Chapter 3

Spring 2010 Edition

Switching Algebra and Analysis and Design of Digital Circuits

A SUMMARY of what you learned in Chapter 2:

1- Inverters.

2- Two- and more-input AND, OR, NAND, NOR, XOR and XNOR gates.

3- Don't care entries in input columns of truth tables (compressed truth tables).

4- Timing diagrams.

5- Higher fan-in using low-fan-in gates (large gates using smaller gates).

6- One-bit partial comparators.

7- NOR-to-NOT conversion.

8- NAND-to-NOT conversion.

9- Programmable inverters.

10- Signal gating.

11- Unused inputs (small gates using larger gates).

12- Transistor-level architectures and switch models of CMOS gates.

Introduction

In this chapter switching algebra is presented. We utilize switching algebra to translate truth tables into logic expressions and also to manipulate logic expressions, which are digital circuits' mathematical models. Additionally, we will learn how to translate a logic expression, hence a truth table, into a network of properly intercommoned logic gates to reach a digital circuit that operates in accordance with the corresponding truth table. The move from truth tables towards logic circuits is called *circuit design*. In this chapter, we will also learn to move in the opposite direction, i.e., from digital circuits to logic expressions and then to truth tables. This move is called *circuit analysis*.

Switching Algebra[1]

In this chapter we frequently use two terminologies: *expression* and *function*. You are already familiar with them in the context of traditional algebra. For example, $(x + 1)^2 - (x - 1)^2$ and $4x$ are two expressions each with one (input) variable, x. An expression is comprised of some variables and/or constants that are operated on by some operators. Informally speaking, when an expression is assigned to an output variable, a function is defined. Therefore, a function is specified by an output variable, one or more input variables and a rule described by an expression (or possibly other means), so that for each set of input value(s) one output value is generated under the corresponding rule. For example, $y = (x + 1)^2 - (x - 1)^2$ represents a function, in which y and x are the output and input variables, respectively; so that if x is, say 2, y would be 8. Two equivalent expressions cannot generate different functions. For example, since $(x + 1)^2 - (x - 1)^2 = 4x$, we say that $y = (x + 1)^2 - (x - 1)^2$ and $z = 4x$, represent the same function. In the context of switching algebra, we still use the same concepts of expression and function.

[1] Or interchangeably called *Boolean* algebra in some textbooks, although they are historically different.

Switching algebra is a mathematical tool that we often need in order to properly and easily manipulate logic expressions, which are logic circuits' mathematical models, as elaborated on in this chapter. In general, the *axioms* (or *postulates*) of a mathematical system are a minimal set of basic definitions to fully describe the system. The axioms of switching algebra are as follows. These definitions are not new to us; they were introduced in Chapter 2 but in a different language.

Axiom No	**Left Axiom**	**Right Axiom**	**Comments**
A1	$a \neq 0 \Rightarrow a = 1$	$a \neq 1 \Rightarrow a = 0$	"a" may take on either 1 or 0
A2	$a = 1 \Rightarrow a' = 0$	$a = 0 \Rightarrow a' = 1$	Definition of NOT
A3	$1 + 1 = 1$	$0 . 0 = 0$	The last 3 lines
A4	$0 + 0 = 0$	$1 . 1 = 1$	define OR and AND
A5	$1 + 0 = 0 + 1 = 1$	$0 . 1 = 1 . 0 = 0$	as explained below.

Table 1. Axioms of switching algebra

In this table there is one left (L) and one right (R) axiom in each row with the same axiom number. Comments are in the last column. We may refer to these axioms pair-wise (e.g. axiom A2) or individually (e.g. A2-L, which means the left axiom in row A2, or A5-R, which means the right axiom in row A5). Here is what these axioms imply:

A1 says that a switching variable may take on either a 0 or a 1.

A2 is the definition of the NOT function.

The left columns of A3, A4 and A5 define the OR function.

The right columns of A3, A4 and A5 define the AND function.

In the definition of switching algebra (the above axioms) there are only 3 types of functions or operators[2], namely AND, OR and NOT, where the first two are binary and the third one is unary. (A binary operator needs two operands, while a unary operator needs only one.) In Chapter 2 we studied some physical devices (gates) realizing not only these basic functions but also other functions (such as 2-input XOR or 3-input AND), which are not included in the definition of switching algebra. However, as we shall see in this chapter, every logic function (including those introduced in Chapter 2 but not included in the definition of switching algebra) can be represented by an appropriate combination of the basic functions included in the definition of switching algebra, i.e., 2-input AND, 2-input OR and NOT.

As demonstrated later in this section, the axioms mentioned above may be utilized to prove *theorems* such as the ones shown in Table 2.

What is a theorem? A theorem has the general form of "an expression = another expression". And remember that an expression is comprised of some logic variables and constants operated on by some operators. Also, in switching algebra a variable may take on a value that is either a 1 or a 0; and a constant is either a 1 or a 0. The equality in a theorem is true for *every* possible combination of values that (input) variables may take. For example, consider the following theorem from the above table:

$a + b = b + a$

[2] In this book we use *operator* and *function* interchangeably, but mathematically speaking they are not the same.

Theorem No	Left Theorem	Right Theorem	Theorem Name
T1	a + 0 = a	a . 1 = a	Identities
T2	a + 1 = 1	a . 0 = 0	Null elements
T3	a + a = a	a . a = a	Idempotency
T4	(a')' = a		Involution
T5	a + a' = 1	a . a' = 0	Complements
T6	a + b = b + a	a . b = b . a	Commutativity
T7	(a + b) + c = a + (b + c)	(a . b) . c = a . (b . c)	Associativity
T8	a . (b + c) = a . b + a . c	a + (b . c) = (a + b) . (a + c)	Distributivity
T9	a + a . b = a	a . (a + b) = a	Covering
T10	a . b + a . b' = a	(a + b) . (a + b') = a	Combining
T11	(a + b)' = a' . b'	(a . b)' = a' + b'	DeMorgan's theorems

Table 2. Some frequently used theorems

There are two variables in this theorem, namely a and b; hence, the possible combinations of values that these variables may take are as follow: a b = 00, 01, 10, 11. Since a + b = b + a is a theorem, the equality is true for each of these four combinations. Conversely, to prove a theorem one method is to show that the theorem is true for every possible combination of values that (input) variables may take. In other words, in this technique (we call it *truth-table technique*) we in fact obtain the truth tables of both expressions of the theorem in hand and show that the tables are identical. But this is not a new technique to us; we used it in Chapter 2 and will continue to use in the current chapter.

Example 1. Use the truth-table technique to prove T9-R from Table 2: a . (a + b) = a.

To prove this theorem we show that the following two functions represented by the two expressions in T9-R have the same truth tables:

z1 = a

z2 = a . (a + b)

z1 is always equal to input a (see Figure 1). For z2 with two variables we may apply all possible input combinations {ab = 00, 01, 10, 11} and use the switching axioms to obtain the output values as shown below:

ab = 00: z2 = 0 . (0 + 0) = 0 . (0) = 0

ab = 01: z2 = 0 . (0 + 1) = 0 . (1) = 0

ab = 10: z2 = 1 . (1 + 0) = 1 . (1) = 1

ab = 11: z2 = 1 . (1 + 1) = 1 . (1) = 1

These output values have been entered in the truth table shown in Figure 1. Columns z1 and z2 are identical; therefore, z1 = z2, hence theorem T9-R is proved. You will not need to carry out all these trivial operations to obtain a truth table when you build up enough experience.

In this example there are two equivalent expressions, namely a and a . (a + b), representing the same function. In general, we may create many expressions equivalent to a given expression. In other words, a function may be represented by many (equivalent) expressions; but remember that every function can be described by a unique truth table.

Row	a b	z1 z2
0	0 0	0 0
1	0 1	0 0
2	1 0	1 1
3	1 1	1 1

Figure 1. Two-output truth table for Example 1

A major problem with this technique is that it does not work efficiently for theorems with 5 or more variables because a truth table doubles in size when the number of variables increases by one. Algebraic techniques are a reasonable solution for this problem, as shown in this chapter.

Similar to the axioms (in Table 1), the theorems shown in Table 2 are also organized in a side-by-side fashion (except for T4), so that there is one left (L) and one right (R) theorem in each row with the same theorem number. We may refer to these theorems pair-wise, (e.g. theorem T6, or the *commutativity* theorem as it is called in the last column) or individually (e.g. T6-R, which refers to the right theorem (column 3) in row T6, or T5-L, which refers to the left theorem (column 2) in row T5.)

Go through these theorems. Comparing them with the traditional algebra in your math courses you will notice that some theorems such as T8-R are much less familiar. But do not worry! You will thoroughly master them as you work on the examples and homework problems provided in this chapter.

The frequently used theorems provided in Table 2 may be utilized to prove more complex theorems. Each theorem (with one right-side expression and one left-side expression, separated by an equal sign) may be used or applied in two different ways or directions in different applications. More specifically, to prove a new theorem we may use an already-proved theorem and replace its left side with its right side or vice versa, depending on what we are going to prove.

In order to unambiguously read and interpret logic expressions, by convention, AND has precedence over OR, and NOT has precedence over AND. Therefore, 0 . 0 + 1 equals 1. However, if OR was assumed to take precedence over AND, 0 . 0 + 1 would equal 0! Also, according to these notational conventions 1 + 0' equals 1. But if OR took precedence over NOT, 1 + 0' would equal 0! Therefore, parentheses would be used to change precedence should the need arise. For example, 0 . (0 + 1) = 0 and (1 + 0)' = 0.

As another example, if the parentheses were removed from T8-L, we would reach the *false* theorem: a . b + c = a . b + a . c . Try a b c = 0 0 1 as a counterexample, which results in the erroneous equation 1 = 0. Substitute for the three variables:

Left side = 0 . 0 + 1 = 0 + 1 = 1

Right side = 0 . 0 + 0 . 1 = 0 + 0 = 0.

One more example: Let us assume that a = 0, b = 1 and c = 1, and then considering the above precedence conventions evaluate the logic expression (a + b') . (a' + b' . c)' + a' . b . c' . If we substitute for the three variables, the expression will change to (0 + 1') . (0' + 1' . 1) + 0' . 1 . 1', which may be simplified as follows:

(0 + 0) . (1 + 0 . 1) + 1 . 1 . 0 = (0) . (1 + 0) + 1 . 0 = (0) . (1) + 0 = 0 + 0 = 0. ◊

You have probably realized the similarity between any two theorems or two axioms located in the same row in the above tables. The *principle of duality* (presented shortly) shows how to easily derive one theorem or axiom from the other one in the same row.

Definition. The dual of a logic expression is another logic expression obtained through the following replacements:

Replace all zeros with ones, all ones with zeros, all AND operators with OR operators, and all OR operators with AND operators, or $0 \leftrightarrow 1$ and $. \leftrightarrow +$, for short[3].

For example, the dual of $(a . b) + c$ is $(a + b) . c$, and the dual of $a + 0$ is $a . 1$.

Principle of Duality Any theorem in switching algebra remains valid if the theorem's right-side expression is replaced with its own dual expression, and the theorem's left-side expression is replaced with its own dual expression; in other words, any theorem in switching algebra remains valid if on both sides of the theorem all zeros are replaced with ones, all ones are replaced with zeros, all AND operators are replaced with OR operators, and all OR operators are replaced with AND operators. The resulting theorem is called the dual of the original theorem and vice versa. Every two theorems located in the same row of Table 2 are duals of each other. For example, the dual of $a + a' = 1$ (T5-L) is $a . a' = 0$ (T5-R), and the dual of $a . a' = 0$ is $a + a' = 1$. Theorem 4 is the dual of itself.

