Missouri S&T Library and Learning Resources

Scholars' Mine

Masters Theses | Student Theses and Dissertations

Spring 1989

Automated translation of digital logic equations into optimized VHDL code

John Evan Stark

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

Part of the Computer Sciences Commons

Department:

Recommended Citation

Stark, John Evan, "Automated translation of digital logic equations into optimized VHDL code" (1989). *Masters Theses*. 729.
https://scholarsmine.mst.edu/masters_theses/729

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

AUTOMATED TRANSLATION OF DIGITAL LOGIC EQUATIONS INTO OPTIMIZED VHDL CODE

BY

JOHN EVAN STARK, 1965-

A THESIS

Presented to the Faculty of the Graduate School of the

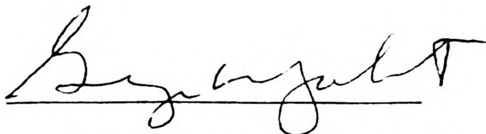
UNIVERSITY OF MISSOURI - ROLLA

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

May 1989

Approved by



Dr. George W. Zobrist, Advisor

Dr. A. K. Rigler

Dr. Darrow F. Dawson

ABSTRACT

It was desired to develop an algorithm for the automated translation of finite state machines from state table form to optimized VHDL form. To do this, algorithms are needed for reducing the state machine to simplest form, making state assignments, producing minimal logic equations to represent the state machine, and producing VHDL code which describes the intended circuit. Various such algorithms were examined and a prototype program written to perform this translation.

ACKNOWLEDGEMENT

I wish to thank everyone who has helped me with this project. In particular I want to thank my advisor Dr. George W. Zobrist for suggesting the topic and guiding me along the way. Special thanks go also to Dr. A. K. Rigler and Dr. Darrow F. Dawson, my committee members. I would also like to express my appreciation for the support and funding provided by the Intelligent Systems Center of the University of Missouri - Rolla and the United States Air Force (Contract No: F49620-85-C-0013/SB5851-0360).

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGEMENT iii

LIST OF ILLUSTRATIONS vi

I. INTRODUCTION 1

II. FROM STATE TABLE TO LOGIC EQUATIONS 3

A. STATE TABLE SIMPLIFICATION 3

1. Removal of Unreachable States 4

2. Removal of Equivalent States 4

a. Implication Tables 6

b. Equivalence Sets 7

B. STATE ASSIGNMENT 8

C. KARNAUGH MAP PRODUCTION 9

D. LOGIC EQUATION PRODUCTION 12

1. Quine/McCluskey Method 12

2. Prather Method. 14

3. Implicant Table Reduction 15

III. VHDL DESCRIPTION OF FINITE STATE MACHINES 19

IV. CONCLUSIONS 21

BIBLIOGRAPHY 22

VITA 24

APPENDICES

A. VHDL 25

B. Functional Flow Diagram 27

C. VHDL Template File 32

D. Sample Output 35

E. Meg Output Comparison 79

LIST OF ILLUSTRATIONS

Figure		Page
1	Sequential Network	1
2	Transition Diagram and State Table	3
3	Algorithm for Removal of Unreachable States	4
4	Equivalent State Removal	5
5	Algorithm for Removal of Equivalent States by Implication Table.	6
6	Algorithm for Removal of Equivalent States by Equivalence Sets.	7
7	Karnaugh Maps	9
8	Algorithm for each Karnaugh Map Production	11
9	Logic Equation Production	13
10	Algorithm for Quine/McCluskey Method	14
11	Algorithm for Prather Method	16
12	Algorithm for Implicant Table Reduction	17
13	Algorithm for Producing VHDL Description	20
14	Functional Flow Diagram	28

1. INTRODUCTION

A finite state machine is a model of a sequential logic network. The term sequential indicates that its outputs are dependent not just on its current inputs but also on past inputs. Therefore, a history of inputs must be kept. This is accomplished by use of a memory. Rather than attempt to keep track of all past inputs, a finite number of states are used, each of which represents a set of equivalent input histories. Each input causes the machine to either enter a new state or stay in the same state, and may affect the machine's output. An electrical circuit for a finite state machine includes inputs, a combinational logic part, a memory, and outputs as shown in figure 1.

