THE BELL SYSTEM

TECHNICAL JOURNAL

DEVOTED TO THE SCIENTIFIC AND ENGINEERING
ASPECTS OF ELECTRICAL COMMUNICATION

Volume 50 | July-August 1971 | Number 6

Mathematical Models of Computation Using Magnetic Bubble Interactions

By A. D. FRIEDMAN and P. R. MENON

(Manuscript received February 10, 1971)

This paper considers the computational capabilities of different mathematical models of magnetic bubble interactions. A specific model was studied earlier by R. L. Graham, who showed that there exist combinational functions of 11 or more variables that cannot be computed by this model. This paper extends his results by introducing different types of interactions which seem to be practical and enable the computation of all combinational functions. The problem of efficient computation from the points of view of time and space requirements and the geometrical requirements imposed by the fact that interactions can occur only between physically adjacent locations are also examined. Finally, a model in which computations are carried out by applying uniform magnetic fields to the entire platelet, with individual access limited to locations along the periphery, is presented.

I. INTRODUCTION

Cylindrical magnetic domains in certain orthoferrite materials have been investigated extensively in recent years.[1–3] These domains, commonly referred to as bubbles, have the property that they can be moved within the material by the application of suitable external magnetic fields. The motion of a bubble is also dependent on other bubbles in its vicinity. The locations of bubbles can be restricted to a finite set of possible positions in an orthoferrite platelet. The presence or absence of a bubble at a particular location may be treated as representing the values of a binary variable. Thus, magnetic bubbles seem to have natural applications in performing memory and logic functions.

In a recent paper, R. L. Graham[4] considered the computational capabilities of a particular mathematical model of magnetic bubble interaction. The only type of interaction allowed in this model is represented by an instruction of the type (x, y) where x and y are distinct adjacent locations in the orthoferrite platelet. Application of this instruction results in moving the bubble in location x to location y if the latter does not contain a bubble prior to the application of the instruction. If x does not contain any bubble or y already has a bubble, no transfer of bubbles takes place. Two locations, only one of which contains a bubble, are used to represent each binary variable and its complement. Graham has shown that only a small fraction of all combinational functions of 11 or more variables can be computed by this model.

In this paper, we review Graham's results and extend his model by introducing different types of interactions which seem to be practical and enable the computation of all combinational functions. We then examine the problem of efficient computation from the points of view of space and time requirements. The geometrical requirements, necessitated by the fact that bubble interactions can occur only between physically neighboring locations, are also examined. Finally, we consider a model in which computations are carried out by applying magnetic fields to the entire platelet and individual access is limited to locations along the periphery.

II: MODELS OF BUBBLE INTERACTIONS

Let us first consider the model studied by Graham.[4] All locations are assumed to be adjacent to one another so that interaction between any pair of locations is possible. If S is the set of all possible bubble locations and n is the number of such locations, let Q be the set of all 2^n subsets of S. If a subset $X \in Q$ containing bubbles represents the values of the variables of a function f, then f is a mapping from Q onto Q. The function f is realized by applying a program P to S with bubbles in locations contained in X. If the set of resultant bubble locations is

X^P, then $X^P = f(X)$ and $X^P \in Q$. The program P consists of a sequence of instructions of the form $e = (a, b)$ such that $X^e = (X - \{a\}) \cup \{b\}$, if $a \in X$, $b \notin X$, and $X^e = X$ otherwise. The nondecreasing overlap (NDO) theorem due to Shockley and proven in Ref. 4 points out some of the limitations of this model of interaction. The NDO theorem is repeated below in the interest of completeness.

Theorem 1 (NDO Theorem): Let X_1 and X_2 be two initial sets of locations of bubbles and let P be any program (of the type discussed above). Then $| X_1^P \cap X_2^P | \geqq | X_1 \cap X_2 |$, where $| X |$ is the cardinality of the set X.

The NDO theorem precludes the possibility of certain types of computation using this model of bubble interaction. For example, it is impossible to have a program P such that $\{a, b\}^P = \{c, d\}$ and $\{a\}^P = \{e\}$. Another consequence of the NDO theorem is the nonexistence of a replicating program. That is, there exists no program P^* such that if S and S' are nonintersecting sets of vertices and $X \subset S$, P^* creates a set $X' \subset S'$ without disturbing X, so that there is a one-to-one correspondence between X and X'. In order to be able to compute all combinational functions, it may be necessary to do one or both of the following: (*i*) Include other types of interactions. (*ii*) Code the inputs and the outputs.

Let us consider the following set of instruction types, where the first type is the one we have discussed above.

Name	Notation	Resultant Bubble Locations
I. Bubble transfer	$e=(a, b)$	$X^e = \begin{cases} (X - \{a\}) \cup \{b\} \text{ if } a \in X, b \notin X; \\ X \text{ otherwise.} \end{cases}$
II. Bubble splitting	$e=(a, b)_S$	$X^e = \begin{cases} X \cup \{b\} \text{ if } a \in X; \\ X \text{ otherwise.} \end{cases}$
III. Bubble annihilation	$e=(0, a)$	$X^e = X - \{a\}$.
IV. Bubble creation	$e=(1, a)$	$X^e = X \cup \{a\}$.

