Logic design

1— Boolean algebra and Karnaugh maps

by B. Holdsworth and L. Zissos *Chelsea College, University of London*

Up to 1969, when the Boolean sequential equations were developed, the design of sequential circuits was achieved through an empirical choice of unrelated informal techniques paying little attention to engineering constraints until, in most cases, the implementation stage. The advent of the sequential equations has made possible the development of clear-cut step-by-step design procedures in which realistic circuit constraints are taken into account at the design level. No engineering or other specialist knowledge is necessary to use these design procedures.

The design philosophy adopted in this series is one that allows the emphasis to be placed on optimal rather than minimal design. This is to enable technicians, users with no specialist knowledge of electronics, and the less experienced designer, to produce sound and economical designs, while at the same time providing the means whereby the specialist designer may improve his technique in dealing with more sophisticated assemblies involving such devices as r.o.ms, r.a.ms, microprocessors, and so on.

The primary design objective is to produce sound and reliable digital systems which are meaningful not only to the designer but also to the user.

Basic concepts

As in conventional algebra, so in Boolean algebra variables are combined into expressions with operators that obey certain laws. The Boolean variables, denoted by letters of the alphabet such as A,B,C etc., are binary variables and may assume one of two values, 0 or 1, or they may be alternatively read as 'false' and 'true' respectively. They are not the 'zero' and 'one' of arithmetic and the operations that can be performed on them are somewhat different and more limited than the normal arithmetical processes.

Although there exists a wide number of Boolean operators, such as NAND, NOR, etc., we need only consider three operators at this stage — all other operators can be expressed in terms of these three. They are:

- Boolean addition,
- Boolean multiplication,
- Boolean inversion.

The addition operator is written as + and may be interpreted as 'OR'. A+B may be read 'A or B' or 'A plus B'. It is true if either A is true or B is true or both are true, otherwise it is false. Thus,

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

The multiplication operator is written as . or ×, or omitted when its factors are variables denoted by single letters, and may be interpreted as 'AND'. A.B (or AB) may be read 'A and B' or as 'A times B'. It is true if A and B are both true, and false otherwise. Thus,

$$0\times0=0$$
$$0\times1=0$$
$$1\times1=1$$
$$1\times0=0$$

The inversion operator is written as a bar over the variable and the bar may be interpreted as "NOT". For example, $\bar{A}$ may be read as "NOT A".

$$\text{If } A=1 \quad \text{then } \bar{A}=0$$
$$\text{and if } A=0 \quad \text{then } \bar{A}=1$$

Boolean theorems

Redundancy.

$$A+AB = A$$

Fig. 1. The redundancy theorem implemented in a relay circuit. From the three relays giving $f=A+AB$ is derived the single-relay circuit giving $f=A$, since AB contains A and is therefore redundant.

Proof:
$$\begin{aligned} A+AB &= A.1+AB \\ &= A(B+\bar{B})+AB \\ &= AB+A\bar{B}+AB \\ &= AB+A\bar{B} \\ &= A.1 \\ &= A \end{aligned}$$

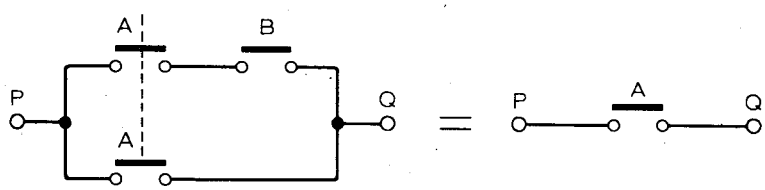
This theorem states that in a sum-of-products Boolean expression, a product that contains all the factors of another product is redundant. As a consequence it allows the elimination of redundant products in a sum-of-products expression. For example, in the Boolean function $f=AB+ABC+ABD$, the products ABC and ABD can be eliminated, because each contains all the factors present in AB.

The application of this theorem to a relay circuit is shown in Fig. 1.

Race-hazards. The main interest of the logic designer in this theorem is in its use in logic circuits for the suppression of race-hazards, which result in the generation of unwanted spikes. For example consider the Boolean function $f=AB+\bar{A}C$. Following changes in A, there is a race-hazard when $B=1$ and $C=1$, since the function then reduces to $f=A+\bar{A}$. The use of an inverter to generate $\bar{A}$ from A implies a delay between the waveforms of A and $\bar{A}$ as shown in Fig. 2. This leads to the generation of a transient signal as indicated in the same diagram.

The unwanted transient can be averted by the introduction of an optional product, that is a Boolean product whose presence in an expression does not affect the value of the Boolean function. A suitable optional product for the function $f=AB+\bar{A}C$ is formed by taking the product of the coefficients A and $\bar{A}$.

Hence, $AB+\bar{A}C = AB+\bar{A}C+BC$



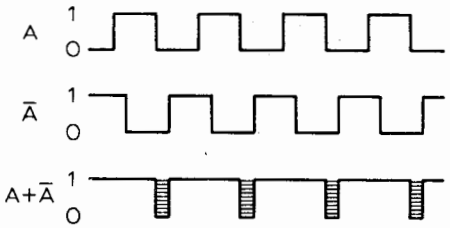


Fig. 2. A race hazard. $\bar{A}$ is obtained by inverting A and is subject to a delay, resulting in the interval during which neither $\bar{A}$ nor A is 'up.' The output $f=A+\bar{A}$ is therefore not true, or 'down' during this time.

Proof: $AB+\bar{A}C+BC$
$=AB+\bar{A}C+(A+\bar{A})BC$
$=AB+\bar{A}C+ABC+\bar{A}BC$
$=AB(1+C)+\bar{A}C(1+B)$
$=AB+\bar{A}C$

The product BC is optional so long as its parent products, AB and $\bar{A}C$ remain in the expression. Should, however, one of its parent products be eliminated (by applying the redundancy theorem), then such a product is no longer optional and cannot be removed from the expression.

If now $B=C=1$ the expression $f=AB+\bar{A}C+BC$ reduces to $f=A+\bar{A}+1$, which always has the value 1 irrespective of the values of A and $\bar{A}$.

The use of optional products will now be demonstrated with the aid of three examples.

(1) **Elimination of parent product.**

$$f=A+\bar{A}BC,$$

Form the optional product BC:

$$f=A+\bar{A}BC+BC,$$

Eliminate parent product $\bar{A}BC$ using theorem of redundancy:

$$f=A+BC.$$

(2) **Elimination of non-parent product.**

$$f=AB+\bar{A}C+BCD$$

Form the optional product BC:

$$f=AB+\bar{A}C+BC+BCD.$$

Eliminate non-parent product BCD using theorem of redundancy:

$$f=AB+\bar{A}C+BC.$$

But BC is an optional product and is redundant, hence

$$f=AB+\bar{A}C.$$

(3) **Elimination of non-parent product and parent product.**

$$f=AB+\bar{A}BC+BCD$$

Form the optional product BC:

$$f=AB+\bar{A}BC+BCD+BC.$$

Eliminate non-parent product BCD and parent product $\bar{A}BC$ using theorem of redundancy:

$$f=AB+BC.$$

De Morgan's theorem. The complement of a Boolean expression can be obtained by replacing each variable by its complement in the corresponding dual expression. For example, the dual of $f=A+BC$ is obtained by replacing the operator + by . and vice versa. Hence the dual expression is

$$f_D=A(B+C)$$

and

$$\bar{f}=\bar{A}(\bar{B}+\bar{C})$$

that this is so can be confirmed with the aid of a truth table as shown in Fig. 3. Examination of columns 8 and 10 of this table show that $\bar{A}(\bar{B}+\bar{C})$ is the complement of $A+BC$.

A	B	C	$\bar{A}$	$\bar{B}$	$\bar{C}$	BC	A+BC	$\bar{B}+\bar{C}$	$\bar{A}(\bar{B}+\bar{C})$
0	0	0	1	1	1	0	0	1	1
0	0	1	1	1	0	0	0	1	1
0	1	0	1	0	1	0	0	1	1
0	1	1	1	0	0	1	1	0	0
1	0	0	0	1	1	0	1	1	0
1	0	1	0	1	0	0	1	1	0
1	1	0	0	0	1	0	1	1	0
1	1	1	0	0	0	1	1	0	0

Fig. 3. The truth table shows that $\bar{A}(\bar{B}+\bar{C})$ is the complement of $A+BC$.

Example. Find the complement of $f=A(BC+\bar{B}\bar{C}+BCD)$.
Apply redundancy

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

dualise: $f_D=A+(B+C)(\bar{B}+\bar{C})$
invert: $f=\bar{A}+(\bar{B}+\bar{C})(B+C)$

$$f=A+B\bar{C}+\bar{B}C$$

Fan in. This theorem has its application in those logic circuits where there is a fan-in restriction placed on the designer by the availability of gate inputs. This matter will be dealt with more fully in a later article.

It is frequently convenient, when multiplying out two Boolean sums to refer to one section of the sum as its head, H, and to the remaining section as its tail, T. The statement of the theorem then is:

$$(H_1+T_1)(\bar{H}_1+T_2)=H_1T_2+\bar{H}_1T_1$$

Proof: l.h.s. $=(H_1+T_1)(\bar{H}_1+T_2)$
$=H_1\bar{H}_1+H_1T_2+\bar{H}_1T_1+T$
Now $H_1\bar{H}_1=0$ and T_1T_2 is redundant by theorem of race-hazards; therefore
l.h.s. $=H_1T_2+H_1T_1$

This theorem allows us to multiply out two Boolean sums, two sections of which are the complement of each other, without generating algebraically redundant products.

The partition of a Boolean sum into head and tail is arbitrary. For example in the case of the Boolean sum $A+B+C$ any of the following partitions is allowable

head	tail
A	B+C
B	A+C
C	A+B
A+B	C
A+C	B
B+C	A

Example. $f=(A+B+C)(\bar{A}+DE+F)$
Let $H_1=A$ and $T_1=B+C$
$H_2=\bar{A}$ and $T_2=DE+F$,
then $(A+B+C)(\bar{A}+DE+F)$
$=A(DE+F)+\bar{A}(B+C)$
$=ADE+AF+\bar{A}B+\bar{A}C$

If there are terms common to both of the sums to be multiplied the process of multiplication can be further simplified by noting that such terms appear in the product in their original form. For example

$(A+BC)(A+DE)$
$=AA+ADE+ABC+BCDE$
$=A+BCDE.$

Hence, if $P=(I+X)$ and $Q=(I+Y)$ where I is the common term,
then $PQ=(I+XY)$.
Finally, if $P=(H_1+T_1+I)$ and $Q=(\bar{H}_1+T_2+I)$,
then $PQ=H_1T_2+\bar{H}_1T_1+I$

Boolean reduction

A Boolean function is said to be irredundant, or reduced, if it contains no optional products. For example, the factor $\bar{A}$ in the function $f=A+\bar{A}B$ is redundant, since $A+\bar{A}B=A+B$. Redundancies in two-level Boolean expressions can be removed in three steps, using the theorems of redundancy and racehazards. If an expression contains more than two levels, it is converted into its two-level sum-of-products form by multiplying out.

