Binary Decision Diagrams

SHELDON B. AKERS, FELLOW, IEEE

Abstract—**This paper describes a method for defining, analyzing, testing, and implementing large digital functions by means of a *binary decision diagram*. This diagram provides a complete, concise, "implementation-free" description of the digital functions involved. Methods are described for deriving these diagrams and examples are given for a number of basic combinational and sequential devices. Techniques are then outlined for using the diagrams to analyze the functions involved, for test generation, and for obtaining various implementations. It is shown that the diagrams are especially suited for processing by a computer. Finally, methods are described for introducing inversion and for directly "interconnecting" diagrams to define still larger functions. An example of the carry look-ahead adder is included.**

Index Terms—**Binary decision diagrams, digital functions, logical analysis, logic diagrams, logic synthesis, test generation.**

I. Introduction

With the ever increasing complexity of digital functions and systems, the researcher who is charged with their analysis, testing, and implementation is faced with a very real "description" dilemma. On the one hand, he has at his disposal a variety of sophisticated design languages which can provide concise *functional* descriptions of the device or system with which he is concerned. However, when he attempts to use such descriptions in any sort of formal analysis procedure, he typically discovers that their very conciseness virtually precludes any detailed logical investigation. On the other hand, when he turns to those descriptions which are amenable to extensive analysis, he finds that these take the form of truth tables, Boolean equations, Karnaugh maps, etc.—all of which have the unpleasant property of growing *exponentially* with the number of variables involved. What he would like to have would be a concise, "implementation-free" description which could still yield meaningful results about the logical structure and testing requirements of the function involved. This paper will explore one possible approach to bridging this "description gap."

The general idea will be to define a digital function in terms of a "diagram" which tells the user how to determine the output value of the function by examining the values of the inputs. We shall begin by describing these diagrams and showing how they may be derived for various digital devices. Techniques will then be described for using the diagrams to analyze the functions involved, for test generation, and for obtaining actual implementations. Finally, methods will be described for directly "interconnecting" diagrams to define larger digital functions.

Manuscript received July 29, 1977; revised February 28, 1978.

The author is with the Electronics Laboratory, General Electric, Syracuse, NY 13221.

II. Binary Decision Diagrams

Consider the switching function,

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

and assume we are interested in defining a procedure for determining the binary value of f given the binary values of A, B, and C. One way to do this would be to begin by looking at the value of A. If $A = 1$, then $f = 1$ and we are finished. If $A = 0$, we look at B. If $B = 1$, then $f = 0$ and again we are finished. Otherwise, we look at C and its value will be the value of f.

Fig. 1 shows a simple diagram of this procedure. We enter at the node indicated by the arrow and then simply proceed downward through the diagram, noting at each node the value of its variable and then taking the indicated branch. When a 0 or 1 value is reached, this gives the value of f and the process ends.

Fig. 2 shows similar diagrams for some simple AND, OR, and EXCLUSIVE-OR functions. In each case, the reader should have little difficulty in confirming that the diagram does indeed describe a procedure for finding the value of the indicated function. We shall refer to these diagrams as *binary decision diagrams*.[1]

Before examining some of the properties and uses of these diagrams, let us look at how they may be derived. A number of procedures are possible depending on the form in which the function f is defined. If, for example, we are given nothing more than the truth table for f, then a simple but possibly cumbersome procedure is to construct a diagram such as that shown in Fig. 3 which has a one-to-one correspondence between the 2^n rows of the table and the 2^n paths to the outputs of the diagram. These outputs may then be labeled with the corresponding binary values of f and the required diagram automatically results. With n variables, there will initially be $2^n - 1$ nodes in such a diagram.[2] There are, however, several obvious ways in which this number can be considerably reduced. Consider the diagram in Fig. 4(a)

[1] Strictly speaking, all paths through the diagram terminate at a single 0-node or a single 1-node (as with the EXCLUSIVE-OR diagram). However, it will be convenient to indicate this by simply terminating a branch with the value of the terminal node to which it is directed. It will be understood that all branches are directed downward.

[2] The reader may recognize the diagram of Fig. 3 as a binary decision *tree* [3]. While these diagrams do share many of the same properties as these trees, they differ in the important aspect that a node in a diagram may have a number of branches directed into it (rather than just one).

0018-9340/78/0600-0509$00.75 © 1978 IEEE

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

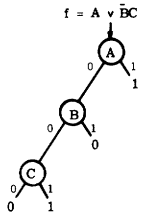


Fig. 1. Diagram for $A \vee \bar{B}C$.

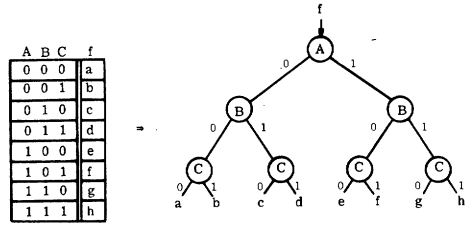


Fig. 3. Deriving a diagram from a truth table.

