Electronics Research Laboratory
University of California
Berkeley, California
Internal Technical Memorandum M-53

TECHNIQUES FOR THE SIMULATION OF COMPUTER LOGIC*

by

Melvin A. Breuer

*Research reported herein was partially supported by National Science Foundation grant G-15965 and Electrodata Division of Burroughs Corporation, Pasadena, California.

February 5, 1964

ABSTRACT

The simulation of a digital computer is an integral part of most computer design automation systems. The evaluation of the Boolean functions which characterize the computer being simulated constitutes one major portion of a simulation system. Four general procedural classes for evaluating these functions are defined. In order to greatly increase the efficiency of a simulation system, methods are presented for simultaneously evaluating many functions for one set of values of the variables, and for evaluating simultaneously one function for many sets of values for the variables.

I. INTRODUCTION

A computer logic simulator can be used as an extremely valuable tool in many problems, such as in the automatic generation of computer diagnostic tests, and in the check-out (debugging) of the logical design of a new computer prior to its actual construction. Since a computer can be completely characterized by a set of Boolean functions, one of the essential functions of a simulation system is the evaluation of logical expressions. The simulation of one bit time of the operation of the simulated computer requires, in general, a sequential evaluation of at least a subset of all the functions characterizing the simulated computer, and hence is a relatively slow process. Usually, many thousands of test cases (sets of data) are to be run. Hence, it is desirable that the evaluation of logical expressions be carried out as efficiently as possible. In a recent paper by Katz,[1] a few techniques for increasing the sophistication of Boolean logic evaluation are introduced. In this discussion, a few new system organizations will be outlined.

A Boolean logic evaluation procedure is defined to be data-independent if the sequence of operations performed is independent of the operands. Otherwise, the procedure is data-dependent, and contains conditional branching operations. A procedure is defined to be symbol-independent if in the expressions being evaluated, when in unfactored form, a change in the name of any variable used in the function does not change any operators in the procedure. Otherwise, the procedure is symbol-dependent. Obviously, all factorization procedures are symbol dependent.

A survey of present operating systems shows that Katz[2] and Wolf[3] employ noninterpretive data and symbol dependent procedures. Stockwell[4] and Larsen[5] employ an interpretive mode of operation, and resort to data-dependent symbol-independent, and data-independent symbol-dependent procedures, respectively.

The choice of the "best" procedures to use is a function of the expressions to be evaluated and a function of the system requirements and applications. The applications may vary from the simultaneous

evaluation of may functions for one set of variable values, to the evaluation simultaneously of one function for many sets of values.

II. SIMULTANEOUS EVALUATION OF MANY FUNCTIONS FOR ONE SET OF VARIABLE VALUES

A. TREE METHOD—DATA AND SYMBOL DEPENDENT PROCEDURE

In general, a variable may be operated on many times within the evaluation of a single function, and in the case of a data and symbol procedure, this sometimes leads to nonoptimal codes. In fact, as will now be shown, for such procedures, each variable need be operated on once, at most, for any given set of values for the variable. Using this fact, the "best" codes for the examples presented in Refs. 1 and 3 can be improved upon. A bi-product of this method will be that for a given set of variable values, many functions can be evaluated simultaneously.

Consider the set of functions

$$F = \left\{ F_1, F_2, \ldots, F_t, \ldots, F_m \right\} ,$$

where all F_t's are either a function of the variables $x_1, x_2, \ldots, x_n$ or some subset of these variables. One can now construct an evaluation tree, analogous to the standard relay-switching tree encountered in combinatorial logic synthesis.[7] Each node (n_i) of the tree corresponds to a conditional branch operator based on the value of a variable x_v associated with this node. Let node n_i have input branch b_i and output branches b_j and b_k. Each branch (b_i) of the tree corresponds to a set of evaluations (e_i) for some set of the variables. Letting b_1 be the root of the tree, then $e_1\{\phi\}$. If x_v is associated with n_i, then $e_j = \{e_i, x_v = 1\}$, and $e_k = \{e_i, x_v = 0\}$. We can also associate with each branch b_i a set of functions F^{e_i}, which is the set of original functions F simplified, under the rules of Boolean logic, according to the value of the variables specified by e_i. Branch b_i is a terminal branch or leaf, when all functions in F^{e_i} simplify to either 0 or 1. Note that if $F^{e_{i_1}} = F^{e_{i_2}}$, then b_{i_1} can be relabeled b_{i_2} and becomes

another input to n_{i_2}, and the sub-tree defined by the root node n_{i_1} may be discarded. This pruning reduces the total number of commands in the code associated with the tree.