Operator precedence must be taken into consideration when the principle of duality is applied. For example, let's apply the principle of duality to the distributivity theorem on the left (T8-L), which has been shown below again for ease of reference:

$a . (b + c) = a . b + a . c$ (T8-L)

There are no 0s or 1s in this theorem. So, what we need is to change all AND operators to ORs, and all OR operators to ANDs, which results in the following equation:

$a + b . c = a + b . a + c$ (T8-R-false)

But this is a false theorem (a counterexample is $a\ b\ c = 0\ 0\ 1$) because operator precedence was not properly taken into consideration when the principle of duality was applied. More specifically, the two OR operators on the right side of T8-R-false have replaced the two AND operators on the right side of T8-L. However, the AND operators have precedence over the OR operator on the right side of T8-L. Therefore, a similar precedence has to be preserved between their counterparts in the resulting theorem by using parentheses as shown below:

$a + b . c = (a + b) . (a + c)$

This is the righ-side theorem (T8-R) in row 8. ◊

Theorems T1 – T5 can easily be proved by the truth-table technique explained on page 3, i.e., we need to show that each of these theorems is true for both $a = 1$ and $a = 0$. (According to the first axiom a switching variable (like a) may take on either a 1 or a 0.) This technique is also called *perfect induction* in some textbooks.

Example 2. Prove that $a + 1 = 1$ (T2-L)

We need to make two substitutions:

$a = 1$: $1 + 1 = 1$ correct, according to A3-L
$a = 0$: $0 + 1 = 1$ correct, according to A5-L ◊

Switching theorems become more powerful when we notice that each variable in a theorem may be replaced with an arbitrary logic expression resulting in a (different) valid theorem.

[3] We assume that only the basic switching operators, AND, OR and NOT, may participate in the logic expression in hand. Other operators such as XOR are considered non-basic. In this context non-basic operators are not allowed to participate unless they have been decomposed into some basic ones as shown in this and the previous chapter.

Example 3. Prove that a’ . (a’ + b) = a’

In T9-R change a to a’ to reach the new theorem.

Example 4. Prove that a + c + (a + c) . b = a + c

In T9-L make the substitution a → (*a* + *c*) to reach the new theorem.

Example 5. Prove that a . b . c + a . b . c’ = a . b

If a . b is renamed d, we need to prove d . c + d . c’ = d. But this is true according to T10-L. ◊

Sometimes we may prefer to partially use perfect induction.

Example 6. Prove that a + (b . c) = (a + b) . (a + c) (T8-R)

We prove that “left side = right side”, for both a = 1 and a = 0.

a = 0: left side = 0 + (b . c) = b . c right side = (0 + b) . (0 + c) = b . c

a = 1: left side = 1 + (b . c) = 1 right side = (1 + b) . (1 + c) = 1 . 1 = 1

In this proof the following theorems and axiom have been used:

T1-L: a + 0 = a

T2-L: a + 1 = 1

A4-R: 1 . 1 = 1

Example 7. Prove that a + a . b = a (T9-L)

Again we may make two substitutions for b (b = 0 and b = 1) to prove this theorem. We may also employ already-proved theorems of switching algebra to reach this goal as shown below:

Apply T1-R	a + a . b	= a . 1 + a . b
Apply T8-L		= a . (1 + b)
Apply T2-L		= a . 1
Apply T1-R		= a ◊

To prove a theorem, we may also wish to prove its dual instead because according to the principle of duality if a theorem is true, its dual is true as well.

Example 8. Prove that (a + b + c) . (a + b + c’) = a + b

This is the dual of the theorem that was proved in Example 5. ◊

Example 9. Prove that (a . b) + (a . b . c’. d) + (a . b . d . e’) = a . b

We use the covering theorem proved in Example 7.

Apply T9-L:	(a . b) + (a . b . c’. d) = a . b
	Left-side expression = (a . b) + (a . b . d . e’)
Apply T9-L:	Left-side expression = (a . b) ◊

Remember that logic expressions are mathematical models of logic circuits. Before we continue with switching algebra it would be helpful to see how logic expressions are translated into logic circuits and vice versa; after all, they describe the same thing but in different domains. And this is what we need to understand as soon as possible.

A digital circuit is a *network of interconnected gates* with one final output (going to the outside world), some primary inputs (coming from the outside world), and some possible internal nodes; so that the output of one gate (except the gate which generates the final output) becomes the input of another gate in the network[4]. (By an internal node we mean the interconnection point between a gate output and a gate input, e.g. node x in Figure 2*c*.) The digital circuit shown in Figure 2*c*, as an example, has three primary inputs, a, b and c, an output, z, and one internal node, x. More examples of digital circuits are shown in different figures throughout this chapter.

A digital circuit is mathematically described by one logic expression and, similarly, a logic expression can be translated into one digital circuit. In other words, a digital circuit *realizes* (makes *real*) a logic expression. For example, the circuit shown in Figure 2*c* is described by expression a . (b + c), and we write z = a . (b + c), which represents a *function*: every 3-bit pattern for triple (a, b, c) generates only one value for z. But the same function may be specified by different expressions. For example, T8-L shows two expressions for the same function.

$a \cdot (b + c) = a \cdot b + a \cdot c$ (T8-L)

Example 10 and Example 11 elaborate on these two expressions.

Example 10. Realize a . (b + c), the left side of T8-L.

Let's call the output of the resulting circuit z; therefore,

$z = a \cdot (b + c)$

We also call (b + c) a new signal, x

$x = (b + c)$

This equation describes a two-input OR gate with inputs {b, c} and output x, as shown in Figure 2*a*.

Substitute x for (b + c) in the original expression:

$z = a \cdot x$

This equation describes a two-input AND gate with inputs {a, x} and output z, as shown in Figure 2*b*. But input x is the output of an OR gate with inputs {b, c}. Therefore, z is the output of a two-input AND gate in which one input is driven by a and the other one is tied to the output of a two-input OR gate with inputs b and c, as shown in Figure 2*c*.

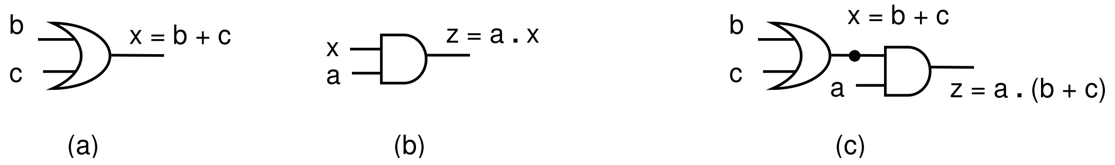


Figure 2. Three steps to realize z = a . (b + c): (a) step 1, (b) step 2, (c) step 3

After some practice, and once you feel comfortable, you may translate from the algebraic domain to the circuit domain directly in one step.

Example 11. Determine the exact logic expression that describes the logic circuit shown in Figure 3 with inputs {a, b, c} and output y.

We use the same procedure examined in Chapter 2:

Label the intermediate nodes and then write the algebraic equation of the output. Now in this equation, substitute for all intermediate signals (from their own equations) to obtain a new equation for the output. Make similar substitutions repeatedly to eventually eliminate all intermediate signals from the output equation.

[4] More specifically, by a digital circuit in this context we mean a *loop-free* network of some interconnected gates in which each gate output drives no more than one gate input, unless otherwise specified.

In Figure 3 y is the output of an OR gate with inputs z1 and z2, so

y = z1 + z2

On the other hand, z1 and z2 are two intermediate signals, where

z1 = a . b
z2 = a . c

Substitute for z1 and z2 in the original equation, y = z1 + z2:

y = a . b + a . c

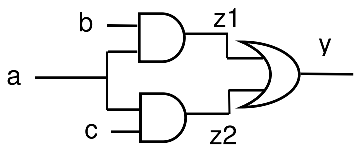


Figure 3. Logic circuit for Example 11

The expression obtained for y and realized in Figure 3 is the right side of T8-L. This circuit needs one 2-input OR gate and two 2-input AND gates. The logic circuit corresponding to the left side of T8-L is illustrated in Figure 2*c*. This circuit needs one 2-input OR gate and one 2-input AND gate, hence is simpler (less expensive) than the circuit shown in Figure 3, demonstrating how switching algebra may be used to simplify digital circuits. ◊

DeMorgan's theorems (T11) deserve special attention. They are shown here again for ease of reference:

(a + b)' = a' . b' (a . b)' = a' + b' (T11)

T11 shows how primed parentheses in a logic expression may be removed by properly applying DeMorgan's theorems. More specifically, using T11-L, primed parentheses around an OR term as in (a + b)' are removed resulting in an AND term (a' . b') with no primed parentheses anymore. Similarly, according to T11-R, expression (a . b)' with primed parentheses around an AND term is logically equivalent to an OR term (a' + b') with no primed parentheses, where a and b each are an arbitrary logic expression. Primed-parentheses removal is a key factor to manipulate and simplify logic expressions as illustrated in Example 12 through Example 14. The following procedure shows how to repeatedly apply DeMorgan's theorems to an arbitrary logic expression to remove all primed parentheses:

```
Start with the outermost primed parentheses and proceed towards the inside.

Repeat until no parentheses remain primed:
    Apply T11-L or T-11-R to the next primed parentheses if an OR or an AND operation, respectively,
    is surrounded by the parentheses. (prime absorption)
```

Example 12. Apply DeMorgan's theorem (and other theorems as needed) to simplify the following expression:

f = ((x' + y)' . x)'

Apply T11-R to f:	f = ((x' + y)')' + x'
Apply T4 to ((x' + y)')':	f = (x' + y) + x' = x' + x' + y
Apply T3-L to x' + x':	f = x' + y

Example 13. Apply DeMorgan's theorem (and other theorems as needed) to simplify the following expression:

f = ((x' + y) . (x + (y . z)'))'

Apply T11-R to f:	f = ((x' + y)' + (x + (y . z)')')

To enhance readability let's process the two primed parentheses, $(x' + y)'$ and $(x + (y \cdot z)')'$, separately.

Apply T11-L to $(x' + y)'$, the 1st parentheses:	$(x' + y)' = (x')' \cdot y'$
Apply T4 to $(x')'$:	$(x' + y)' = x \cdot y'$
Apply T11-L to $(x + (y \cdot z)')'$, the 2nd parentheses:	$(x + (y \cdot z)')' = x' \cdot ((y \cdot z)')'$
Apply T4 to $((y \cdot z)')'$:	$(x + (y \cdot z)')' = x' \cdot y \cdot z$
Put both parentheses together:	$f = x \cdot y' + x' \cdot y \cdot z$

Example 14. Apply DeMorgan's theorem (and other theorems as needed) to simplify the following expression:

$f = ((x' + y)' + (x \cdot y' + z)')'$

Apply T11-L to f:	$f = ((x' + y)')' \cdot ((x \cdot y' + z)')'$
Apply T4 to both outermost parentheses:	$f = (x' + y) \cdot (x \cdot y' + z)$
Apply T8-L to f:	$f = x \cdot y' \cdot (x' + y) + z \cdot (x' + y)$
Apply T8-L to both parentheses:	$f = x \cdot y' \cdot x' + x \cdot y' \cdot y + z \cdot x' + z \cdot y$
Apply T5-R to $x \cdot y' \cdot x'$ and $x \cdot y'\, y$:	$f = 0 \cdot y' + 0 \cdot x + x' \cdot z + y \cdot z$
Apply T2-R to $0 \cdot y'$ and $0 \cdot x$:	$f = 0 + 0 + x' \cdot z + y \cdot z$
Apply A4-L to $0 + 0$:	$f = 0 + x' \cdot z + y \cdot z$
Apply T1-L to f :	$f = x' \cdot z + y \cdot z$ ◊

You have just seen how to apply DeMorgan's theorems to logic expressions to eliminate primed parentheses. Now let's see how these transformations are reflected on the corresponding digital circuits. Notice that a prime on parentheses corresponds to a bubble at the output of a gate. Therefore, prime removal from parentheses in the algebraic domain corresponds to bubble removal (in the circuit domain) from the output of the gate corresponding to those primed parentheses. More specifically, using these theorems (and appropriate inversions) we may replace a NAND gate with an OR gate, and also replace a NOR gate with an AND gate as shown below.