The three steps for eliminating redundancies in Boolean expressions are:

(1) Multiply out.

Consider the Boolean function

$$f=BC+(AB+C)\bar{C}+A$$

Apply (1):

$=BC+AB\bar{C}+C\bar{C}+A$
$=A+BC+AB\bar{C}$

(2) Apply redundancy theorem:

In (1) the expression $f=A+BC+AB\bar{C}$ was derived. Step (2) is commenced by considering the first product, in this case A. Now scan the products to the right of A, looking for a product that contains the factor A. Here $AB\bar{C}$ is such a product and this is eliminated, resulting in $f=A+BC$. Since there are no products to the right of BC the step is not repeated.

(3) Apply theorem of race hazards:

The first variable in the first product is selected and the remainder of the expression is scanned for a product that contains the complement of the selected variable. When such a product is found, an optional product is formed using the second theorem. The optional product is used to eliminate non-parent products and/or to replace parent products as previously described. If a parent product has been replaced, the optional product is inserted at the beginning of the expression and (3) is repeated. If the optional product has not been used, it is discarded. Step (3) is repeated until all first-level optional products have been generated. Repeat (3) if necessary using higher level optional products[1]. For example:

$$f=A+\bar{A}B+BC+\bar{A}\bar{B}D.$$

Form the optional product B:

$$f=A+\bar{A}B+BC+\bar{A}\bar{B}D+B.$$

Eliminate parent product $\bar{A}B$ and non-parent product BC:

$$f=B+A+\bar{A}\bar{B}D.$$

Form optional product $\bar{A}D$:

$f = B + A + \bar{A}\bar{B}D + \bar{A}D.$

Eliminate parent product $\bar{A}\bar{B}D$:

$f = \bar{A}D + B + A.$

Form optional product D:

$f = \bar{A}D + B + A + D.$

Eliminate parent product $\bar{A}D$:

$f = A + B + D,$

which is the required result.

Minimisation

A Boolean sum-of-products expression is said to be minimal if (a) no other sum-of-products expression for the same function has fewer products, and (b) of other sum-of-products expressions for the same function with the same number of products, none has fewer factors.

There are three main methods for minimising Boolean expressions.

These are:

- The Karnaugh map method. In this method the function is displayed on a map and by suitable looping arrangements the minimal form is obtained.
- The Quine-McCluskey method[2]. In this method all irredundant forms of a given Boolean function are generated and the shortest one chosen.
- A step-by-step algebraic method[3] which does not involve expansion of the function.

In this article the Karnaugh map method will be described.

Consider the Boolean function:

$$\begin{aligned} f &= \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC + \bar{A}B \\ &= (A+\bar{A})\,BC + (A+\bar{A})\,\bar{B}C + \bar{A}B \\ &= BC + \bar{B}C + \bar{A}B \\ &= (B+\bar{B})\,C + \bar{A}B \\ &= C + \bar{A}B \end{aligned}$$

The original expression has been transformed algebraically into a simpler Boolean function which requires less hardware for implementation. Certainly in the era before the advent of the integrated circuit, minimization of Boolean functions was a positive advantage. In these days of integrated circuits the advantages of Boolean minimisation at the gate level are less obvious and the designer is now thinking in terms of minimizing the number of chips, both from the point of view of economy of space and cost. However, the formal process of simplification does lead the designer to a facility for handling Boolean equations and in that sense it is still useful.

The simplest and most widely used method of simplification employs a mapping technique. Maps for two, three, four and five variables are shown in Fig. 4, and are called Karnaugh maps.

For the two-variable map there are four cells, each of which represent one of the four possible combinations of the two variables. In the top left hand cell of the map A = 0 and B = 0, that is, the cell represents the minterm $m_0 = \bar{A}\bar{B}$, where a minterm may be defined as a Boolean product which contains all the variables in their true or inverted form. The decimal number in a cell is the decimal equivalent of the binary representation of the minterm associated with that particular cell. For example, the minterm associated with the top right hand cell of the two variable Karnaugh map is $\bar{A}B$ and its binary representation is 01 which has a decimal equivalent of 1.

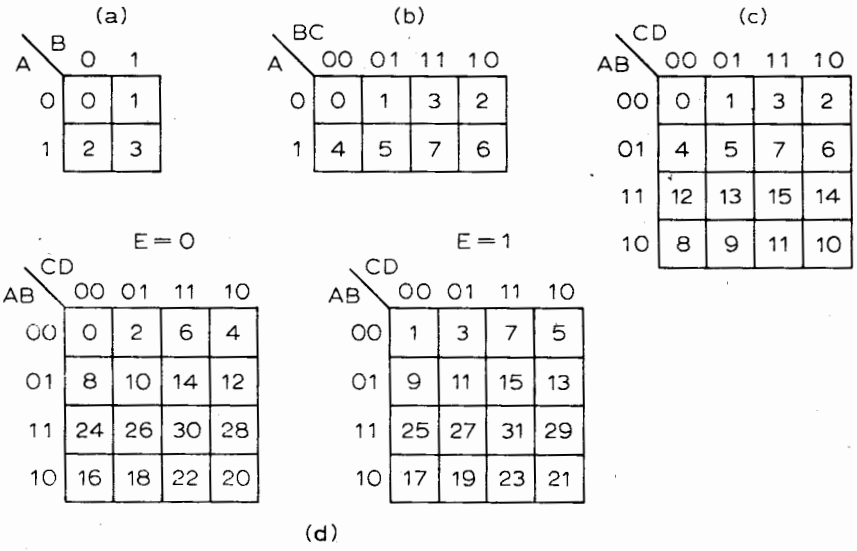


Fig. 4. Karnaugh maps for two(a), three(b), four (c) variables. In the case of five variables, two maps are needed, as shown at (d).

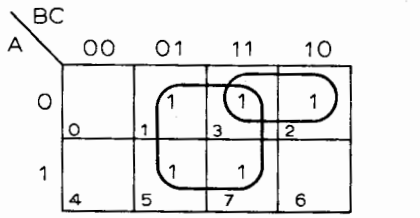


Fig. 5. The Karnaugh map for $f = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC + \bar{A}B$. The ringed '1's show that the expression can be minimized to $f = C + \bar{A}B$.

Any Boolean function of a given number of variables can be plotted on a Karnaugh map. For example, consider again the function:

$f = \bar{A}\bar{B}C + \bar{A}BC + A\bar{B}C + ABC + \bar{A}B$

The first term in the expression $\bar{A}\bar{B}C$ has a binary representation of 001 and the cell corresponding to 001 on the map shown in Fig. 5 is marked with a 1. It follows that the terms $\bar{A}BC$, $A\bar{B}C$, and ABC can be plotted on the map using the same method. The remaining term $\bar{A}B = \bar{A}B(C + \bar{C}) = \bar{A}BC + \bar{A}B\bar{C}$ and the binary representation of these two terms is 011 and 010 respectively, corresponding to cells 2 and 3, but cell 3 has already been covered by the term $\bar{A}BC$ and it is only necessary to enter a 1 in cell 2 to complete the plot of the function.

The above example has shown that a 3-variable term occupies one cell only on a 3-variable map, a two variable term occupies two adjacent cells on the map and a single variable term will occupy four adjacent squares on the map. For example, the term A would be plotted in the cells marked 4, 5, 7 and 6 on the map and these four adjacent squares represent that term.

The process of simplification therefore reduces to the process of identifying plotted adjacencies on the Karnaugh map and then looping these adjacencies as shown in Fig. 5. The four-cell adjacency represents the term C and the two cell adjacency represents the term $\bar{A}B$ and the minimal function is

$f = C + \bar{A}B$

as was previously determined by algebraic methods.

Clearly to get the minimal form of the function the largest possible adjacencies should be chosen.

Example Minimize the Boolean function:

$f = BD + \bar{A}\bar{B}C + A\bar{B}\bar{C} + \bar{A}C\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + AB\bar{C}\bar{D}.$

The function is shown plotted on the Karnaugh map in Fig. 6 and the adjacencies giving the minimal function are shown looped.

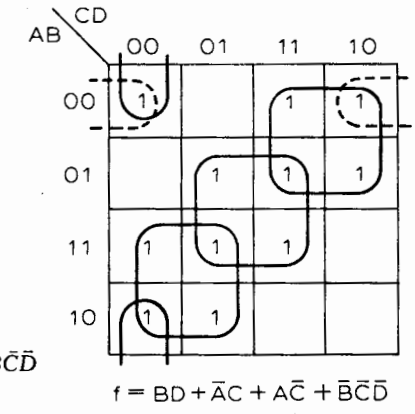


Fig. 6. Minimal function of $f = BD + \bar{A}\bar{B}C + A\bar{B}\bar{C}ABC + \bar{A}C\bar{D} + \bar{A}\bar{B}\bar{C}\bar{D} + AB\bar{C}\bar{D}$ is shown to be $f = BD + \bar{A}C + A\bar{C} + \bar{B}\bar{C}\bar{D}$.

From the map

$$f = BD + \bar{A}C + A\bar{C} + \bar{B}\bar{C}\bar{D}$$

or $$f = BD + \bar{A}C + A\bar{C} + \bar{A}\bar{B}\bar{D}$$

Example Minimize the Boolean function shown plotted in Fig. 7.

For five-variable functions two maps are required as shown in Fig. 7 and the minimisation process can then be carried out in two steps:

Step (1): Minimize the functions plotted in the E=0 and E=1 maps as if dealing with two separate four-variable problems.

This gives $f_1 = \bar{B}\bar{D}\bar{E} + ABD\bar{E} + BCD\bar{E}$

and $f_2 = BDE + A\bar{B}\bar{D}E + A\bar{C}\bar{D}E$

Note that in this case there are two equally valid minimal solutions for the E=1 map, one of which has been chosen.

Step 2: Look for combinations between cells on the E=0 and E=1 maps which will lead to the elimination of factors from any of the terms in f_1 or f_2.

For example, the term $\bar{B}\bar{D}\bar{E}$ in f_1, may be written as $\bar{B}\bar{D}\bar{E} + A\bar{B}\bar{D}\bar{E}$ and the term $A\bar{B}\bar{D}\bar{E}$ can be combined with the term $A\bar{B}\bar{D}E$ in f_2 to generate the term $A\bar{B}\bar{D}$ thus eliminating the factor E between these two terms. The minimal sum is then given by the logical sum of f_1 and f_2 after all possible combinations have been made between the two maps. This leads to the following minimal solution.

$$f = \bar{B}\bar{D}\bar{E} + BDE + ABD + BCD + A\bar{B}\bar{D} + \bar{A}\bar{C}\bar{D}E$$

Obviously, the process of minimization using maps becomes more complicated as the number of variables in a problem increases. However, the method is readily usable up to six variables.