In the above set, type II instructions are necessary for replication. Type III instructions have the property that $|X^e| < |X|$ and are therefore necessary to compute $\{a, b\}^P = \{c\}$. Instructions of type IV are necessary for performing computations of the type $\varphi^P = \{a\}$, where φ is the null set.

Theorem 2: The four types of instructions (viz., bubble transfer, splitting, annihilation, and creation) are not sufficient for performing all computations.

Proof: Consider a program P such that $X^P = S - X$. Let the first instruction of the program be e_1. We show that a program P for computing $S - X$ using only the four types of instructions discussed above does not exist, by showing that for each type of instruction there exist two sets $X_1 \neq X_2$ such that $X_1^{e_1} = X_2^{e_1}$. Since $S - X_1 \neq S - X_2$, the program cannot compute $S - X$ for all X.
Case 1: Let $e_1 = (a, b)$.

Let $X_2 = (X_1 - \{a\}) \cup \{b\}$; $X_1, X_2 \subset S$; $a \in X_1$; $b \notin X_1$. Then

$$X_1^{e_1} = X_2^{e_1} = X_2 \quad \text{and} \quad X_1^P = X_2^P .$$

Case 2: Let $e_1 = (a, b)_S$.
Let $a \in X_1$, $b \notin X_1$, and $X_2 = X_1 \cup \{b\}$. Then

$$X_1^{e_1} = X_2^{e_1} = X_2 \quad \text{and} \quad X_1^P = X_2^P .$$

Case 3: Let $e_1 = (0, a)$.
Let $a \in X_1$, and $X_2 = X_1 - \{a\}$. Then

$$X_1^{e_1} = X_2^{e_1} = X_2 \quad \text{and} \quad X_1^P = X_2^P .$$

Case 4: Let $e_1 = (1, a)$.
Let $a \in X_1$, and $X_2 = X_1 - \{a\}$. Then

$$X_1^{e_1} = X_2^{e_1} = X_1 \quad \text{and} \quad X_1^P = X_2^P .$$

Thus there exists no program P using only the four types of instructions that can compute $S - X$ for all $X \subset S$. This implies that complementation cannot be done (using only these types of instructions) if the value of a Boolean variable is represented by the presence or absence of a bubble in a particular location.

The other approach mentioned earlier for improving the computational capabilities of magnetic bubbles is the use of suitable codes. Graham[4] has used two bubble locations x_0 and x_1 to represent the value of a binary variable x. $x = 0$ is represented by $x_0 = 1, x_1 = 0$ and $x = 1$ by $x_0 = 0, x_1 = 1$. However, Graham has shown that this representation and the bubble transfer instruction are not sufficient for realizing all Boolean functions. By enumerating the number of distinct programs, he has proven the following theorem.
Theorem 3: There exists a Boolean function of 11 variables which cannot be realized by a program of bubble transfer instructions.

It is not known whether there exists some coding of the variables which permits the realization of all Boolean functions using only the bubble transfer instruction. However, the four types of instructions together with a coding of the type used by Graham are sufficient for realizing all Boolean functions.

Theorem 4: All combinational functions can be computed using the four types of instructions and a suitable coding of inputs.

Proof: Any combinational function can be realized using the following method. Let the set of all possible bubble locations be denoted by M, where $M = S \cup T$ and $S \cap T = \varphi$. As before, values of the input variables are represented by bubbles in subsets of S, and let each variable be represented by two locations x_0 and x_1. Let T be the set of possible bubble locations that serves as temporary storage. Any given combinational function f and its complement $\bar{f}$ are expressed in the sum of products form. The following procedure is used to realize the function f:

(*i*) Clear the output locations f_0 and f_1 by $(0, f_0)(0, f_1)$.
(*ii*) Clear temporary storage by $(0, y_{i0})(0, y_{i1})$ for all $y_{i0}, y_{i1} \,\varepsilon\, T$.
(*iii*) Copy the values of the input variables into temporary storage using bubble splitting instructions $(x_{i0}, y_{i0})_S(x_{i1}, y_{i1})_S$ for all $x_{i0}, x_{i1} \,\varepsilon\, S$.
(*iv*) If x_i^* is a variable appearing in a product term, where x_i^* represents x_i or $\bar{x}_i$, the term $\prod x_j^*$ is computed by the set of instructions (y_{1a}, y_{jb}) for all $j \neq i$ such that x_j^* is contained in the product term and $a = 0$ if $x_i^* = \bar{x}_i$, $a = 1$ if $x_i^* = x_i$, $b = 0$ if $x_j^* = \bar{x}_j$, and $b = 1$ if $x_j^* = x_j$.
(*v*) (y_{ia}, f_1) puts the OR of all terms computed so far in f_1.
(*vi*) If there are additional terms, go to step *ii* and repeat for next term.
(*vii*) If there are no more terms in f, repeat steps *ii* through *vi* for $\bar{f}$, replacing step *v* by (y_{ia}, f_0).

The above procedure merely shows that all combinational functions can be computed using the four types of instructions and the particular type of coding used. Both the program and the coding could be made more efficient.