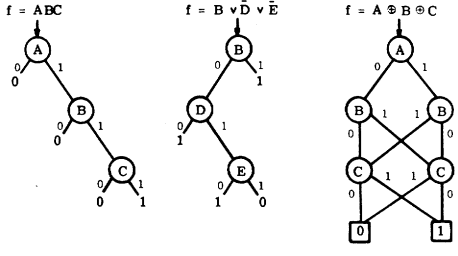


Fig. 2. Some 3-variable diagrams.

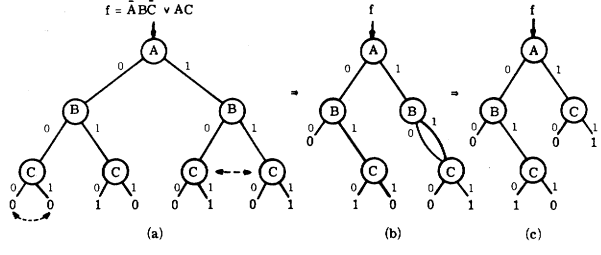


Fig. 4. Simplifying a diagram.

which results from the truth table for $f = \bar{A}B\bar{C} \vee AC$. We note that the value of f obtained at the leftmost C-node is 0 regardless of the value of C. Accordingly, we can remove this node and replace it by 0. Likewise, the two rightmost C-nodes are identical in the sense that they lead to identical output values, so we can combine them into a single node. The result is Fig. 4(b). But now we note that the rightmost B-node is superfluous, since both of its branches go to the same node. Thus, we can remove it to obtain the simplified diagram of Fig. 4(c). (We shall return to this diagram in Section VI to show that even further simplification is possible.)

Normally, switching functions are specified in more compact forms than that afforded by the truth table with its 2^n values. One of the most common is as a Boolean expression. In this case, a "top-down" procedure can be used to derive the diagram by repeated applications of the classical Shannon expansion formula:

$$f(A, B, C, \cdots) = Af(1, B, C, \cdots) \vee \bar{A}f(0, B, C, \cdots).$$

Fig. 5 shows this procedure for the 5-variable function,

$$f = B(\bar{A}C \vee \bar{C}\bar{E}) \vee \bar{E}(\bar{A}B \vee \bar{B}D).$$

We begin by setting $A = 0$ in f to obtain the function f_0 which must be realized below the $A = 0$ branch. We then do the same for $A = 1$ to obtain f_1 as shown in Fig. 5(a). Now the process is repeated for variable B to obtain the four functions in Fig. 5(b). Now we may note that two of the branches lead to the same function $(D\bar{E})$ so these may be directed to the same node. [See Fig. 5(c).] We continue in this fashion—merging identical subfunctions, and then expanding each about one of its remaining variables—until all paths have terminated with a 0 or 1. Clearly, all paths will terminate in at most n steps.

Note that in the process of deriving this diagram we have actually obtained the diagrams for a number of functions. If, for example, we enter the diagram of Fig. 5(d) at the node indicated by f_0, we will exit with the value of f_0. If we enter at the f_1-node, we will exit with the value of f_1, etc. Thus, we can use a single diagram to specify a number of functions depending on the node at which we enter the diagram. If, in fact, we had initially wanted to derive a diagram for the *two* functions, f_0 and f_1, we would have followed the same steps described above, beginning at Fig. 5(b) and ending with the diagram of Fig. 5(d) (with the A-node missing). Thus, this same procedure can be used to derive a diagram for an arbitrary number of functions.

Fig. 6 shows the diagrams for some common combinational functions derived by these procedures.[3] The generalization to larger numbers of variables for functions such as the multiplexers or the majority and parity functions should be obvious. In general, an n-input parallel-to-serial multiplexer will have a diagram with $n - 1$ nodes; the n-output serial-to-parallel multiplexer will have $2(n - 1)$. Fig. 7 shows the diagrams for the next-state equations for several common sequential devices. All of these equations are for edge-triggered devices. While it can be shown that in the worst case the number of nodes in a diagram of n variables can be $0(2^n/n)$, for almost all common digital devices this number grows *linearly with n.*

Before turning to some of the uses of these diagrams, let us look at how they may be stored and processed by a digital computer. Fig. 8 shows the diagram for the eight variable function,

$$f = M(A\bar{B}, C, \bar{D}E \vee DF) \oplus (\bar{G} \vee H)$$

where M denotes the 3-variable majority function.

In order to completely define this diagram in a computer,

[3] The notation and equations used in this figure and in Fig. 7 are taken from [4] where the reader may find formal descriptions of the devices involved. Where a path terminates at a variable, this simply indicates that the value of the function will be the value of that variable. We shall refer to these variables as *exit variables.*

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

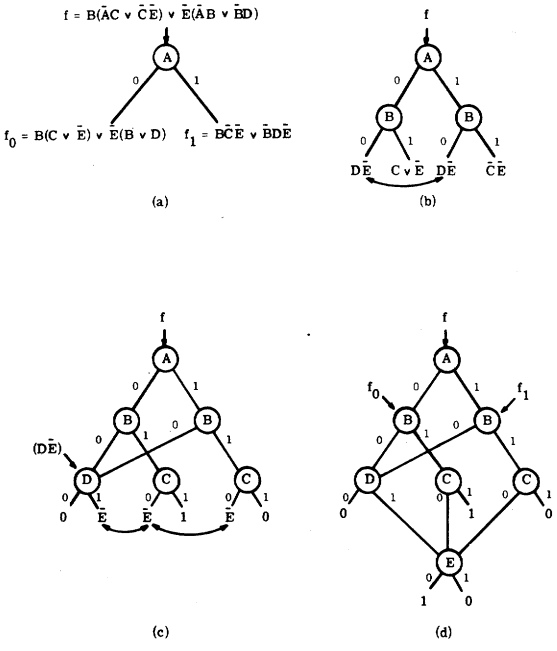


Fig. 5. Deriving a diagram by expansion.

