Representation of Switching Circuits by Binary-Decision Programs

By C. Y. LEE

(Manuscript received May 8, 1958)

A binary-decision program is a program consisting of a string of two-address conditional transfer instructions. The paper shows the relationship between switching circuits and binary-decision programs and gives a set of simple rules by which one can transform binary-decision programs to switching circuits. It then shows that, in regard to the computation of switching functions, binary-decision programming representation is superior to the usual Boolean representation.

I. INTRODUCTION

In his 1938 paper,[1] Shannon showed how relay switching circuits can be represented by the language of symbolic logic and designed and manipulated according to the rules of Boolean algebra. This far-reaching step provided an algebraic language for a systematic treatment of switching and logical design problems and provided a root system from which new art can grow and flourish.

We may want to know, however, if there might not be other ways of representing switching functions and circuits, and to compare such representations with the algebraic representation of Shannon. In this paper we will give a new representation of switching circuits, and will call this representation a "binary-decision program."

Binary-decision programs, as the reader will see, are not algebraic in nature. They are, therefore, less easily manipulated. A switching circuit may be simplified not by simplifying its binary-decision program, but by essentially finding for it a better binary-decision program. A good binary-decision program generally means one which is well-knit and makes efficient use of subroutines; it is good in the sense then that a computer program is good. Binary-decision programs do not seek out series-parallel circuits, but are more suited for representing circuits with a large number of transfers. In these respects, binary decision programs therefore differ very greatly from the usual Boolean representation.

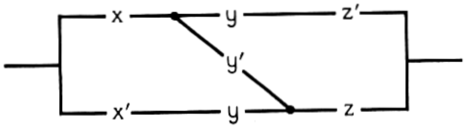

Fig. 1 — Typical switching circuit.

The characteristic which sets binary decision programs still further apart from Boolean representation and gave this study its initial stimulation is in the computation of switching functions. It is here that we will give direct evidence of the superiority of binary-decision programs.

II. STRUCTURE OF BINARY-DECISION PROGRAMS

A binary decision program is based on a single instruction

$$T \quad x; A, B.$$

This instruction says that, if the variable x is 0, take the next instruction from program address A, and if x is 1, take the next instruction from address B. Every binary-decision program is made up of a sequence of instructions of this kind.

Take, for example, the switching circuit shown in Fig. 1. This circuit is described exactly by the following binary-decision program:

1. $T \quad x; 2, 4.$
2. $T \quad y; \theta, 3.$
3. $T \quad z; \theta, I.$
4. $T \quad y; 3, 5.$
5. $T \quad z; I, \theta.$

The program is actually a sequential description of the possible events that may occur. We begin at program address 1 by examining the variable x. If x should be 0, we go to address 2 and examine y. If y is 0, we go to address θ; otherwise we go to address 3, and so forth. The symbols θ and I indicate whether the circuit output is 0 or 1. From a computer viewpoint, they can be the exit addresses once the circuit output value is known.

III. CONSTRUCTION OF SWITCHING CIRCUITS FROM BINARY-DECISION PROGRAMS

The question that we will consider here is this: Suppose the logical requirements of a switching circuit are given, when would it be possible to design the circuit according to the following procedure:

$$\begin{tikzcd}
\begin{array}{l}\text{Logical}\\ \text{requirements}\end{array} \arrow[r] & \begin{array}{l}\text{Binary}\\ \text{decision}\\ \text{program}\end{array} \arrow[r] & \begin{array}{l}\text{Switching}\\ \text{circuit}\end{array} \text{ ?}
\end{tikzcd}$$

In various examples we have tried, this approach has given us a fresher look at things and, in several instances, has given us rather good circuits. The process of going from binary-decision programs to switching circuits is very well defined, so that how good a circuit we get depends entirely upon how good a binary-decision program we can write. Roughly speaking, if a problem has a fairly sizable set of logical requirements to begin with, it would call for a well-organized array of subroutines in the binary-decision program, which are called in as the need arises. Nevertheless, there are many exceptions, and it is very hard to say where ingenuity ends and routine process begins.

