Direct Transition Memory and its Application in Computer Design

HOWARD A. SHOLL, MEMBER, IEEE

Abstract—This paper investigates a potential application of microprogrammable memories to the problem of sequential network synthesis and computer design. It is shown that by allowing a controllable amount of memory redundancy, a microprogrammed emulation of a state table can be organized such that decision branches in the microprogram are achieved in an immediate multiplex manner, without the need of additional branch steps in the microprogram, thus increasing operational speed. A design technique is developed which, for a given state table, allows a minimum number of memory address variables to be used while minimizing the dependence of the variables on both the input and current state information. This technique has immediate practical application in the design of sequential networks, and is shown to be feasible in the controller design of a general purpose computer.

Index Terms—Computer design, microprogramming, partition algebra, sequential network design, state assignment.

I. INTRODUCTION

THERE ARE two topics in the current field of logic and computer design that promise to have significant effect on the current and future designs of digital systems: the decreasing cost and increasing complexity of logic and memory components; and the use of microprogramming methods to achieve organization and flexibility in design [1]. Two areas of research involving microprogramming that reflect these trends are computer architecture, on a macroscopic level, and the logic design methods appropriate for designing microprogrammed control units, (or sequential networks) on a microscopic level; this paper is primarily concerned with the latter. The basic structure of a microprogrammed control unit and its variations have been established and used in a number of computer systems [2]-[4]. The basic structure is illustrated in Fig. 1. In this configuration the memory can be read only (static microprogramming) or read/write (dynamic microprogramming). For operation, a set of control signals is encoded into words of memory in a control information field, and a set of memory addresses (usually one) is written into an address information field to be used for branching operations. A microprogram then consists of a set of these words organized to solve a specific problem. The system executes at a starting address and sequences through words of memory, thus emitting a corresponding sequence of control signals. When a branch control signal occurs, the next address is selected from the address information field, conditioned by the status of the input data (for a conditional branch).

The logic design of microprogrammed controllers or sequential networks has two areas of concern: organization of a set of control signals into a control information field of a memory word; and implementation of a next address calculation technique. The first problem is concerned with a tradeoff between the number of bits required in the control information field versus the complexity of the corresponding decoder (combinational network A). The two extremes in potential coding procedures are called: horizontal microprogramming, in which each control signal is assigned an independent bit of the control information field—thus requiring no decoding; vertical microprogramming, in which the field is divided into operation code and operand subfields such that only one operation code can be specified per word—thus requiring entire decoding. Practical design tradeoffs generally allow some degree of parallelism (horizontal) between sets of operation code subfields (vertical). An additional technique that is sometimes used encodes a set of sequential operations into each word, requiring a multiple number of clock pulses per word to recover them in sequence. This technique allows the memory operation time to be longer for a fixed control signal rate, and requires a smaller word dimension at the cost of a larger bit dimension.

The coding problem of obtaining a minimum bit dimension in the control information field which retains a specified amount of parallelism was considered by Schwartz [5], and a solution to the problem was given by Grasselli and Montanari [6].

The second problem—that of implementing the next desirable memory address—has customarily been implemented by using the address information field to contain a set of branch address subfields, in conjunction with a set of branch microcontrol signals. The address calculation network then allows the current address to be incremented, if no branch microcontrol signals were generated, or substitutes a branch address field for the current address, if a branch microcontrol signal were generated. Often a single branch address field is used, thus allowing only binary decisions to be made.

The technique suggested in this paper proposes to use memory in a more redundant manner to gain processing speed—a practical incentive being the lowering of costs of memory components. This is accomplished by allowing the address calculation network to determine the next memory address as a direct (combinational) function of the input parameters and current address information. This approach obviates the need for branch microcontrol signals,
reduces the bit size of both the address information and control information fields, and allows immediate multiport branching. The advantages are a reduction in bit dimension and an increase in processing speed; the disadvantages are an address calculation network that is suitable for only a specific problem, and a word dimension redundancy since input parameters always affect the next address calculation—not just when a branch command is given—thus requiring additional copies of memory words.

This technique can also be envisioned as a means of realizing a hardware implementation of a decision table [8]—a finite state machine type of program organization. It should be emphasized at this point that, from a software viewpoint, the speed advantage to be gained results from two factors. First, since no branch or branch on condition instructions would exist in the machine implementation, the execution time is reduced by the amount of their fetch and execute times. Secondly, the ability to implement multiport decisions at any decision point in a program provides a means of replacing the typical binary decision chain of a program and recovering nearly its entire time cost. Clearly the amount of increase in processing speed is context dependent. From a hardware viewpoint, the technique provides a means of using memories to realize sequential networks without the time overhead caused by branching costs evident in more conventional microprogrammed designs. The intent of this paper is not to attempt to classify types of problems and the benefit they might receive, but rather to investigate the feasibility and characteristics of a proposed computing structure which has the above potential.

This paper considers such a direct transition microprogrammed computing structure, and presents a solution to the problem of controlling the tradeoff between the size of the word dimension and the complexity of the address calculation network required for a given microprogram. The results are directly applicable to sequential network design problems and are shown to be feasible in the design of the controller for a general purpose computer.

II. REPRESENTATION OF PROGRAMS AS FINITE STATE PROCEDURES

Since the starting point of the subsequent development is a state table, and since it is desirable to show its programming applicability, this section defines a programming system and its description by means of a state table. Consider a program, or routine, as a pure control procedure operating upon a data structure. The following definitions and algorithm illustrate a means of representing such a program in state table form.

Definition 1: A flow chart is a directed graph whose nodes consist of data operation elements, subroutine call elements, decision elements, terminal elements, and a starting node, and whose branches represent the flow of control.

Component Specification: A subroutine call element is identified by a circle whose label, d, represents the identity of the called subroutine.

A data operation element is identified by a rectangle whose label, z_i, represents a prespecified operation, where no decisions are required.

A decision element is identified by a polygon whose label, z_i; j = 0,1,...,n_i - 1, represents an n_i-valued variable for decision i.

A terminal point is identified by a triangle whose label is φ or r. φ is the empty set and indicates the end of a procedure; r is a return to the calling program if the procedure is a subroutine.