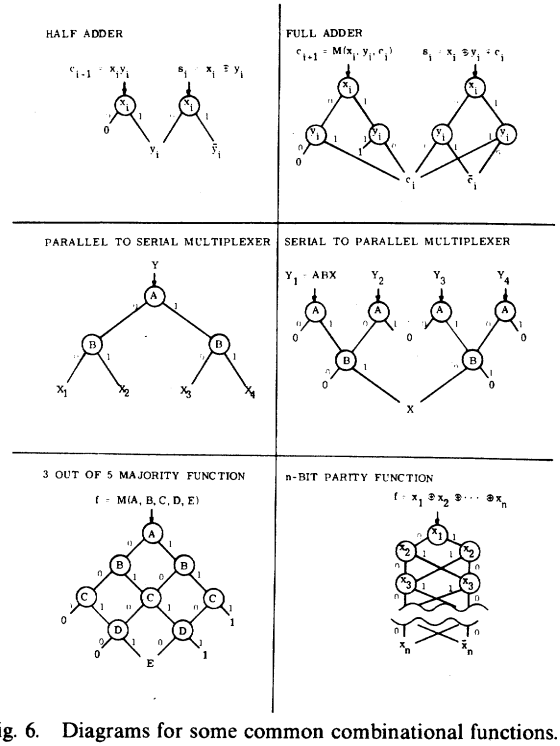


Fig. 6. Diagrams for some common combinational functions.

it is merely necessary to store for each node the binary variable involved plus "pointers" to the two nodes to which its 0 and 1 branches are directed. Thus, if we assign the 9 nodes in Fig. 8 to locations $a, b, c, \cdots, i$ as shown, then the diagram may be completely specified by simply storing an ordered "triple" at each location as shown in the list in Fig. 8. Procedures such as those to be described become quite simple with such lists since they merely involve manipulation of appropriate "pointers."

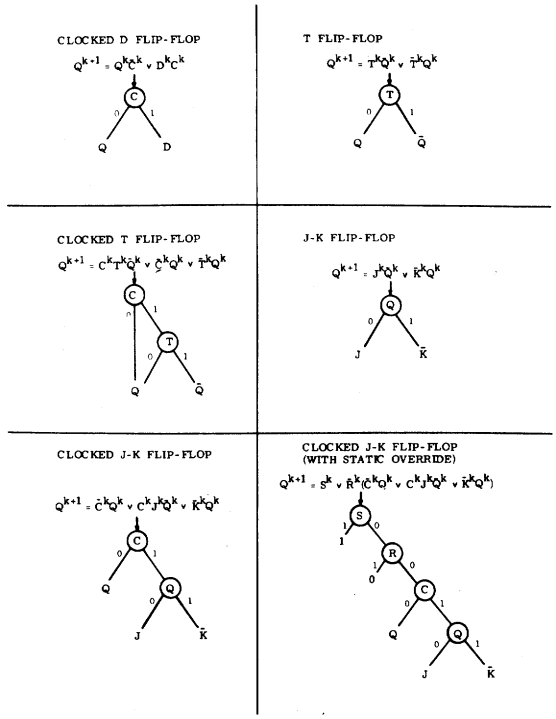
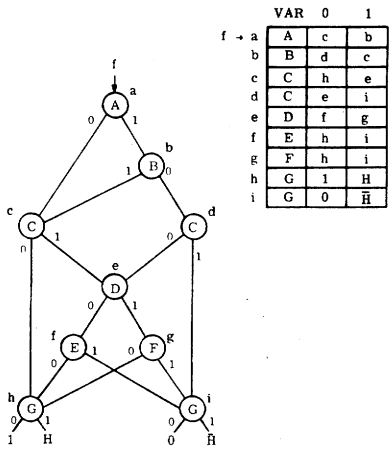


Fig. 7. Diagrams for some common sequential devices.



	VAR	0	1
f → a	A	c	b
b	B	d	c
c	C	h	e
d	C	e	i
e	D	f	g
f	E	h	i
g	F	h	i
h	G	1	H
i	G	0	$\bar{H}$

Fig. 8. An 8-variable diagram with its list of triples.

III. Analyzing a Diagram

Since a diagram is intended to provide a convenient means of finding the output of one or more functions for any given input, let us now examine a diagram in the presence of one such input. Fig. 9 shows the diagram of Fig. 8 in the presence of the 8-bit input 00110100. The darkened branches correspond to these input values and thus indicate the "active" branch out of each node. We shall refer to such a diagram as an *activated* diagram. It is a simple matter to start at the f-node and follow down the path defined by the activated branches to determine that for this input the value of the function would be 0.