The logically equivalent expressions on the left and right of T11-L have been realized in Figure 4*a* and Figure 4*b*, respectively. The circuit in Figure 4*a* is a two-input NOR gate with inputs a and b. In Figure 4*b* the same inputs are first inverted and then applied to a two-input AND gate.

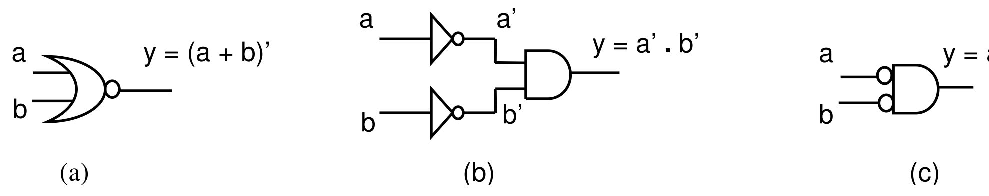


Figure 4. Circuits realizing T11-L: (a) (a + b)', (b) a' . b', (c) equivalent symbol for a NOR gate

Take a closer look at the above transformation. A two-input NOR gate is in fact logically equivalent to a two-input AND gate with inverted inputs. An important outcome of this transformation is that the output bubble has been removed from the NOR gate in Figure 4*a* (output-bubble absorption). This bubble removal is algebraically equivalent to prime removal from the parentheses on the left of T11-L. Using *inversion bubbles* the circuit in Figure 4*b* may be abbreviated as shown in Figure 4*c*, which illustrates an equivalent symbol for a two-input NOR gate. This procedure may be extended to three- or more-input NOR gates, and summarized as follows:

To convert a NOR gate to an AND gate:

- Replace the NOR gate with a same-size AND gate. (output bubble absorption)
- Invert all inputs.

We may also wish to perform this transformation in the reverse order, i.e., to convert an AND gate to a NOR gate:

- Replace the AND gate with a same-size NOR gate.
- Invert all inputs.

A similar transformation for T11-R is shown in Figure 5. Here a two-input NAND gate shown in Figure 5*a* is converted to a two-input OR gate with inverted inputs as shown in Figure 5*b*. The latter may be abbreviated using inversion bubbles as shown in Figure 5*c*, which illustrates an equivalent symbol for a two-input NAND gate. This procedure may be extended to three- or more-input NAND gates, and summarized as follows:

To convert a NAND gate to an OR gate:

- Replace the NAND gate with a same-size OR gate. (output bubble absorption)
- Invert all inputs.

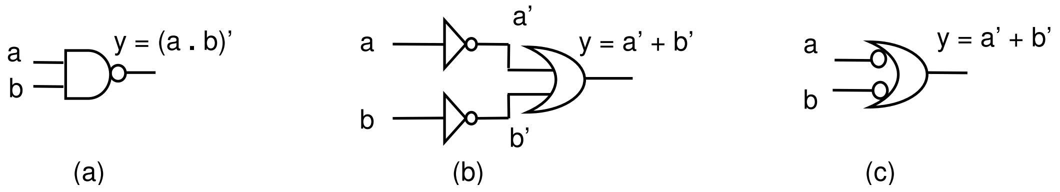


Figure 5. Circuits realizing T11-R: (a) (a . b)', (b) a' + b', (c) equivalent symbol for a NAND gate

We may also need to perform this transformation in the reverse order, i.e., to convert an OR gate to a NAND gate. The procedure for this conversion is as follows:

- Replace the OR gate with a same-size NAND gate.
- Invert all inputs.

Example 15. Apply DeMorgan's theorem T11-R to the logic circuit shown in Figure 6*a*.

In this circuit two inputs already have inverters, but the rule is the same: replace the NAND gate with a same-size OR gate and then complement the inputs, as shown in Figure 6*b*, where the already-inverted inputs (top and bottom ones) have received a second inverter each. Since two cascaded inverters cancel out each other (as we saw in Chapter 2 and also shown by T4), the circuit is eventually simplified as illustrated in Figure 6*c*. ◊

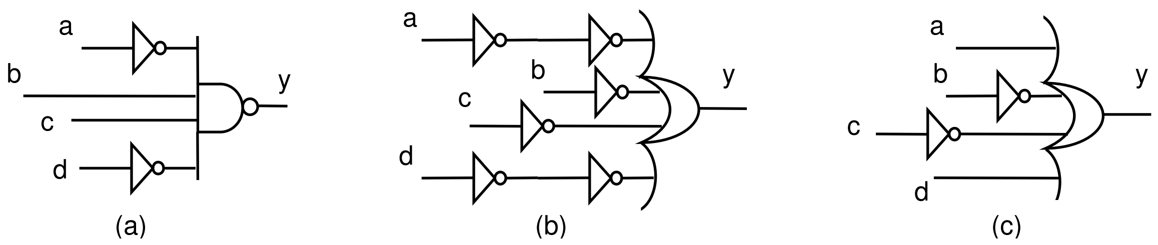


Figure 6. An example for T11-R: (a) original circuit, (b) intermediate circuit, (c) resulting circuit

Using DeMorgan's theorems we can eliminate all inversion bubbles from the outputs of all NAND/NOR gates in any digital circuit in which every gate output drives no more than one gate input as shown in Example 16. This is algebraically equivalent to prime removal from all parentheses in the corresponding logic expression. The one-output-to-one-input constraint mentioned above will be considered in more detail in Example 17. In this book we assume that this constraint is always in place unless otherwise specified.

Here is the procedure to remove all (output) bubbles from NAND/NOR gates in a digital circuit to obtain an AND/OR-only equivalent circuit. Notice that the circuit topology is not supposed to change; i.e., the number of gates and their relative positions in the network should remain unchanged. Inverters are allowed to complement only the primary inputs.

Begin with the gate generating the output of the circuit in hand and move backwards towards the primary inputs.

Repeat until no inverted gate remains

> If the gate under consideration has an output bubble, apply the appropriate form of DeMorgan's theorems and then move the newly generated input bubbles to the outputs of gates (if any) driving the gate under consideration. If a newly generated bubbled input (of the gate under consideration) is driven by a primary input (either inverted or non-inverted), then replace the bubble with an (explicit) inverter. Remember that two consecutive inverters cancel out each other.
>
> If the gate under consideration has no output bubble, move its input bubbles (if any) to the outputs of gates (if any) driving the gate under consideration. If a bubbled input (of the gate under consideration) is driven by a primary input (either inverted or non-inverted), then replace the bubble with an (explicit) inverter.

Example 16. Apply DeMorgan's theorems to the logic circuit shown in Figure 7*a* to obtain a logically equivalent circuit comprised of AND/OR gates only.

Starting with gate 1 in Figure 7*a* apply T11-L to this gate to reach Figure 7*b*. Now the bubble at the output of gate 2 cancels the upper input bubble of gate 1. Also move the bubble at the lower input of gate 1 to the output of gate 4, as shown in Figure 7*c*, where AND 4 becomes NAND.

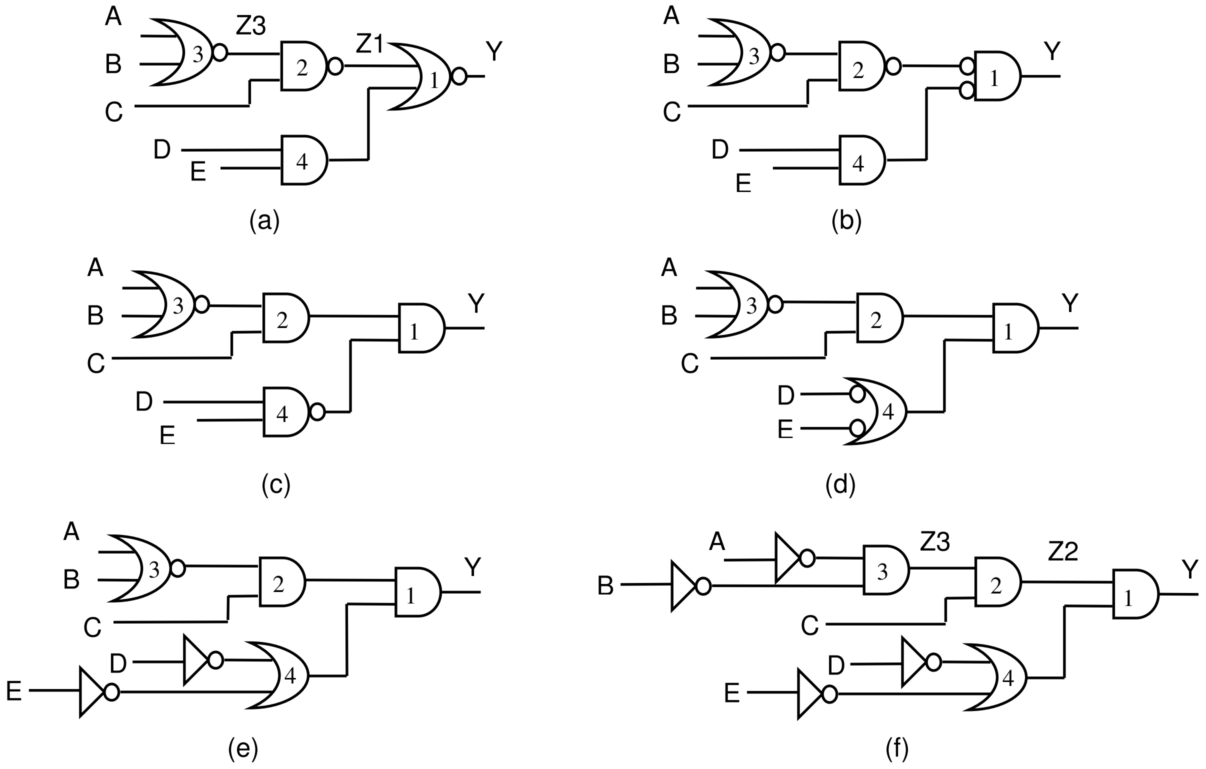


Figure 7. Bubble removal in a typical multistage logic circuit

Now consider gate 2 in Figure 7*c*; it is not bubbled, hence it does not undergo any changes. But gate 4 in Figure 7*c* is bubbled; so apply T11-R to this NAND gate to reach Figure 7*d*. Since gate 4 is not driven by any other gates, its input bubbles are replaced with two explicit inverters as shown in Figure 7*e*. Finally, consider NOR 3 which is not driven by any other gates either. Apply T11-L to this gate and then replace the resulting input bubbles with two inverters as shown in Figure 7*f*, which illustrates the AND/OR-only version of the original circuit. However, keep in mind that the original internal nodes may have been replaced with new ones. For example, node Z1 in Figure 7*a* does not exist in Figure 7*f* anymore; it has been replaced with node Z2. But, node Z3, as an example, is still there.

Example 17. The above conversion to AND/OR-only circuits may not be feasible if the output of one gate drives two or more gates, as shown in Figure 8*a*. To solve this problem we need to clone gate 4 as illustrated in Figure 8*b*, resulting in the AND/OR-only circuit shown in Figure 8*c*. Otherwise, an inverter has to be used between gate 4 and gate 3 as depicted in Figure 8*d*. But in a true AND/OR-only circuit internal nodes (such as Z in Figure 8*d*) are not allowed to get inverted.

