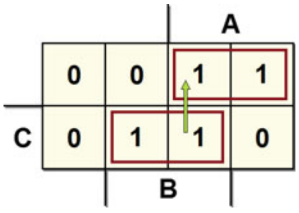


$$OUT = A\,\overline{C} + B\,C$$

Time simulation shows that the hazard occurs since, for a short time, the two logical products $A\,\overline{C}$ e $B\,C$ will be reduced to zero due to the delay induced by NOT.



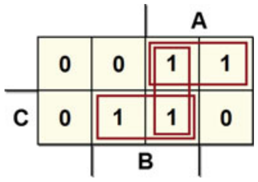
If we look at the Karnaugh map, we see that the hazard develops in the transition indicated by the arrow (i.e., when C goes from 1 to 0, while $A = 1$ and $B = 1$).



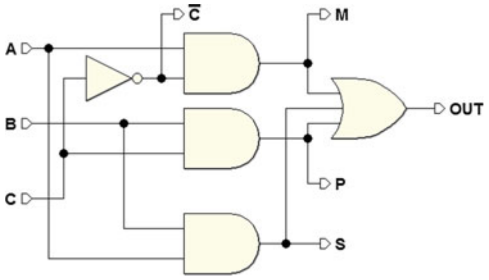
There is a theorem (not proven here) that affirms: a two-level combinational network synthesized as a sum of products is free of hazards if there are no hazards in its $1 \rightarrow 1$ transitions (from one set of inputs that produce 1 to another set of inputs that produce 1).

In a two-level AND-OR combinational network, there can be a hazard when there is a pair of 1s near each other on the map that does not belong to the same implicant, as in the map above.

Let's check for a hazard in this network since there are two adjacent 1s in different implicants. To eliminate the hazard, we must add the implicant that contains these two 1s (term $A\ B$):

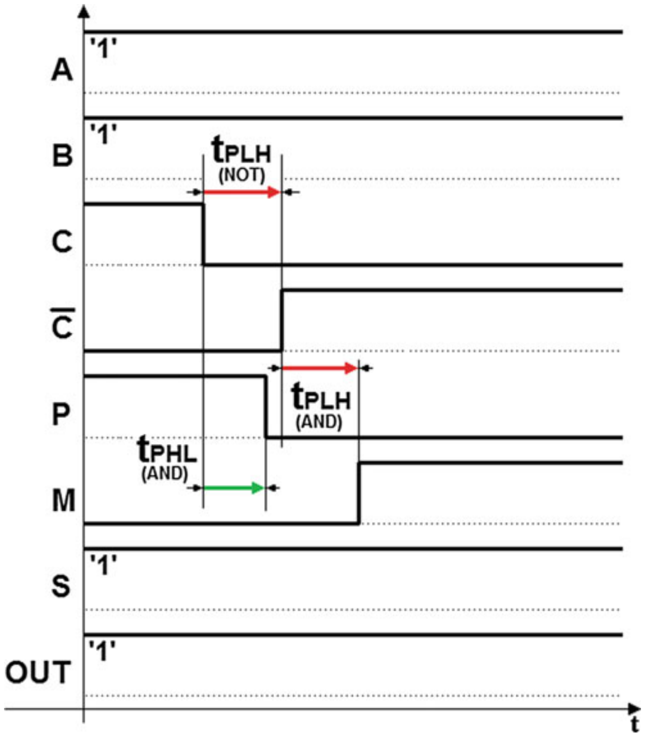


The result is: $OUT = A\,\overline{C} + B\,C + A\,B$. See the schematic of the new network below:



This is no longer a minimal synthesis, but the added product term masks (or *covers*) the hazard, which disappears from the time simulation.

In fact, in the short time in which the two logical products $A\,\overline{C}$ e $B\,C$ go to zero, the term $A\,B$ keeps a 1 in the input of OR, as we can see in the figure below:



In the OR-AND networks, the situation is symmetrical: the hazard occurs in $0 \rightarrow 0$ transitions and is eliminated by adding the implicants containing two nearby zeros that are not contained in the same subcube.

2.8.4 Notes on Eliminating Hazards

Generally, to eliminate static hazards, we systematically add nonessential implicants in all the situations that can produce hazards. However, let's note that eliminating hazards is necessary only when they are damaging, typically in *asynchronous sequential circuits*.

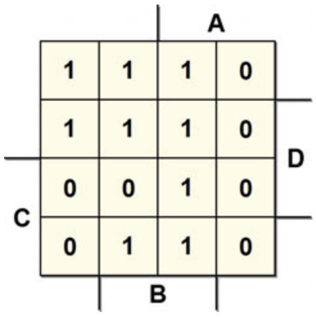
Hazards can generally be tolerated in *synchronous sequential circuits*, in which the signals are read in well-defined times where possible hazards cannot occur. We will see more on this later in the book.