Now several interesting facts may be noted. Since each node has two output branches and since one and only one of these is activated for a given input, it follows that for any

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

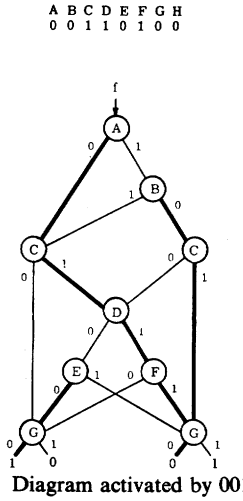


Fig. 9. Diagram activated by 00110100.

input exactly half of the branches in a diagram are activated. Moreover, since each node has one and only one active output branch, it follows that from *every node there is one and only one active path to an output value of 0 or 1*. This is an important property of these activated diagrams which will prove quite useful in what follows.

Note also that while any input induces an active path from each node to an output value Φ it does *not* always follow that for any path from a node to a Φ there exists an input which activates it. This is for the simple reason that the same variable may be involved twice in the path—once with a branch value of 1 and once with 0. On the other hand, any path which does not have this property (i.e., both a $V = 0$ and a $V = 1$ branch) can be activated by simply setting the variables involved to their indicated values. We shall call such a path a *feasible path*. Diagrams which are derived using the truth table or expansion procedures can be shown to automatically have this property, and in others it is a simple matter of path tracing to check for its existence.

Now let us look at how we can use a diagram to determine various logical properties of the function(s) which it represents. It is easily seen that in Fig. 8 there are no paths containing nodes with the same variable and hence that all paths in the diagram are feasible. This means that if we arbitrarily trace out a path from the f-node to (say) a 1 then we have automatically found a set of input variable values for which the function will be 1 *regardless* of the values of the other variables. Assume, for example, that we choose the leftmost path in the diagram of Fig. 8 exiting to 1 at the $G = 0$ branch. We now know that any input with $A = C = G = 0$ will cause f to be 1. In other words, that $\bar{A}\bar{C}\bar{G}$ is an *implicant* of f. From this it follows that if we trace out *all possible* paths from f to a 1, we will automatically generate a sum-of-products form for f.[4] (Conversely, tracing out all paths to 0's will yield a product-of-sums form.)

[4] For the readers who are concerned with methods for finding absolutely minimal 2-level forms for a function, we hasten to point out that the implicants obtained are not necessarily *prime* implicants. However, they are collectively *disjoint* and *essential* in the sense that each minterm is covered by one and only one such implicant.

Unless we have a burning desire to actually see a sum-of-products form for f, tracing out all paths is not a particularly rewarding task since the number of such paths (products) may well become quite large (e.g., 2^{n-1} for the n-bit parity function). What we can do with very little effort is to determine the exact *number* of such paths. This is easily accomplished by a branch labeling procedure. We begin with the f-node and assign each of its branches a 1 indicating that we have found 1 path containing that branch. (See Fig. 10.) We now look for a node having numbers on all of its input branches. (A well-known property of directed acyclic graphs [2] insures that this will always be so.) We add up the numbers on its input branches and assign this number to *both* of its output branches, thus giving the total number of paths from f which pass through each such branch. In this case, we would assign a 1 to each branch out of B. Now C has a 1 on both its input branches so we assign 2 to its output branches. Continuing in this way, we end up with all branches labeled as shown in Fig. 10. Now we have only to add up the numbers on the terminal 1-branches to determine that there exists a sum-of-products form for f with exactly 23 product terms. Likewise, the terminal 0-branch numbers indicate a product-of-sums form with 22 sum terms.

Similar procedures may be used to count the number of literals in these two-level forms or, if desired, to count the exact number of min- and max-terms in the function. Thus, we can very easily obtain meaningful information about the two-level and min- and max-term forms of a large function without resorting to the often horrendous task of actually generating such forms.

IV. Test Generation

Another area in which these diagrams can be particularly useful is that of *test generation*, i.e., finding a set of inputs which can be used to confirm that a given implementation performs correctly. Since the diagram provides only a *functional* description of the device or devices involved, it would certainly be overly optimistic to expect that we can use it to find a set of tests which will automatically provide complete fault coverage for *all* implementations. (In fact, given *any* possible test it is always possible to come up with a contrived implementation in which only that test will detect certain faults.)

On the other hand, if we generate a set of tests which fully "exercises" all of the various nodes and branches of the diagram, then certainly it would not be unreasonable to expect that this same set will likewise be quite useful for testing almost any reasonable implementation. Also, as we shall see in the next section, there exists a variety of implementations in which the diagram tests have a one-to-one correspondence with tests of the implementation. How then can we generate a set of tests for fully "exercising" a given diagram?

For smaller devices, one obvious choice is to generate a set of tests which will collectively activate all 0- and 1-paths through the diagram. (Using procedures such as those described in the previous section, we can initially check to insure that the number of such tests will not be prohibitively

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

large.) These, in turn, will not only constitute test sets for two-level sum-of-products and product-of-sums realizations of the device, but they will also constitute "universal test sets" for various "AND-OR" implementations (see [5] and [6]).