Definition 2: A programming system is a set of flow charts closed under subroutine calls.

We should notice that the definitions force all subroutines to be used in a conventional manner (i.e., entering and exiting must always be the call and return). Fig. 2 illustrates these ideas.

Using these definitions we can now describe an algorithm to represent any programming system by a set of state tables, where each state table represents a flow chart. The state table is described in Mealy form, where the outputs represent the data operations, and the inputs represent the encodings of the decision variables. It is
assumed that the operations are carried out by a lower
level process, and are just initiated by the state table.

Step 1: Tabulate all unique data operations as \( z_i \),
\( i = 1,2,\ldots,m \).

Step 2: Form a set of outputs, \( Z \), from the union of the \( m \)
operations from Step 1 and two operations whose func-
tions are to call (c) and return (r) from subroutines.

\[
Z = (z_1,z_2,\ldots,z_m)\).
\]

Step 3: Tabulate all decision variables for a specific
flow chart as \( x_i \).

\[
i = 1,2,\ldots,r.
\]

Step 4: Form a set of inputs, \( X \), from the decision
conditions of Step 3.

\[
X = (I_1,I_2,\ldots,I_{(m)}(n_2)\ldots(n_n))
\]

representing the \( (n_1)(n_2)\ldots(n_n) \) combinations of the
\( x_i \) variable values.

Step 5: Associate states with branches of the graph.

a) Let \( q_0 \) be the input (initial) state.

b) Assign a unique \( q_i \), \( i \neq 0 \), to each junction of

branches.

c) Assign a unique \( q_j \), \( j \neq i,0 \), to each remaining

branch excluding those at decision outputs. State

set \( Q = (q_0,q_1,\ldots,q_{n-1}) \).

Step 6: Specify the next state mapping \( \delta \) and output
mapping \( \omega \):

a) If there exists a path from \( q_i \) to \( q_j \) having one data

operation node \( z_k \) and no decision nodes:

\[
\delta(q_i,I_\lambda) = \begin{cases} q_j & \text{if } I_\lambda \text{ can exist in state } q_i \\
\text{don't care} & \text{otherwise} \end{cases}
\]

\[
\omega(q_i,I_\lambda) = \begin{cases} z_k & \text{if } I_\lambda \text{ can exist in state } q_i \\
\text{don't care} & \text{otherwise} \end{cases}
\]

b) If there exists a path from \( q_i \) to \( q_j \) having one data

operation node \( z_k \) and one decision node with \( x_n \).

\[
\delta(q_i,I_\lambda) = q_j
\]

\[
\omega(q_i,I_\lambda) = z_k
\]

for those \( I_\lambda \) which represent the value of \( x_n \).

Step 7: Set the next state mapping corresponding to the
terminal element to itself for all inputs. If \( q_{n-1} \) is the branch
to the terminal node,

\[
\delta(q_{n-1},I_\lambda) = q_{n-1}
\]

\[
\omega(q_{n-1},I_\lambda) = \begin{cases} \phi & \text{otherwise} \\
\text{for all } I_\lambda. \end{cases}
\]

Step 8: Repeat Steps 3–7 for each flow chart of the
programming system.

Theorem 1: All programming systems can be described
by a set of state tables.

Proof: Since all flow charts contain a finite number of
elements, implementation of the preceding algorithm will
create a state table for each flow chart, or routine.

It can be observed that we have shown only that the
control flow of a program is no more than the finite state
controller of a Turing machine. Also it should be clear at
this point that the state table representation, in terms of a
program, is intended to apply only to the control flow of the
machine level program and does not include the data
structure. A program is thus executed by the sequence of
data operations on a separate data structure that ensue
during state transitions.

III. REPRESENTATION OF A STATE TABLE
IN A DIRECT TRANSITION MEMORY

This section describes means of representing a state
table in memory, so that all next state transitions can be
direct functions of the input and current state. A cell of a
state table, identified by a current state and current input,
contains both the next state and output mappings. Thus
an association can be established between cells of a state
table and words of a memory in which each mapping (next
state and output) specified in the state table cell is
assigned a field in a memory word. A state table is defined
by sets of states \( (Q) \), inputs \( (X) \), and outputs \( (Z) \), and
by a next state \( (\delta) \) and output \( (\omega) \) mapping (Fig. 3).

There are then two approaches that can be considered.
1) We can assign a memory word for every defined cell in
the state table. 2) We can assign a memory word for every
uniquely defined cell in the state table.

In general, the first approach requires more memory
space, but will be shown to require an address calculation network which is universally manageable. The second approach effectively creates a Moore machine representation, with each Moore state represented by a memory word, and its state code represented by the memory address. Thus the address calculation network is of the same complexity as an ordinary logic component implementation of the state table, and the memory just serves as the state register. Obviously this latter approach is not desirable in most applications, since memory would not be effectively used. Henceforth the first approach will be assumed.

The next step is to decide how the addresses of the defined cells can be positioned in memory. In order to evaluate comparative approaches to this problem we can define a measure of memory redundancy and a measure of address calculation complexity.

Definition 3: Memory Redundancy, R, is the fractional part of required memory space which is not used in the representation.

\[
R = \frac{\text{amount of unused space}}{\text{total space required}}.
\]

Definition 4: Network Complexity, C, of a combinational network is the total number of input variable dependencies for all output variables relative to the maximum possible number of such dependencies.

\[
C = \sum_{i=1}^{s} \frac{|a_i|}{(S)(L)}
\]

where

\[
|a_i| = \text{number of variables of which } a_i \text{ is a function}
\]

\[
S = \text{number of output variables } a_i
\]

\[
L = \text{number of input variables } x_j
\]

\[x_ja_i \text{ will be assumed to be binary variables.}\]

Thus a nonredundant network has a maximum complexity of

\[
S \text{ terms}
\]

\[
C = \frac{L + L + \cdots + L}{(S)(L)} = 1.0
\]

(all output variables are functions of all input variables) and a minimum complexity of

\[
C = \frac{L}{(S)(L)} = \frac{1}{S} = \frac{1}{L}
\]

(all input variables are only represented once).