Let us now state the rules on how a switching circuit can be constructed from a binary-decision program.

Rule 1. Each address of the binary-decision program corresponds to a node of the circuit.

Rule 2. If at address A the instruction is $T \quad x; B, C$, a variable x' should be connected between nodes A and B and a variable x should be connected between nodes A and C.

Rule 3. The node corresponding to address 1 is the input node. The node corresponding to address I is the output node.

A simple change in Rule 3 will enable us to get the negative of the switching circuit. This is done by making the output node the node corresponding to address θ rather than to address I.

We will illustrate our procedure by two examples.

3.1 *Example 1*

We wish to design a circuit with six switching variables, a,b,c and x,y,z. Let M be the binary number abc and N be the binary number xyz. Then the output is to be 1 whenever $M \geqq N$.

The problem says that two binary numbers M and N are to be compared; these two numbers may be compared one bit at a time. We may compare the most significant bits a and x first. If $a = 0$ and $x = 1$, then $M < N$, so that there is no output. If $a = 1$ and $x = 0$, then $M > N$, and the output will be 1. If $a = 0$ and $x = 0$ or $a = 1$ and $x = 1$, the comparison process must be continued to the next pair of bits.

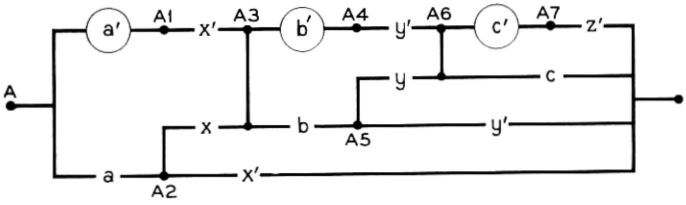

Fig. 2 — Switching circuit for Example 1.

The program proceeds by first comparing the pair of variables a and x. Depending on the values of a and x, b and y are compared next and, lastly, c and z, if necessary. The program instructions are

Address	*Instruction*
A	$T \quad a; A1, A2$
$A1$	$T \quad x; A3, \theta$
$A2$	$T \quad x; I, A3$
$A3$	$T \quad b; A4, A5$
$A4$	$T \quad y; A6, \theta$
$A5$	$T \quad y; I, A6$
$A6$	$T \quad c; A7, I$
$A7$	$T \quad z; I, \theta.$

Following the three rules, we begin with the first instruction and let A correspond to the input node of the circuit. A branch labeled a' leads to the node $A1$ and a branch labeled a leads to the node $A2$. $A1$ and $A2$ thus become internal nodes of the circuit. From $A1$ we need to put down only the branch x' leading to internal node $A3$, since a branch labeled x would give no output. Continuing in this way, we get all the paths from the input to the output; the addresses tell us where the interconnections between these paths are to be made. The circuit for this example is given in Fig. 2. From this circuit it can be seen that the three circled variables are superfluous and can be deleted.

3.2 *Example 2*

We wish to design a switching circuit with eight variables, a,b,c,d and w,x,y,z. Let L be the number of the variables a,b,c,d which are in the 0 state and let R be the number of the variables w,x,y,z which are in the 0 state. Then the output is to be 1 whenever $L \geqq R$.

The problem tells us that, if all of the variables a,b,c,d are 0, the output would be 1 regardless of what w,x,y,z are; if exactly three of the variables a,b,c,d are 0, then the output would be 1 whenever three or fewer of the variables w,x,y,z are 0; and so forth. To program this problem, we will therefore begin with the following subroutines:

S1. The output is to be 1 whenever three or fewer of the variables w,x,y,z are 0.

S2. The output is to be 1 whenever two or fewer of the variables w,x,y,z are 0.

S3. The output is to be 1 whenever one or fewer of the variables w,x,y,z is 0.

S4. The output is to be 1 whenever none of the variables w,x,y,z is 0.