For larger devices where this number of tests becomes unmanageable, a useful alternative is to look for sets of tests which induce "sensitized paths" [7] between the various inputs and outputs of the device. Again, the diagram affords a convenient means for quickly and systematically carrying out such a procedure. It is a simple matter of path tracing, for example, to determine that the input shown in Fig. 9 would serve as a test (detectable at f) for variables C, D, and F s-a-0 and for variable G s-a-1.

A more comprehensive test set can be obtained by postulating various "diagram faults" analogous to the "stuck-at" faults in an actual implementation and then generating tests for discovering each of these faults. (The input of Fig. 9, for example, would detect "s-a-1 faults" on all of the branches along the activated path beginning at the f-node.) Again, such a procedure will insure that the device is systematically put through a variety of different modes of logical operation.

A discussion of precise methods for generating test sets under these various conditions would take us far beyond the scope (and permitted length) of this paper. Let it suffice to say that the reader who has labored through test generation procedures for logic networks will find it a far more simple and straightforward task. In particular, he will find that standard techniques and results in the literature, such as path tracing [3] and the Max Cut/Min Flow theorem [1], can greatly enhance the process. Instead, we shall now look at how actual implementations of a device may be derived from the diagram and thus directly related to the generated tests.

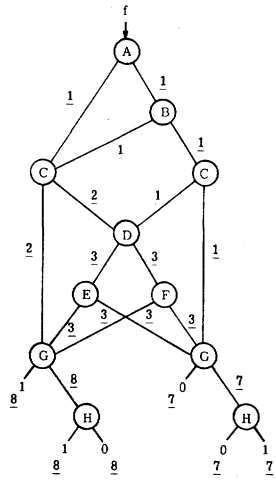


Fig. 10. Counting products and sums.

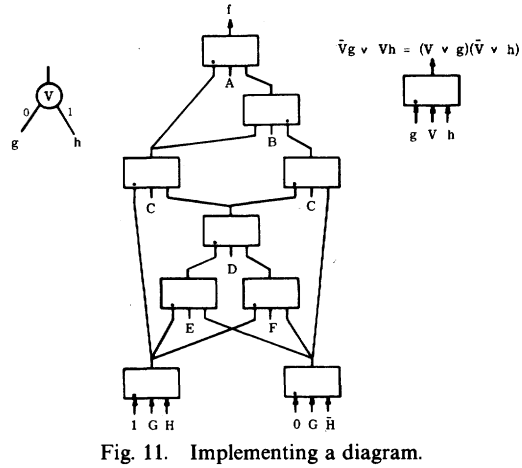


Fig. 11. Implementing a diagram.

V. Implementing a Diagram

Assume we are given the diagram for a digital function f and we wish to construct a logic network which will perform precisely as indicated by the diagram. If the entry node for f has node variable V, and its 0- and 1-branches are directed to entry nodes for functions g and h, respectively, then it follows that the logical value of f will be

$$f = \bar{V}g \vee Vh = (V \vee g)(\bar{V} \vee h).$$

Thus, if g and h are implemented, then f can be implemented by a single "1 out of 2" selector having V, g, and h as inputs. Accordingly, to implement any diagram, we simply replace each node by a "1 out of 2" selector with the resulting inputs and outputs connected just as indicated by the diagram. Fig. 11 illustrates this implementation for the diagram of Fig. 8. (The small dot on each logic element distinguishes the g-input from the h-.) Note that the logic signals now flow *upward* through the resulting implementation.

Clearly, a variety of such structures is possible depending on the technology involved. (Some elements may often be realized more simply than others because constant inputs are involved or because the particular input variable involved is unate.) What is of immediate interest is the fact that there is a one-to-one correspondence between postulated faults in the diagram and the stuck-at faults on the inputs and interconnecting leads in the implementation.

Consider again the input used to activate the diagram in Fig. 9. If this same input is applied to the implementation of Fig. 11, it automatically sets the "1 out of 2" selectors as shown in Fig. 12. Now it can be seen that for every *active* path from a node to an output value ϕ in Fig. 9 there now exists a *sensitized* path in Fig. 12 from that ϕ-value to the output of the logic element at that node. In other words, there is a one-to-one correspondence between the (downward) *active* paths in the diagram and the (upward) *sensitized* paths in the implementation.

In particular, this means that for each of the five *branch* s-a-1 faults which would be detected in the diagram there exists a corresponding *lead* s-a-1 fault which will be detected in the implementation. Likewise, there is a direct correspondence between the four detectable node "stuck-at" faults in the diagram and "stuck-at" faults on the corresponding

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

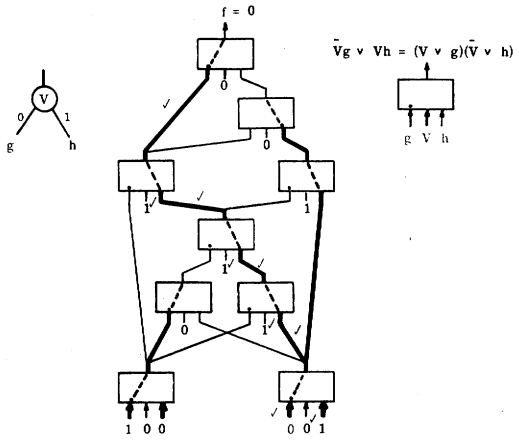


Fig. 12. Testing an implementation.