Bubble-to-bubble logic diagrams

Take a closer look at the two functionally-equivalent logic circuits in Figure 7*a* and Figure 7*f*. Due to bubbles at the outputs of some of the gates in Figure 7*a* it may not be that easy to read and understand this circuit. For example, what would be your answer to this question: "Does an asserted A assert Y in Figure 7*a*?" you will not be able to propagate signal A = 1 easily to reach Y because of the inversion bubbles that exist on this path. But in Figure 7*f* you may easily recognize that an asserted A will deassert the upper input of AND gate 3, and this will eventually deassert output Y. So, we may arrive at this conclusion that AND/OR-only circuits are more readable than *ill-formed* circuits such as the one shown in Figure 7*a*. However, these gates are typically slower than their inverted versions, NAND and NOR gates; in other words, although AND/OR-only circuits are more readable (than ill-formed circuits), they are slower than their functionally equivalent NAND/NOR-only counterparts. The ideal choice is to utilize only NAND and NOR gates but in a *readable* (as opposed to ill-formed) fashion called *bubble-to-bubble* circuits. In other words, while bubble-to-bubble circuits are faster than their AND/OR-only counterparts, they are as readable as AND/OR-only circuits, as illustrated in Example 18. The following are some rules to obtain a bubble-to-bubble circuit:

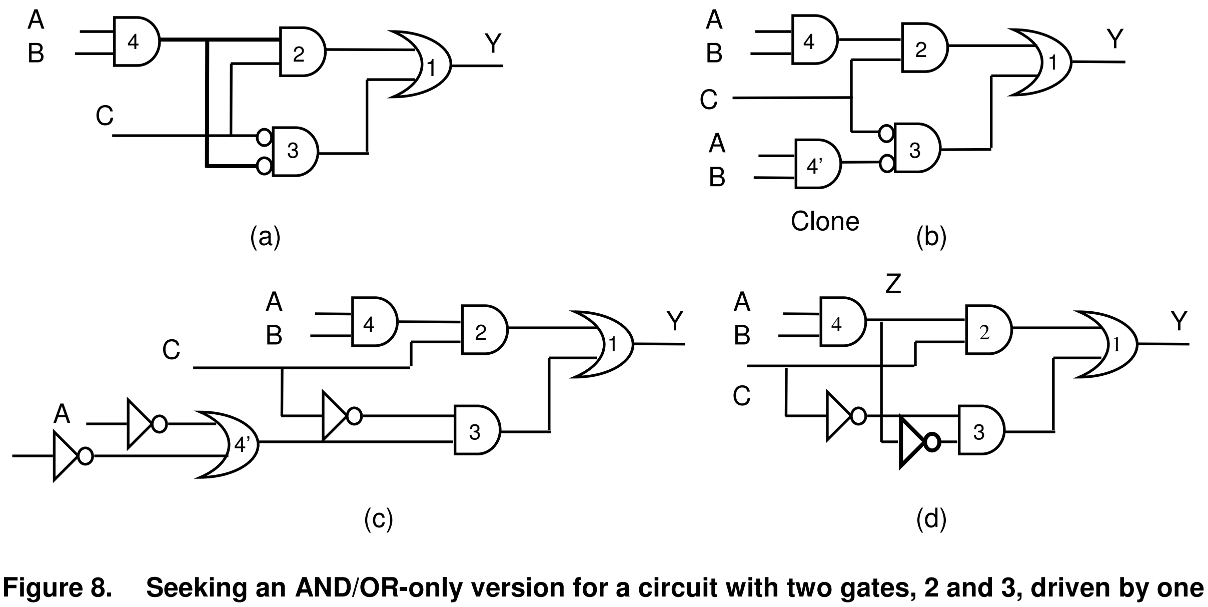


Figure 8. Seeking an AND/OR-only version for a circuit with two gates, 2 and 3, driven by one gate, 4: (a) original circuit, (b) gate 4 is cloned, (c) AND/OR-only circuit with one redundant gate, 4', (d) an inverter placed between gates 4 and 3 to avoid redundancy at the cost of inverting internal node z

Rule 1: Gates and gate symbols
In bubble-to-bubble circuits we use only NAND and/or NOR gates each with two possible symbols shown again in Figure 9*a*/Figure 9*b* (NOR) and Figure 9*c*/Figure 9*d* (NAND) for ease of reference. Three- or more-input gates can of course be used if necessary. Inverters are allowed for complementing input variables only. Two different symbols shown in Figure 9*e* and Figure 9*f* may be used for inverters to adhere to the bubble-to-bubble format.

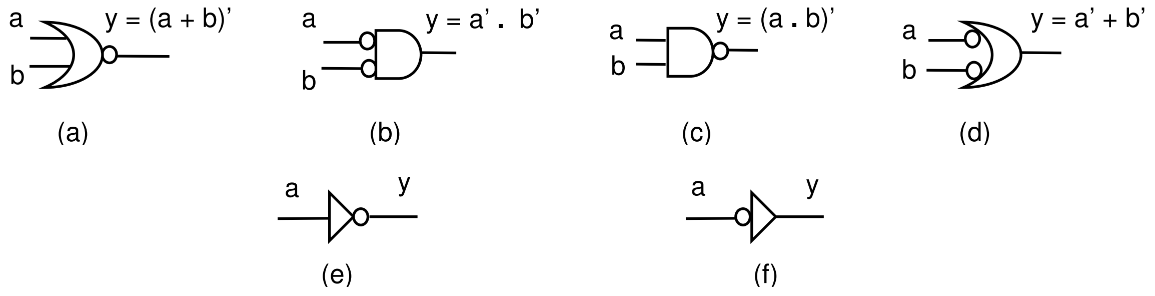


Figure 9. Symbols for: (a), (b) 2-input NOR gate, (c), (d) 2-input NAND gate, (e), (f) inverter

Rule 2: Final gate
The final output is taken from a gate with no output bubble, i.e., a symbol shown in either Figure 9*b* or Figure 9*d*. Rule 2 will be modified later in this chapter to make circuits even more readable.

Rule 3: Interconnections between gates
A bubbled output may get connected to a bubbled input only and vice versa. An unbubbled output may get connected to an unbubbled input only and vice versa. This type of connection is always possible by choosing appropriate symbols for NAND and NOR gates, provided that no gate drives more than one gate.

Rule 4: It cannot always be guaranteed that all primary input signals (coming from the outside world) will arrive at unbubbled inputs, as illustrated in Example 18. This might be considered a *violation* of the ideal bubble-to-bubble format.

Example 18. Obtain a bubble-to-bubble version for the logic circuit shown in Figure 7*e*. This circuit is shown again in Figure 10*a* for ease of reference.

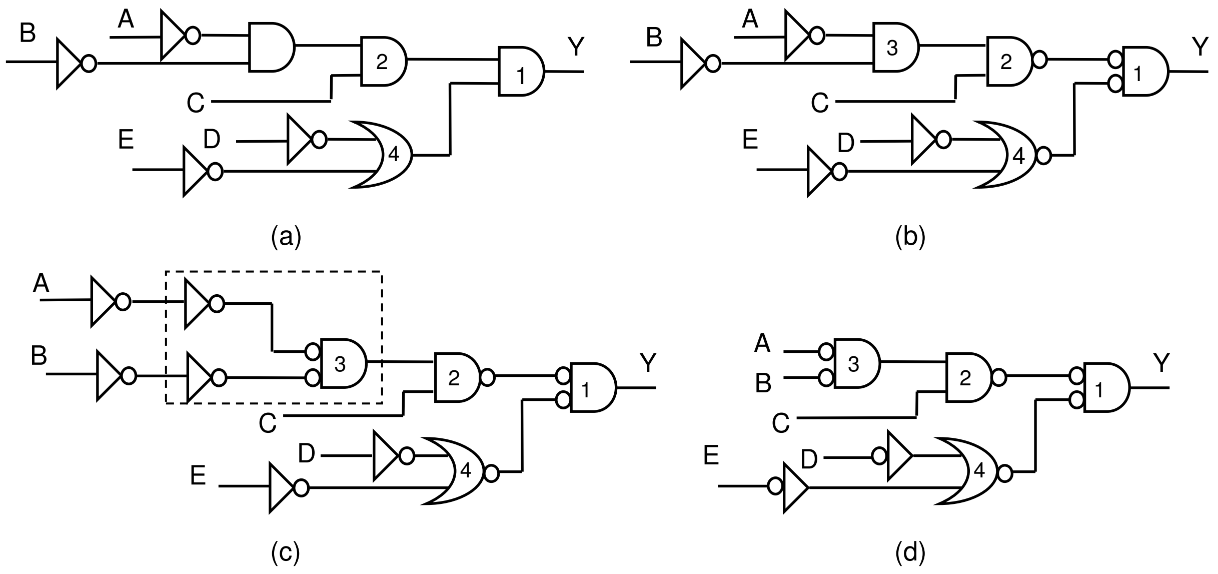


Figure 10. Converting a non-bubble-to-bubble circuit to a bubble-to-bubble circuit

AND 1 is replaced with the NOR symbol shown in Figure 9*b*, and then the two input bubbles are moved to the outputs of gates 2 and 4, as illustrated in Figure 10*b*. Now gate 3 is replaced with the NOR symbol shown in Figure 9*b* preceded by two inverters, as illustrated in Figure 10*c*. Finally, the two cascaded inverter pairs on the top-left corner of Figure 10*c* are removed to reach the circuit shown in Figure 10*d*. Additionally, in this figure the two inverters located at the inputs of gate 4 have been replaced with *bubble-first* symbols to meet the bubble-to-bubble requirements for gate 4.

If every bubble pair (located at a gate output and the following gate input) was removed from the circuit illustrated in Figure 10*d*, the original AND/OR-only circuit shown in Figure 10*a* would be reached. As we build up sufficient experience we may also perform the opposite conversion in one step (instead of the multiple steps shown in Figure 10) to reach the bubble-to-bubble format from an AND/OR-only circuit.

Canonical Representations of Logic Functions
We are already familiar with truth tables and how they represent logic functions. Figure 11 shows an eight-row truth table describing a three-variable logic function. Row 6 (for example) of this table reads "if a = 1, b = 1 and c = 0, then y = 0", i.e., the output value of the function (represented by this table) at 110 is 0. In other words, one *sufficient* condition to deassert y is to have {a = 1, b = 1 and c = 0}. In this section we introduce two more representations for logic functions. But to reach this goal conveniently we first need some definitions.

Row	a b c	y
0	0 0 0	0
1	0 0 1	1
2	0 1 0	1
3	0 1 1	0
4	1 0 0	0
5	1 0 1	1
6	**1 1 0**	**0**
7	1 1 1	1

Figure 11. An eight-row truth table

Definitions:
A *literal* is a variable or its complement, e.g.: a, b'

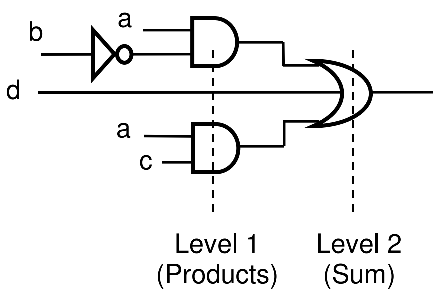
A *product term* or *p-term* for short is a single literal or a logical product (AND) of two or more literals, e.g. a', a . b', a . b' . a

Remember that a logic *expression* is created when one or more variables are operated by one or more operators, e.g. a', a + b

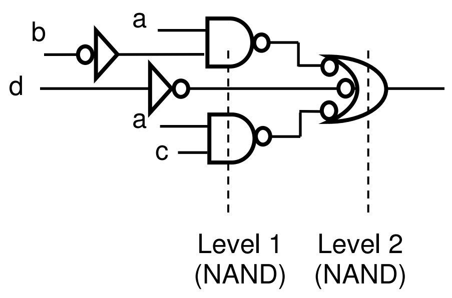
A *sum-of-products* (SOP) expression is a logical sum (OR) of product terms, e.g. a . b' + d + a . c.

Figure 12*a* shows the exact circuit diagram for Y = a . b' + d + a . c. We call it a two-level logic because each primary input has to pass through at the most two gates in series to reach the output. Inverters do not count. A bubble-to-bubble notation for this circuit is illustrated in Figure 12*b*. As clearly shown in this figure, any SOP expression can be realized with a NAND-NAND circuit.

A logic circuit may have more levels. We will see different examples as we move forward. But keep in mind that every three- (or more) level logic circuit can always be translated into a logically equivalent two-level circuit. Chapter 03 teaches us how to do this algebraically. A non-algebraic technique will be covered in Chapter 04.