Consider an m-input, n-state table in each of the three following address assignment strategies.

A. Minimal Complexity Assignment

Let each of the S address variables be a direct function of an input or state variable, then

\[
S = M + N, \quad \text{where } M = \lceil \log_2 m \rceil, N = \lceil \log_2 n \rceil
\]

\[
C = \frac{1}{S}
\]

\[
R = 1 - \frac{p}{2^{m+n}},
\]

where \(p\) = number of defined state table cells.

While this assignment minimizes the address calculation network (reduces it to straight-through connections), it may greatly increase redundancy, as shown by the following theorem.

Theorem 2: A minimal complexity assignment for a state and input reduced state table has a limiting worse case memory redundancy of 1.0 as the state table size increases.

Proof: The proof consists of two parts.

1) Determining the smallest number of possible unique cells in a state table.
2) Evaluate \(R\) as \(m\) or \(n\) approaches infinity.

Part 1: The following three characteristics are evident in all state and input-reduced state tables.

1) Each column and row has at least one entry. (Otherwise that column or row is compatible with all the others.)
2) There must exist for each unique pair of columns (inputs) at least one row (state) with entries in those columns. (Otherwise that pair of columns would be compatible.)
3) Characteristic 2 with rows/columns interchanged. Thus the smallest number of cells \(p\) that can exist
with these characteristics is \( p = m + n - 1 \) where \( m \) cells are positioned in a row (to satisfy 2) and \( n - 1 \) additional cells are positioned in a column (to satisfy 3). Any other arrangement would require more total entries.

**Part 2:**

\[
R = 1 - \frac{p}{2^{M+N}}.
\]

If we represent \( m, n \) by \( m = r_m2^{m'}; n = r_n2^{n'} \) where \( 1 \leq r_m, r_n < 2 \), and \( m', n' \) are appropriate constants then \( 2^{M+N} = (mn)2^{m+n}2^c \) where \( c = [\log_2 r_m] + [\log_2 r_n] = 0, 1, 2 \). Since \( p = m + n - 1, R = 1 - ((m + n - 1)/mn)[K] \) where \( K = (r_m2^{m'}/2^c) \)

\[
\lim_{n \to \infty} R = 1.0.
\]

**B. Minimal Redundancy Assignment**

If we force each defined cell to lie in adjacent positions (addresses) in memory, then: \( R = 0, \) all required space is used, and

\[
C = \sum_{i=1}^{s} |a_i|/S(M+N) \quad \text{where} \quad S = [\log_2 p].
\]

This assignment minimizes required memory space, but places a restriction on the memory address coding assignment; thus we expect the complexity of the address calculation network to be greater than in the approach below.

**C. Minimal Address Variable Assignment**

This approach minimizes the number of address variables required for a given state table, but allows address coding to assume any value. The effect is that the smallest \( 2^p \) memory space is required for a state table. \( S = [\log_2 p], S \) is the number of address variables, and \( p \) is the number of defined state table cells.

\[
C = \sum_{i=1}^{s} |a_i|/S(M+N) \quad \text{where} \quad R = 1 - \frac{p}{2^s}.
\]

It will be shown that this approach maintains redundancy below 0.5 while limiting complexity in a manageable manner.

**Theorem 3:** A minimal address variable assignment for a state and input reduced state table has a memory redundancy \( 0 \leq R < 0.5. \)

**Proof:**

\[
R = 1 - \frac{p}{2^{[\log_2 p]}}.
\]

Let \( p = r_p2^p \) where \( 1 \leq r_p < 2 \). Substituting:

\[
R = 1 - \frac{r_p2^p}{2^{[\log_2 r_p]2^p}} = 1 - \frac{r_p}{2^{[\log_2 r_p]}}
\]

for \( r_p = 1, R = 0; \) for \( 1 < r_p < 2, [\log_2 r_p] = 1. \) Thus \( 0 \leq R < 0.5 \). Henceforth, the minimum address variable assignment will be assumed.

The basic problem, now, is to determine a minimum complexity address calculation network and the related assignments for input, state, and address variables, given an unassigned state table and a minimum address variable assignment approach. The remainder of this section formulates a solution in terms of partition algebra. It is assumed that the reader is familiar with the application of partition algebra to the state assignment problem as described by Hartmanis and Stearns [7]. Assume a state table is given as described in Fig. 3, and that the outputs can be handled as described by [6].

Let the state, input, and address variables be \( y_i, x_j, a_k \), respectively. Let \( \tau_k \) be a partition corresponding to the coding assigned to state variable \( y_i \), \( i = 1, 2, \ldots [\log_2 n] \) for \( n \) states. Let \( \tau_j \) be a partition corresponding to the coding assigned to input variable \( x_j \), \( j = 1, 2, \ldots [\log_2 p] \) for \( p \) defined cells. To achieve a useful minimal dependence assignment for the address variables as a function of the input and state variables, the codings must satisfy two criteria: uniqueness and minimal dependence. The uniqueness requirement forces each element in the state, input, and address sets to have a unique code.

1) Input assignment:

\[
\prod_{i=1}^{[\log_2 n]} \tau_{y_i} = (y_0)(X_1) \cdots (X_{n-1}).
\]

2) Next state assignment:

\[
\prod_{j=1}^{[\log_2 p]} \tau_{y_j} = (y_0x_0)(q_1X_1) \cdots (q_{n-1}X_{n-1}).
\]

3) Address assignment:

\[
\prod_{k=1}^{[\log_2 p]} \tau_k = (q_0X_0)(q_0X_1) \cdots (q_{n-1}X_{n-1}).
\]

The minimal dependence requirement forces

\[
\sum_{k=1}^{[\log_2 p]} \mid a_k \mid
\]

to be minimum. In partition terms, we can define a partition mapping \( \tau = m(\pi) \) where \( m(\pi) \) is the partition formed by the mapping of the blocks of \( \pi \) onto the address cells of a state table, and a corresponding partition pair \( (\pi, \tau) \) such that \( \tau = m(\pi) \). Let \( n_\pi = \{m(\pi_m), m(\pi_m)\} \).