2.9 Exercises

2.9.1 Maps

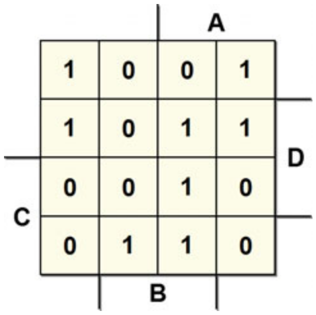
1. Draw the truth tables and maps for the following functions:
   (a) $G = A\,B\,C + B\,\overline{C}$
   (b) $H = (A + \overline{B})(B + \overline{C})$
2. Construct the maps of the following functions:
   (a) $F = \overline{A}\,\overline{B}\,D + \overline{A}\,B\,C + A\,B\,D + A\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C}\,D$
   (b) $M = (\overline{A} + C)(\overline{A} + \overline{C} + \overline{D})(\overline{A} + B)(A + B + \overline{C} + D)$
3. Minimize the logical functions in the maps below as sums of products.

   (a) F1:

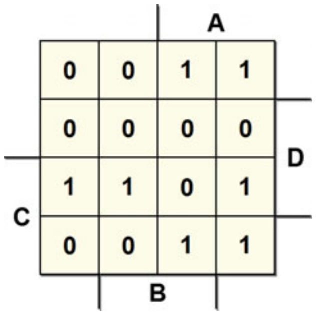


1	1	1	0
1	1	1	0
0	0	1	0
0	1	1	0

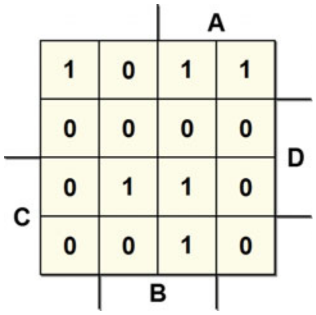
(b) F2:



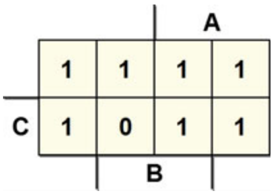
(c) F3:



(d) F4:

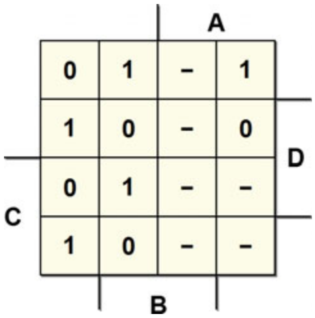


4. Minimize the function $F = \overline{B}\,C\,\overline{D} + \overline{A}\,B\,\overline{C}\,\overline{D} + A\,\overline{B}\,\overline{D}$ as a sum of products, keeping in mind that inputs $ABCD =$ "$11--$" and $ABCD =$ "$--11$" are never present (combinations $A = B = 1$ or $C = D = 1$ can never arise).
5. Synthesize the logical function in the map below as a sum of products.



6. Synthesize the logical function in the map of the previous exercise as a product of sums.

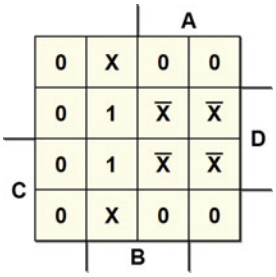
7. Synthesize the following map, which contains don't-cares.



8. Using only NAND gates, draw the simplest circuit that generates the function:

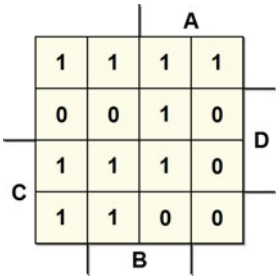
$$F = (A\,B + \overline{A}\,\overline{B})\,C\,D + (A\,\overline{C} + \overline{A}\,C)\,B\,D + \overline{A}\,C\,D + A\,\overline{B}\,C\,D.$$

9. Synthesize the following map with entered variables (group only the don't-cares that give us the largest groupings; ignore those that would add groups).