The program is then completed by counting the number L of the variables a,b,c,d which are 0. If L is 4, the output is made 1 directly; if L is 3,2,1 or 0, the program enters subroutine $S1$, $S2$, $S3$ or $S4$ respectively.

We will begin with the subroutine programs. The subroutine $S1$ is a successive scan of the states of the variables w,x,y,z:

Address	*Instruction*
$S1$	$T \quad w; S11, I$
$S11$	$T \quad x; S12, I$
$S12$	$T \quad y; S13, I$
$S13$	$T \quad z; \theta, I.$

The subroutine $S2$ can be written likewise. A moment's reflection will show, however, that $S2$ can make use of a portion of the instructions of $S1$. Similarly, $S3$ can make use of $S2$ and $S4$ can make use of $S3$. The subroutine programs come out to be:

Address	*Instruction*
$S1$	$T \quad w; S11, I$
$S11$	$T \quad x; S12, I$
$S12$	$T \quad y; S13, I$
$S13$	$T \quad z; \theta, I$
$S2$	$T \quad w; S21, S11$
$S21$	$T \quad x; S22, S12$
$S22$	$T \quad y; \theta, S13$
$S3$	$T \quad w; S31, S21$
$S31$	$T \quad x; \theta, S22$
$S4$	$T \quad w: \theta, S31.$

The main program which evaluates L and selects the appropriate subroutine is

Address	*Instruction*
A	$T \quad a; A1, A2$
$A1$	$T \quad b; A3, A4$
$A3$	$T \quad c; A5, A6$
$A5$	$T \quad d; I, S1$
$A6$	$T \quad d; S1, S2$
$A4$	$T \quad c; A6, A7$
$A7$	$T \quad d; S2, S3$
$A2$	$T \quad b; A4, A8$
$A8$	$T \quad c; A7, A9$
$A9$	$T \quad d; S3, S4.$

Following the three rules of construction, the final circuit is given in Fig. 3. To show how the subroutine circuits can be combined in stages, the circuit for $S1$ is given in Fig. 4 and the combined circuit for $S1$ and $S2$ is given in Fig. 5.

Generally, we find that the switching circuits constructed from binary-decision programs have several distinct characteristics. The construction does not distinguish among series-parallel, bridge and nonplanar circuits, but it is restricted to unidirectional flow of current in any

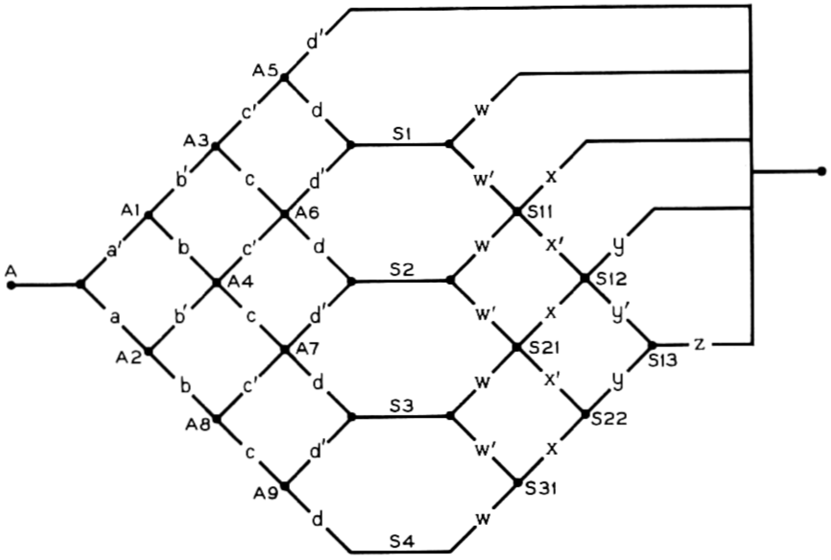

Fig. 3 — Switching circuit for Example 2.