The property of the code that made the computation of all combinational functions possible is the elimination of the need for complementation, which was shown to be impossible to perform with the four types of instructions used. A more general code which has this property is an m-out-of-n (m/n) code[5] where n bubble locations, out of which exactly m contain bubbles, are used to represent each input combination. A procedure analogous to that given above may be used for computing any combinational function using this code. The case discussed above, where each variable is represented by two bubble locations, is a special case of the m/n code and is referred to as the autosynchronous code.[5]

The m/n code is more efficient than the autosynchronous code in terms of the number of bubble locations required for encoding a given number of binary variables. Whereas $2k$ locations are necessary to represent k binary variables, the number of locations required with an m/n code is such that $\binom{n}{m} \geqq 2^k$. Since $m = [n/2]$, where $[x]$ is the integral part of x, gives the largest value of $\binom{n}{m}$, we have $\binom{n}{[n/2]} \geqq 2^k$. Using Stirling's approximation, we have $[n - \frac{1}{2} \log_2 n] \geqq k$. For large values of k, the $[n/2]/n$ code requires fewer locations than the autosynchronous code as the following sample values indicate:

k	$n([n/2]/n$ code)	$2k$ (autosynchronous code)
14	16	28
29	32	58
61	64	122

A given type of instruction is said to be *computationally complete* if all combinational functions can be computed using programs containing only instructions of the given type, assuming that any location may be initially cleared and a bubble may be initially inserted in any location. That is, instructions of the type $(0, x)$ and $(1, x)$ may be used for initialization, prior to the application of the program. Note that the bubble transfer instruction discussed earlier is not computationally complete under our definition, although bubble transfer and bubble splitting together are sufficient to compute any combinational function. (The need for clearing locations within the program in the procedure discussed earlier can be eliminated by using a sufficient number of temporary locations.) We shall now consider some computationally complete instruction types that may be realizable by bubble interactions.

Consider an instruction $e = (x, y)_d$, defined by

$$\begin{aligned} X^e &= X - \{y\} \qquad \text{if } x, y \in X \\ &= (X - \{x\}) \cup \{y\} \qquad \text{if } x \in X,\ y \notin X \\ &= X \text{ otherwise.} \end{aligned}$$

If the presence or absence of a bubble in a location is treated as the value of a binary variable, and if the value of a variable after the application of an instruction $(x, y)_d$ is denoted by the corresponding primed variable, the instruction can be represented by the following equations:

$$x' = xy \quad \text{and} \quad y' = x \oplus y$$

where $\oplus$ represents exclusive-OR.

We prove that the instruction $(x, y)_d$ is computationally complete by showing that we can perform duplication and also realize the NAND function using only instructions of this type. It is well known that any combinational function can be realized using only two-input NAND gates if duplication (fan-out) is allowed. Denoting locations that initially contain bubbles by lower-case letters with the subscript 1, the NAND of two variables x and y can be realized by the sequence of instructions $(x, y)_d(x, a_1)_d$; which results in, $x' = xy$, $y' = x \oplus y$, $a_1' = 1 \oplus xy = \overline{xy}$. Duplication can be performed by the sequence $(x, b_1)_d(b_1, c_1)_d$, resulting in $x' = x$, $b_1' = 1 \oplus x = \bar{x}$, $c_1' = 1 \oplus \bar{x} = x$. Thus all combinational functions can be computed without special encodings.

A timing problem associated with this instruction* must be noted. Two applications of the instruction $(x, y)_d$ results in $xy(x \oplus y) = 0$ in location x and $xy \oplus x \oplus y = x + y$ in location y. Physically, if $x = y = 1$ when the instruction $(x, y)_d$ is applied, the bubble in y is destroyed. If the field effecting the execution of the instruction is still applied, the bubble in location x is transferred to y. Thus, this instruction requires application of the field for a fixed interval of time. We may say that the instruction is pulse-width dependent. For each of the other instruction types considered, application of the same instruction a second time does not result in any further change in either location.

Another type of computationally complete instruction is the conditional transfer† denoted by $e = (x, y)z$ and defined by

$$\begin{aligned} X^e &= (X - \{x\}) \cup \{y\} \qquad \text{if } x, z \in X,\ y \notin X \\ &= X \text{ otherwise.} \end{aligned}$$

Duplication can be performed by the instruction $(a_1, a_0)x$ where a_0 and a_1 denote an empty location and a location with a bubble respectively. After the application of the instruction, locations a_0 and x will contain the variable x. Similarly, the sequence $(x, y)b_1$, $(b_1, b_0)x$,

* In practice, there exists an interaction involving three locations x, y, z, such that if $y = 0$, $y' = x \oplus z$; $x' = z' = xz$. The timing problem mentioned above does not occur in this case.

† The interaction involving three locations, mentioned in the previous footnote, may be represented by the following sequence of conditional transfer instructions, if $y = 0$ initially: $(x, y)\bar{z}$; $(z, y)\bar{x}$. The order of execution of these two instructions is immaterial.

yields the function $\overline{xy}$ in location b_1. All combinational functions can be computed using only conditional bubble transfers without special encodings.

III. TIME AND SPACE CONSIDERATIONS

In the preceding section, we showed how any combinational function can be computed using magnetic bubble interactions, assuming that interactions are possible between any pair of bubble locations. Since interactions can occur only between bubbles in physically adjacent locations, this implies that bubbles that are required to interact are brought into adjacent locations prior to the application of the instruction. The time taken for computing a function depends not only on the number of instructions in the program but also on the layout of the bubble locations. It is also necessary to be able to move a bubble from any arbitrary location to any other location without affecting the bubbles in other locations. We shall examine these problems in this section.