(a)



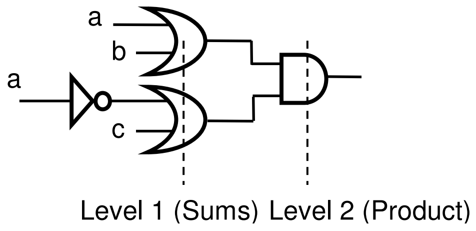
(b)

Figure 12. Two-level SOP logic: (a) AND-OR realization, (b) NAND-NAND realization

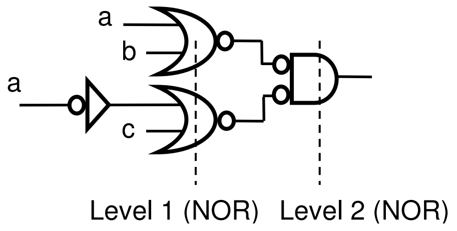
A *sum term* or *s-term* for short is a single literal or logical sum (OR) of two or more literals, e.g. b, a' + c, a + b' + b

A *product-of-sums* (POS) expression is a logical product (AND) of sum terms, e.g. b. (a' + c) . (a + b)

Figure 13*a* shows the exact circuit diagram for Y = (a' + c) . (a + b), which is two level as well. A bubble-to-bubble notation for this circuit is illustrated in Figure 13*b*. As clearly shown in this figure, any POS expression can be realized with a NOR-NOR circuit.



(a)



(b)

Figure 13. Two-level POS logic: (a) OR-AND realization, (b) NOR-NOR realization

A *normal term* is a product or sum term in which no variable appears more than once, e.g. a, (a' + b), (a . b'. c). Here are some examples of non-normal terms: (a + a' + b), (a . b . b).

A *minterm* is a normal product term that has all of the variables, e.g. (a' . b . c') , (a . b . c), assuming a function of three variables, a, b, and c.

A *maxterm* is a normal sum term that has all of the variables, e.g. (a + b + c'), (a + b' + c'), assuming a function of three variables, a, b, and c. ◊

Each row of a truth table corresponds to one specific minterm and also one specific maxterm, as shown in Figure 14 for a truth table with three variables, a, b, and c.

Row	a b c	Minterm	Maxterm
0	0 0 0	a' . b' . c'	a + b + c
1	0 0 1	a' . b' . c	a + b + c'
2	0 1 0	a' . b . c'	a + b' + c
3	0 1 1	a' . b . c	a + b' + c'
4	1 0 0	a . b' . c'	a' + b + c
5	1 0 1	a . b' . c	a' + b + c'
6	**1 1 0**	**a . b . c'**	**a' + b' + c**
7	1 1 1	a . b . c	a' + b' + c'

Figure 14. Definitions of 3-literal minterms and maxterms

Pay close attention to how minterms and maxterms are assigned to different rows. First, remember that a minterm (and also a maxterm) does have all input variables, either inverted or non-inverted. Additionally, in a minterm the literals are ANDed (while in a maxterm the literals are ORed). To find out whether a variable in a minterm must be inverted or non-inverted we have to check the value of that variable in the input column of the same row. Here is the rule: A variable participating in a minterm is inverted if that variable has a 0 value in the input column of the corresponding row; otherwise, the variable is not inverted. For example, consider row 6 where the input column reads: a b c = 1 1 0. Since c = 0, c must be inverted in the minterm of this row. On the other hand, since a = 1 and b = 1 in the input column, these variables must not be inverted in the minterm of this row. Therefore, the minterm representing this row would be (a . b . c'), as summarized in Figure 15.

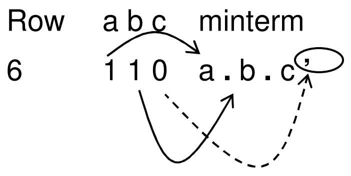


Figure 15. Minterm 6 is obtained: 1 → non-inverted variable, and 0 → inverted variable

Conclusion 1. In a truth table each row has a unique minterm because each row has its own unique bit pattern in the input column.

Remember that a maxterm does have all variables as well, either inverted or non-inverted. Additionally, in a maxterm the literals are ORed. A procedure similar to what is described above for minterms, is carried out to determine whether or not a variable must be inverted in a maxterm. Here is the rule: A variable participating in a maxterm is inverted if that variable has a 1 value in the input column of the corresponding row; otherwise it is not inverted. For example, consider row 6 where the input column reads: a b c = 1 1 0. Since c = 0, c must not be inverted in the maxterm of this row. On the other hand, since a = 1 and b = 1 in the input column, these variables must be inverted in the maxterm of this row. Therefore, the maxterm representing this row would be (a' + b' + c), as summarized in Figure 16.

Conclusion 2. In a truth table each row has a unique maxterm because each row has its own unique bit pattern in the input column.

Conclusion 3. The maxterm and minterm of each row are the complement of each other. For example, for row 6 we have (a . b . c')' = a' + b' + c.

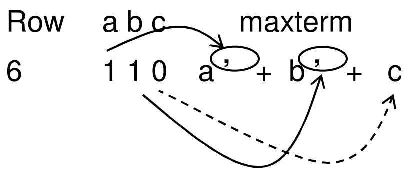


Figure 16. Maxterm 6 is obtained: 1 → inverted variable, and 0 → non-inverted variable

Definition: The set of all *minterms* each with a 1 in the output column of a truth table is called the *on-set* minterms of the corresponding function.

Definition: The set of all *maxterms* each with a 0 in the output column of a truth table is called the *off-set* maxterms of the corresponding function.

Off-set minterms and on-set maxterms may also be defined in a similar way.

By simply on-set or off-set we mean the on-set minterms or the off-set maxterms, respectively.

Conclusion 4. Each row in a truth table belongs to either the on-set or the off-set. In other words, a truth table is *partitioned* into the on-set and off-set rows[5].

Conclusion 5. In order to pull the output up, an on-set minterm has to be pulled up by applying proper inputs. No more than one on-set minterm may be pulled up at a time.

Conclusion 6. In order to pull the output down, an off-set maxterm has to be pulled down by applying proper inputs. No more than one off-set maxterm may be pulled down at a time.

Conclusion 7. Every function may be algebraically written as the *sum* of all its on-set minterms. Similarly, every function may be algebraically written as the *product* of all its off-set maxterms.

Definition. *Canonical SOP* of a function is the sum of all on-set minterms of that function, e.g. f(a, b, c) = a . b . c + a . b' . c + a . b' . c' + a'. b . c' + a' . b' . c

Definition. *Canonical POS* of a function is the product of all off-set maxterms of that function, e.g. f(a, b, c) = (a' + b' + c) . (a + b' + c') . (a + b + c)

Figure 17*a* and Figure 17*b* show the exact logic diagrams for the above canonical SOP and canonical POS, respectively. Remember that (**.**) has precedence over (+), so a + a . b means a + (a . b).

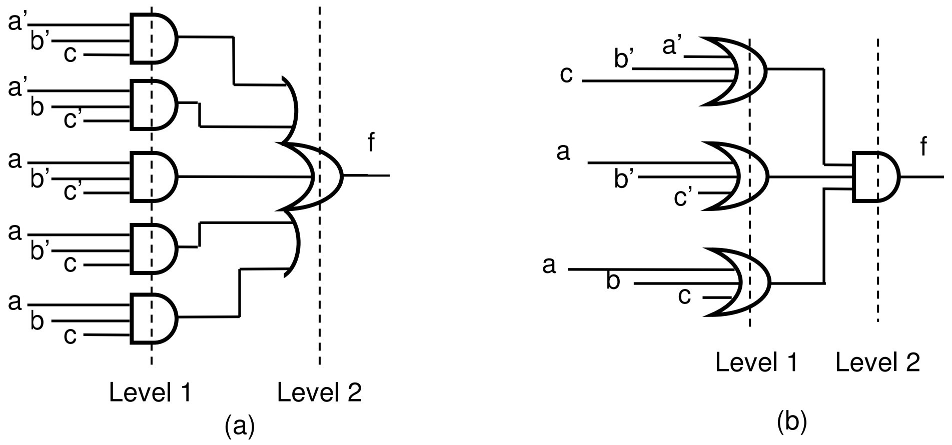


Figure 17. Logic diagrams for: (a) canonical SOP, (b) canonical POS

Example 19. Consider the truth table shown in Figure 18. Obtain the on-set, off-set, canonical SOP and canonical POS of this function.

Row	a b c	y
0	0 0 0	0
1	0 0 1	1
2	0 1 0	0
3	0 1 1	1
4	1 0 0	0
5	1 0 1	0
6	1 1 0	0
7	1 1 1	1

Figure 18. Three-variable truth table for Example 19

On-set = {a' . b'. c, a' . b . c, a . b . c}

Off-set = {a + b + c, a + b' + c, a' + b + c, a' + b + c', a' + b' + c}

[5] This will change when the concept of *don't care* in output columns is introduced.

Canonical SOP: $y = a' . b' . c + a' . b . c + a . b . c$

Canonical POS: $y = (a + b + c) . (a + b' + c) . (a' + b + c) . (a' + b + c') . (a' + b' + c)$ ◊

For more-variable functions the detailed notations used above could be error-prone and hard to read. Therefore, two shorthand notations, *minterm list* and *maxterm list*, may instead be adopted to describe functions:

The minterm list to describe the function in Example 19: $y(a, b, c) = \Sigma_{a, b, c} (1, 3, 7)$
where 1, 3 and 7 are the (decimal) row numbers corresponding to the on-set of y.

The maxterm list to describe the function in Example 19: $y(a, b, c) = \prod_{a, b, c} (0, 2, 4, 5, 6)$
where 0, 2, 4, 5 and 6 are the (decimal) row numbers corresponding to the off-set of y.

Example 20. Obtain the canonical POS and maxterm list of $y(a, b, c) = \Sigma (0, 2, 6)$.

Remember that any row which does not belong to the on-set belongs to the off-set, and vice versa. Since the on-set is (0, 2 and 6) the remaining rows (1, 3, 4, 5 and 7) belong to the off-set of this three-variable function; therefore

$y(a, b, c) = \prod (1, 3, 4, 5, 7)$

$y = (a + b + c') . (a + b' + c') . (a' + b + c) . (a' + b + c') . (a' + b' + c')$

As a reminder, maxterm $(a + b + c')$ corresponds to row 1 ($a\ b\ c = 0\ 0\ 1$) and is obtained by applying the assignment rule: a variable which is 1 in the input column is inverted in the maxterm, otherwise it is not.

Example 21. Obtain the canonical SOP and minterm list of $y(a, b, c) = \prod (2, 4)$.

Since the off-set is (2 and 4), the remaining rows (0, 1, 3, 5, 6 and 7) belong to the on-set of this three-variable function; therefore

$y(a, b, c) = \Sigma (0, 1, 3, 5, 6, 7)$

$y = a' . b' . c' + a' . b' . c + a' . b . c + a . b' . c + a . b . c' + a . b . c$

As a reminder, minterm $(a' . b' . c)$ corresponds to row 1 ($a\ b\ c = 0\ 0\ 1$) and is obtained by applying the assignment rule: a variable which is 0 in the input column is inverted in the minterm, otherwise it is not.

Example 22. Obtain the truth table of the following three-variable function:
$g = \prod_{u, v, w} (1, 4, 5, 7)$

This statement reads that in the output column of g's 8-row truth table, rows 1, 4, 5 and 7 (the off-set rows) each have a 0, and the remaining rows, 0, 2, 3 and 6 (the on-set rows), each have a 1. This interpretation can directly be transferred to a truth table as shown in Figure 19.