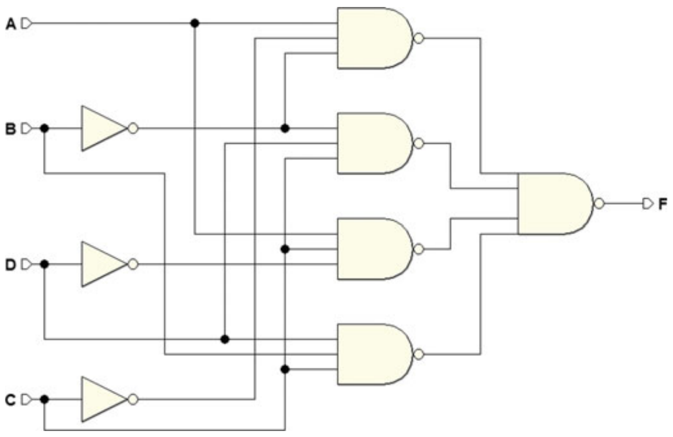

A
D
C
B
0 X 0 0
0 1 $\overline{X}$ $\overline{X}$
0 1 $\overline{X}$ $\overline{X}$
0 X 0 0

2.9.2 *Hazards*

1. Derive the minimal synthesis from the following map:
   (a) Ignoring the hazards;
   (b) Eliminating the hazards.



2. Identify the hazards in the circuit and eliminate them.

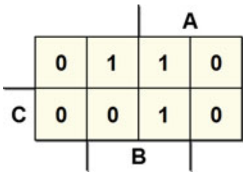


2.10 Solutions

2.10.1 Maps

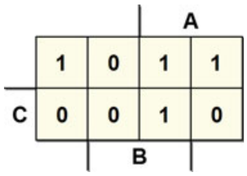
1. (a) $G = A\,B\,C + B\,\overline{C}$

A	B	C	G
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

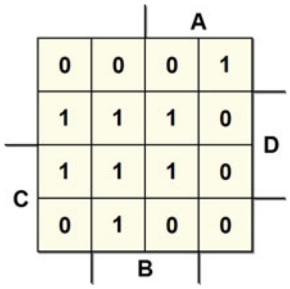


(b) $H = (A + \overline{B})(B + \overline{C})$

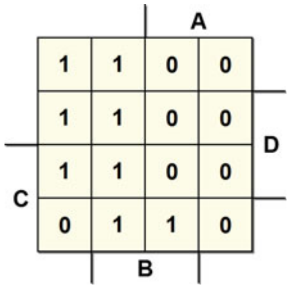
A	B	C	H
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1



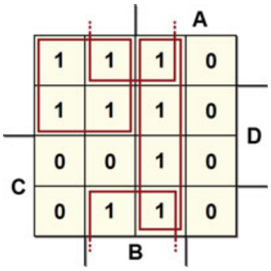
2. (a) $F = \overline{A}\,\overline{B}\,D + \overline{A}\,B\,C + A\,B\,D + A\,\overline{B}\,\overline{C}\,\overline{D} + \overline{A}\,B\,\overline{C}\,D$



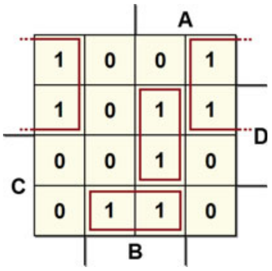
(b) $M = (\overline{A} + C)(\overline{A} + \overline{C} + \overline{D})(\overline{A} + B)(A + B + \overline{C} + D)$



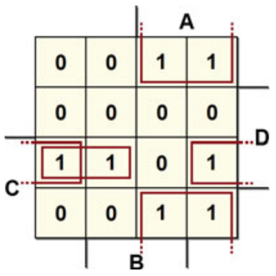
3. (a) $F1 = \overline{A}\,\overline{C} + A\,B + B\,\overline{D}$:



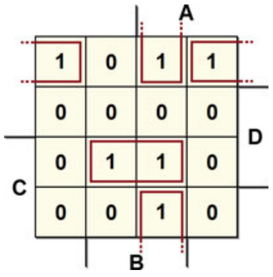
(b) $F2 = \overline{B}\,\overline{C} + A\,B\,D + B\,C\,\overline{D}$:



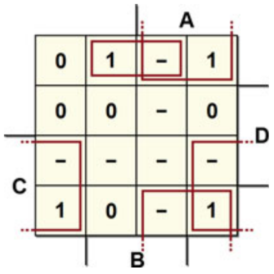
(c) $F3 = A\,\overline{D} + \overline{B}\,C\,D + \overline{A}\,C\,D$:



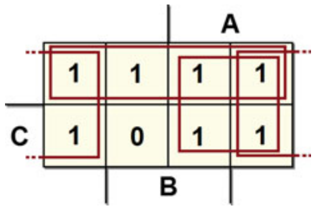
(d) $F4 = \overline{B}\,\overline{C}\,\overline{D} + A\,B\,\overline{D} + B\,C\,D$:



4. $F = A\,\overline{D} + \overline{B}\,C + B\,\overline{C}\,\overline{D}$:

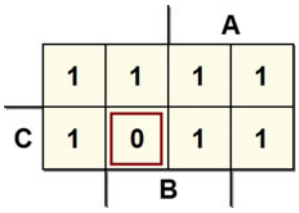


5. These are the three order-2 subcubes for AND-OR synthesis:



We obtain the expression: $F = \overline{A} + \overline{B} + \overline{C}$.