For minimal dependence, select \( \tau_k, \tau_k \geq a \) a product of \( n_\pi \) partitions, \( n_k \subseteq n_\pi \) such that

\[
\sum_{k=1}^{[\log_2 p]} \mid n_k \mid
\]

is minimized.

The requirements can be satisfied by forming the list of partition pairs \( (\pi, \tau) \) for each unique 2-block input and state partition, and selecting exhaustively through the list for a minimal dependence solution.
Example 1:

<table>
<thead>
<tr>
<th>present state</th>
<th>input</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 2 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 4 3 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In this example, 11 defined cells are specified, requiring 4 address variables. In addition 2 input variables and 3 state variables are required. Thus the address calculation network compresses five variables to four.

Partition pairs:

<table>
<thead>
<tr>
<th>input</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(ab) (c)</td>
<td>(0a,2a,3a,0b,1b,2b,3b,4b,4c) (0c,3c,4c)</td>
</tr>
<tr>
<td>(ac) (b)</td>
<td>(0a,2a,3a,0c,3c,4c) (0b,1b,2b,3b,4b)</td>
</tr>
<tr>
<td>(a) (bc)</td>
<td>(0a,2a,3a) (0b,1b,2b,3b,4b,0c,3c,4c)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0) (1234)</td>
<td>(0a,0b,0c,1b,2a,2b,3a,3b,4b,4c)</td>
</tr>
<tr>
<td>(1) (0234)</td>
<td>(1b,0a,0b,0c,2a,2b,3a,3b,4b,4c)</td>
</tr>
<tr>
<td>(2) (0134)</td>
<td>(2a,2b) (0a,0b,0c,1b,3a,3b,4b,4c)</td>
</tr>
<tr>
<td>(3) (0124)</td>
<td>(3a,3b,3c) (0a,0b,0c,1b,2a,2b,4b,4c)</td>
</tr>
<tr>
<td>(4) (0123)</td>
<td>(4b,4c) (0a,0b,0c,1b,2a,2b,3a,3b,4c)</td>
</tr>
<tr>
<td>(01) (234)</td>
<td>(0a,0b,0c,1b) (2a,2b,3a,3b,4b,4c)</td>
</tr>
<tr>
<td>(02) (134)</td>
<td>(0a,0b,0c,3a,2a,2b) (1b,3a,3b,4b,4c)</td>
</tr>
<tr>
<td>(03) (124)</td>
<td>(0a,0b,0c,3a,3b,3c) (1b,2a,2b,4b,4c)</td>
</tr>
<tr>
<td>(04) (123)</td>
<td>(0a,0b,0c,4b,4c) (1b,2a,2b,3a,3b,3c)</td>
</tr>
<tr>
<td>(12) (034)</td>
<td>(1b,2a,2b) (0a,0b,0c,3a,3b,4b,4c)</td>
</tr>
<tr>
<td>(13) (024)</td>
<td>(1b,3a,3b,3c) (0a,0b,0c,2a,2b,4b,4c)</td>
</tr>
<tr>
<td>(14) (023)</td>
<td>(1b,4b,4c) (0a,0b,0c,2a,2b,3a,3b,3c)</td>
</tr>
<tr>
<td>(23) (014)</td>
<td>(2a,2b,3a,3b,3c) (0a,0b,0c,1b,4b,4c)</td>
</tr>
<tr>
<td>(24) (013)</td>
<td>(2a,2b,4b,4c) (0a,0b,0c,1b,3a,3b,3c)</td>
</tr>
<tr>
<td>(24) (012)</td>
<td>(3a,3b,3c,4b,4c) (0a,0b,0c,1b,2a,2b)</td>
</tr>
</tbody>
</table>

It is now necessary to search the two lists of partitions to find a set satisfying the requirement for minimal dependence. Several equally good solutions may exist; the decision of which of the selected partitions to apply to a specific variable is arbitrary.

One minimal dependence solution is then

- \( \pi_{x_1} = (ac)(b) \)
- \( \pi_{x_2} = (ab)(c) \)
- \( \pi_{x_3} = (34)(012) \)
- \( \pi_{x_4} = (03)(124) \)
- \( \pi_{x_5} = (01)(234) \)

Applying to \( m(\pi_{x_1}) = (0a,2a,3a,0c,3c,4c) (0b,1b,2b,3b,4b) \)

\( \tau_1 = m(\pi_{x_1}) = (0a,0b,0c,3a,3b,3c) (1b,2a,2b,4b,4c) \)

\( \tau_4 > m(\pi_{x_4}) \cdot m(\pi_{x_4}) \)

let

\( \tau_4 = (0a,0b,1b,3c,4c) (2a,3a,2b,3b,4b,0c) \).

The logic design of the network is then

- \( a_1 = x_1 \)
- \( a_2 = y_1 \)
- \( a_3 = y_2 \)
- \( a_4 = x_2 \oplus y_4 \), where \( \oplus \) is exclusive or.

Since each of the state, input variables are represented only once, the assignment has minimal dependence. The final structure of the design is shown in Fig. 4.

IV. PARTITION SELECTION PROCESS

This section describes a partition selection procedure which can be applied in a graphical manner, and which avoids the formation of the lists of partition pairs described previously. This approach allows a complete solution to be defined for completely specified state tables, and is applicable in a heuristic manner for incompletely specified state tables.

Assume we have a completely specified state table with \( m \) inputs and \( n \) states (Fig. 5). For a single dependent address variable a partition on the \( m \) inputs is mapped into a partition of the address space by drawing a vertical line dividing the address space into the blocks \( \alpha_i, \beta_i \) indicated. The corresponding logic implementation is just a straight through connection. In the double dependent case, two types of mappings are possible, one corresponding to a logical AND and the other to an EXCLUSIVE OR. The following observations can now be made.

Observation 1: Initially, before assignment, all rows (and all columns) are interchangeable for a given partition.

Proof: Since the assignment is independent of state table content, and the state table is completely full, all rows are equivalent and all columns are equivalent.

Observation 2: For an unassigned table, all double dependent address variables are dependent upon one state variable (row) and one input variable (column).