The variable x_v associated with node n_i is such that x_v appears in at least one function in the set F^{e_i}. In principle, one would like to chose the variable to be associated with a node such that the program obtained from the tree would optimize some objective. For example, one such criterian would be to minimize the average number of execution cycles (T) when running the program assuming all variables take on the values 0 and 1 with equal probability. Unfortunately, the determination of the optimal choice for each x_v is, in general, a hard problem. A dynamic programming solution does exist, and though the procedure is not exhaustive, it does require the construction and analysis of a great many trees. One can of course base the choice of x_v on some "rule of thumb." One such rule which has been found to be fairly efficient is to associate with n_i that variable x_v which appears most often in the set of functions F^{e_i}.

As an example of this method, consider the set F and a tree generated from F as shown in Fig. 1. The code for evaluating the functions in F can be easily found from this tree, and one such code is given in Fig. 2.

We assume the simulating computer to be of the IBM 704 family,* that each variable and its complement are stored in unique words in core memory, and that the value of these variables is stored in some arbitrary, say the i-th, bit position of these words, all other bit positions containing zero. Assume also that F_1, F_2, F_3, and F_4 have been preset to zero, and that the accumulator contains a single one in the i-th bit position.

*See Appendix A for definition of operators.

Example

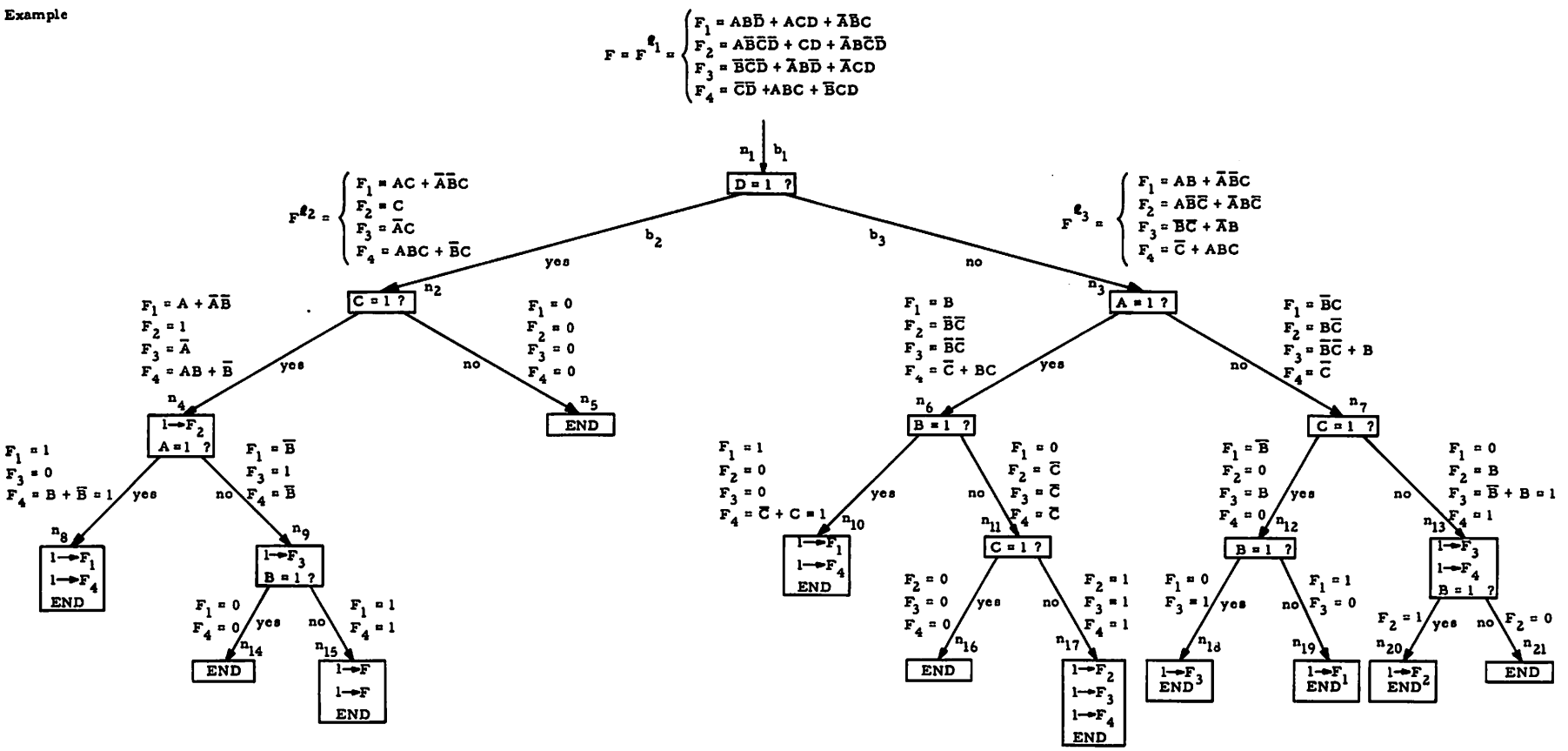

Fig. I.

**	*	n_1	NZT	D			STO	F_4
**			TRA	n_3			TRA	END
	*	n_2	NZT	C		n_{11}	ZET	C
	*	n_5	TRA	END		n_{16}	TRA	END
		n_4	STO	F_2		n_{17}	STO	F_2
			NZT	A			STO	F_3
			TRA	n_9			STO	F_4
		n_8	STO	F_1			TRA	END
			STO	F_4	**	n_7	NZT	C
			TRA	END	**		TRA	n_{13}
		n_9	STO	F_3		n_{12}	NZT	B
			ZET	B			TRA	n_{19}
		n_{14}	TRA	END		n_{18}	STO	F_3
		n_{15}	STO	F_1			TRA	END
			STO	F_4		n_{19}	STO	F_1
			TRA	END			TRA	END
**		n_3	NZT	A	**	n_{13}	STO	F_3
**			TRA	n_7	**		STO	F_4
		n_6	NZT	B	**		ZET	B
			TRA	n_{11}	**	n_{20}	STO	F_2
		n_{10}	STO	F_1	END	n_{21}	---	

Fig. 2.