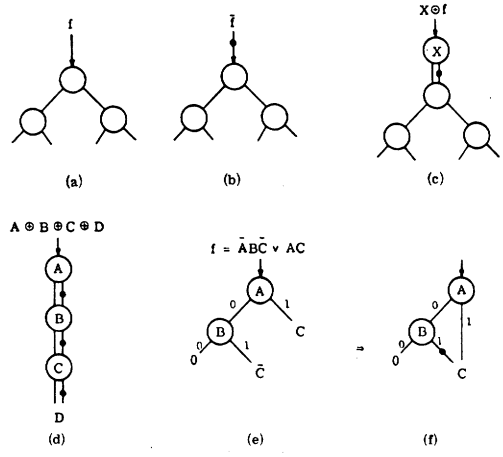


Fig. 13. Diagrams with inverters.

inputs in the implementation. (Checks in Fig. 12 indicate nine lead and input faults which would be tested in this case.)

VI. Inverters and Interconnections

The process of designing large digital systems is typically one of interconnecting smaller devices, such as flip-flops, registers, multiplexers, etc., in such a way that the resulting system has the required performance characteristics. Normally, these devices are interconnected directly to each other with various inverters being inserted as needed. Accordingly, we shall conclude this paper by showing, first, how inversion may be introduced into the diagram process and, second, how various diagrams may then be directly "interconnected." These techniques will then be used to derive the diagram for a 4-bit carry look-ahead adder.

A simple way to introduce inversion into the diagram process is to allow for the insertion of a small dot (or inverter) on the entry branch to a node with the understanding that whenever such a dot is encountered, the user will automatically *complement* the binary value which he ultimately obtains. (See Fig. 13.) During the process of following the branches of an activated diagram, he may encounter a number of such dots, but he need only keep track of whether this number is odd or even in order to decide whether or not to complement the final value.

The use of such inverters can reduce still further the number of nodes required in the diagrams. Fig. 13(c) shows, for example, how the function, $X \oplus f$, may be represented by simply inserting a single node ahead of the diagram for f. (In these and subsequent diagrams, it will be understood that unless otherwise indicated the left and right exit branches from a node correspond to 0- and 1-values, respectively.) Now, the n-bit parity function may be represented by just $n-1$ nodes (Fig. 13(d)] and other diagrams, such as that derived in Fig. 4, may be similarly simplified [Fig. 13(e) and (f)] by inserting appropriate inverters.

In all of the diagrams discussed thus far, we have assumed that the variables within the nodes were primary input variables. However, there is no reason that we cannot permit the node variable to itself be a subfunction (say, g) having its own diagram. In this case, the user who encounters this node would proceed to the diagram for g, determine the value of g, and then *return* to the original node and take the indicated branch. This, of course, complicates the path tracing process but, as we shall see, is more than offset by the ease with which the resulting diagrams may now be manipulated.

To illustrate this process, consider the following equations defining the operation of a full carry adder in terms of a subfunction $(A_i \oplus B_i)$:

$$C_{i+1} = E_i C_i \vee \bar{E}_i A_i$$
$$S_i = E_i \oplus C_i$$

where

$$E_i = A_i \oplus B_i.$$

Fig. 14 shows the diagrams for these three equations (together with the three "triples" which define their operation). Now, if the user wanted the value of C_{i+1} when all the primary inputs were 1, he would enter at the C_{i+1} node where he would be directed to the E_i node. Here, he would traverse the E_i diagram to obtain 0 as the value of E_i. He would then return to the C_{i+1} node and take the 0-branch, exiting with $C_{i+1} = A_i = 1$.

The primary advantage of allowing the node variables to refer to other subfunctions is the ease with which diagrams may then be interconnected. To see this, consider the two devices shown in Fig. 15(a) and (b) having diagrams, D_1 and D_2, respectively.

Assume that we wish to interconnect these two devices as shown in Fig. 15(c). How can we obtain a diagram D_3 which describes the operation of this new device? All that is necessary is to simply replace each variable, V in D_1, by g, the output which drives it. If g is complemented, then V is replaced by $\bar{g}$. Thus, in D_1, we would replace B by g_1, C by $\bar{g}_2$, D by g_3, and E by $\bar{g}_3$.

With *exit* variables (such as D and E in D_1), direct the branches into these variables directly to the corresponding g's, inserting inverters as required. Fig. 15(c) shows the diagram D_3 which results from these operations.