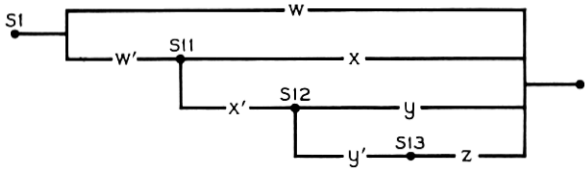

Fig. 4 — Circuit for S1 of Example 2.

branch. Because of the transfer characteristic of the program instruction, the program in most cases gives bridge or nonplanar circuits and very rarely gives series-parallel circuits. Again, because of the transfer characteristic of the instruction, the procedure tends to give circuits having a large number of transfers, causing unnecessary appearance of variables in the circuits. On the other hand, the presence of the transfers prevents sneak paths, which are often a source of worry.

IV. COMPUTATION OF SWITCHING FUNCTION BY BINARY-DECISION PROGRAMS

The problem that we wish to consider here is this: Suppose, in carrying out a complicated task, a complex decision depending on many variables is to be made and made repeatedly. Question: What procedure should one follow so as to arrive at the decision quickly and without having to go through a large amount of computation?

To make the problem more tractable, let us say that the decision function is a switching function of n variables. The problem is to find a good procedure for the computation of this function.

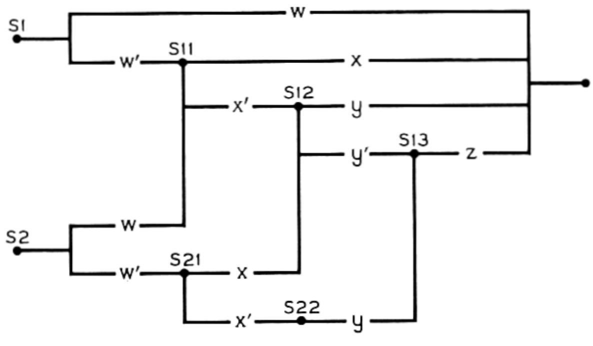

Fig. 5 — Combined circuit for S1 and S2 of Example 2.

The first question one should ask is, perhaps, what are the choices? If the switching function is, say,

$$x(y'z \vee yz') \vee x'yz,$$

what alternatives in computation are there?

There are indeed many alternatives. One may, for instance, carry out the following direct computation:

1. y AND z', store result in location a;
2. y' AND z, store result in location b;
3. (contents of a) OR (contents of b), store result in location a;
4. x AND (contents of a), store result in location a;
5. x' AND y, store result in location b;
6. z AND (contents of b), store result in location b;
7. (contents of a) OR (contents of b), store answer in location b.

Or, one may go back to the switching function itself and rewrite it as

$$(y \vee z)\ [x \oplus (yz)],$$

where $\oplus$ (called SUM) stands for addition modulo 2. A direct computation now becomes:

1. y OR z, store result in location a;
2. y AND z, store result in location b;
3. x SUM (contents of b), store result in location b;
4. (contents of a) AND (contents of b), store answer in location b.

We have done in four steps what took seven above.

Finally, we may write for this switching function a binary-decision program:

1. T x; 2, 4.
2. T y; $\mathit{0}$, 3.
3. T z; $\mathit{0}$, I.
4. T y; 3, 5.
5. T z; I, $\mathit{0}$.

This computation scheme differs from the other two in one major aspect, namely, the number of steps one needs to go through in the binary-decision program before one gets an answer depends upon the initial values of the variables. For example, we will arrive at an answer in two or three steps according as $(xyz) = (001)$ or $(xyz) = (110)$. To compare this program with the second procedure (with AND, OR and SUM), therefore, let us sum over all eight combinations of the three variables. This yields a total of 22 steps for the binary-decision program, and 32 for the AND-OR-SUM procedure. On the other hand, the binary-decision program is longer by one instruction.

The questions that we want to answer here are these:

1. For an arbitrary switching function of n variables what is the order of magnitude of the number of instructions in its binary-decision program?