Proof: Assume a double dependent assignment is made for row only or column only dependence. The four blocks from which the address partition would be formed are then four disjoint row or column sections. Formation of the address partition would then yield two column or row sections. However, since observation 1 is true, any such assignment can be reduced to a single dependence case by interchanging columns or rows to merge the two blocks together. Therefore no such double dependence can exist.

Definition 5: \( k \)-Decomposable State Table: A state table is called \( k \)-decomposable if its minimum variable address calculation network can contain \( k \) single dependent variables.
Definition 6: Completely Decomposable State Table: A state table is called completely decomposable if its minimum variable address calculation network can be completely realized by \( k \) single dependent variables.

Definition 7: Nondecomposable State Table: A state table which is 0-decomposable is called nondecomposable.

Theorem 4: A state table is completely decomposable with \( k \) address variables if \( \lceil \log_2 m \rceil + \lceil \log_2 n \rceil - \lceil \log_2 mn \rceil = 0 \) where \( k = \lceil \log_2 mn \rceil \).

*Proof:* If the above condition is true, then the required number of address variables is equal to the sum of state and input variables and the assignment has minimum complexity.

Corollary 1: All completely specified state tables in which \( m \) or \( n \) is a power of two are completely decomposable.

*Proof:* By substitution.

We will next show that, given a state table we can decompose it into an array of \( 2^s \) subtables by sequentially extracting \( k \) single dependent address variables. In order to be able to apply this result in a recursive manner, we must show that a single dependent address variable can be described as a composite of separate partitions on an array of subtables. We can first describe the structure of a state table, \( T \), after it has been partitioned (see Fig. 6) by describing each block of \( T \) as a subtable \( T_{ij} \) whose input dimension is \( m_i \) and whose state dimension is \( n_i \).

Theorem 5: A state table, \( T \), is recursively decomposable if one of the following conditions holds for all subtables, \( T_{ij} \).

Condition 1: \( m_j \) is even.

Condition 2: \( n_i \) is even.

Condition 3: \( \lceil (m_j + 1)/2 \rceil n_i \leq 2^{k_{ij}-1} \).

Condition 4: \( \lceil (n_i + 1)/2 \rceil m_j \leq 2^{k_{ij}-1} \)

where

\[
1 \leq i \leq 2^{p_m-n}, \quad 1 \leq j \leq 2^{p_n-n},
\]

\[
k_{ij} = \lceil \log_2 mn \rceil - p_m - p_n.
\]

\( p_m \) = the number of single dependent input variables already selected.

\( p_n \) = the number of single dependent state variables already selected.

\( m_n, n_n \) = the number of empty blocks occurring along the \( m, n \) dimension.

*Proof:* We must first show that the uniqueness requirement can be preserved for both the address variable partition and its single independent variable partition. Then we can show that the independently selected parti-
Partition Mapping
\[
\pi_x = (\pi_0)_{(a_1)} \quad \pi_x = (\pi_1)_{(a_1)}
\]

Logic Equation
\[ a_1 = \pi_1 \]

Logic Implementation
\[
\begin{array}{c}
\text{a}1 \\
\end{array} \quad \begin{array}{c}
\text{a}1 \\
\end{array}
\]

Example of a decomposed state table.

Fig. 5. (a) Graphical interpretation of single. (b) Double dependent address variable assignments.

Fig. 6. Example of a decomposed state table.

Definitions on the subtables are sufficient to determine a composite partition on the state table. Consider a subtable with dimensions \( n_i, m_j, k_{ij} \) remaining address variables, where \( k_{ij} = \lfloor \log_2 m_j n_i \rfloor \). For 1-decomposability, a single dependent variable must be able to partition the address space into two blocks such that the magnitude of the largest block can be subsequently divided by the remaining address variables.

\[
\max (|a_0^1|, |a_1^1|) \leq 2^{k_{ij}}.
\]

If \( m(n) \) is even, the partition which clearly allows uniqueness for the \( m(n) \) dimension is \( m/2 \) \((n/2)\). Thus

\[
\max (|a_0^1|, |a_1^1|) = mn/2.
\]

If \( m, n \) are odd,

\[
\max (|a_0^1|, |a_1^1|) = \frac{1}{2}(m + 1)n, \frac{1}{2}(n + 1)m.
\]

Again it follows that a partition of \((m + 1)/2\) \((m - 1)/2\) satisfies the uniqueness requirements. We must show that

\[
\frac{m + 1}{2} \leq 2^\log_2 m - 1.
\]

Since the worst case occurs for

\[
m = 2^\log_2 m - 1 - 1,
\]

the requirement is clearly satisfied.

Now we must show that a single dependent composite partition can be determined from a set of partitions formed on subtables. Assume we have the state table shown in Fig. 5(a), which has two subtables—blocks of the address assignment partition—\( a_0^1, a_1^1 \). Assume each subtable \( T_i, j = 1, 2 \) is independently considered and partitioned into two subtables \( a_j, \beta_j \) along the \( m \) axis. We can then form the following composite address assignment partition, and the corresponding \( m \)-dimension partition.

\[
(a_0^2)(a_1^2) = (a_1 \lor a_2)(\beta_1 \lor \beta_2) \quad \text{or} \quad (a_1 \lor \beta_2)(a_2 \lor \beta_1).
\]
By induction this result applies to a state table with any number of subtables. It is clear that in order to form a composite address assignment partition all subtables must be decomposable. Since a unit cell state table is recursively decomposable, and might exist as a subtable, empty blocks $m_a n_0$ can occur for these subtable partitions as $(a) (f)$.

Applying Theorem 5, we can extract all single dependent variables from a state table for a minimum address variable assignment. At this point we have an array of subtables, at least one of which is nondecomposable (Fig. 6).