Now let us use these procedures to derive the diagram for a 4-bit adder, i.e., a device having two 4-bit input words, $A_3 - A_2 - A_1 - A_0$ and $B_3 - B_2 - B_1 - B_0$, plus a carry bit, C_{IN}, whose output is their binary sum, $S_4 - S_3 - S_2 -$

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

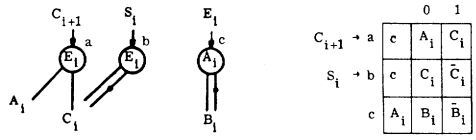
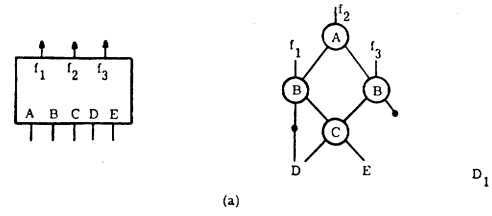
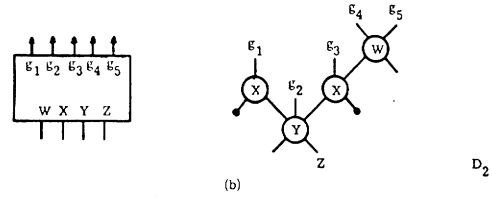


Fig. 14. Full adder diagram.





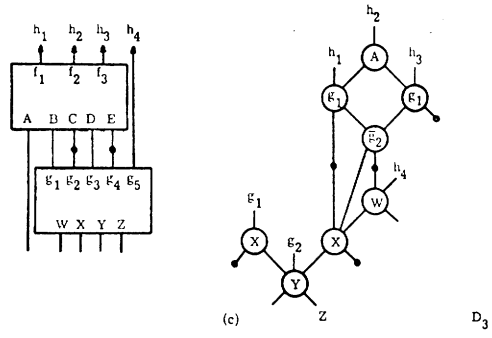


Fig. 15. Interconnecting two diagrams.

$S_1 - S_0$. (S_4 is normally denoted as C_{OUT}.) To do this, we simply make four copies of the full adder of Fig. 14 and then interconnect them as described above. The result is the "ripple-thru" adder of Fig. 16. Note that, in general, an n-bit "ripple-thru" adder requires just $3n$ nodes.

In order to make this device into a carry look-ahead adder, it is necessary to generate two additional functions, G and P, which are used exclusively for generating the carry bit C_{OUT}. The functions, G and P, can be defined as those which result when C_{IN} is set to 0 and 1, respectively, in the function for C_{OUT}. Thus, to obtain the diagram for G, we take the diagram for C_{OUT} in Fig. 16, prune off the S's since they do not affect C_{OUT}, and set $C_{IN} = 0$. Likewise, P is obtained by repeating the process and setting $C_{IN} = 1$. Finally, the diagram for C_{OUT} is obtained by inserting C_{IN} as shown in Fig. 17, and the full diagram for the 4-bit carry look-ahead adder results.[5] (Note that the C_{OUT} node is removed from the ripple-thru adder since it is now generated separately.)

The extension of this diagram to any number of bits

[5] The G and P functions used here are those which result when C_{OUT} is "expanded" about C_{IN}. For a discussion of carry look-ahead logic and other ways of defining G and P, see [8].

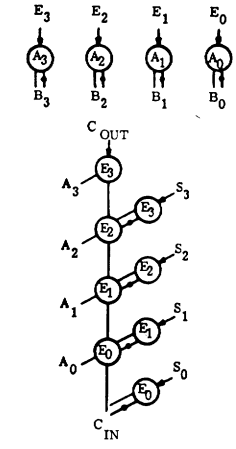


Fig. 16. Diagram for ripple through adder.

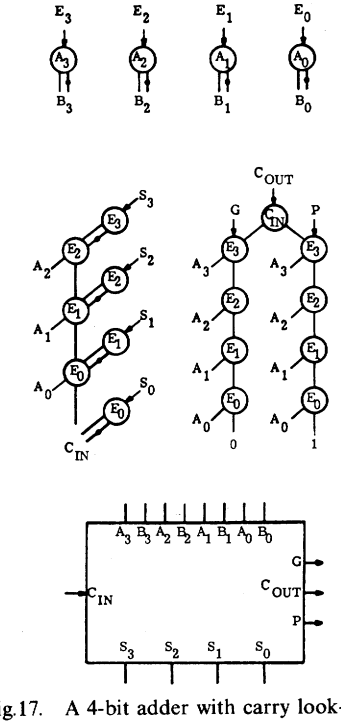


Fig.17. A 4-bit adder with carry look-ahead.

should be obvious. In general, $5n$ nodes will be required. This is in contrast to the truth table for such a device which would require $(n + 3) \cdot 2^{2n+1}$ bits.

VII. Conclusion

Our primary goal in this paper has been to present a "testimonial" to the potential usefulness of binary decision diagrams for defining, analyzing, testing, and implementing large digital functions and systems. With this in mind, we have tried to emphasize the *breadth* of their applicability rather than dwelling on precise methods for exploiting this usefulness. We shall conclude, therefore, with some questions which the interested reader may wish to explore.

How can the various methods for deriving these diagrams be modified and extended so that a minimal (or near minimal) number of nodes results? In the procedures of Section II, for example, we blindly processed the input variables in alphabetical order. Clearly, other orders and

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.

techniques would lead to different and often simpler diagrams.