3.1 *Bounds on Memory Requirements*

Consider the layout shown in Fig. 1. If the locations in the shaded area are used as bubble locations for a program, the locations in the unshaded part can be used as transit points. These memory locations are initialized so as not to contain bubbles. Denoting the shaded and unshaded regions by S and T respectively, an instruction of the form (a, b) $a, b \in S$ will be replaced by a sequence of instructions

$$(a, x_i)(x_i, x_{i+1}) \cdots (x_{j-1}, x_j)(x_j, b)(x_j, x_{j-1}) \cdots (x_{i+1}, x_i)(x_i, a)$$

where

$$x_i, x_{i+1}, \cdots x_j$$

are adjacent locations in T so as not to affect any other location in S. With this layout $3n/2 - 1$ locations are required for n locations which are useful for computation. (We shall refer to these as data points.) If p is the total number of locations, the maximum number of data points can be shown to approach $2p/3$ asymptotically (K. C. Knowlton, private communication).

The maximum path length (one-way) between two data points is $n/2$. The maximum path length can be greatly reduced at the expense of a slight increase in the required memory (i.e., total number of locations) by the layout shown in Fig. 2a. The total number of locations

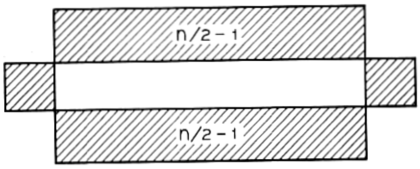

Fig. 1—Memory layout.

is $(n/2k - 3)\cdot 3k + 12k = 3n/2 + 3k$, and the number of data points is $2\cdot 3k + (k - 1)\cdot(n/k - 6) + 2(n/2k - 3) = n$. The maximum path length (one-way) is $n/2k + 3k - 1$. [By changing the corners as shown in Fig. 2b, the maximum one-way path length can be reduced by 2, and the total number of locations can be reduced by 4 (M. D. McIlroy, private communication).] The maximum path length is dependent on k. Since k must be an integer and $2k$ must divide n, the maximum path length is minimal when $k = \sqrt{n/6}$ if this is an integer. Thus if $n = 6k^2$ the maximum path length for the layout of Fig. 2a is $(\sqrt{6}\,\sqrt{n} - 1)$. Some typical values of n and the minimum maximal path length for various values of k are shown below:

k	n	path length
2	24	11
3	54	17
4	96	23
5	150	29

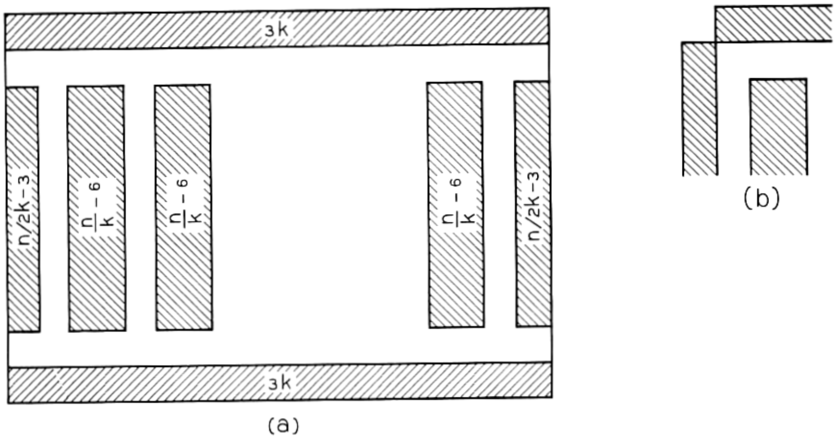

Fig. 2—Memory layout to reduce maximum path length.

In order to obtain a lower bound on the maximum path length for a memory with n locations, allowing interactions between any pair of locations, consider the closest packing of these locations in two-dimensional space, which is a circle. If the radius of the circle is r, $\pi r^2 = n$ or $r = \sqrt{n/\pi}$. The maximum distance between two points is $2r = \sqrt{4n/\pi}$. Clearly, this bound cannot be attained because all points in the circle are treated as data points in deriving the bound. Thus the best we can hope to do is to obtain a maximum path length of $C\sqrt{n}$ where $C > \sqrt{4/\pi}$. The coding of Fig. 2 has maximum path length $< \sqrt{6}\sqrt{n}$ and hence requires approximately twice as much time as the lower bound.

The memory layouts discussed above allow interactions between all pairs of data points and are therefore suitable for realizing any arbitrary program. For realizing any specific function, interactions would be necessary only between a subset of these pairs of data points and consequently a smaller memory would suffice. The maximum path length can also be reduced in general. In this case, the memory requirements and maximum path lengths derived above serve only as upper bounds.

3.2 *Number of Instructions for Realizing a Function*

The number of instructions and bubble locations required for realizing any given function using any given type(s) of instructions may vary over a wide range depending upon the realization, as shown in the following examples.