It was shown earlier in this article in the section on the race-hazard theorem that unwanted transient signals can be averted by the introduction of optional products. For example, for the Boolean function $f = \bar{A}B + AC$ a race-hazard occurs, following changes in A, when B=C=1, and it is eliminated by introducing the optional product BC so that the function becomes $f = \bar{A}B + AC + BC$. The original function is shown plotted in Fig. 8(a) and the new function including the optional product is plotted in Fig. 8(b).

Before the introduction of the optional product the Boolean function was irredundant in that it contained no loops, when plotted on Fig. 8(a), that are already covered by other loops. The function was also minimal. However with the introduction of the optional product a loop BC is formed which is already covered by the loops for $\bar{A}B$ and AC. The function is now no longer minimal in that it contains a redundant product BC. This example shows that the introduction of redundancy into a Boolean function is necessary to eliminate race hazards and that the minimal solution is not always the best solution.

Clearly the possibility of a race-hazard occurring can easily be spotted on a Karnaugh map plot of the Boolean function to be minimized.

The minimal form of the function shown plotted in Fig. 9(a) is $f = \bar{A}D + A\bar{B}C + ABC$ but race-hazards will occur at the places indicated by arrowheads on the map. To eliminate these race-hazards two extra loops should be added to the Karnaugh map plot as shown in Fig. 9(b) and the minimum hazard-free function becomes

$$f = \bar{A}D + A\bar{B}C + ABC + B\bar{C}D + \bar{B}CD$$

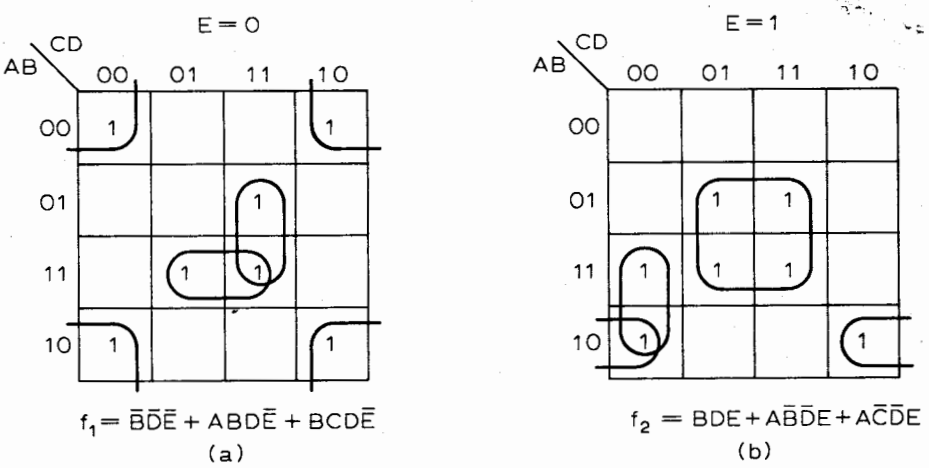


Fig. 7. A further example of minimization.

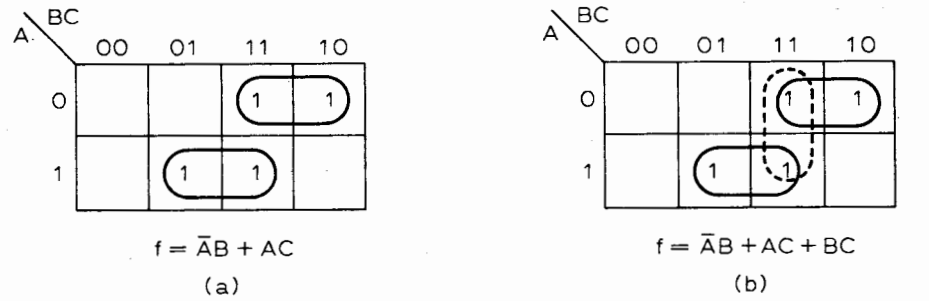


Fig. 8. The use of optional product BC in (b) eliminates the race hazard with changes in A.

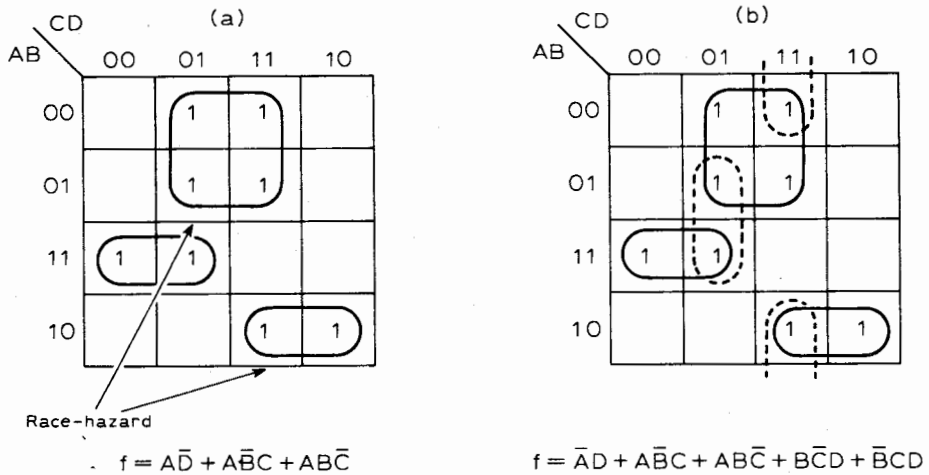


Fig. 9. More elimination of race hazards, shown by arrows in (a) by optional products shown at (b).

References

1. "Problems and Solutions in Logic Design," D. Zissos, Oxford University Press, 1976.
2. "Minimization of Boolean Functions," E. J. McCluskey, *Bell System Technical Journal*, November 1956.
3. "Boolean Minimization," D. Zissos and F. Duncan, *The Computer Journal* vol. 16, No. 2, May 1972.

Logic design — 2

Combinational logic

by B. Holdsworth* and L. Zissos†

†Department of Computing Science, University of Calgary, Canada.
*Chelsea College, University of London

Two of the most essential features that must be met in the design of logic circuits are the imposed gate *fan-in restrictions* and *hazard-free operation*. Gate fan-in is the number of imput terminals provided in a gate, i.e. the maximum number of input signals to a gate. Race-hazards are unwanted transient signals (signal spikes), which under certain changes of an input signal and with certain relationships of circuit delays appear in a logic circuit.

Combinational circuits can be constructed using AND, OR and INVERTER gates, NOR gates or NAND gates. It is possible to construct curcuits using all of the above elements but such circuit configurations are not, at present, common. Circuits composed entirely of NAND or entirely of NOR gates are generally more economical and convenient to use than circuits using AND, OR and INVERTER gates.

The truth table for a two-input NAND gate is shown in Fig. 1(a) and that of a two-input NOR gate in Fig. 1(b). A NAND gate can be used as an INVERTER if all except one of the inputs are tied to logic 1, a practice which, though not always necessary, is strongly advised. For example, if the input A of the gate shown in Fig. 1(a) is tied to logic 1, then the output of the gate is $\bar{B}$ as indicated by the entries in the bottom two rows of the truth table.

Similarly a NOR gate can be used as an INVERTER if all except one of the inputs are tied to logic 0. The remaining input then appears inverted at the output of the gate. In the case of the gate shown in Fig. 1(b), if input A is connected to logic 0 the output of the gate is $\bar{B}$, as indicated by the entries in the top two rows of the truth table.

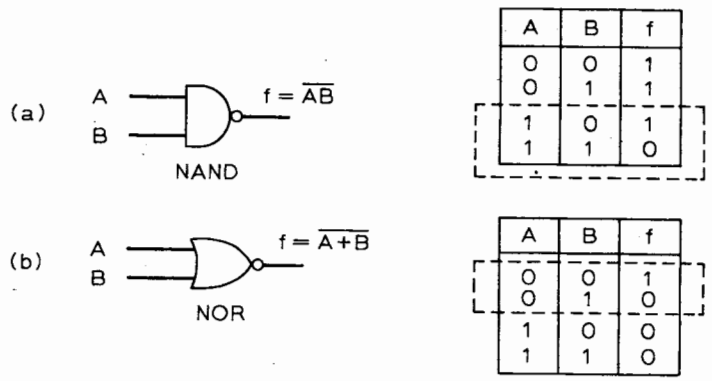
NAND and NOR gates can also be used to generate the OR and AND functions. For example, the output of a NAND gate driven by signals $\bar{A}$ and $\bar{B}$ is $\bar{A}\,\bar{B}$, which may be written as A + B, as shown in Fig. 2(a). The AND function can be generated by connecting two NAND gates in cascade, the first one generating the NAND function of the two input variables A and B, whilst the second gate acts as an INVERTER, as shown in Fig. 2(b). It follows that a NOR gate fed with inverted variables generates the AND function of the true values of the input variables, whilst two NOR gates in cascade generate the OR function of the variables fed to the inputs of the first gate.

Two levels of NAND gates generate a two-level sum-of-products expression, as shown in Fig. 3(a), which indicates the one-to-one relationship that exists between a sum-of-products expression and its NAND implementation. The reader's attention is drawn to the fact that the realisation of a minimal sum-of-products expression does not necessarily result in a minimal circuit. For example, the implementation of the "Exclusive OR" function $f = A\bar{B} + \bar{A}B$, which is a minimal expression, requires five gates, if inverted variables are not available as shown in Fig. 3(b), whereas the NAND circuit satisfying its non-minimal form $f = A(\bar{A} + \bar{B}) + B(\bar{A} + \bar{B})$ requires one gate less, as shown in Fig. 3(c).

In order to implement a function, such as $f = (\bar{A} + BC)E + (\bar{G} + \bar{H})F$ using NAND gates, it is simpler to work backwards from the output gate. The equation is of the form PQ + RS, where

$$P = (\bar{A} + BC) \qquad R = F$$
$$Q = E \qquad S = (\bar{G} + \bar{H})$$

This type of two-level sum-of-products has already been realised in Fig. 3 (a) and is repeated with the relevant input signals in Fig. 4(a). The input line $\bar{G} + \bar{H}$ to gate 3 is the output line of a two-input NAND gate, whose inputs are found by inverting the variables $\bar{G}$ & $\bar{H}$. Similarly, the input line $\bar{A}$ + BC to gate 2 is the output line of a two-input NAND gate, whose inputs are found by inverting the variable $\bar{A}$ and the product BC, as



A	B	f
0	0	1
0	1	1
1	0	1
1	1	0

A	B	f
0	0	1
0	1	0
1	0	0
1	1	0

Fig. 1. Symbols and truth tables for NAND (a) and NOR (b).