What are some useful ways in which these diagrams can and should be generalized? One possibility is to allow for "DON'T CARE's" or "DON'T KNOW's" by simply including "X" as a third terminal value.

How can the diagrams be used for various *synthesis* procedures? We have seen (in Section III) that path tracing can yield a many gate, two-level form while a direct substitution method, such as that in Section V, tends to result in few gates but many levels.

REFERENCES

[1] C. Berge, *The Theory of Graphs and Its Applications*. London: Methuen, 1962.
[2] F. Harary, R. Z. Norman, and D. Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs*. New York: Wiley, 1966.
[3] D. E. Knuth, *Fundamental Algorithms, The Art of Computer Programming*, Vol. I. Reading, MA: Addison-Wesley, 1969.
[4] H. T. Nagle, Jr., B. D. Carroll, and J. D. Irwin, *An Introduction to Computer Logic*. Englewood Cliffs, NJ: Prentice-Hall, 1975.
[5] S. B. Akers, "Universal test sets for logic networks," *IEEE Trans. Comput.*, vol. C-22, pp. 835–839, Sept. 1973.
[6] S. M. Reddy, "Complete test sets for logic functions," *IEEE Trans. Comput.*, vol. C-22, pp. 1016–1020, Nov. 1973.
[7] D. B. Armstrong, "On finding a nearly minimal set of fault detection tests for combinational logic nets," *IEEE Trans. Comput.*, vol. EC-15, pp. 66–73, Feb. 1966.
[8] C. Ghest and J. Springer, *Advanced Micro Devices Data Book*, Advanced Micro Devices, Inc., pp. 8-9–8-28, 1974.

Sheldon B. Akers (SM'62–F'75) was born in Washington, DC. He received the B.S. degree in electrical engineering and the M.A. degree in mathematics both from the University of Maryland, College Park, in 1948 and 1952, respectively.

From 1948 to 1956 he was employed in the Washington, DC, area at the National Bureau of Standards, the U.S. Coast Guard Headquarters, and ACF Industries. In 1956 he joined the Electronics Laboratory, General Electric, Syracuse, NY, where he is presently employed as a Staff Computer Scientist. His primary areas of research include switching theory, graph theory, combinatorial analysis, operations research, and design automation. He is a coauthor of *Design Automation of Digital Computers* (Englewood Cliffs, NJ: Prentice-Hall, 1972). He is also an Adjunct Professor at Syracuse University, Syracuse, NY.

Mr. Akers is a member of Pi Delta Epsilon, Omicron Delta Kappa, Sigma Xi, and the Mathematical Association of America. He belongs to the IEEE Computer Society Technical Committee on Design Automation and is Secretary and Publicity Chairman of the Technical Committee on Mathematical Foundations of Computing. He has served in the Computer Society's Distinguished Visitor Program.

Efficiency of Random Compact Testing

JACQUES LOSQ, MEMBER, IEEE

Abstract—**Random compact testing uses random inputs to test digital circuits. Fault detection can be achieved by comparing some statistic of the circuit under test, e.g., the frequency of logic ones at an output, with the value of that statistic previously determined for the fault-free circuit. In this paper, we show that random compact testing can efficiently detect failures in both combinational and sequential circuits. Although this testing method cannot guarantee detection of all faults, it provides a simple way to detect the vast majority of failures in most circuits. The effects of failures inside combinational circuits are modeled in relation to the statistical property measured by the test and a general evaluation of the testing efficiency is obtained. The probability of detection is shown to increase with the test length and to be dependent upon test parameters such as the statistics of the input sequence. For sequential circuits, the uncertainty of the initial state necessitates an initialization step, which is a long sequence of random inputs. The length of such an initialization sequence is circuit dependent, but for most circuits, proper initialization can be achieved in a few seconds. Most failures inside the memory elements are easily detected, even with short tests. Random compact testing can also detect most of the failures inside the excitation logic and the output circuitry. There, as for combinational circuits, its efficiency is largely dependent upon the test length. Some of the requirements and tradeoffs to achieve efficient detection are presented.**

Index Terms—**Combinational digital circuits, compact testing of digital circuits, random testing of digital circuits, sequential digital circuits.**

Manuscript received August 11, 1977; revised February 23, 1978. This work was supported by the National Science Foundation under Grant MCS 76-05327, the Joint Services Electronics Program (JSEP) under Contract N00014-75-0601, and the Air Force Office of Scientific Research under Grant 77-3325.

The author was with the Digital Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305. He is now with the IBM T. J. Watson Research Center, Yorktown Heights, NY10598.

I. INTRODUCTION

THE INCREASING complexity of digital circuits has made the testing problem extremely difficult. Deterministic methods for test generation (*D*-Algorithm [1], [2], Boolean difference [3], Poage's method [4]) become prohibitively expensive for large circuits. The number of multiple stuck-at faults increases exponentially with the number of gates. For large LSI chips, like microprocessors, the amount of computation required to generate a vector test set that

0018-9340/78/0600-0516$00.75 © 1978 IEEE

Authorized licensed use limited to: East China Normal University. Downloaded on April 10,2023 at 07:36:29 UTC from IEEE Xplore. Restrictions apply.