2. Comparing specifically the binary-decision program approach with the AND-OR-SUM procedure, which will in general need fewer instructions and which will need less time to execute?

Questions of this nature but pertaining to the number of relay contacts or electronic components have been studied by Shannon[2] and Muller.[3] As is the case with their investigations, we are not able to answer these questions for individual functions but our answers apply to an overwhelming fraction of switching functions of n variables.

4.1 *Computation by Binary-Decision Programs.*

Let f be a switching function of n variables, and let $\mu(f)$ be a number such that no binary-decision program representing f has fewer than $\mu(f)$ instructions. We will let μ_n be the smallest number of instructions sufficient to represent any switching function of n variables. That is,

$$\mu_n = \max\{\mu(f) \mid f \in F(n)\},$$

where $F(n)$ is the set of all switching functions of n variables. Then

Lemma 1: $\mu_n > 2^n/2n$.

Proof: Let $N(n,p)$ denote the number of possible binary-decision programs involving n variables with p instructions. Since each instruction can be chosen from at most np^2 instructions, it follows that $N(n,p) \leqq (np^2)^p$ and, in particular, $N(n,\mu_n) \leqq (n\mu_n{}^2)^{\mu_n}$.

Now suppose $\mu_n \leqq 2^n/2n$. Then

$$N(n,\mu_n) \leqq \left(n\,\frac{2^{2n}}{4n^2}\right)^{2^n/2n} = \left(\frac{2^{2n}}{4n}\right)^{2^n/2n} < 2^{2^n},$$

and the lemma follows.

To find an upper bound for μ_n requires an interesting subroutine technique. Let a set of programs each of which computes a switching function be called a *library*. Let $L(n)$ be the library of programs which represent all 2^{2^n} switching functions of n variables. Then

Lemma 2: The library $L(n)$ can be written so that it contains not more than 2^{2^n} instructions.

Proof: For $n = 2$, the 16 functions of two variables are

1. θ.
2. I.
3. x_1.
4. x_2.
5. x_1'.
6. x_2'.
7. x_1x_2.
8. $x_1'x_2$.
9. x_1x_2'.
10. $x_1'x_2'$.
11. $x_1 \vee x_2$.
12. $x_1' \vee x_2$.
13. $x_1 \vee x_2'$.
14. $x_1' \vee x_2'$.
15. $x_1'x_2 \vee x_1x_2'$.
16. $x_1x_2 \vee x_1'x_2'$.

Now $L(2)$ can be the following program:

1. $T \quad x_1 ; \theta, \theta$.
2. $T \quad x_1 ; I, I$.
3. $T \quad x_1 ; \theta, I$.
4. $T \quad x_2 ; \theta, I$.
5. $T \quad x_1 ; I, \theta$.
6. $T \quad x_2 ; I, \theta$.
7. $T \quad x_1 ; \theta, 4$.
8. $T \quad x_1 ; 4, \theta$.
9. $T \quad x_1 ; \theta, 6$.
10. $T \quad x_1 ; 6, \theta$.
11. $T \quad x_1 ; 4, I$.
12. $T \quad x_1 ; I, 4$.
13. $T \quad x_1 ; 6, I$.
14. $T \quad x_1 ; I, 6$.
15. $T \quad x_1 ; 4, 6$.
16. $T \quad x_1 ; 6, 4$.

Therefore, $L(2)$ can be written in exactly 2^{2^2} instructions.

Now suppose the lemma is true for all n, $2 \leqq n \leqq m$. Consider $n = m + 1$. The library $L(m)$ by hypothesis has not more than 2^{2^m} instructions and covers all functions of m variables among the $2^{2^{m+1}}$ functions of $m + 1$ variables. For each of the other $2^{2^{m+1}} - 2^{2^m}$ functions of $m + 1$ variables, the program can be written

$$T \quad x_{m+1} ; A, B,$$

where A and B refer to addresses in the library $L(m)$. Hence $L(m + 1)$ can be written with not more than

$$(2^{2^{m+1}} - 2^{2^m}) + 2^{2^m} = 2^{2^{m+1}}$$

instructions. This proves the lemma.