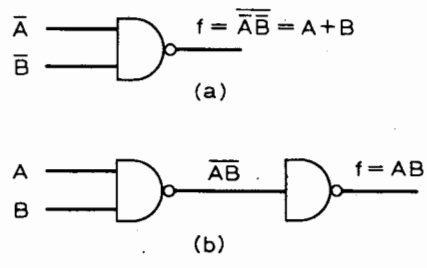


Fig. 2. The OR function using a NAND gate at (a) and the AND using NANDs at (b).

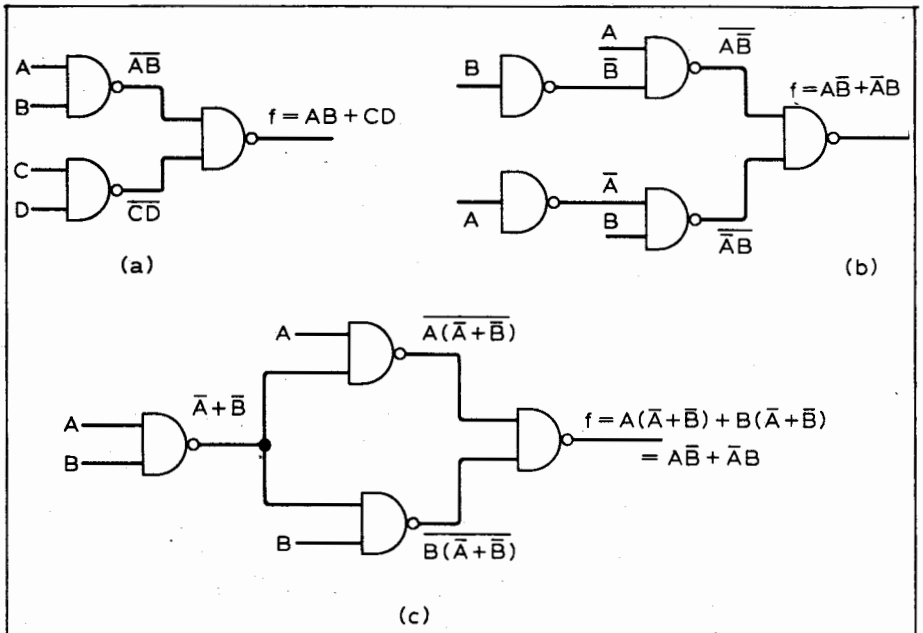


Fig. 3. The use of NAND gates to obtain a sum-of-products function (a). The minimal form of expression need not give a minimal circuit; minimal expression $f = \bar{A}\,\bar{B} + \bar{A}\,B$ in (b) needs one more gate than non-minimal expression $f = A\,(\bar{A} + \bar{B}) + B\,(\bar{A} + \bar{B})$ at (c).

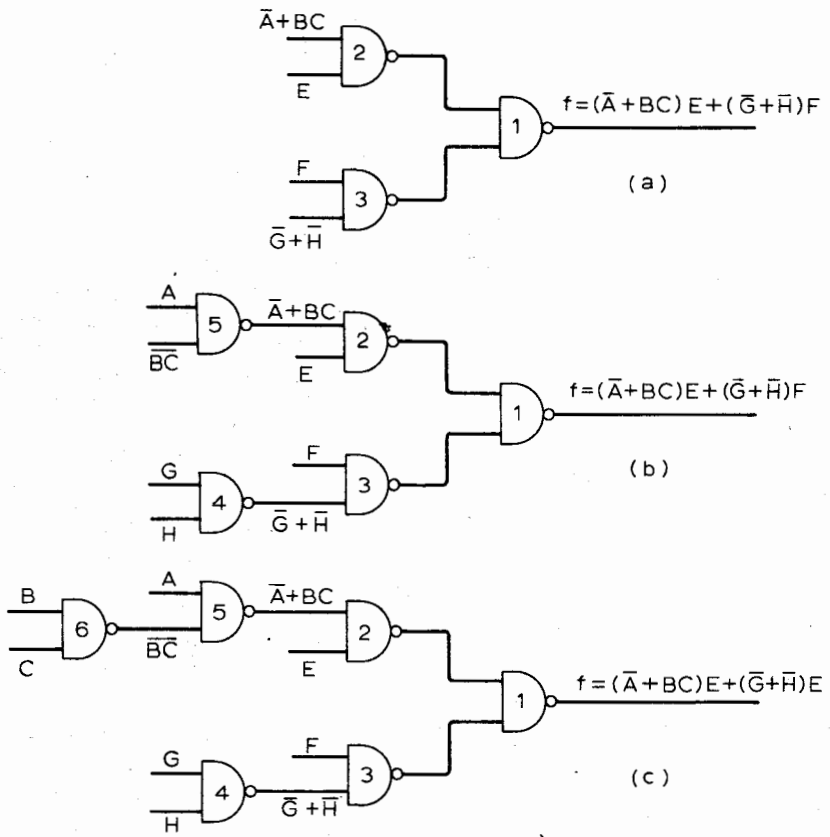


Fig. 4. Building up the expression $f = \bar{A} + B\,C)E + (\bar{G} + \bar{H})F$ from the output end in three levels.

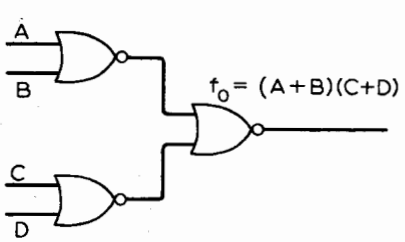


Fig. 5. Dualizing $f = A\,B + C\,D$ with NOR gates.

shown in Fig. 4(b). For the final stage in the implementation it is only necessary to precede gate 5 with a two-input NAND gate whose input variables are B and C as shown in Fig. 4(c).

If the NAND gate in Fig. 3(a) were replaced by NOR gates as shown in Fig. 5 the output function, which the reader can check for himself, will be

$$f_D = (A + B)\,(C + D)$$

which is the dual of the output function of the circuit shown in Fig. 3(a). Hence to implement the NOR circuit of a Boolean function first derive the NAND-circuit of the dual function and replace the NAND gates by NOR gates.

Example. Implement the function

$$f = \bar{A}\bar{B}\bar{C} + ABC.$$

Dualise:

$$f_D = (\bar{A}+\bar{B}+\bar{C})\,(A+B+C)$$

Express in Sum-of-Products form:

$$f_D = \bar{A}B + \bar{A}C + A\bar{B} + \bar{B}C + A\bar{C} + B\bar{C}$$

minimising using the method of Part 1:

$$f_D = \bar{A}B + \bar{B}C + A\bar{C}.$$

The NAND circuit of this function is shown in Fig. 6(a) and the NOR function $f = \bar{A}\bar{B}\bar{C} + ABC$ is given by replacing the NAND gates by NOR gates, as shown in Fig. 6(b).

Hazard-free operation

Race-hazards are unwanted transient signals (signal spikes) which, under certain changes of an input signal and with certain relationships of circuit delays, appear in a logic circuit. The NAND circuit of Fig. 7 shows a combinational logic circuit in which "spikes" are generated during a change of input signal A from 1 to 0 when B = C = 1. The cause of the race-hazard is that immediately following a change in the signal A, $A = \bar{A}$ = either 0 or 1. Hence if a Boolean expression of a signal in a circuit reduces to either $A + \bar{A}$ or $A\bar{A}$, a race-hazard exists at the output of the corresponding gate, otherwise the signal is hazard-free.

In the example shown in Fig. 7, $f = AB + \bar{A}C$ reduces to $A + \bar{A}$ when B = C = 1, revealing the existence of a race-hazard at the output of gate 4. Race-hazards in a circuit can be suppressed by preventing its Boolean expression from reducing to either $A + \bar{A}$ or $A\bar{A}$. This is achieved by the application of the theorem of race-hazards in Part 1. Hence

$$AB + \bar{A}C = AB + \bar{A}C + BC$$

or, alternatively, expressing the same function as a product-of-sums

$$(\bar{A} + B)\,(A + C) = (\bar{A} + B)\,(A + C)\,(B + C).$$

The introduction of the third term prevents the first expression from being reduced to $A + \bar{A}$, since when B = C = 1, $\bar{A}B + AC + BC$ now reduces to $A + \bar{A} + 1 = 1$. Similarly, the second expression, when B = C = 0, reduces to $(\bar{A}+0)\,(A + 0)\,(0 + 0) = \bar{A}.A.0 = 0$.

Fan-in restrictions

The implication of a fan-in restriction (the number of gate inputs) on the realisation of a Boolean function is equivalent to imposing a restriction on the maximum size of the products and sums in the expression of the function to be satisfied. For example the direct realisation of the function $f = AB + A\bar{C} + A\bar{D}$ shown in Fig. 8 requires one three-input NAND gate, three two-input NAND gates and two single-input NAND gates, six gates in all.

If the fan-in restriction is two, implying the use of two-input NAND gates, there are two possible methods of

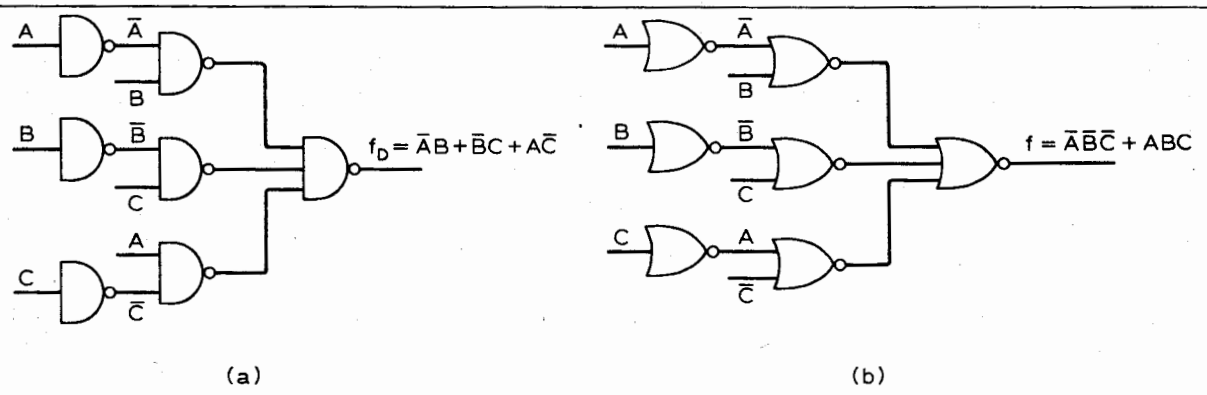


Fig. 6. Generating a function using NOR gates. Function $f = \bar{A}\,\bar{B}\,\bar{C} + A\,B\,C$ is first dualized, minimized and implemented in NAND logic, as at (a). This circuit is then converted to NOR gates to provide the required output.