Row	u v w	g
0	0 0 0	1
1	0 0 1	0
2	0 1 0	1
3	0 1 1	1
4	1 0 0	0
5	1 0 1	0
6	1 1 0	1
7	1 1 1	0

Figure 19. Three-variable truth table for Example 22

Example 23. Obtain the truth table of the following four-variable function:
y (a, b, c, d) = Σ (1, 4, 6, 9, 11, 12)

This statement reads that in the output column of y's 16-row truth table, rows 1, 4, 6, 9, 11 and 12 (the on-set rows) each have a 1, and the remaining rows, 0, 2, 3, 5, 7, 8, 10, 13, 14 and 15 (the off-set rows), each have a 0. This interpretation can directly be transferred to a truth table as shown in Figure 20.◊

Row	a b c d	y	Row	a b c d	y
0	0 0 0 0	0	8	1 0 0 0	0
1	0 0 0 1	1	9	1 0 0 1	1
2	0 0 1 0	0	10	1 0 1 0	0
3	0 0 1 1	0	11	1 0 1 1	1
4	0 1 0 0	1	12	1 1 0 0	1
5	0 1 0 1	0	13	1 1 0 1	0
6	0 1 1 0	1	14	1 1 1 0	0
7	0 1 1 1	0	15	1 1 1 1	0

Figure 20. Four-variable truth table for Example 23

The following is a direct conclusion from Conclusion 7:

Conclusion 8. Every logic function may be realized with three types of gates: NOT, AND and OR. (Remember that in the axioms of switching algebra only these three types are defined.) ◊

In the following two conclusions we take one more step forward, and interestingly state that every logic function may be realized with only one type of gate with two inputs, although not necessarily efficiently.

Conclusion 9. Every logic function can be realized with two-input NAND gates. No other gates are necessary.

Conclusion 10. Every logic function can be realized with two-input NOR gates. No other gates are necessary.

Example 24. Realize g with only two-input NAND gates.
g = x . y + x' . y . z

The above expression has first been realized in Figure 21*a* with one two-input AND gate for the first product term, one three-input AND gate for the second product term, one two-input OR gate for the OR operator, and one inverter to complement x. In Figure 21*b* the OR gate is converted to a NAND gate preceded by two inverters (bubbles) by applying DeMorgan's theorem, T11-R. Also the three-input AND gate is split into two two-input AND gates, gate 3 and gate 5, as shown in Figure 21*b*. The bubbles at the inputs of NAND 1 in Figure 21*b* are then moved to the outputs of gates 2 and 3 to convert them to NAND gates, as shown in Figure 21*c*. On the other hand, gate 5 in Figure 21*b* is replaced with one NAND gate (No 5) followed by one NAND-based inverter (No 4) in Figure 21*c*. The single inverter to complement input x is also realized with a NAND gate (No 6). Figure 21*d* shows a 2-input-NOR-only equivalent for this circuit, which may be obtained in a similar way. ◊

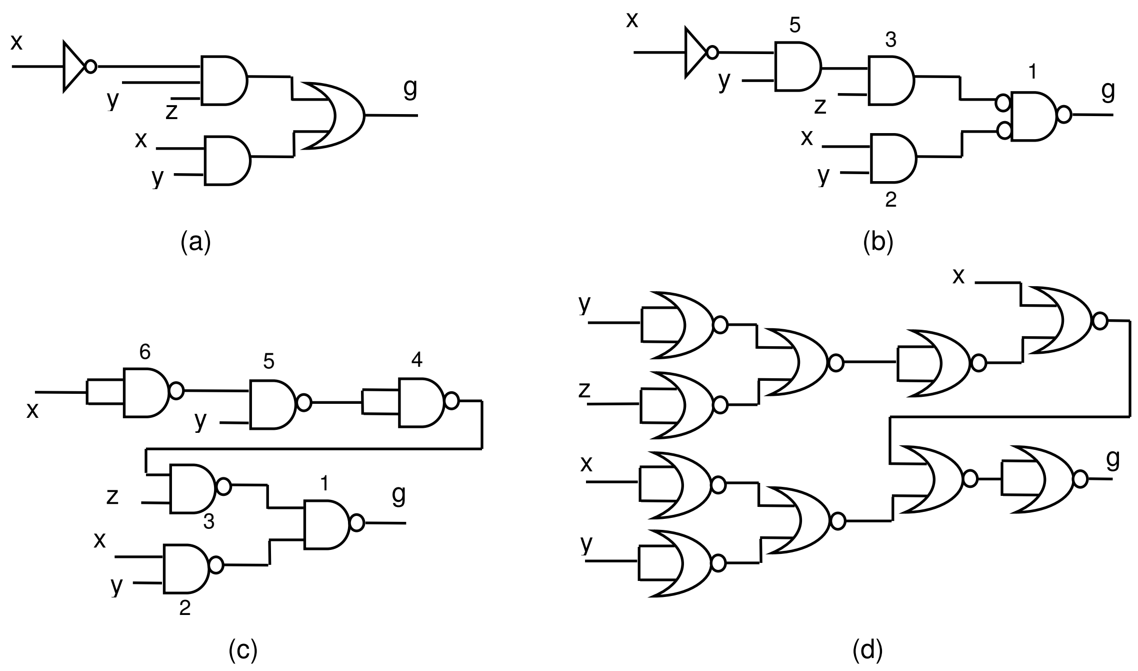


Figure 21. Realizing a logic function with only one type of gate: (a) original circuit, (b) intermediate circuit, (c) two-input-NAND-only circuit, (d) two-input-NOR-only circuit

Example 25. Obtain the truth table and then the canonical SOP and POS of y(a, b, c) = a + b’ . c.

The truth table is shown in Figure 22. There are three variables involved in this function; therefore we need a 2^3 = 8-row truth table. In this table a separate column has been allocated to each of b’ and b’.c to make the translation (from logic expression to truth table) as smooth as possible. The b’ column is obtained by bit-wise complementing the b column and then the b’ . c column is obtained by row-wise ANDing the b’ and c columns. Once the b’ . c column is determined, the y column (final output) is obtained by row-wise ORing the b’ . c and a columns.

Row	a b c	b’	b’ . c	y = a + b’ . c
0	0 0 0	1	0	0
1	0 0 1	1	1	1
2	0 1 0	0	0	0
3	0 1 1	0	0	0
4	1 0 0	1	0	1
5	1 0 1	1	1	1
6	1 1 0	0	0	1
7	1 1 1	0	0	1

Figure 22. Three-variable truth table for Example 25

Now, we may obtain the canonical SOP and POS from the truth table.

y (a, b, c) = Σ (1, 4, 5, 6, 7)

y (a, b, c) = a’ . b’ . c + a . b’ . c’ + a . b’ . c + a . b . c’ + a . b . c

y (a, b, c) = ∏ (0, 2, 3)

y (a, b, c) = (a + b + c) . (a + b’ + c) . (a + b’ + c’) ◊

To reach the canonical representation we do not have to obtain the function's truth table; we may use switching algebra instead, as shown in Example 26 and Example 27.

Example 26. Convert $f(a, b, c) = a + b' . c$ to its canonical SOP:

Apply T10-L: $a = a . b + a . b'$
Apply T10-L: $a = a . b . c + a . b . c' + \mathbf{a . b' . c} + a . b' . c'$ (1)
Apply T10-L: $b' . c = \mathbf{a . b' . c} + a' . b' . c$ (2)

In the original expression substitute for a and $b' . c$ from (1) and (2), respectively, and then remove the repeated term $(a . b' . c)$ according to T3-L:

$f = a . b . c + a . b . c' + a . b'. c + a . b'. c' + a' . b' . c$

Example 27. Convert $g(a, b, c) = a . (b + c)$ to its canonical POS:

Apply T10-R: $a = (a + b) . (a + b')$
Apply T10-R: $a = (\mathbf{a + b + c}) . (a + b + c') . (a + b' + c) . (a + b' + c')$ (3)
Apply T10-R: $(b + c) = (\mathbf{a + b + c}) . (a' + b + c)$ (4)

In the original expression substitute for a and $(b + c)$ from (3) and (4), respectively, and then remove the repeated term $(a + b + c)$ according to T3-R:

$g = (a + b + c) . (a + b + c') . (a + b' + c) . (a + b' + c') . (a' + b + c)$

If you do not feel comfortable yet with T10-R you may take the following steps of conversions:

Apply T1-L: $a = a + (0)$
Apply T5-R: $a = a + (b . b')$
Apply T8-R: $a = (a + b) . (a + b')$
Apply T1-L: $a = (a + b + (0)) . (a + b' + (0))$
Apply T5-R: $a = (a + b + (c . c')) . (a + b' + (c . c'))$
Apply T8-R: $a = (\mathbf{a + b + c}) . (a + b + c') . (a + b' + c) . (a + b' + c')$ (5)
Apply T1-L: $(b + c) = b + c + (0)$
Apply T5-R: $(b + c) = b + c + (a . a')$
Apply T8-R: $(b + c) = (\mathbf{a + b + c}) . (a' + b + c)$ (6)

In the original expression substitute for a and $(b + c)$ from (5) and (6), respectively, and then remove the repeated term $(a + b + c)$.

Example 28. Convert $f(a, b, c) = a + b' . c$ to its canonical POS.

First use T8-R to convert f to POS and then use the procedure mentioned in Example 27, to reach the canonical POS.

Apply T8-R: $a + b' . c = (a + b') . (a + c)$

Apply T10-R: $a + b' . c = (\mathbf{a + b' + c}) . (a + b' + c') . (a + b + c) . (\mathbf{a + b' + c})$

Remove the repeated term $(a + b' + c)$ according to T3-R:

$f(a, b, c) = (a + b' + c) . (a + b' + c') . (a + b + c)$ ◊

Some simple circuits were analyzed in Chapter 2 (see Example 4 in Chapter 2). Now in Example 29 a more complex circuit is analyzed following the same procedure described in Chapter 2.

Example 29. Obtain the (exact) logic expression of the circuit shown in Figure 23.

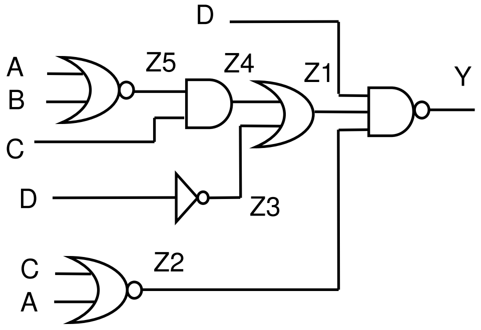


Figure 23. Digital circuit for Example 29

Here is the procedure:

Label the intermediate nodes and then write the algebraic equation of the final output. Now in this equation, substitute for all intermediate signals (from their own equations) to obtain a new equation for the output. Make similar substitutions repeatedly to eventually eliminate all intermediate signals from the output equation.

In Figure 23 the intermediate signals are Z1, Z2, Z3 and Z3.

Y = (D . Z1 . Z2)'

In this equation there are two intermediate signals, namely Z1 and Z2:

Z1 = Z4 + Z3
Z2 = (A + C)'

Z2 has nothing to be substituted for. However, Z1 still has two intermediate signals, Z4 and Z3:

Z4 = Z5 . C
Z3 = D'

The last substitution is for Z5:

Z5 = (A + B)'

Therefore:

Z4 = (A + B)' . C
Z1 = (A + B)' . C + D'
Y = (D. ((A + B)' . C + D') . (A + C)')' ◊

Example 30. Obtain the bubble-to-bubble version of the circuit shown in Figure 23. This circuit has been redrawn in Figure 24*a* for ease of reference.