**Lemma 1**: All nondecomposable state tables have odd dimensions.

**Proof**: From Theorem 5, if a dimension were even, then the state table would become recursively decomposable.

**Lemma 2**: The difference, $\Delta$, in subtable sizes, along either dimension, is $\Delta \leq 1$ for all subtables $T_{ij}$ of a state table, $T$, where $\Delta = \max \, \Delta_{ij}$, $1 \leq i, j \leq p$, where $p$ is the number of subtables along a dimension, $m$, $\Delta_{i,j} = |b_i - b_j|$, and $b_i$ is the dimension of subtable $i$ along dimension $m$.

$$m = \sum_{i=1}^{p} b_i.$$  

**Case 1**: $m = 2^r$ then $b_i = m / p$, $\Delta_{ij} = 0$ for all $i, j$ and $\Delta = 0$.

**Case 2**: $m$ is an odd number. On the first decomposition

$$b_1 = \frac{m - 1}{2}, \quad b_2 = \frac{m + 1}{2}, \quad \Delta = 1.$$  

a) If $(m - 1)/2$ is even; the next decomposition is

$$b_1, \quad b_2 = \frac{m - 1}{4}, \quad b_3 = \frac{(m + 1)/2 + 1}{2},$$  

$$b_4 = \frac{(m + 1)/2 - 1}{2}, \quad \Delta = 1.$$  

b) If $(m - 1)/2$ is odd, by substitution $\Delta = 1$. By induction, Case 2 is applicable to all subsequent decompositions.

**Case 3**: $m$ is an even number. $m$ can be represented by $m = 2m_s$, where $m_s$ is an odd number; thus Cases 1 and 2 hold.

**Lemma 3**: Only one unique nondecomposable subtable will exist for a given state table.

**Proof**: Since nondecomposable state tables have odd dimensions, and $\Delta \leq 1$, only one odd number can occur along a dimension.

**Lemma 4**: All occurrences of a unique nondecomposable subtable of a state table, $T$, have the same orientation.

**Proof**: Same as Lemma 2.

**Lemma 5**: The unique nondecomposable subtable is always the largest of any subtable of a state table, $T$, along both dimensions.

**Proof**: By contradiction. If this were not the case there would exist an even and odd numbered subtable along a dimension, say $n$. Then the nondecomposable subtable would have an area of $[(n - 1)/2]m$ and the other subtable an area of $[(n + 1)/2]m$.

To be nondecomposable

$$\left(\frac{n - 1}{2}\right) m < 2^{\lceil \log_2 mn \rceil - 1}$$

and obviously

$$\left(\frac{n + 1}{2}\right) m < 2^{\lceil \log_2 mn \rceil - 1}.$$  

But then

$$\left(\frac{n - 1}{2}\right) m + \left(\frac{n + 1}{2}\right) m < 2^{\lceil \log_2 mn \rceil}, \quad mn < 2^{\lceil \log_2 mn \rceil}$$

which is contradictory.

**Lemma 6**: Only the nondecomposable subtable need be considered to complete the selection of address assignment partitions for the state table.

**Proof**: By Lemma 4, we know that all $k$-decomposable ($k > 0$) subtables are smaller than the nondecomposable subtable along both dimensions. Thus an assignment for the nondecomposable subtable is also an assignment for the $k$-decomposable subtables. (The decomposable subtable can be pictured as a subset of the nondecomposable table.)

These results now allow us to extract all possible single dependent variables for the assignment by a series of simple tests upon the dimensional sizes of the state tables. The problem is thus reduced to one of finding a double, or more, —dependent solution to just the nondecomposable subtables. The data shown in Table I were determined for tables with dimensions through 32, using the prior results.

In order to complete this solution, it is necessary to find minimal dependence assignments for the nondecomposable state tables. Although the properties described below simplify the assignment process, some searching is necessary to find a final solution.

**TABLE I**

**Nondecomposable State Tables for Dimensions from 2 to 32**

<table>
<thead>
<tr>
<th>NONDECOMPOSABLE TABLE</th>
<th>NUMBER OF OCCURRENCES</th>
<th>% OCCURRENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 x 5</td>
<td>161</td>
<td>16.7%</td>
</tr>
<tr>
<td>3 x 11</td>
<td>11</td>
<td>1.2%</td>
</tr>
<tr>
<td>5 x 25</td>
<td>6</td>
<td>0.6%</td>
</tr>
<tr>
<td>7 x 9</td>
<td>42</td>
<td>4.4%</td>
</tr>
<tr>
<td>11 x 11</td>
<td>6</td>
<td>0.6%</td>
</tr>
<tr>
<td>11 x 23</td>
<td>4</td>
<td>0.4%</td>
</tr>
<tr>
<td>13 x 19</td>
<td>4</td>
<td>0.4%</td>
</tr>
<tr>
<td>15 x 17</td>
<td>6</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

**TOTAL NUMBER OF TABLES**

**TOTAL NUMBER OF DISTINCT TABLES**

**TOTAL NUMBER OF COMPLETELY DECOMPOSABLE TABLES**

**TOTAL NUMBER OF UNIQUE NONDECOMPOSABLE TABLES**
Lemma 7: The maximum possible address variable reduction is 1.

Proof: This requires verifying \( \max (\log_2 m' + \log_2 n' - \log_2 mn) = 1 \) for arbitrary \( m,n \). Let \( m = 2^M - i_m, \ n = 2^N - i_n \), where \( M(N) = \log_2 m(n) \). \( 0 < i_m(n) < 2^{M(N)} - 1 \) and \( \log_2 mn = \log_2 (2^M - i_m) + \log_2 (2^N - i_n) \). Substituting, max \( (M + N - \lceil r_m + r_n \rceil) \). If \( i_m(n) \to 0; \ (r_m + r_n) \to M + N. \) If \( i_m(n) \to 2^{M(N)} - 1; \ (r_m + r_n) \to M + N - 2. \) Thus \( 0 \leq \log_2 m' + \log_2 n' - \log_2 mn < 2 \), and max \( [\log_2 m' + \log_2 n' - \log_2 mn] = 1. \)

We can now summarize the properties of \( m,n \) for the nondecomposable state tables.