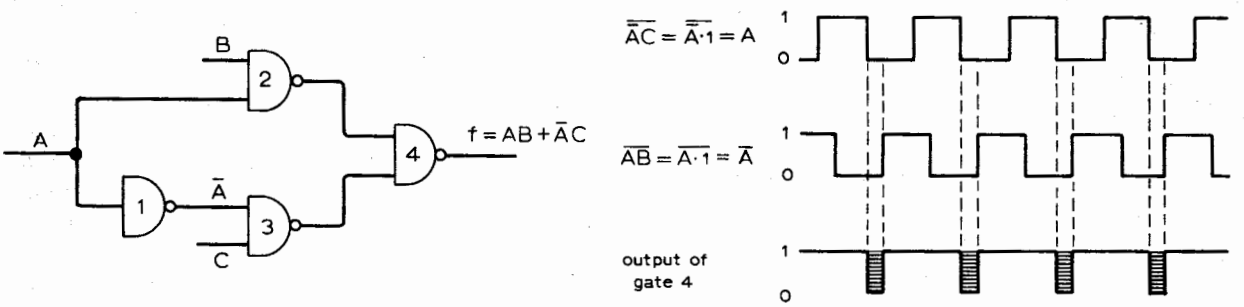


Fig. 7. Mechanism of "spike" generation.

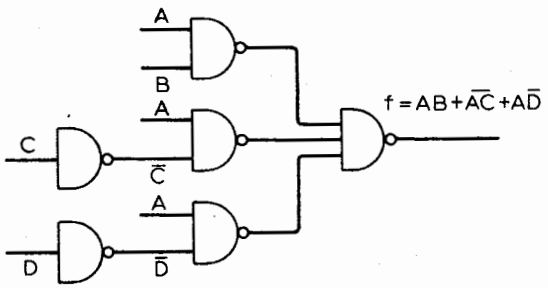


Fig. 8. "Direct" generation of $f = A\,B + A\,\bar{C} + A\,\bar{D}$ when 3-input gates can be unused.

rearranging the given function to satisfy this restriction.

Method 1: bracket two of the three products.

The function is $f = AB + A\bar{C} + A\bar{D}$
bracketing: $f = (AB + A\bar{C}) + A\bar{D}$.
The implementation of this function is shown in Fig. 9(a). It meets the fan-in restriction of two but it requires eight gates, two more than in Fig. 8.

Method 2: remove a common factor:

The function can then be written
$$f = AB + A(\bar{C} + \bar{D}).$$
The realisation of this function is shown in Fig. 9(b). It meets the fan-in restriction of two and requires only four gates, two less than in Fig. 8. Alternatively the function may be written
$$f = A(B + \bar{C}) + A\bar{D}.$$
The implementation of this function is shown in Fig. 9(c). Again it meets the fan-in restriction of two and it requires the same number of gates as realised in Fig. 8. There is one further factorization of interest and that is
$$f = A(B + \bar{D}) + A\bar{C}$$
but this function has the same form as $f = A(B + \bar{C}) + AD$ and can be implemented with six NAND gates, the same number as in Fig. 8. Obviously the optimal implementation is given when the function is written in the form $f = AB + A(\bar{C} + \bar{D})$ even if a fan-in restriction of two had not been imposed.

A systematic method can be used to arrive at an optimal expression for a logic function which to be realised using gates with a specified fan-in. The method described is based on the use of the merging table [1,2].

For the case of NAND circuits the starting point is the irredundant sum-of-products expression of the function to be implemented.
$$f = AB + A\bar{C} + A\bar{D}$$
The function is dualised and the brackets numbered:
$$f_D = (A + B)^1(A + \bar{C})^2(A + \bar{D})^3$$
Next the change in the gate count ΔN, which occurs when pairs of brackets are merged is determined with the aid of the merging table shown in Fig. 10, which has been developed for the case when, there is no increase in the size of the sum ($\Delta Z = 0$) upon merging brackets.

Merging is the process described in the Fan-in theorem in the first article of this series, where two brackets are replaced by a single bracket i.e.
$$(H_1 + T_1)(\bar{H}_1 + T_2) = H_1T_2 + \bar{H}_1T_1$$
It is essential to note that merging does not affect terms which are present in both brackets i.e.
$$(I + X)(I + Y) = I + XY$$
To determine the value of ΔN the components of the two brackets are counted in the following manner.

x= the number of terms in the smaller bracket, excluding common terms.

y= the number of terms in the larger bracket, excluding common terms.

r= the number of terms in the head section of the smaller bracket.

$n = 1$ if a group of terms in one bracket, called the head, is the complement of a group of terms in the other, otherwise $n = 0$.

l= the number of variables true or inverted counted in x and y.

t= the number of true variables in x and y such that for each

(1) its complement does not occur as a variable in any of the other brackets.

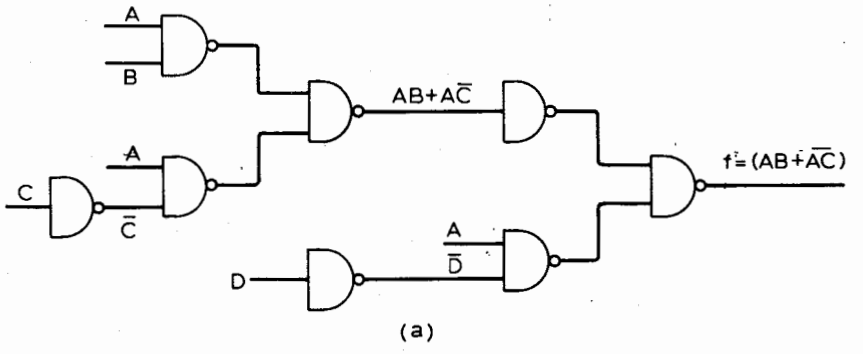
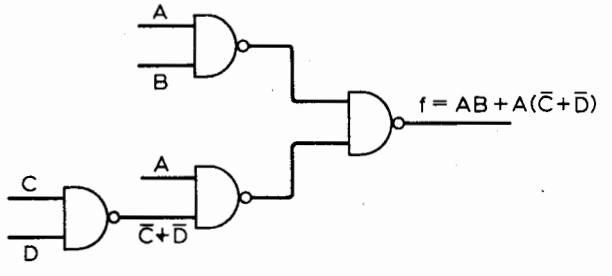
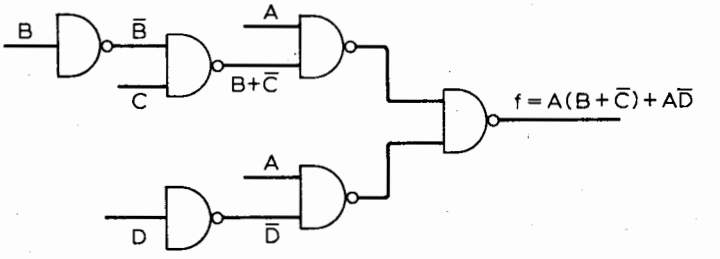


Fig. 9. Bracketing two products in $f = A\,B + A\,\bar{C} + A\,\bar{D}$ enables use of 2-input gates but requires eight instead of six, as in (a). Removing a common factor again meets fan-in restriction to 2 inputs, with varying savings in number of gates, as seen in (b) and (c).





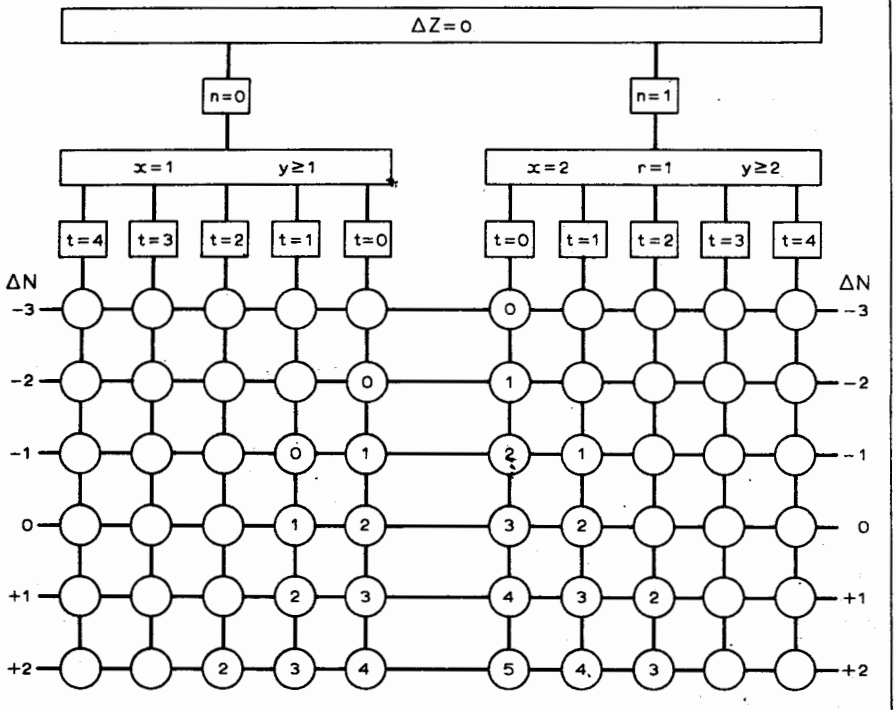


Fig. 10. Merging table for $\Delta Z = 0$.

(2) it does not occur in its true form in a product within the expression.

i= the number of inverted variables such that for each

(1) it is not repeated in the expression as an inverted variable

(2) it does not occur in its true form in a product within the expression.

N is the gate count and ΔN is the change in the value of N caused by merging two brackets.

The quantities detailed above are tabulated below for each bracket pair of the dual function, ΔN being obtained from the table of Fig. 10.

$$f_D = (A+B)^1(A+\bar{C})^2(A+\bar{D})^3$$

b/p	n	x	y	r	t	l	i	l-i	ΔN
1/2	0	1	1	–	1	2	1	1	0
1/3	0	1	1	–	1	2	1	1	0
2/3	0	1	1	–	0	2	2	0	−2

The above tabulation shows that merging brackets 1 and 2 or brackets 1 and 3 does not result in a change in the gate count but that merging brackets 2 and 3 gives a reduction in the gate count by 2, which is the same result obtained working directly with the circuits in Fig. 9.

Merging 1/2 gives $f_D = (A+B\bar{C})\,(A+\bar{D})$
redualising: $f = A(B+\bar{C})+A\bar{D}$
see Fig. 9 (c)
Merging 1/3 gives $f_D = (A+BD)\,(A+\bar{C})$
redualising: $f = A(B+\bar{D}) + A\bar{C}$
Merging 2/3 gives $f_D = (A+\bar{C}\bar{D})\,(A+B)$
redualising: $f = A(\bar{C}+\bar{D})+AB$
see Fig. 9(b).