The shortest path through the tree occurs for D = 1, C = 0, in which case 3 instructions (marked by *) are executed, and the cycle time is 5. The longest path occurs for A = B = D = 0, B = 1, in which case 10 instructions

(marked by **) are executed, and the cycle time is 16. A total of 41 instructions are required. Note that the structure of the tree is determined by F_1, F_2, and F_3, and the evaluation of F_4 requires no additional tests or branches, though it does necessitate the use of five store instructions. Applying this technique to the function given in Ref. 1, and assuming the conditions as outlined there, * we get the following code,

```
BEGIN   CAL   C̄                      CAL   Ā
        TLQ   STORE                  TLQ   LOC
        CAL   D                      CAL   B̄
        TLQ   STORE                  TRA   STORE
        CAL   E              LOC     CAL   B
        TLQ   STORE          STORE   STO   F
```

which requires 12 instructions and gives a $T = 7\ 15/16$, which is to be compared to the results shown in Table III of Ref. 1, where 13 instructions are required, and $T = 10\ 21/22$.

For the general case, one can construct a tree with 2^n final branches, each one corresponding to a canonical term C_k, i.e., to a unique valuation for the n variables. If $C_k = 1 \Longrightarrow F_t = 1$, we would set $F_t = 1$ in the last block of the k-th branch. With this tree, all functions of the variables $(x_1, x_2, \ldots, x_n)$ could be evaluated. The order in which the value of the n variables are inspected will not effect T, hence there is no optimization problem.

The code would contain $\sum_{i=1}^{n} 2^i$ value test commands (e.g., NZT or ZET). However, during evaluation, only n such commands would be executed, since each variable is inspected at most just once. On the average n/2 transfer instructions would be executed. Assume that on the average, m/2 of the m functions F_t assume the value 1, and are set to 1 in the last branch of the tree. Assuming all final

---

* Value of variable stored in sign position of each word.

branches are equally likely outcomes, we have

$$T \cong 2 \times n + 1 \times n/2 + 2 \times m/2 + \underset{\text{TRA END}}{1} = (5n)/2 + m + 1.$$

For $n = 4$, $m = 4$, we obtain $T \cong 15$.