Using Lemma 2 and combining it with Lemma 1, we have

Theorem 1: For all n,

$$\frac{1}{2}\frac{2^n}{n} < \mu_n < 4\frac{2^n}{n} - 1.$$

Proof: The lower bound was given by Lemma 1. To get the upper bound, let us write $n = (n - j) + j$, where j may vary from 0 to n. Let f be a switching function of n variables. Then f may be expanded about $n - j$ of its variables in its canonical expansion:

$$f(x_1, x_2, \cdots, x_n) = x_1'x_2' \cdots x_j' f(0,0, \cdots, 0, x_{j+1}, \cdots, x_n) \vee \\ \cdots \vee x_1x_2 \cdots x_j f(1,1, \cdots, 1, x_{j+1}, \cdots, x_n).$$

Now, we may write a program with exactly

$$1 + 2 + 2^2 + \cdots + 2^{j-1} = 2^j - 1$$

instructions to give all the 2^j functions of j variables

$$x_1'x_2'x_3' \cdots x_j', \cdots, x_1x_2x_3 \cdots x_j .$$

Also, by Lemma 2, we may construct a library $L(n - j)$ with not more than $2^{2^{n-j}}$ instructions. Hence f can be programmed with not more than $2^{2^{n-j}} + 2^j - 1$ instructions. Now set

$$j = n - [\log_2(n - \log_2 n)],$$

where $[x]$ denotes the largest integer less than or equal to x. Then

$$2^j = \frac{2^n}{2^{[\log_2(n - \log_2 n)]}} \leqq 2\,\frac{2^n}{n - \log_2 n} \leqq 3\,\frac{2^n}{n} \quad \text{for} \quad n \geqq 4.$$

Therefore, for $n \geqq 4$,

$$\mu_n \leqq \min \{2^{2^{n-j}} + 2^j - 1 \mid 4 \leqq j \leqq n\} \leqq 4\,\frac{2^n}{n} - 1.$$

Now, by direct computation, we find $\mu_1 = 1$, $\mu_2 \leqq 4$ and $\mu_3 \leqq 6$. Hence for all n, we have $\mu_n \leqq 4\ (2^n/n) - 1$, and the theorem follows.

Theorem 2: Given any ϵ, $0 < \epsilon < 1$, a fraction $1 - 2^{-\epsilon 2^n}$ of switching functions of n variables will need at least $2^n(2n)^{-1}(1 - \epsilon)$ binary-decision program instructions to program.

Proof: The number of possible binary-decision programs with not more than $2^n(2n)^{-1}\ (1 - \epsilon)$ instructions cannot exceed

$$\left\{2n\left[\frac{2^n}{2n}\,(1 - \epsilon)\right]^2\right\}^{2^n(2n)^{-1}(1-\epsilon)},$$

which is less than $2^{2^n(1-\epsilon)}$. Therefore, the fraction of switching functions of n variables which need not more than $2^n(2n)^{-1}(1 - \epsilon)$ instructions to program cannot exceed $2^{-\epsilon 2^n}$. Hence the rest must need at least $2^n(2n)^{-1}$ $(1 - \epsilon)$ instructions to program and the proof follows.

The procedure outlined in the proof of Theorem 1 yields for each switching function of n variables a binary-decision program. We will call this program the *normal* binary-decision program for that function. A close examination of this procedure will show that the number of program instructions executed in the computation of a particular value of any switching function of n variables never exceeds n. That is, in the computation no variable is examined more than once. Therefore, we have

Corollary 1: The number of instructions which has to be executed in the normal binary-decision program for the computation of each value of any switching function of n variables never exceeds n.

These results together give us a fairly good idea of how efficient it is to compute switching functions with binary-decision programs. For $n = 20$, for instance, practically all switching functions need more than 25,000 instructions to program, although none needs more than 200,000 instructions. The number of instructions that one needs to go through to compute a single value is never more than 20, however. We want now to compare these results with the AND-OR-SUM procedure mentioned earlier.