This part will be concluded with two examples, the first one demonstrating the process of minimal design using the merging table and the second one demonstrating the development of a minimal, hazard-free design.

Example 1 Design a minimal two-input NAND circuit to realise the following Boolean function.

$$f = AB+\bar{A}\bar{C}+C\bar{D}$$

This equation is already in its minimal form.

Dualise: $f_D = (A+B)^1(\bar{A}+\bar{C})^2(C+\bar{D})^3$

Attempt merging:

b/p	n	x	y	r	t	l-i	ΔN
1/2	1	2	2	1	2	4−2=2	+1
1/3	cannot be merged						
2/3	1	2	2	1	1	4−3=1	−1

Merging 2 and 3 will result in a

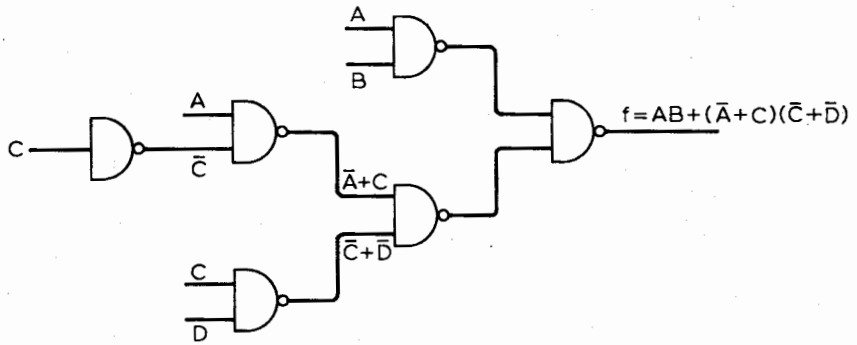


Fig. 11. Minimal circuit for $f = A\,B\,(\bar{A} + C)(\bar{C} + \bar{D})$, using the merging operation.

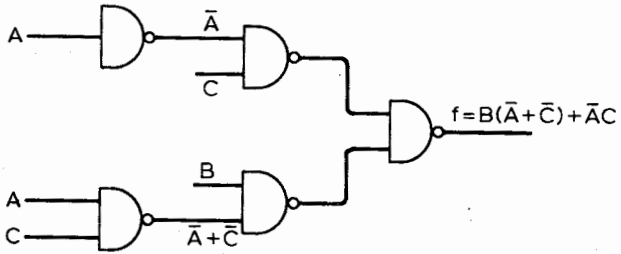


Fig. 12. Two-input NANDS used to realise $f = B\,(\bar{A} + \bar{C}) + \bar{A}C$, which is hazard-free form of $= \bar{A}C + B\bar{C}$.

reduction of the gate count by 1
Merge 2, and 3: $f_D = (A+B)(\bar{A}C+C\bar{D})$.
Redualise: $f = AB + (\bar{A}+C)(\bar{C}+\bar{D})$
Implement as in Fig. 11.

Example 2. Under what circumstances will a spike be generated at the output gate if a direct NAND implementation of the function $f=\bar{A}C+B\bar{C}$ is made? Derive an equivalent hazard-free expression that can be implemented minimally using two-input NAND gates.

If A=0 and B=1 the function $f=\bar{A}C+B\bar{C}$ reduces to $f=A+\bar{A}$ which is the condition for generating a spike when C changes from 1 to 0.

The hazard-free expression is $f = \bar{A}C+B\bar{C}+\bar{A}B$
Dualise: $f_D = (\bar{A}+C)^1(B+\bar{C})^2(\bar{A}+B)^3$

Attempt merging:

b/p	*n*	*x*	*y*	*r*	*t*	*l-i*	ΔN
1/2	1	2	2	1	2	4−1=3	+2
1/3	0	1	1	–	1	2−0=2	+1
2/3	0	1	1	–	0	2−1=1	−1

Merge 2 and 3: $f_D = (B+\bar{A}\bar{C})(\bar{A}+C)$
Redualise: $f = B(\bar{A}+\bar{C}) + \bar{A}C$
Implement, as in Fig. 12.

References

1. Logic Design Algorithms, D.Zissos, Oxford University Press,1972.
2. "Fan-in Restrictions in Logic Circuits," D. Zissos and F. G. Duncan,*Proc. I.E.E.*,Vol. 118, No. 2, Feb. 1971.

Logic design — 3

Event-driven circuits

by B. Holdsworth* and D. Zissos†

A four-step procedure based on the sequential equations, for the design and implementation of event-driven logic circuits is described in this article. Realistic circuit constraints are automatically taken into account by the design process.

The principal factors to be considered in the design of event-driven circuits are:

- Circuit reliability. The circuit must operate correctly and reliably.
- Gate fan-in and fan-out restrictions. These must be observed.
- Speed tolerances. Gate speed tolerances of ± 33⅓ per cent are automatically accommodated by the design process used.

Generally speaking, the solutions obtained do not necessarily use a minimum number of gates, but the design requires minimum effort. The design steps are easy to apply and require no specialist knowledge.

State diagrams

State diagrams can be used to describe both the external and internal operations of event-driven sequential circuits. In a state diagram, states can be represented by squares, rectangles or circles and lines linking the states represent transitions between states. The direction of a transition is indicated by an arrow pointing in the direction of the destination state, and the signal condition that initiates the transition is indicated by its Boolean function inserted either above or below the line. For example the part of a state diagram shown in Fig. 1 indicates that the circuit moves from state S_0 to S_1 when $XY = 1$, i.e. when $X = 1$ and $Y = 0$.

The external and internal-state diagrams of a circuit in which the activation of a switch X in Fig. 2(a) operates, in turn, two lights L_1 and L_2 are shown in Figs. 2(b) and 2c) respectively. Variables X_n and X_{n+1} are used to indicate the n^{th} and $(n+1)^{th}$ activation of the switch. The external-state diagram closely resembles a flow chart, which can be drawn with very little regard to circuit implementation.

There are no hard-and-fast rules for developing internal-state diagrams. Since such diagrams describe the internal operation of a circuit, the designer usually makes arbitrary choices depending on past experience, his understanding of the problem and availability of components, which can lead to different but equivalent results.

The following example is used to illustrate typical variations in the internal-state diagrams of the relatively simple light circuit shown in Fig. 3(a). The function of the circuit is to turn lamp L_1 on when the two switches X and Y are made in that order, and lamp L_2 on when the switches are made in the reverse order. Two different but correct versions of the internal operation of the circuit are shown in Figs. 3(b) and 3 (c).

Most persons attempting this problem would probably derive internal-state diagram 3(b) which uses five internal states. State S_1 is used to record that switch X has been made and state S_3 that switch Y only has been made. In both cases there is no change in the circuit output, although clearly there is a change in the internal-state of the circuit. Very few designers, if any, would arrive at Fig. 3(c) the first time round.

As might be expected the circuit implementation of the state diagram of Fig. 3(c) is the simplest. This state diagram can be obtained by constructing a state table from the state diagram shown in Fig. 3(b), the state table then being reduced by the application of Caldwell's merger procedure. This technique will be described later in this article.

State variables

Each state of an event-driven logic circuit is defined by a unique combination of logic signals called state variables or secondary signals. Clearly one state variable A, defines two states, one by $A = 0$ and the other by $A = 1$. Two state variable define four circuit states each state corresponding to a combination of their values, i.e. 00, 01, 11, and 10. In general, n variables will define 2^n circuit states. As an example of state allocation, the states S_0, S_1, S_2 and S_3 in Fig. 2(c) can be defined by the state variables $AB = 00$, 01, 11, 10. In allocating state variables to states in event-driven circuits it is necessary to ensure that each circuit transition involves a change in the value of a single variable only. The reasons for

* Chelsea College, University of London
† Dept of Computing Science, University of Calgary, Canada

Fig. 1. Elements of a state diagram.

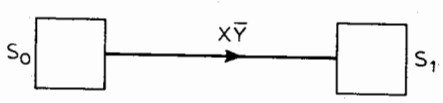
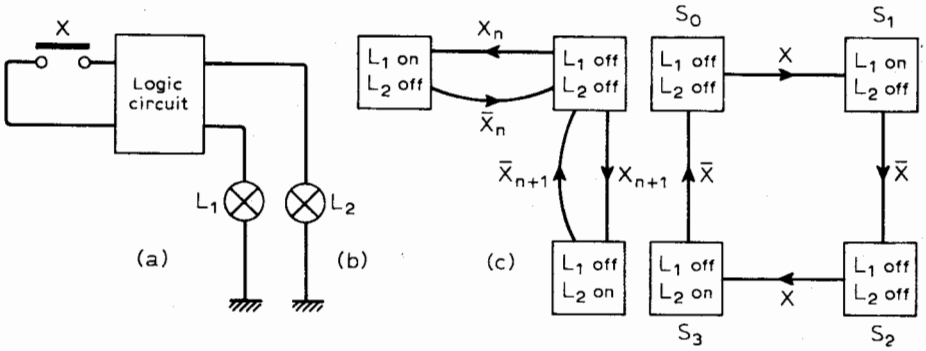


Fig. 2. Internal and external state diagrams of a logic circuit.



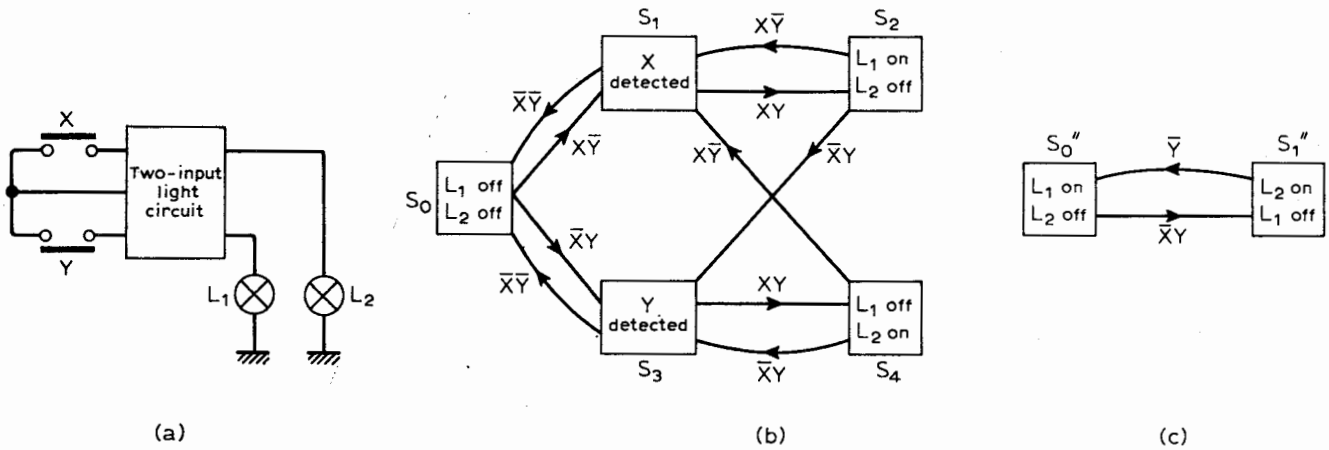


Fig. 3. Internal state diagrams of a two-switch logic circuit.