Starting with NAND 1, replace it with the second NAND symbol shown in Figure 9*d* to make the final output bubble-free as shown in Figure 24*b*, in which gate-5-to-gate-1 is bubble-to-bubble; however gate-2-to-gate-1 is not. So, apply T11-R to gate 2 to make gate-2-to-gate-1 bubble-to-bubble as shown in Figure 24*c*. Then move the input bubbles of gate 2 backwards to convert AND 3 to a NAND and also to remove the inverter as shown in Figure 24*d*. Now we need to make gate-3-to-gate-2 bubble-to-bubble. Use the second NAND symbol for gate 3 to eventually reach the circuit shown in Figure 24*e* in which all inter-gate connections are consistent with the bubble-to-bubble notation. Since two (consecutive) bubbles located on every gate-output-to-gate-input connection cancel each other out, we may easily convert the bubble-to-bubble circuit to its AND/OR-only equivalent, as illustrated in Figure 24*f*. Now it is easy to

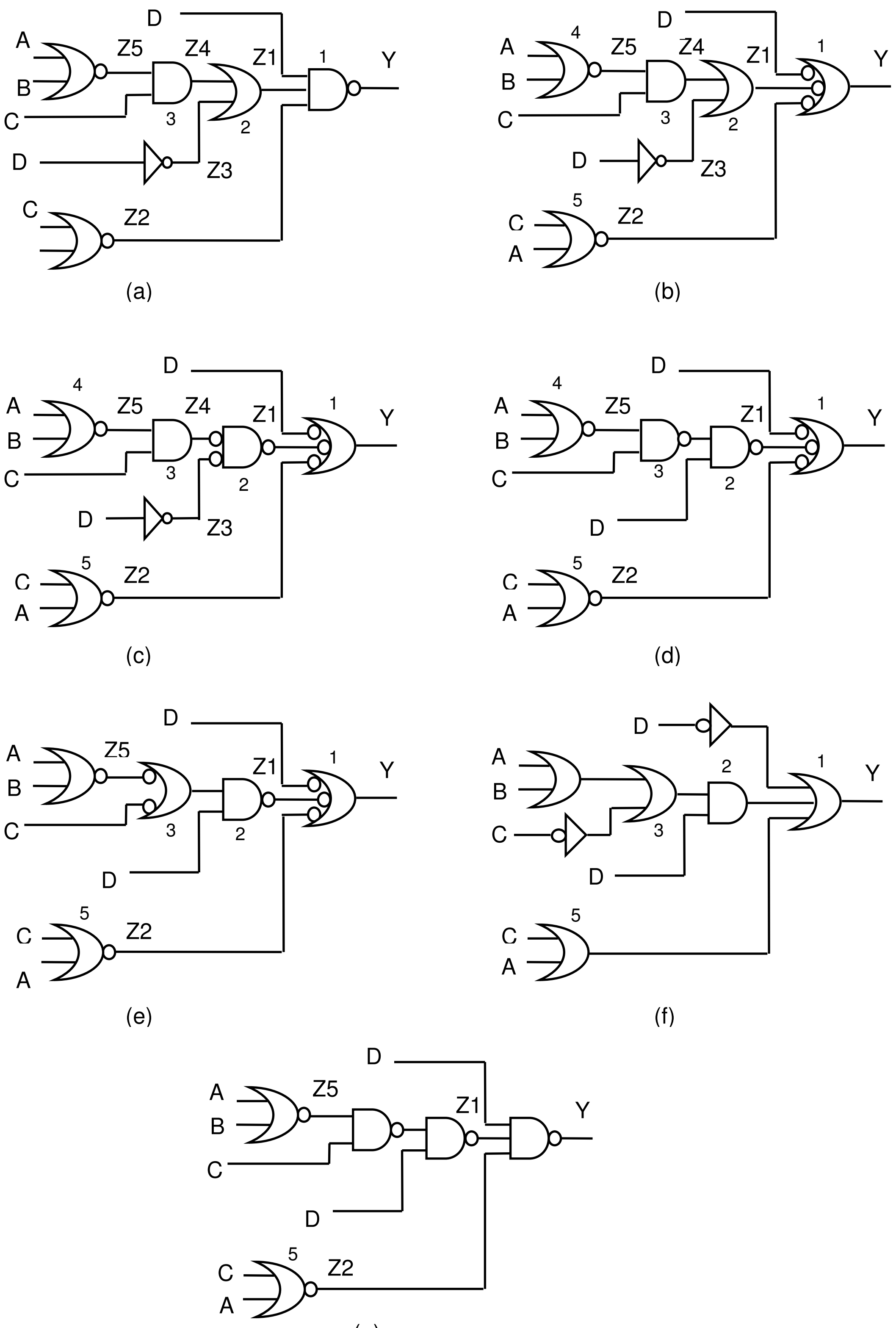


Figure 24. Developing a bubble-to-bubble circuit in Example 30: (a) original circuit, (b)-(d) intermediate circuits, (e) bubble-to-bubble circuit, (f) AND/OR-only version, (g) final circuit with more-traditional symbols for NAND and NOR

obtain an AND/OR-only logic expression for Y:

Y = D' + D . (A + B + C') + (A + C)

The same circuit, but with the more-traditional NAND/NOR symbols, is shown in Figure 24*g*, which is much more difficult to read as compared with the circuit shown in Figure 24*e*.

Some internal nodes might not be preserved during the bubble removal/relocation procedure. For example, while node Z1 (the output of gate 2 in Figure 24*a*) still exists in Figure 24*e*, node Z4 (the output of gate 3 in Figure 24*a*) has been eliminated in Figure 24*e*. If we need Z4 (as another output to go to the outside world) we have to use an additional inverter to complement the output of gate 3 in Figure 24*e*. ◊

Remember that the logic circuit corresponding to the left-side expression of T8-L was obtained in Example 10. Example 31 shows the same procedure but for a more complex expression:

Example 31. Obtain the (exact) logic circuit which realizes the following expression:

Y = ((A + B . C)' . C + A' . D) + (A . C)'

Similar to Example 10, it might be helpful to replace some of the terms of the expression with auxiliary (intermediate) signals. We use as many intermediate signals as we feel adequate to clarify the structure of the circuit. Let's use four intermediate signals (Z1, Z2, Z3 and Z4) in this example:

Z1 = (A . C)'
Z2 = (A + B . C)'
Z3 = A' . D
Z4 = Z2 . C

Therefore, Y could be stated as a simpler expression:

Y = (Z4 + Z3) + Z1

The above equations can be interpreted as follows:

Y is the output of a two-input OR gate: one input is driven by Z1, and the other input is driven by the output of another two-input OR gate (represented by the parentheses) with one input driven by Z3, and the second one driven by Z4. On the other hand,

Z4 is the output of an AND gate with two inputs, Z2 and C.

Z2 is the output of a two-input NOR gate, with one input driven by A, and the other one driven by the output of an AND gate with two inputs, B and C.

Z3 is the output of an AND gate with two inputs, A' and D.

Z1 is the output of a NAND gate with two inputs, A and C.

Based on this analysis, the digital circuit realizing Y = ((A + B . C)' . C + A' . D) + (A . C)' is obtained as shown in Figure 25. ◊

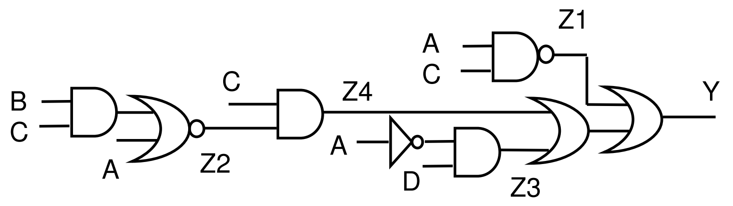


Figure 25. Logic circuit for Example 31

Again, we obtained the *exact* translation of an algebraic expression. But what we normally do in the design of digital circuits is to simplify the expression to save as much hardware as possible. To reach this goal we may use switching algebra as we did in Example 9. An alternative technique will be presented in Chapter 4.

Example 32. Realize both sides of the theorem which was proved in Example 9. The theorem is again shown below for ease of reference:

(a . b) + (a . b . c'. d) + (a . b . d . e') = a . b

To realize the left-side expression we need one two-input AND gate for the first parentheses, two four-input AND gates for the last two parentheses, and one three-input OR gate for the two OR operators in the expression as shown in Figure 26*a*. If you do not feel comfortable yet with this direct translation you may break it down to some intermediate stages similar to what we did in Example 31.

The right-side expression, a . b, needs only one two-input AND gate to get realized as shown in Figure 26*b*, which is much simpler (less expensive) than what the left-side expression needs. ◊

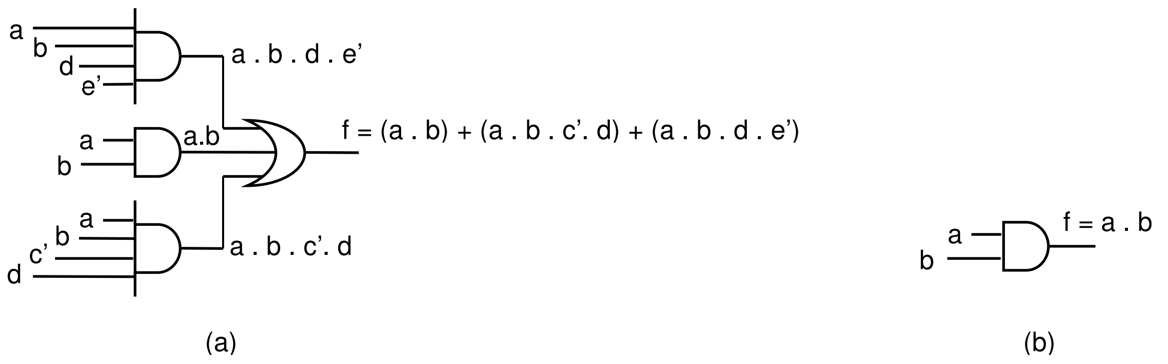


Figure 26. Logic circuits for: (a) original expression, (b) simplified expression in Example 32

The theorem proved in Example 33 is known as the *consensus* theorem, which basically says: if Y . Z = 1, then X . Y or X' . Z must be 1 as well. (But remember that X . Y and X' . Z. cannot both be 1 at the same time.)

Example 33. Prove that **X . Y** + **X' . Z** + Y . Z = X . Y + X' . Z

T10-L: Y . Z = X . Y . Z + X' . Y. Z (7)

Substitute for Y . Z from (7) in the left expression of the theorem:

Left expression = **X . Y** + X . Y . Z + **X' . Z** + X' . Y . Z (8)

Apply T9-L to the first two terms and also to the last two terms in (8):

First two terms: **X . Y** + X . Y . Z = X . Y
Last two terms: **X' . Z** + X' . Y . Z = X' . Z

Therefore:

X . Y + **X' . Z** + Y . Z = X . Y + X' . Z ◊

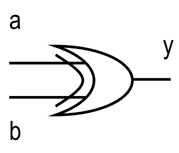
An XOR gate (similar to any other logic function) may be built up using the three basic gates specified in the definition of switching algebra as shown in the following example.

Example 34. Obtain the canonical SOP of a two-input XOR gate.

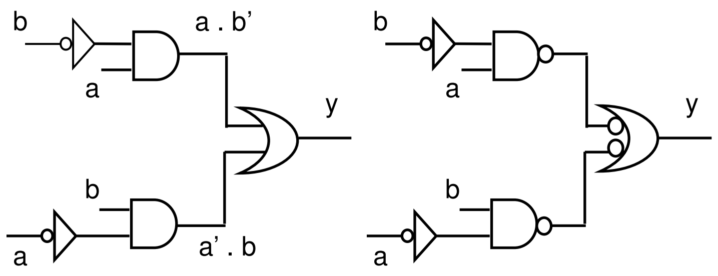
The logic symbol and truth table of a two-input XOR are shown again in Figure 27*a* and Figure 27*b*, respectively. Considering this truth table the canonical SOP of a two-input XOR gate is obtained as follows:

y (a, b) = Σ (1,2) = a' . b + a . b'

The AND/OR-only and bubble-to-bubble logic circuits realizing this gate are shown in Figure 27*c* and Figure 27*d*, respectively.



Row	a b	y
0	0 0	0
1	0 1	1
2	1 0	1
3	1 1	0



(a) (b) (c) (d)

Figure 27. XOR gate: (a) logic symbol, (b) truth table, (c) SOP realization, (d) bubble-to-bubble notation

Active levels for input/output signals

Consider the simple alarm system shown in Figure 28*a*, which receives two inputs, a and b, from a temperature sensor and a pressure sensor, respectively, in a chemical plant. The system places an appropriate logic level on the output line y to sound the alarm if both the temperature and pressure are high; otherwise, the alarm does not sound. Notice that it would be more descriptive to call the output sound, instead of the non-descriptive y. This meaningful naming now raises the following question: is the alarm supposed to go off when sound = 1 or when sound = 0? In other words, there are two choices for the logic level at the output of this logic block to do the job; if the job is carried out by a logic 1, then that output is called *active high*; otherwise, if the job is carried out by a logic 0, then the output is called *active low*. Of course the buzzer to be used has to be consistent with the specific type of the output signal (either active high or active low) generated by the alarm control circuit.