4.2 *An Alternative Procedure*

Before we consider the AND-OR-SUM procedure illustrated previously, it might be well for us to show why this particular procedure is chosen for comparison. A switching function is commonly written in terms of its variables and their complements connected by AND and OR. Besides AND and OR, there are eight other binary operations, denoted by $/$, $\downarrow$, $\oplus$, $\leftrightarrow$, $\supset$, $\subset$, $\not\supset$, and $\not\subset$, where we have called $\oplus$ the SUM operation. These can be written in terms of AND and OR operation:

$$\begin{array}{ll}
x \,/\, y = x' \vee y'. & x \supset y = x' \vee y. \\
x \downarrow y = x'y'. & x \subset y = x \vee y'. \\
x \oplus y = x'y \vee xy'. & x \not\supset y = xy'. \\
x \leftrightarrow y = xy \vee x'y'. & x \not\subset y = x'y.
\end{array}$$

In order not to be restrictive with our alternative computational procedure, let us be allowed to use any of these 10 operations in a computation. The first thing we wish to show is that we lose nothing by throwing away seven of these operations. In order to do this, let us call any switching function expression involving the variables and their complements, in which any of the 10 operations may appear, a *binary expression*. Let us also say that two binary expressions are *equivalent* if they represent the same function. Then

Theorem 3: Let $\bar{f}$ be a binary expression with r operations. Then there is an equivalent binary expression $\bar{g}$ having r or fewer operations such that the only binary operations appearing in the expression $\bar{g}$ are AND, OR and SUM.

For example, the expression

$$(x' \not\supset y) \downarrow [(z' \oplus u')/w']$$

is equivalent to the expression

$$(x \vee y)\,[w'(z \oplus u)]$$

in which no operation other than AND, OR and SUM appears.

To prove this theorem, we note that, if $\bar{g}$ is any binary-expression of r operations, $r \geqq 1$, then $\bar{g}$ is expressible as

$$\bar{g} = \bar{h} * \bar{k},$$

where $\bar{h}$ and $\bar{k}$ are binary-expressions each of $(r - 1)$ or fewer operations and $*$ is one of the 10 binary operations. Also,

$$\bar{g}' = (\bar{h} * \bar{k})' = \bar{h} *' \bar{k},$$

where $*'$ is again one of the 10 binary operations. Therefore, if $\bar{g}$ is any binary-expression with r operations, then its complement expression $\bar{g}'$ requires not more than r operations.

Going back to Theorem 3, we note that the theorem is true for $r = 1$. Now suppose that the theorem is true for all r, $1 < r \leqq R$. Let $\bar{f}$ be an expression with $R + 1$ operations. Then $\bar{f}$ is expressible in the form

$$\bar{f} = \bar{g} * \bar{h},$$

where $\bar{g}$ and $\bar{h}$ are expressions each with not more than R operations and $*$ is one of the 10 binary operations. Since for the seven operations $/$, $\downarrow$, $\leftrightarrow$, $\supset$, $\subset$, $\not\supset$ and $\not\subset$ we have $\bar{g}/\bar{h} = \bar{g}' \vee \bar{h}'$, $\bar{g} \downarrow \bar{h} = \bar{g}'\,\bar{h}'$, $\bar{g} \leftrightarrow \bar{h} = \bar{g}' \oplus \bar{h}$, $\bar{g} \supset \bar{h} = \bar{g}' \vee \bar{h}$, $\bar{g} \subset \bar{h} = \bar{g} \vee \bar{h}'$, $\bar{g} \not\supset \bar{h} = \bar{g}\,\bar{h}'$ and $\bar{g} \not\subset \bar{h} = \bar{g}'\,\bar{h}$, it follows that $\bar{f}$ can be expressed as

$$\bar{f} = \bar{k} *' \bar{m},$$

where $*'$ is one of the three operations AND, OR or SUM, $\bar{k}$ is either $\bar{g}$ or its complement $\bar{g}'$ and $\bar{m}$ is either $\bar{h}$ or its complement $\bar{h}'$. The theorem now follows from the previous assertion and the induction hypothesis.