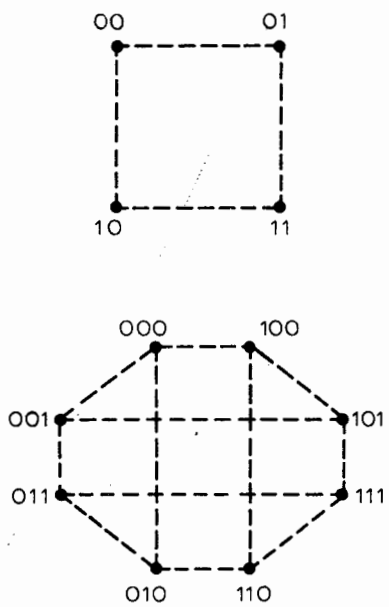


Fig. 4. Race-free diagrams for two and three variables.

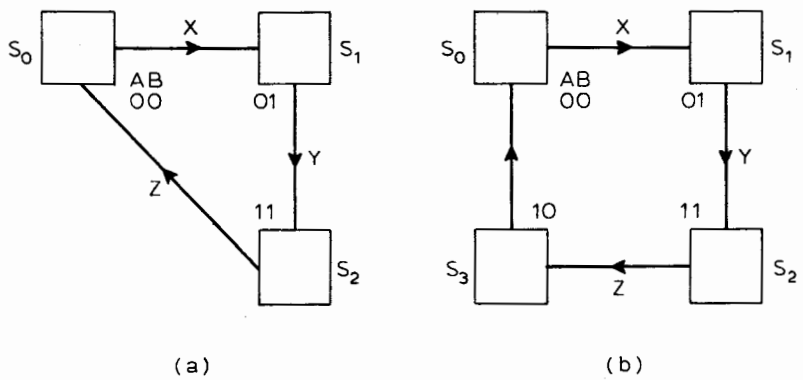


Fig. 5. Use of a dummy state (S_3) avoids simultaneous two-variable change from S_2 to S_0.

doing this is to ensure that races between state variables or secondary signals are automatically avoided.

A race-free assignment of states can be achieved with the aid of a race-free diagram. This is a two-deminsional diagram containing 2^n coded nodes, where all nodes whose codes differ in one variable only are joined by interrupted lines. Hence, races between secondary signals are automatically avoided if each circuit transition lies on a race-free line. Race-free diagrams for two and three variables are shown in Fig. 4.

Dummy states

There are certain patterns of internal-state diagrams that cannot be assigned race-free codes. Such a pattern is shown in Fig. 5(a). If the state codes for S_0, S_1, and S_2 are AB= 00, 01, and 11 respectively the direct transition from state S_2 to S_0 cannot be implemented as this would involve the simultaneous change of two variables. In this case the link between S_2 and S_0 can be turned into a race-free link by interposing a fourth state S_3 between S_2 and S_0, coded A=1 and B=0. This is called a dummy state and replaces line S_2–S_0 with race-free links S_2–S_3 and S_3–S_0, as shown in Fig. 5(b). The S_3–S_0 transition is unconditional and once the circuit assumes state S_3 it moves automatically to S_0. In terms of state variables this ensures that signal B is turned off first and this automatically turns signal A off.

Unused states.

If the number of states to be implemented is N, where $2^{n-1}<N<2^n$, there will be 2^n-N unused or redundant states. For example, in the case of the three-state diagram shown in Fig. 5(a) there will be one unused state. It can never be assumed in practice that a circuit will not move into an unused state either when switching the circuit on or due to the interference of a noise signal. For example, when in state $S_0=00$, a noise signal may turn A on and the circuit enters the unused state $S_3=10$. The circuit may be operating in conjunction with other circuits and moving into state $S_3=10$ may result in the incorrect behaviour of the overall system.

The designer is therefore strongly advised to take such a possibility into account at the design level and take the necessary action. For example, if the misoperation of the above circuit can result in the jamming of a production

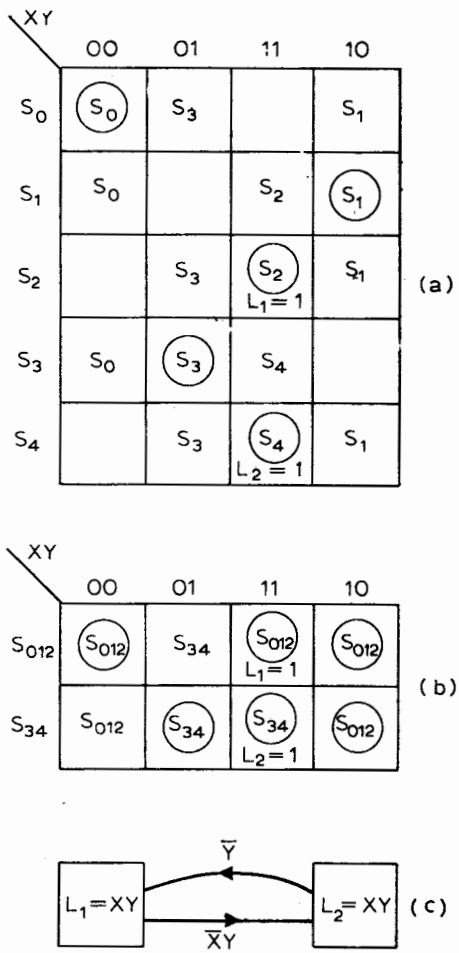
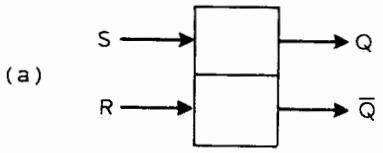


Fig. 6. State-table reduction.



(b)

	S^t	R^t	Q^t	$Q^{t+\delta t}$	
1	0	0	0	0	
2	0	0	1	1	
3	0	1	0	0	
4	0	1	1	0	
5	1	0	0	1	
6	1	0	1	1	
7	1	1	0		Forbidden combinations
8	1	1	1		

Fig. 7. SR flip-flop and its truth table.

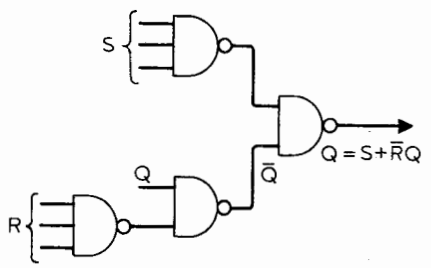


Fig. 8. Implementation of the NAND sequential equation for S and R primary signals.

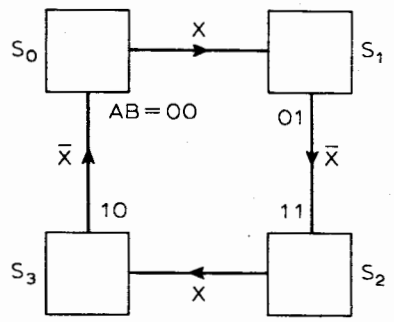
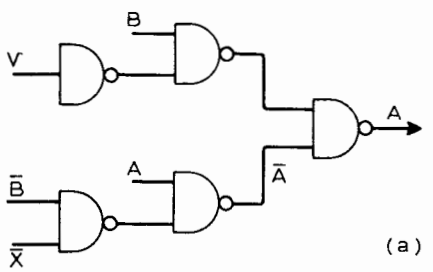
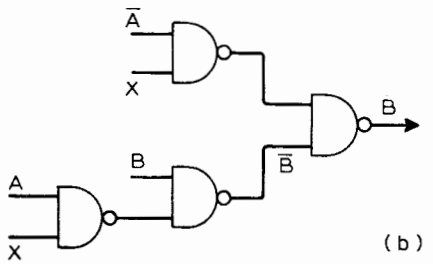
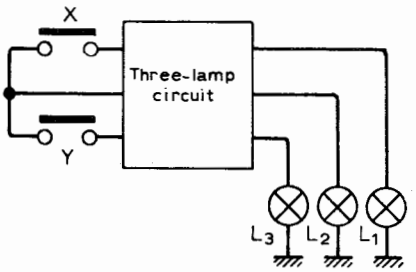


Fig. 9. Determination of turn-on and turn-off sets.







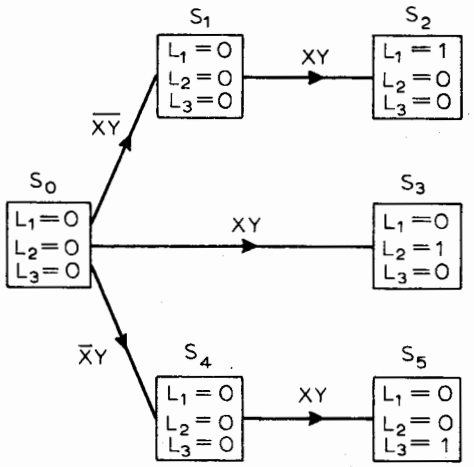


Fig. 11. Three-lamp circuit and its state diagram.

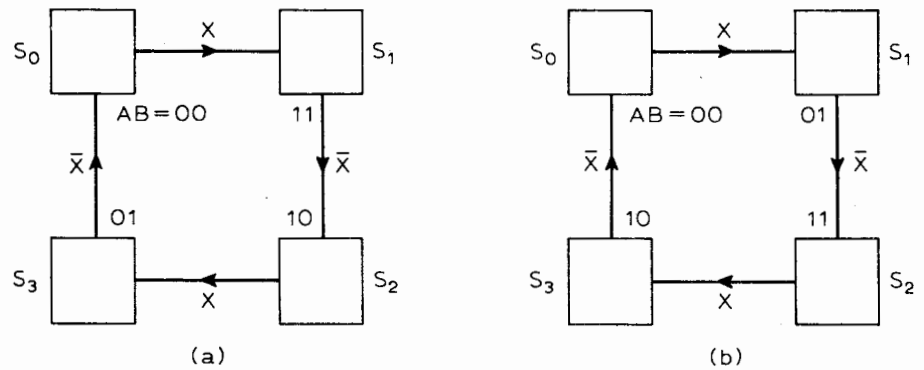


Fig. 12. Elimination of races between secondary signals.