1) \( \log_2 m' + \log_2 n' - \log_2 mn = 1. \)
2) \( m,n \) odd.
3) \( \left( \frac{m + 1}{2} \right) n > 2^{\log_2 mn - 1}. \)
4) \( \left( \frac{n + 1}{2} \right) m > 2^{\log_2 mn - 1}. \)

Theorem 6: A double dependent AND partition with block sizes of \( 2^{\log_2 mn - 1} \), \( mn - 2^{\log_2 mn - 1} \)
can be formed from nondecomposable state tables by selecting dimensional partitions having block sizes of

\[
m_0(n_0) = 2^{\log_2 m(n) - 1}, \quad m_1(n_1) = m(n) - 2^{\log_2 m(n) - 1}.
\]

Proof: This requires the following to be true:

\[
(2^{\log_2 m - 1})(2^{\log_2 n - 1}) = 2^{\log_2 mn - 1}.
\]

Reducing,

\[
\log_2 m' + \log_2 n' = \log_2 mn + 1,
\]

which was shown by Lemma 7.

At this point we can consider the possibility of forming subsequent AND partitions. Fig. 7 shows the two types of subsequent AND partitions that might be possible. It is easy to show that any other arrangement is equivalent to these by just interchanging rows and columns. The problem now is to determine whether we can always establish a type 1 or 2 partition for nondecomposable subtables, and if not, what conditions are necessary. Obviously, we just need to calculate the block areas that are created in the \( m,n \) area and establish the conditions for them to be divided into the proper size. The following theorem summarizes the results.

Theorem 7: A second AND partition can be formed on a nondecomposable subtable if the following conditions are satisfied \( (m > n) \).

Condition 1:

\[
n - \frac{3N}{4} \leq v_n \leq \frac{1}{n - m} \left( M - n \left( \frac{M}{2} \right) \right).
\]

Condition 2: Same as Condition 1 with \( M,m,v_n \) exchanged for \( N,n,v_n \) where \( M(N) = 2^{\log_2 m(n)} \) and \( 0 \leq v_m \leq m(n) \leq MN/2 \).

Proof:

1) Because of the input and state coding restrictions \( (\log_2 m' \text{ input, } \log_2 n' \text{ state variables}) \), a subsequent AND partition must be a mapping of only additional partitions that will divide the \( m,n \) area by two. In addition, each new input or state partition must divide \( m_0 \) or \( n_0 \) by two. Thus only the two types shown are possible.

2) The restriction on \( v_n,v_m \) forces the \( m,n \) areas to be further partitioned such that the subsequent block areas can be completed subdivided by the remaining variables.

We notice that if we were to continue this policy, we would, for each new partition, generate an additional set of simultaneous inequalities, whose solution requires an iterative, or exhaustive tabulation. Actual implementation of this procedure is more easily done graphically by visual inspection and tabulation of block sizes. We can also notice that there exist nondecomposable state tables from which only one AND partition can be formed \( (1 \times 11) \).

This same approach can be applied to XOR partitions, and alternate selections of AND, XOR partitions, and the same type of results apply.

Theorem 8: A double dependent XOR partition with block sizes of \( (mn + 1)/2 \) and \( (mn - 1)/2 \) can be formed from nondecomposable state tables by selecting dimensional partitions having block sizes of

\[
m_0(n_0) = \frac{m(n) + 1}{2}, \quad m_1(n_1) = \frac{m(n) - 1}{2}.
\]

Proof: By direct calculation the difference in address block size is 1, clearly guaranteeing that further partitioning can be attempted.

Using these results, and visually applying the partitioning procedure the data in Table II were obtained for the set of eight nondecomposable state tables of dimensions 32 or less.

For state tables \( 3 \times 5, 9 \times 7, 15 \times 17 \) the assignment solutions need only double dependence, and are shown in the Appendix. Dependence properties in excess of double dependence were not investigated.
TABLE II
MINIMAL DEPENDENCE ASSIGNMENTS OF NONDECOMPOSABLE STATE TABLES WITH DIMENSIONS OF 32 OR LESS

<table>
<thead>
<tr>
<th>STATE TABLE DIMENSIONS</th>
<th>MAXIMUM NUMBER OF DOUBLE DEPENDENT PARTITIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 x 5</td>
<td>4</td>
</tr>
<tr>
<td>3 x 21</td>
<td>4</td>
</tr>
<tr>
<td>5 x 25</td>
<td>5</td>
</tr>
<tr>
<td>9 x 7</td>
<td>6</td>
</tr>
<tr>
<td>11 x 11</td>
<td>4</td>
</tr>
<tr>
<td>11 x 23</td>
<td>4</td>
</tr>
<tr>
<td>13 x 19</td>
<td>6</td>
</tr>
<tr>
<td>15 x 17</td>
<td>8</td>
</tr>
</tbody>
</table>

Example 2: Partitioning of the 3 x 5 nondecomposable state table.

Step 1 Theorem 6
(AND)

Step 2 Theorem 7
(AND)

Step 3 (XOR)

Step 4 (XOR)

From these results, we notice that of the 961 possible state tables, 75 percent of the cases are completely decomposable. Also, the remaining 25 percent of the cases are split into 21.7 percent requiring no more than double dependence (3 x 5, 7 x 9, 15 x 17) and 33.3 percent requiring more than double dependence. Overall, 96.7 percent of the possible cases can be handled with no more than double dependent variables, where the double dependence function is either an AND or EXCLUSIVE OR. We also note that by adding a redundant row or column to the remaining nondecomposable tables, a double dependence solution is possible, and redundancy is increased to no more than 0.527 (11 x 11 nondecomposable table).

VI. INCOMPLETELY SPECIFIED STATE TABLES

Even though the prior results apply only to completely specified state tables, the graphical partitioning process can be applied to incompletely specified tables by just considering the nonspecified cell positions as ones which do not have to be partitioned. Let $p$ be the number of specified cells, and let $m, n$ be the state table dimensions.

1) If $\lceil \log_2 mn \rceil = \lceil \log_2 p \rceil$ the completely specified solution can be used, but may not be a minimal dependence solution.

2) If $\lceil \log_2 mn \rceil > \lceil \log_2 p \rceil$ the memory word dimension can be reduced beyond the completely specified case.

Theorem 9: An incompletely specified state table, $T$, is recursively decomposable if the following conditions hold.