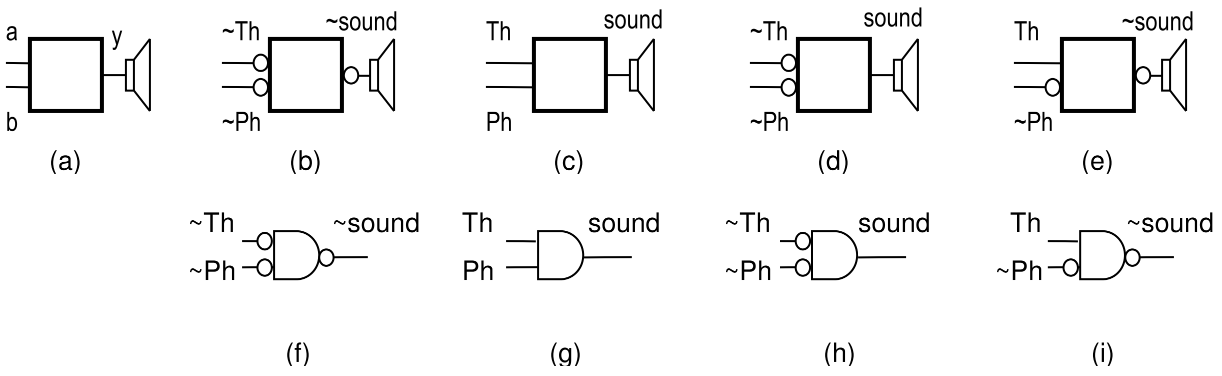


Figure 28. A simple alarm system: (a) initial logic symbol, (b) active-low inputs/output, (c) active-high inputs/output, (d) active-low inputs, active-high output, (e) active-low Ph and sound, active-high Th, (f)-(i) implementations for (b)-(e), respectively

A similar concept exists for the inputs as well. Let's first give descriptive names Th (temperature high) and Ph (pressure high) to inputs a and b, respectively. Th is called active high if the high temperature is signified by a 1 at Th, otherwise Th is called active low. Similarly, Ph is called active low if the high pressure is indicated by a 0 at Ph, otherwise Ph is called active high. To distinguish between these two types of inputs/outputs the name of each active-low input/output receives a suffix or prefix. In this book we use the prefix ~. Additionally, in logic circuits an active-low input/output is signified by a bubble, as shown in different parts of Figure 28. Active-high inputs/outputs are left with no bubbles/prefixes. Figure

28*b* to Figure 28*e* show different notations for four different versions of this alarm system. The implementations for the control circuit are shown in Figure 28*f* to Figure 28*i*, respectively. As an example, in Figure 28*d* and Figure 28*h* the inputs are active low while the output is active high.

With the input/output or *signal* classification mentioned above, we now make a modification to the definition of an asserted signal: a signal is *asserted* if it takes its active logic value; otherwise it is *deasserted* (or *negated*). In other words, an active-low signal is asserted if it is 0; otherwise it is deasserted or negated. Similarly, an active-high signal is asserted if it is 1; otherwise it is deasserted or negated.

With this new interpretation for asserted and deasserted signals, we may reach a concise definition for an n-input AND gate with *any* combination of bubbles at the inputs and output: the output of an n-input AND gate is asserted if all inputs are asserted, otherwise the output is deasserted. Four 2-input examples are shown in Figure 28*f to* Figure 28*i*.

A similar conclusion is valid for an n-input OR gate with *any* combination of bubbles at the inputs and output, such as the four 2-input examples shown in Figure 29*a* to Figure 29*d*: for each of these gates we could say that the output is asserted if at least one input is asserted, otherwise the output would be deasserted. The same input/output names used in the alarm control circuits in Figure 28*f* to Figure 28*i* have again been used for the circuits shown in Figure 29*a* to Figure 29*d*, respectively. Therefore, each circuit in Figure 29 can be considered again as an alarm control circuit, but now a high temperature, a high pressure, or both will be able to sound the alarm. ◊

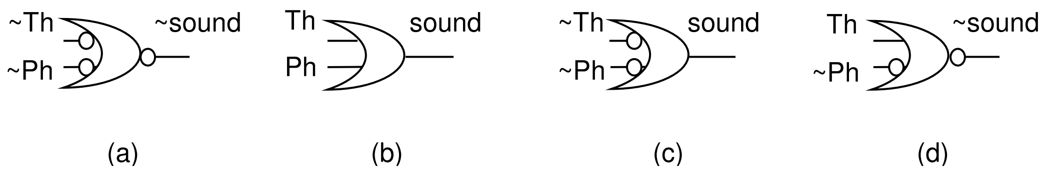


Figure 29. Four different bubbled/unbubbled I/O combinations for a two-input OR gate: (a) active-low inputs/output, (b) active-high inputs/output, (c) active-low inputs, active-high output, (d) active-low Ph and sound, active-high Th

A second look at bubble-to-bubble logic circuits

The output and all inputs of the circuit in hand have implicitly been assumed active high in developing the rules described earlier on page 13. With the new concept of two different assertion levels and to cover any assertion levels for inputs/outputs, rules 2 and 4 are going to be modified in this section. The unchanged rules (rules 1 and 3) are also repeated here for ease of reference:

Rule 1: Gates and gate symbols

In the bubble-to-bubble logic we use only NAND and/or NOR gates each with two possible symbols shown in Figure 9*a*/Figure 9*b* (NOR) and Figure 9*c*/Figure 9*d* (NAND). Three- or more-input gates can of course be used if necessary. Inverters are allowed for complementing input variables only. Two different symbols shown in Figure 9*e* and Figure 9*f* may be used for inverters to adhere to the bubble-to-bubble format.

Rule 2 (modified): Final gate

If the (final) output of a circuit is active high, then the output should be taken from a gate with no output bubble, i.e., a symbol shown in either Figure 9*b* or Figure 9*d*. However, if the (final) output of a circuit is active low, then the output should be generated by a gate with an output bubble, i.e., a symbol shown in either Figure 9*a* or Figure 9*c*.

Rule 3: Interconnections between gates

A bubbled output may get connected to a bubbled input only and vice versa. An unbubbled output may get connected to an unbubbled input only and vice versa. This type of connection is always possible by choosing appropriate symbols for NAND and NOR gates, provided that no gate drives more than one gate.

Rule 4 (modified): Ideally, in a bubble-to-bubble notation active-low inputs should be bubbled and active-high inputs should be unbubbled. This, however, cannot always be guaranteed, resulting in a *violation* of the bubble-to-bubble format, as illustrated in Example 35.

Example 35. Obtain a bubble-to-bubble version for the logic circuit shown in Figure 10*a*. This circuit is shown again in Figure 30*a* for ease of reference. In this circuit output Y and inputs C and E are active low.

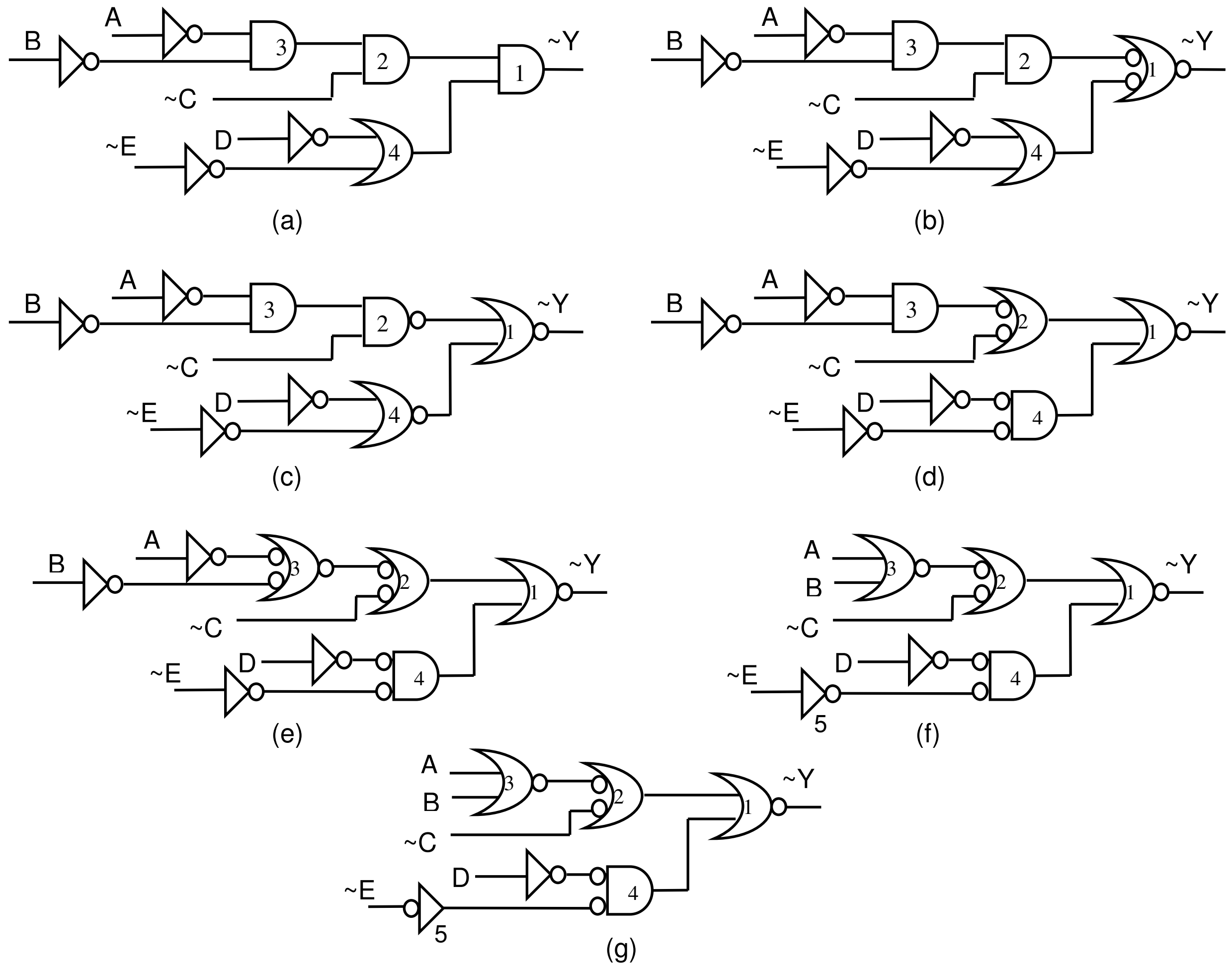


Figure 30. Converting a non-bubble-to-bubble circuit to a bubble-to-bubble circuit in Example 35

1- Replace AND 1 with a NOR gate with inverted inputs as shown in Figure 30*b*.

2- Move the input bubbles of NOR 1 to the outputs of gates 2 and 4 as shown in Figure 30*c*.

3- Replace NOR 4 and NAND 2 with their equivalent symbols shown in Figure 9*b* and Figure 9*d*, respectively, to reach Figure 30*d*.

4- Replace AND 3 with a NOR gate with inverted inputs as shown in Figure 9*e*.

5- Remove the two input bubbles from gate 3 and also remove the two inverters in the path of input signals A and B, to eventually reach Figure 30*f*.

The circuit of Figure 30*f* could also be reached by replacing each gate symbol in Figure 10*d* with its equivalent gate symbol. Equivalent gate symbols are shown in Figure 9.

Take a second look at input E in Figure 30*f*: E is an active-low input but it is not bubbled, violating Rule 4 described above. This inconsistency may be fixed by using a bubble-first symbol for inverter 5 as shown in Figure 30*g*, but now a bubble-to-bubble violation emerges between this inverter and the following NOR gate. Therefore, Rule 4 cannot always be satisfied.