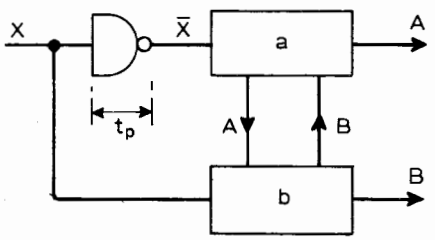


Fig. 13. Races between primary and secondary signals.

line, a possible action would be to use the signal $A\overline{B}$, which defines the unused state, to turn off all machines and raise an alarm.

Frequently, such states are referred to as "don't-care" states. Boolean expressions defining the "don't care" conditions are used as optional products to reduce the circuit equations and hence the complexity of the circuit. This is based on the assumption that a "don't care" condition does not arise in practice, which is only valid for normal operation. Since one cannot exclude the possibility of circuit misoperation, the designer is strongly advised not to leave undefined the response of the circuit under such conditions. In other words the designer "cares" about all circuit conditions.

Summarizing, no state diagram containing other than 2^n states should be implemented. Referring to the light circuit of Fig. 3, only the state diagram in Fig. 3(c) can be implemented. The implementation of the state diagrams in Fig. 3(b) would require the addition of three states. Additionally the reader is strongly advised against the mathematically convenient use of "don't care" states for circuit simplification.

Fig. 10. Implementation of NAND sequential equations for turn-on and turn-off sets obtained from state diagram of, for example, Fig. 9.

State tables

The design restriction of always implementing 2^n states can be met either by introducing dummy states or by reducing the number of internal states. State reduction is carried out by using Caldwell's merging procedure which is based on the state table. Such a table has a row for every state of the circuit and a column for every combination of the input signals. The rows and columns are headed by labels representing the corresponding states and inputs. In each cell the circuit destination is entered, i.e. the next state assumed by the circuit when it is in a state corresponding to the row heading, and it receives input signals defined by the column heading. If the designer does not wish to specify the next state the entry in the appropriate cell is left blank. A second entry is made in each cell which specifies the circuit output,

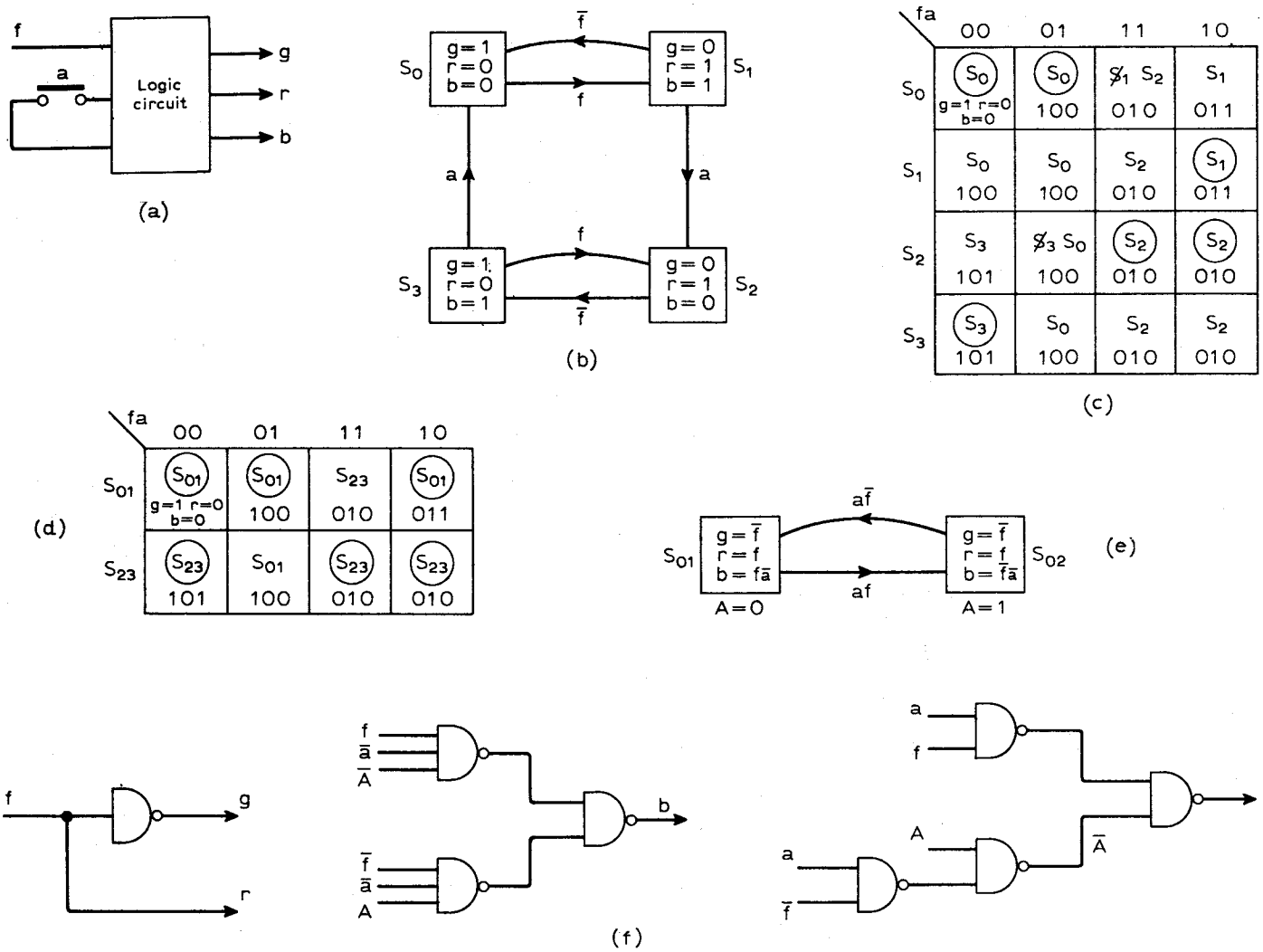


Fig. 14. Function to be realized in Example 1 is at (a) and its state diagram is at (b), while the state table is shown in (c) and in merged form at (d). Initial state diagram based on (d) is shown at (e) and realization of the circuit is (f).

unless it is a blank cell. If the circuit destination is the same as its current state, the circuit is stable and the entry is encircled. The state table corresponding to Fig. 3(b) is shown in Fig. 6(a).

State reduction

The process of combining the rows of a state table is made in accordance with Caldwell's merging rules.

Two rows may be merged if the state numbers and the circuit outputs appearing in corresponding columns of each row are alike, or if the entry in one or both of the rows is blank.

When circled and uncircled entries of the same state number are to be combined, the resulting entry is circled. Thus the two rows

3 ⑤
3 5 6 ⑧

combine into 3 ⑤ 6 ⑧

A change of state from 5 to 8 now involves a change of the input state only.

When a row S_m is combined with row S_n the new row is marked S_{mn}.

Examination of Fig. 6(a) indicates that rows S_0, S_1, and S_2 can be merged to give a new row S_{012} and also that rows S_3 and S_4 can be merged to give a new row S_{34}. The reduced state table is shown in Fig. 6(b) with its corresponding state diagram in Fig. 6(c), and this is identical to the state diagram of Fig. 3(c).

Sequential equations

The sequential equations, allow a state diagram to be translated directly into a circuit, as a consequence, lead to a much simpler solution of event-driven circuit problems. These equations can be obtained directly from a consideration of the logical behaviour of an SR flip-flop.

The SR flip-flop is shown symbolically in Fig. 7(a), the set and reset inputs being labelled S and R respectively, whilst the complementary outputs are labelled Q and $\bar{Q}$. The truth table for the flip-flop is shown in Fig. 7(b).

In the first three columns of this table, all combinations of the present states of S, R, and Q, i.e. their states at time t, are tabulated. In the fourth column the next state of the flip-flop, i.e. its state at time $t + \delta t$, is tabulated.

Examination of this table shows that a change of flip-flop state occurs in rows 4 and 5 only. In row 4 the flip-flop is being reset, i.e. its state is being changed from 1 to 0, by the application of inputs S=0 and R=1. In row 5 the flip-flop is being set, i.e. its state is being changed from 0 to 1, by the application of inputs S=1 and R=0. The reader should also notice that with this type of flip-flop it is inadmissible for S and R both to be logical 1 simultaneously. This restriction can be expressed algebraically as SR=0.

One form of the sequential equations is obtained by taking the logical sum of the combinations in the truth table for which $Q^{t+\delta t}=1$ and adding in the product SR=0. This does not affect the value of $Q^{t+\delta t}$ but leads to a simpler equation for it.

Hence:

$Q^{t+\delta t}=(\bar{S}\bar{R}Q+S\bar{R}\bar{Q}+S\bar{R}Q+SR)^t$.

Minimizing:

$Q^{t+\delta t}=(S+\bar{R}Q)^t$.

The second form of the sequential equations is obtained by excluding the product SR from the equation for $Q^{t+\delta t}$ so that

$Q^{t+\delta t}=(\bar{S}\bar{R}Q+S\bar{R}\bar{Q}+S\bar{R}Q)^t$.

Minimizing:

$Q^{t+\delta t}=[(S+Q)\bar{R}]^t$.

Time is inferred in these equations and they are written

$$Q=S+\bar{R}Q$$

and $$Q=(S+Q)\bar{R}$$

where S is referred to as the turn-on set of Q and R is referred to as the turn-off set of Q.

The most general form of the equations is:

$$Q = \Sigma \text{ turn-on sets of } Q + Q(\Sigma \overline{\text{turn-off sets of } Q})$$

and

$$\overline{Q} = (\Sigma \text{ turn-on sets of } Q + Q)(\Sigma \overline{\text{turn-off sets of } Q})$$

The first of these two equations is used when the design is to be implemented with NAND gates and the second equation when NORs are to be used.

The implementation of the NAND sequential equation, $Q = S + \bar{R}Q$, is shown in Fig. 8. In this circuit S and R, the turn-on and turn-off signals respectively, are the primary signals, whilst Q is the secondary signals which is turned either on or off by the primary signals. When designing an event-driven logic circuit the turn-on and turn-off sets are derived directly from the state diagram; for example, by reference to Fig. 9.

Turn-on set of $A = B\bar{X}$
Turn-off set of $A = \bar{B}\bar{X}$
Turn-on set of $B = \bar{A}X$
Turn-off set of $B = AX$

Substituting these values in the NAND equations

$A = B\bar{X} + A(B + X)$
$B = \bar{A}X + B(\bar{A} + \bar{X})$

and the implementation of these equations is shown in Fig. 10.

Causes of misoperation

Circuit misoperation is said to occur when a circuit assumes an internal state other than the one intended. For example, if on leaving state S_1 in Fig. 11, with $X = 1$ and $Y = 1$ it assumes a state other than S_2, circuit misoperation occurs. Excluding component failure, the causes of circuit misoperation in event-driven circuits are races between primary signals, secondary signals or both. The above three causes will be examined in turn and solutions suggested in the next article.