In general, the efficiency of this tree method increases as the number of appearances of a variable or its negation in the functions being evaluated increases. If each variable appears just once, the method is generally inefficient. This technique is most applicable in decoding or combinatorial logic circuits. Note also that if the complement of variables is not stored, then we can replace

$$\begin{Bmatrix} \text{NZT} & \overline{X} \\ \text{TRA} & \end{Bmatrix} \text{ by } \begin{Bmatrix} \text{ZET} & X \\ \text{TRA} & \end{Bmatrix} \text{ and } \begin{Bmatrix} \text{ZET} & \overline{X} \\ \text{TRA} & \end{Bmatrix} \text{ by } \begin{Bmatrix} \text{NZT} & X \\ \text{TRA} & \end{Bmatrix}.$$

The saving in computer storage by not storing the value of the complement of each variable can sometimes be quite considerable.

B. CANONICAL FORM METHOD

An extension of an idea suggested by Professor D. C. Evans yields a second method for simultaneously evaluating many equations for one set of data. In this approach, however, any one of the four procedural classes can be employed.

Assume we wish to evaluate the set of functions $F = \{F_1, F_2, \ldots, F_t, \ldots, F_w\}$, $1 \leq w \leq W$, where W is the number of bits in a word in the simulating computer. Let $F_C = P_1 f_1 + P_2 f_2 + \ldots + P_s f_s + \ldots + P_{2^n} f_{2^n}$, where P_s is a vector with w elements, and f_s is the s-th canonical minterm of n variables. For example, $f_1 = \overline{x}_1 \overline{x}_2 \ldots \overline{x}_n$, $f_2 = \overline{x}_1 \overline{x}_2 \ldots \overline{x}_{n-1} x_n$, and $f_{2^n} = x_1 x_2 \ldots x_n$. P_s is a presence indicator or operator, and we have

$$P_s = \begin{Bmatrix} P_{s1} \\ \vdots \\ P_{st} \\ \vdots \\ P_{sw} \end{Bmatrix}$$

where $P_{st} = 1$ if $f_s = 1 \Longrightarrow F_t = 1$, otherwise $P_{st} = 0$. If $P_s = P_v$ and if $f_s + f_v$ can be logically simplified to f_{sv}, then we can replace $P_s f_s + P_v f_v$ by $P_{sv} f_{sv}$. If $P_s = \underline{0}$, then $P_s f_s$ may be deleted. If $P_s = \underline{1}$, then P_s may be deleted. The final reduced form of F_C is labelled F_{CR}. P_s is a word in memory, the t-th bit position containing $P_{st} \epsilon (0,1)$. The value of variable x_i must now be stored in <u>all</u> W bit positions of word x_i, for all i. Expression F_{CR} can be coded using any of the procedures defined. In the final result, the value of F_t will be found in the t-th bit position of word F_{CR}. The results must therefore be unpacked from F_{CR} and transferred to their appropriate locations.

Example:

$$\text{Let } F = \left\{ \begin{array}{l} F_1 = AB + AC + \bar{A}\bar{B} \\ F_2 = A\bar{B} + C + \bar{A}B \\ F_3 = \bar{A} + \bar{B}\bar{C} \\ F_4 = A + \bar{B} + \bar{C} \end{array} \right\}$$

$$\left\{ \begin{array}{l} F_1 = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + A\bar{B}C + AB\bar{C} + ABC \\ F_2 = \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} + A\bar{B}C + ABC \\ F_3 = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + \bar{A}BC + A\bar{B}\bar{C} \\ F_4 = \bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + \bar{A}B\bar{C} + A\bar{B}\bar{C} + A\bar{B}C + AB\bar{C} + ABC \end{array} \right\}$$

Therefore $F_C = P_1\bar{A}\bar{B}\bar{C} + P_2\bar{A}\bar{B}C + P_3\bar{A}B\bar{C} + P_4\bar{A}BC + P_5A\bar{B}\bar{C} + P_6A\bar{B}C + P_7AB\bar{C} + P_8ABC$

where

$$P_1 = \left\{ \begin{array}{l} 1 \\ 0 \\ 1 \\ 1 \end{array} \right\}, \quad P_2 = \left\{ \begin{array}{l} 1 \\ 1 \\ 1 \\ 1 \end{array} \right\}, \quad P_3 = P_5 = \left\{ \begin{array}{l} 0 \\ 1 \\ 1 \\ 1 \end{array} \right\}, \quad P_4 = \left\{ \begin{array}{l} 0 \\ 1 \\ 1 \\ 0 \end{array} \right\}$$

$$P_6 = P_8 = P_{68} = \left\{ \begin{array}{l} 1 \\ 1 \\ 0 \\ 1 \end{array} \right\}, \quad P_7 = \left\{ \begin{array}{l} 1 \\ 0 \\ 0 \\ 1 \end{array} \right\}.$$

Hence $F_{CR} = P_1\bar{A}\bar{B}\bar{C} + \bar{A}\bar{B}C + P_3\bar{A}B\bar{C} + P_4\bar{A}BC + P_5 A\bar{B}\bar{C} + P_{68}AC + P_7AB\bar{C}$.