Because of Theorem 3, we may as well consider only those operations AND, OR and SUM in our computation procedure. To make specific comparisons possible, let us define a Boolean program as one that is made up of three kinds of Boolean instructions:

```
AND    A, B, C,
OR     A, B, C,
SUM    A, B, C,
```

Here A, B or C refers to one of $4n$ possible locations in which values of the n variables and their complements as well as intermediate results may be stored. We have reserved $2n$ locations for the storage of intermediate results; this is sufficient for computing the most complex of functions of n variables. To standardize location reference we will use locations $1, 2, \cdots, n$ for storing variable values; $1', 2', \cdots, n'$ for storing complement values; and $1'', 2'', \cdots, (2n)''$ for storing intermediate results. The Boolean program for the function

$$[x \oplus (yz)]\,(y \vee z),$$

for example, can now be written

```
1. OR    2, 3, 1″;
2. AND   2, 3, 2″;
3. SUM   1, 2″, 2″;
4. AND   1″, 2″, 2″;
```

where x, y and z values are stored in locations 1, 2 and 3 respectively; x', y' and z' in locations $1'$, $2'$ and $3'$ respectively; and the final result is in $2''$.

To each function f, let $v(f)$ be a number such that no Boolean program which computes f can have fewer than $v(f)$ instructions. Let

$$v_n = \max \{v(f) \mid f \in F(n)\},$$

where, as before, $F(n)$ is the set of all switching functions of n variables. Then

Lemma 3:

$$v_n \geqq \frac{2^n}{3 \log_2 n + 8}.$$

Proof: Let $M(n,p)$ be the number of possible Boolean programs with p instructions. Then

$$M(n,p) \leqq [3(4n)^3]^p,$$

and

$$M(n,v_n) \leqq (192n^3)^{v_n}.$$

Suppose

$$v_n < \frac{2^n}{3 \log_2 n + 8}.$$

Then

$$M(n,v_n) \leqq (192n^3)^{2^n(3\log_2 n+8)^{-1}} < (192n^3)^{2^n(\log_2 192n^3)^{-1}} = 2^{2^n},$$

and the proof follows.

Lemma 3 affords us a comparison of Boolean programs and binary-decision programs. Combining Theorem 2 and Lemma 3, we see that, for $n \geqq 64$, the inequality

$$\mu_n < v_n$$

strictly holds, and it most probably holds for much smaller values of n as well. In fact, we see that, for large n, v_n is bigger than μ_n by an order of magnitude. It therefore follows that Boolean programs are in general much longer than binary-decision programs. Moreover, since for each computation every instruction in a Boolean program has to be executed, the difference in speed of computation becomes indeed astronomical.

It has been amply clear that, although Boolean representation of switching circuits has been the foundation on which switching theory had been built, the inherent limitations in the Boolean language seem to be difficult hurdles to surmount. Boolean representation is algebraic and highly systematic, but so inflexible that it is powerless against all but series-parallel circuits. Moreover, as this paper shows, it is extremely inefficient as an instrument for computation. Binary-decision programming is our attempt of a way to get beyond these limitations. It works well for computation. Further studies will be required to find efficient ways of minimizing binary-decision programs and to make binary-decision programming an instrument for circuit synthesis.

REFERENCES

1. Shannon, C. E., A Symbolic Analysis of Relay and Switching Circuits, A.I.E.E. Trans., **57,** 1938, p. 713.
2. Shannon, C. E., The Synthesis of Two-Terminal Switching Circuits, B.S.T.J., **28,** January 1949, p. 59.
3. Muller, D. E., Complexity in Electronic Switching Circuits, I.R.E. Trans., **EC-5,** 1956, p. 15.