Example 1: Consider the computation of $f = x \oplus y$, using bubble transfer instructions and bubble splitting, if necessary. Let two locations x_0 and x_1 be used to represent each variable x, as discussed earlier. Using the canonical realization in the sum of products form discussed earlier, four minterms (two each for the function and its complement) have to be computed. Each minterm is computed using four bubble splitting instructions and one bubble transfer instruction. Finally, two bubble transfer instructions are required to compute the sum of two minterms. Thus the total number of instructions in the program will be $5 \times 4 + 2 \times 2 = 24$. The total number of data points required is 10 (4 for x and y, 2 for f, and 4 temporary storage). Using the layout of Fig. 1, 14 bubble locations are required in the memory.

The following program containing only six instructions also computes this function, if f_0 and f_1 are both initially empty:

$$P = (x_1, y_1)(x_0, y_0)(x_1, f_0)(x_0, f_0)(y_1, y_0)(y_1, f_1).$$

Six data points are necessary, and these can be obtained using eight locations and the layout of Fig. 1. Since interaction between all pairs of data points is not required by the program, it can be shown that seven locations are necessary and sufficient to permit the required interactions. However, the following program which also contains six instructions and requires six data points can be realized with only six bubble locations:

$$P' = (x_1, y_1)(x_0, y_0)(x_1, x_0)(x_0, f_0)(y_0, y_1)(y_0, f_1).$$

Figure 3 shows the memory layout for this program.

This example shows that the canonical realization may be inefficient in the number of instructions, computation time, and also memory requirements. The inefficiency of the canonical realization becomes even more apparent if we compare the canonical realization of $f = x_1 \oplus x_2 \oplus \cdots \oplus x_n$ with the realization in the form $[(x_1 \oplus x_2) \oplus x_3 \cdots \oplus x_n]$, where each exclusive-OR operation is performed by a program of the type given above.

The above example points out several open problems associated with the design of efficient programs for computing combinational functions. One problem is that of obtaining a program with the smallest number of instructions for computing a given function. For a given program, the assignment of locations so as to minimize the total number of locations required for its implementation is also an important problem. Since the number of memory locations required is dependent not only on the number of instructions, but also on the specific program, another open problem is that of obtaining a program with minimum memory requirements.

The problem of minimizing the number of instructions in a program for computing a combinational function corresponds to that of finding a realization of the function using the smallest number of modules of given types. For example, the bubble transfer instruction can be represented by the module shown in Fig. 4a. The bubble location at which each output appears is given in parentheses. Note that fan-out is not

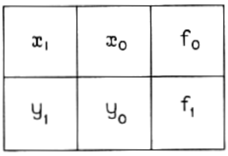

Fig. 3—Memory layout for program P'.

allowed in the realization. Bubble splitting is represented by the module of Fig. 4b. Instructions of the type $(x, y)_d$ and the conditional transfer $(x, y)z$ can be represented by modules of Fig. 4c and d respectively. Creation and annihilation of bubbles can be represented by appropriate constant inputs (0 or 1). If a realization of the function can be obtained using a subset of these module types, then the realization can be readily translated into a program which computes the function. All bubble locations used in the program will correspond to inputs of the circuit. Although efficient realizations of functions using only modules representing the allowed bubble interactions lead directly to efficient bubble programs, there are at present no known algorithms for the former. The following example shows how a program is obtained from a modular realization of a function.

Example 2: Consider the function $f = a\bar{c} + bcd + \bar{b}\bar{c}\bar{d}$, which is to be computed using the bubble transfer instruction only. Let each variable be represented by two locations. Thus, we wish to realize f and $\bar{f}$ using only modules of Fig. 4a, with no fan-out. The inputs to the circuit consist of the variables and their complements. $\bar{f} = c\bar{d} + \bar{b}c + \bar{a}\bar{c}d + \bar{a}b\bar{c} = c(\bar{b} + \bar{d}) + \bar{a}\bar{c}(b + d)$. We obtain the circuit of Fig. 5 (by trial) which realizes both f and $\bar{f}$. Translating each module of Fig. 5 to the corresponding instruction, we obtain the following program:

$$P = (b_0, d_0)\ (d_0, c_1)\ (b_1, d_1)\ (a_0, c_0)\ (d_1, a_0)\ (d_0, d_1)$$
$$(c_1, b_1)\ (b_0, a_0)\ (c_0, a_1)\ (b_0, c_0)\ (c_1, c_0).$$

At the end of the program f and $\bar{f}$ will be in c_0 and d_1 respectively. In translating the circuit to the program, no module should be translated to the corresponding instruction until the modules connected

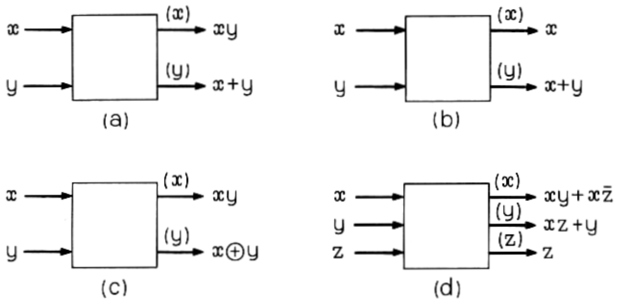

Fig. 4—Modules representing instructions: (a) Bubble transfer, (x, y). (b) Bubble splitting, $(x, y)_S$. (c) Type $(x, y)_d$. (d) Conditional transfer $(x, y)z$.