III. SIMULTANEOUS EVALUATION OF ONE FUNCTION FOR SEVERAL SETS OF VALUES

When checking the logical design of a new computer or when simulating diagnostic tests, the simulation is usually repeated for each test case. Usually many thousands of test cases exist. Bashkow[6] states in his discussion on detecting machine malfunctions:

> "Ideally the accumulator should be tested with all possible bit configurations. The word length of the SPE is 32 bits, however, and the time required to form 2^{32} configurations would be excessive. As a compromise the accumulator is caused to count by ones from 1 to 2^{12}, 2^5 to 2^{17}, ..."

If W is the number of bits in a computer word of the simulating computer, then $w(1 \leq w \leq W)$ tests can be simulated simultaneously if the values of the variables for the i-th tests are stored in the i-th bit position of their respective words. The actual time required to evaluate w tests simultaneously is equal to the time required to evaluate the longest test. Note that the maximum increase in efficiency of the simulation system of almost a factor of w is essentially realized by efficient usage of memory. In most of the papers reviewed, the authors used an entire word of memory to store the binary valued output of a single logical element.

When coding an expression which will be used to simultaneously evaluate many test cases, the tree procedure outlined in Sec. II.A cannot be employed. This is due to the fact that in general, a variable will have the value 0 for some tests, and 1 for others. Hence, if a data dependent procedure is to be employed, it is necessary to branch on either all 1's or all 0's, but not both.

IV. SUMMARY

In this paper we have defined the various procedures for evaluating Boolean functions. Techniques for simultaneously evaluating many functions for one set of variable values, and for evaluating

simultaneously one function for many sets of variable values have been presented. No claim that one procedure is better than another has been made. The best procedure to use for any given system is a function of the simulation system requirements, the characteristics of the simulating computer, and the exact expressions to be evaluated. The inherent efficiency in the procedures proposed lies not in their being optimally implemented, but rather in that many evaluations are carried on simultaneously.

APPENDIX A

DEFINITION OF INSTRUCTIONS USED IN SIMULATING COMPUTER

Instruction	Address	Cycles	Definition
NZT	Y	2	If C(Y) $\neq$ 0, computer skips next instruction.
ZET	Y	2	If C(Y) = 0, computer skips next instruction.
TLQ	Y	2	If C(MQ) $<$ C(AC), computer takes its next instruction from location Y.
TRA	Y	1	Computer takes its next instruction from location Y.
ORA	Y	2	OR to accumulator
ORS	Y	2	OR to storage
ANA	Y	3	AND to accumulator
CAL	Y	2	Clear and Add Logical Word
CLA	Y	2	Clear and Add
STO	Y	2	Store contents of accumulator in C(Y).

REFERENCES

1. J. H. Katz, "Optimizing bit-time computer simulation," Communications of the ACM, Vol. 6, No. 11, pp. 679-685; Nov. 1963.
2. J. Katz, H. Adler, H. Jacobs, "The R-W logic simulation program," AIEE Conference Paper CP 60-1063, 9 pages; August 8, 1960.
3. R. E. Wolfe, "Logical simulation techniques using the IBM 709," Dynamic Digital Logic, Conference sponsored by Computer Control Company.
4. G. N. Stockwell, "Computer logic testing by simulation," IRE Trans., PGME, pp. 275-282; July 1962.
5. R. P. Larsen, "Logic simulation program," presented at the AIEE Design Automation Workshop 1962, Michigan State University, 6 pages; May 1962.
6. J. R. Bashkow, J. Friets, A. Karson, "A programming system for detection and diagnosis of machine malfunctions," IEEE Trans., PGEC, pp. 10-17; Feb. 1963.
7. S. H. Caldwell, Switching Circuits and Logical Design, John Wiley & Sons, New York, pp. 211-226; 1958.