6. For OR-AND synthesis, we just need to consider the only 0.

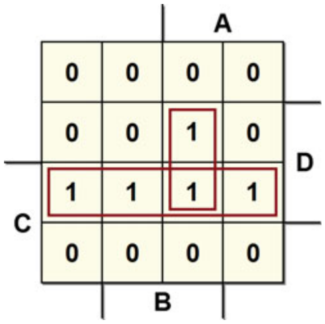


We derive the same expression as in the previous exercise, but using only one grouping.

7. $F = \overline{A}\,\overline{B}\,\overline{C}\,D + \overline{B}\,C\,\overline{D} + B\,\overline{C}\,\overline{D} + B\,C\,D + A\,\overline{D}$
8. We expand the given expression in terms of minterms.

$$\begin{aligned}F &= (AB + \overline{A}\,\overline{B})\,CD + (A\,\overline{C} + \overline{A}\,C)\,BD + \overline{A}\,CD + A\,\overline{B}\,CD =\\ &= ABCD + \overline{A}\,\overline{B}\,CD + AB\,\overline{C}\,D +\\ &\qquad + \overline{A}\,BCD + \overline{A}\,(B + \overline{B})\,CD + A\,\overline{B}\,CD =\\ &= ABCD + \overline{A}\,\overline{B}\,CD + A\,B\,\overline{C}\,D + \overline{A}\,BCD + A\,\overline{B}\,CD\end{aligned}$$

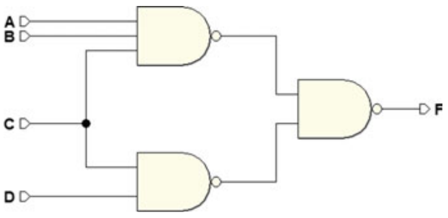
The resulting map is:



which produces the minimized expression: $F = C\,D + A\,B\,C$.
We transform this into NAND-NAND with De Morgan's Theorem.

$$F = \overline{\overline{C\,D} \cdot \overline{A\,B\,C}}$$

This is the network:

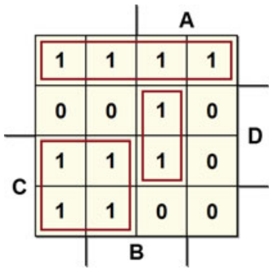


9. With the method, we studied for Variable-Entered Maps, and we have:

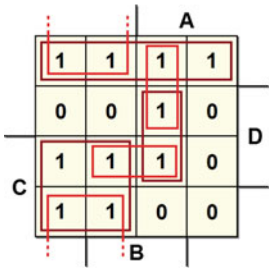
$$F = \overline{A}\,B\,D + \overline{A}\,B\,X + A\,D\,\overline{X}$$

2.10.2 *Hazards*

1. Synthesis of a function considering the hazards issue:



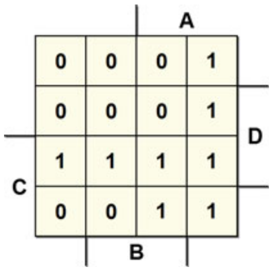
a) With hazards:
$\overline{C}\,\overline{D} + \overline{A}\,C + A\,B\,D$:



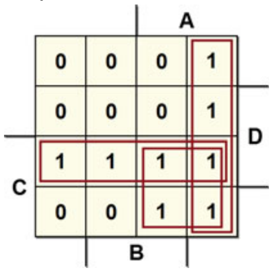
b) With hazards "*covered*":
$\overline{C}\,\overline{D} + \overline{A}\,C + A\,B\,D +$
$\overline{A}\,\overline{D} + B\,C\,D + A\,B\,\overline{C}$:

2. Keeping NAND-NAND network equivalence in mind, we derive the expression of the AND-OR network, and from this, we get the map.

$$F = A\,\overline{B}\,\overline{C} + \overline{B}\;C\;D + A\;C\;\overline{D} + B\,C\,D$$



We group the map for minimal synthesis.



On the map, all the adjacent 1s are masked, so the minimal synthesis is already free of hazards. See its expression and the logical network below.

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