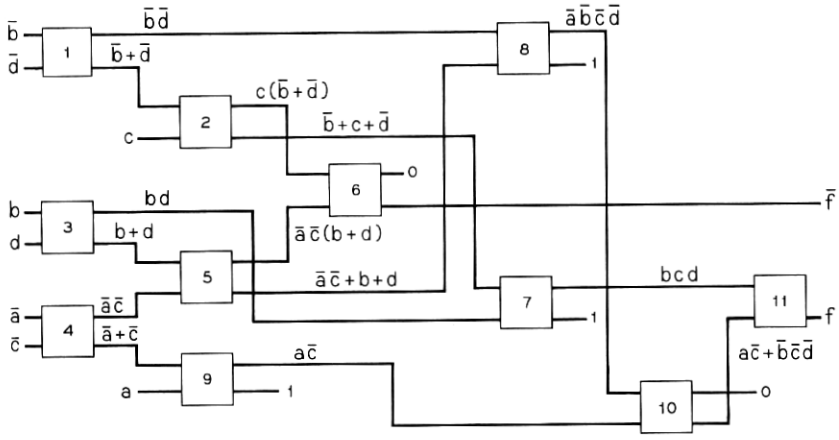

Fig. 5—Circuit for example 2.

to its input terminals have been translated. It is also important to observe the proper correspondence between bubble locations and the output terminals of a module. After applying an instruction (x, y), xy appears at location x, and $x + y$ appears at location y.

The other types of modules shown in Fig. 4 can be used in a similar manner for deriving programs using other types of instructions. However, there are no known algorithms for realizing any arbitrary function using only modules of a given type, so as to minimize the number of modules used. When such algorithms become available they can also be used for obtaining efficient programs of magnetic bubble interactions.

3.3 *Speed-up of Computations*

We have seen how a combinational function can be computed by different programs having different numbers of instructions and memory requirements. If the instructions in a program are executed one at a time, as was implicitly assumed so far, then the time for computing a function depends on the number of instructions in the program. However, it may be possible to speed up the computation by executing several instructions simultaneously.

An obvious method of obtaining parellelism in a program is by executing independent instructions simultaneously. Instructions that may be executed simultaneously can be determined from the circuit corresponding to the program, discussed in the preceding section. The modules in the circuit are arranged according to levels as follows: Modules whose inputs are only input variables (or their complements) are assigned to level 1. The level of any module is defined to be the smallest integer greater than the levels of the modules connected to its input terminals. With levels assigned to all modules in the circuit in this manner, the instructions corresponding to these modules are executed in the order of the level numbers, and the instructions corresponding to all modules in one level may be executed simultaneously. However this may necessitate a further restriction on a valid memory allocation scheme.

Example 3: Consider the realization of the function $f = a\bar{c} + bcd + \bar{b}\bar{c}\bar{d}$ in Fig. 5. The modules can be arranged according to levels as follows: level 1–(1, 3, 4); level 2–(2, 5, 9); level 3–(6, 7, 8); level 4–(10); level 5–(11). The program P can be executed in five steps as follows:

(*i*) (b_0, d_0) (b_1, d_1) (a_0, c_0).

(*ii*) (d_0, c_1) (d_1, a_0) (c_0, a_1).

(*iii*) (d_0, d_1) (c_1, b_1) (b_0, a_0).

(*iv*) (b_0, c_0).

(*v*) (c_1, c_0).

Another situation where simultaneous execution is possible becomes obvious by considering the instructions $(a, b)(a, c)$. If these instructions are executed simultaneously, the location a will contain abc, whereas the contents of locations b and c are indeterminate because they depend on which of the two instructions is actually executed first. However, if we are interested only in the product term abc, the two instructions may be executed simultaneously. In general, all $(n - 1)$ instructions for forming a product of n variables can be executed simultaneously. Similarly, the simultaneous execution of the instructions (a, s) (b, s) $\cdots$, leaves the sum $a + b + \cdots + s$ in location s with the contents of $a, b, \cdots$ indeterminate. Using this technique, the program of Example 3 can be executed in four steps by replacing the instructions of steps *iv* and *v* by $(b_0, c_0)(c_1, c_0)$, executed simultaneously. Instructions of the type $(x, y)_d$ can also be executed simultaneously with similar results. For instance, the simultaneous execution of $(a, b)_d$ and $(a, c)_d$ leaves the product abc in location a, with the contents of locations b and c indeterminate. Similarly, if $(a, c)_d$ and $(b, c)_d$ are executed simultaneously, the location c will contain $a \oplus b \oplus c$, but the contents of locations a and b will be indeterminate.

It may also be possible to speed up computations in a manner similar to the use of completion signals in asynchronous circuits.[5–7] A combinational function f can be expressed as a sum of disjoint minterms. The function can be computed by computing these minterms, one at a time. As soon as a 1 is generated, the computation is, in effect, complete because the value of the function is known to be 1. The computation can be terminated if the presence of a bubble in the output location can be detected by the control circuitry. The conditional transfer instruction $(x, y)z$ discussed earlier may be useful for this purpose.