Condition 1: Each column of subtables, $T_{ij}$, with number of defined cells, $P_{ij}$ for $i = 1, 2, \ldots, 2^m - n$ must be partitionable into subtables $T_{ij}^k, T_{ij}^l$ by a single dependent variable such that for each new subtable

$$P_{ij}^k \leq 2^{k-1}$$

and

$$P_{ij}^l \leq 2^{k-1}$$

Condition 2: Each row of subtables, $T_{ij}$, with number of defined cells, $P_{ij}$ for $j = 1, 2, \ldots, 2^n - m$ must be partitionable into subtables $T_{ij}^k, T_{ij}^l$ by a single dependent variable such that for each new subtable

$$P_{ij}^k \leq 2^{k-1}$$

and

$$P_{ij}^l \leq 2^{k-1}.$$
The address register of the control memory has two fields: one is \( N \) bits long for next state information; the other is \( M \) bits long and exists as a programmer-controlled input register. The universal logic network performs two functions. Each output address variable may be a double-dependent function of an address and state variable. One of \( N \) and one of \( M \) selector-multiplexers direct desired state and input variables to the double-dependent functions. A logic register specifies the function (one of sixteen) for each double-dependent logic element, and also the wiring connections for the multiplexers. Since each state table, or subroutine, would require a separate logic function, a separate memory of logic functions would be necessary to allow rapid modification of the logic register for subroutine linkages.

It is useful to note that the cost of the universal logic network is quite low. A universal combinational function of two variables is realizable as \( \frac{1}{2} \) of a 74153 logic chip. Also, complete multiplexers are available as single chips.

The production of coding for such a machine then requires, in effect, a hardware-like design for each state table—undesirably complex for most programmers. However, an assembler, using the results of Sections III and V, can be designed to automate the production of machine code from a mnemonic language more amenable to programmers. The code produced for such a system would consist of two parts. A state table encoding of the control structure of a program (for the program memory) and the coding of nonbranching sequences of data operations (for the primitive memory).

The processor organization is compatible with conventional system design techniques such as instruction pipelining, cache memories, virtual memory. Since the flow of control depends only upon the state and input (decision) information, the next control state can be determined while the current output (data operation is) is executing, until a data operation modifies the decision information (corresponding to a conditional branch). Thus conventional pipeline techniques are possible. A base register organization allows state table representation to be localized in pages, whose address position in memory is then controllable. Clearly long programs can be divided into a set of interactive state tables. Thus page-based cache memory and virtual memory organizations are quite consistent with the described computing structure.

It should be pointed out that, while horizontally microprogrammed systems of conventional design can sometimes achieve similar branching properties in specific instances, they can and do not achieve these properties in general. For example, it is often possible to specify (at the microprogram level) branch operators in parallel with other nonconflicting operations, thus masking their time cost. Also, branching relative to a register content provides an indirect means of accomplishing multiport branching. However, conditional branch operations must be delayed until the condition has been established, thus causing a time penalty. In multiport branching the programmer must be careful that none of the branching alternatives creates a conflict with the rest of the sequentially-coded program. Also, the indirect nature of the branch creates a time penalty.

Table III compares the properties of the direct transition structure with conventional systems.

The feasibility of this approach in general purpose computer design has been established. What remains is to establish in an objective and quantitative manner the
TABLE III
SUMMARY OF COMPARATIVE DESIGN FACTORS

<table>
<thead>
<tr>
<th>Area</th>
<th>Current System</th>
<th>Proposed</th>
</tr>
</thead>
<tbody>
<tr>
<td>capability for immediate multiport</td>
<td>limited</td>
<td>unlimited</td>
</tr>
<tr>
<td>branching implementation</td>
<td>software</td>
<td>hardware</td>
</tr>
<tr>
<td>memory assignment related to multiport</td>
<td>programmer</td>
<td>no programmer</td>
</tr>
<tr>
<td>branching</td>
<td>responsibility</td>
<td>responsibility</td>
</tr>
<tr>
<td>memory cost</td>
<td>lower-determined by the required instruction count</td>
<td>higher - determined by the state table representation and mapping redundancy.</td>
</tr>
<tr>
<td>execution time</td>
<td>higher</td>
<td>lower, if branching present.</td>
</tr>
<tr>
<td>program preparation</td>
<td>lower complexity</td>
<td>higher complexity</td>
</tr>
</tbody>
</table>

VII. CONCLUSIONS

A direct transition memory computing structure has been defined and investigated. The structure allows immediate multiport branching at all decision points of a procedure at a cost of introducing memory redundancy. Design methods were developed that allow a controlled tradeoff between memory redundancy and the complexity of the address computation of the next memory state. It was shown that memory redundancy, as defined, could be maintained at 50 percent or less for 96.7 percent of all possible completely specified state tables (with input and state dimensions through 32), and at no more than 52.7 percent overall with a maximum network complexity of double dependent variables. A graphic means of implementing the network design was presented, and shown applicable to both completely—and incompletely—specified state tables. The design methods are directly applicable to sequential network synthesis problems; and are shown to be feasible in general purpose computer design.

APPENDIX

PARTITION ASSIGNMENT MAPS FOR DOUBLE DEPENDENT NONDECOMPOSABLE STATE TABLES WITH DIMENSIONS THROUGH 32

\[ 7 \times 9 \]

\[ 3 \times 5 \]
REFERENCES


Howard A. Sholl (M’68) was born in Northampton, Mass. on October 14, 1938. He received the B.S. and M.S. degrees in electrical engineering from Worcester Polytechnic Institute, Worcester, Mass., in 1960 and 1963, respectively, and the Ph.D. degree in computer science from the University of Connecticut, Storrs, in 1970.

From 1961 to 1963 he was a Graduate Teaching Assistant at Worcester Polytechnic Institute. He worked as a Senior Engineer in computer and logic design for Sylvania Electric Co., Needham, Mass., from 1963 to 1966. Since 1966 he has been with the Electrical Engineering and Computer Science Department at the University of Connecticut as an Instructor from 1966 to 1970 and as an Assistant Professor since 1970. His research interests are in the areas of logic and computer system design, and software system organization.

Dr. Sholl is a member of the Association for Computing Machinery, Eta Kappa Nu and Sigma Xi.