3.4 *Iterative Array Realizations*

Our discussions so far have been restricted to realizations which minimize the number of instructions or the computation time. In this section, we shall consider some techniques for obtaining regular structures, which are capable of performing computations by the application of uniform magnetic fields. In such a structure, the same type of instruction is executed simultaneously in all parts by the application of a uniform magnetic field, the type of instruction being controlled possibly by the direction of the magnetic field.

Since regularity of interconnections is a highly desirable feature in integrated circuits also, a great deal of effort has been directed toward the realization of functions as iterative arrays.[8,9] Such arrays may be classified as uniform cell function arrays (in which the function realized by each cell is identical and different functions are realized by the array by changing the inputs to some cells) and nonuniform cell function arrays in which different cells realize different functions, depending on the position of the cell in the array. The operation of uniform cell function arrays can be simulated by magnetic bubble interactions with uniform magnetic fields.

The rectangular array shown in Fig. 6a may be used for realizing any combinational function of n variables by merely changing the connections to the last row of cells. A typical cell in the array is shown in Fig. 6b. It can be shown that the vertical outputs of the nth row consist of the 2^n minterms of n variables. Since at most one of the minterms will be 1 at any time, the last row will compute the sum (OR) of all minterms connected to it. Any function can be realized by connecting to the cells in the last row only those vertical outputs of the nth row that correspond to 1-points of the function.

Note that the number of cells in the array is $(n + 1)2^n$. The longest

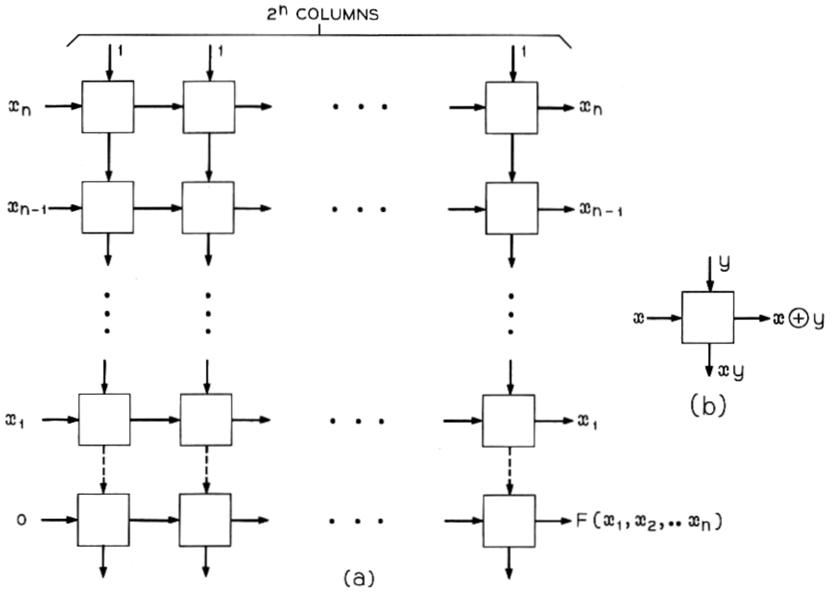

Fig. 6—(a) Rectangular array. (b) Typical cell in array.

path in the array is $2^n + n + 1$. Therefore, the time required for computing a function of n variables is proportional to $2^n + n + 1$.

The operation of this array can be simulated by magnetic bubble interactions as follows: Let each cell in the array be replaced by two bubble locations as shown in Fig. 7. Let the application of a magnetic field to a cell cause the execution of the instruction $(x, y)_d$ and transfer of the resultant bubbles as shown by the arrows in Fig. 7. (It may be necessary to apply a magnetic field in one direction followed by a field in a different direction to accomplish this.) Let the cells in the ith row and jth column be denoted by C_{ij}. If the left and right locations in C_{ij} represent x and y respectively, application of the field to C_{ij} results in xy in the right location of $C_{i+1,j}$ and $x \oplus y$ in the left location of $C_{i,j+1}$. Initially, we set the left locations of all cells in the first column, except the cell in the last row, to correspond to the values of the n variables. The left location of the first cell in the last row should be empty. The right locations of all cells in the first row initially contain bubbles. Paths from the nth row to the $(n + 1)$st row which correspond to minterms for which the function is 0 are inhibited. At $t = 1$, the field is applied to C_{11}. At $t = 2$ the field is applied to C_{12} and C_{21};

at $t = 3$ to C_{13}, C_{22}, and C_{31}; and so on. That is, at $t = k$, $1 \leqq k \leqq 2^n + n$, the field is applied to all cells C_{ij} such that $i + j = k + 1$. At $t = 2^n + n + 1$, the value of the function will be in the right location of $C_{n+1,2^n}$.

Instead of applying fields to successive diagonals, the fields may be applied to the entire platelet. Now, the right locations of all cells in the top row should be connected to "bubble generators," so that they always contain bubbles. The values of the n variables, contained in the left locations in the first column, may be changed every n units of time. The output corresponding to any input combination will appear at the output location $2^n + n + 1$ units of time after the application of the input combination. If input changes occur n units of time apart, the correct outputs will also appear n units of time apart. However, the contents of the output location will not be correct between these fixed instants of time. Therefore, it is necessary to read the output at the appropriate instants of time.

The preceding discussion shows how a uniform cell function iterative array can be simulated by magnetic bubbles by applying identical sequences of instructions to successive sets of bubble locations. This provides us with a systematic way of realizing all combinational functions. If conditional operations are permitted, nonuniform cell function arrays can also be simulated. Such arrays can be made smaller than uniform cell function arrays. The number of cells required for realizing any arbitrary function of n variables is proportional to 2^n with nonuniform arrays compared to $n \cdot 2^n$ required for uniform cell arrays.[8] However, the cells of the nonuniform array are likely to be more complex than those in a uniform array.

It is interesting to compare the size of array realizations to the size of realizations without any restrictions on the interconnection structure. Bounds on the latter will provide us with an indication of the space and time requirements for computing an arbitrary combina-

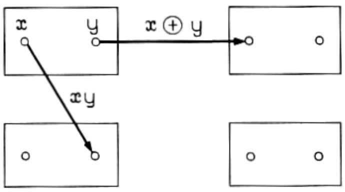

Fig. 7—Simulation of array by magnetic bubble interactions.

tional function with magnetic bubbles. D. E. Muller[10] has shown that for any set of elements capable of realizing all combinational functions, the number of elements N for realizing any arbitrary function of n variables satisfies the inequality

$$\frac{C_1 2^n}{n} \leqq N \leqq \frac{C_2 2^n}{n},$$

where C_1 and C_2 are constants whose values depend on the set of elements used. The proof of the upper bound is constructive and utilizes only one type of cell. It is outlined below because of its possible applicability to magnetic bubble computations.

The cell used realizes a function of three inputs $ab + \bar{a}c$. Any function of k variables can be expressed as

$$f(x_1, x_2, \cdots, x_k) = x_k f(x_1, x_2, \cdots, x_{k-1}, 1) + \bar{x}_k f(x_1, x_2, \cdots, x_{k-1}, 0).$$

If all functions of $k - 1$ variables are available, every function of k variables can be realized using exactly one cell. There are 2^{2^k} functions of k variables (which also include all functions of $k - 1$ variables, $k - 2$ variables, etc). Thus 2^{2^k} cells are sufficient for realizing all functions of k variables.

By expanding about any variable, the output function can be formed from two functions of $n - 1$ variables. Repeating the procedure, it may be formed from 2^{n-k} functions of k variables, using $2^{n-k} - 1$ elements. Since 2^{2^k} elements are sufficient for realizing all functions of k variables,

$$N = 2^{2^k} + 2^{n-k} - 1.$$

The value of k may now be chosen so as to minimize N. An approximate minimum is $N < C_2(2^n/n)$, for some constant C_2. Thus the bound on the number of cells required is greatly reduced for nonregular structures.

IV. SUMMARY

Three different mathematical models of magnetic bubble interactions were studied and each of them was shown to be sufficient for computing all combinational functions. Some methods of speeding up computations were presented. Geometrical patterns in which it is possible to move bubbles between any pair of data locations and also

minimize the maximum path length were examined. A uniform structure for computing arbitrary functions was also presented. These structures have the advantage that computations can be carried out by the application of uniform magnetic fields to the entire platelet.

REFERENCES

1. Bobeck, A. H., "Properties and Device Applications of Magnetic Domains in Orthoferrites," B.S.T.J., *46,* No. 8 (October 1967), pp. 1901–1925.
2. Bobeck, A. H., Fischer, R. F., Perneski, A. J., Remeika, J. P., and Van Uitert, L. G., "Applications of Orthoferrites to Domain-Wall Devices," IEEE Trans. Magnetics, *MAG-5,* No. 3 (September 1969), pp. 544–553.
3. Perneski, A. J., "Propagation of Cylindrical Magnetic Domains in Orthoferrites," IEEE Trans. Magnetics, *MAG-5,* No. 3 (September 1969), pp. 554–557.
4. Graham, R. L., "A Mathematical Study of a Model of Magnetic Domain Interactions," B.S.T.J., *49,* No. 8 (October 1970), pp. 1627–44.
5. Armstrong, D. B., Friedman, A. D., and Menon, P. R., "Design of Asynchronous Circuits Assuming Unbounded Gate Delays," IEEE Trans. Computers, *C-18,* No. 12 (December 1969), pp. 1110–1120.
6. Muller, D. E., "Asynchronous Logics and Application to Information Processing," Proc. Symposium on Application of Switching Theory in Space Technology, Stanford University Press (1963), pp. 289–297.
7. Miller, R. E., *Switching Theory,* Vol. 2, New York: John Wiley and Sons, 1965, Chap. 9, 10.
8. Minnick, R. C., "A Survey of Microcellular Research," J. ACM, *14,* No. 2 (April 1967), pp. 203–241.
9. Arnold, T. F., Tan, C. J., and Newborn, M. M., "Iteratively Realized Sequential Circuits," IEEE Trans. Computers, *C-19,* No. 1 (January 1970), pp. 54–66.
10. Muller, D. E., "Complexity of Electronic Switching Circuits," IRE Trans. Electronic Computers, *EC-5,* No. 1 (March 1956), pp. 15–19.