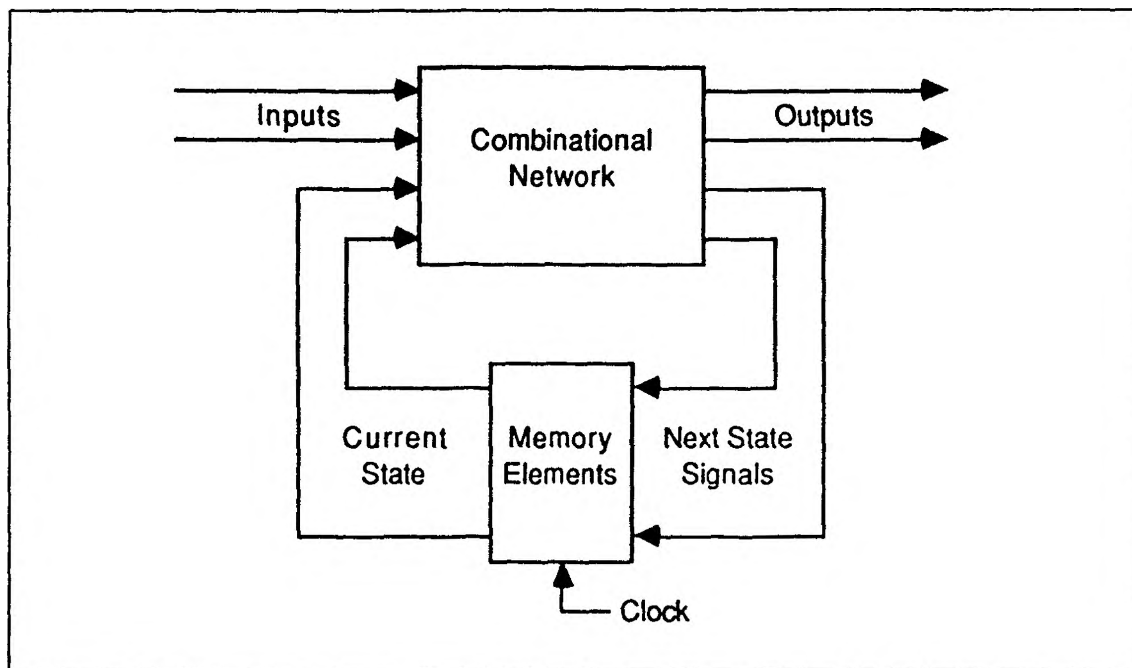


Figure 1. Sequential Network

VHDL is a hardware description language intended for the design, description, and simulation of electrical hardware systems and components. The description of an object is in two parts, an interface and an architecture. This allows for separation of function and implementation. For versatility, objects can be described by behavior,

structure, data flow, or any combination of the three architectures [13, 14]. See appendix A for a more detailed description.

The intent of this research was to provide for high level design of of electronic circuits using the finite state machine model. High level design relieves concerns for lower level details, allowing the designer to concentrate on the purpose of the design and reducing error.

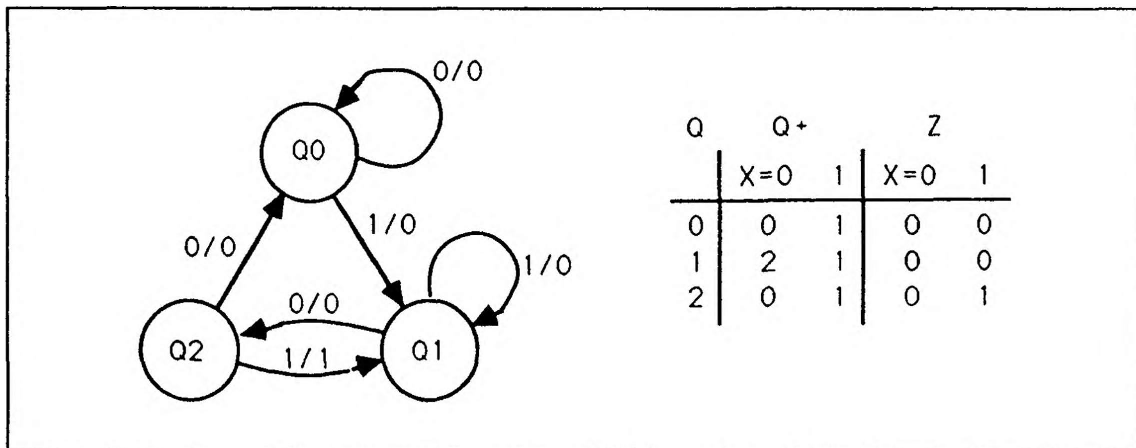
Only completely specified, synchronous, single input/single output machines were considered for the translation from state table to VHDL form. A prototype program, FSM, to perform this translation was written using Pascal on an IBM PC [10]. The following sections outline algorithms available and identify those used for the prototype program. Complete examples of the process of translating a state table to logic equations is given in appendix D.

Input for the prototype program, read from a file, includes a short (80 character) description of the finite state machine, the number of states in the machine, and the state transitions pairs. Each transition is specified by its next state and associated output. Since only completely specified single input/single output machines are considered, there are exactly two transitions for each state. States are assumed to be numbered sequentially starting with zero which is assumed to be the initial state. Additional input accepted directly from the user consists of the name the finite state machine is to be given in the VHDL code, the name of the file containing the state transitions, the name of the file to which the VHDL code is to be written, the type of flipflop to use and its delay time, and the implementation of the combinational logic and its delay.

II. FROM STATE TABLE TO LOGIC EQUATIONS

A. STATE TABLE SIMPLIFICATION

A state table is a tabular description of a transition diagram listing the transitions from each state and the outputs produced either at the state (a Moore machine) or on transition to the next states (a Mealy machine--used by FSM, the prototype program) [3]. Figure 2 shows a transition diagram and corresponding state table for a finite state machine.



Q	Q+		Z	
	X=0	1	X=0	1
0	0	1	0	0
1	2	1	0	0
2	0	1	0	1

Figure 2. Transition Diagram and State Table

Reducing the number of states in a state machine can reduce the number of memory elements needed to represent the states of the machine and help minimize the combinational logic used to determine the machine's outputs and next states. The number of memory elements required to represent n machine states is the ceiling of $\log_2 n$. Having fewer states than the maximum a set of memory elements could represent introduces don't-care terms into the logic, possibly simplifying it.

To reduce a state table to its simplest form, unnecessary states must be removed. These include redundant, unreachable, and equivalent states. As redundant states are a subset of equivalent states, they need not be considered separately although

algorithms exist for their removal. Unreachable states however can only be equivalent to other unreachable states and must therefore be handled separately.

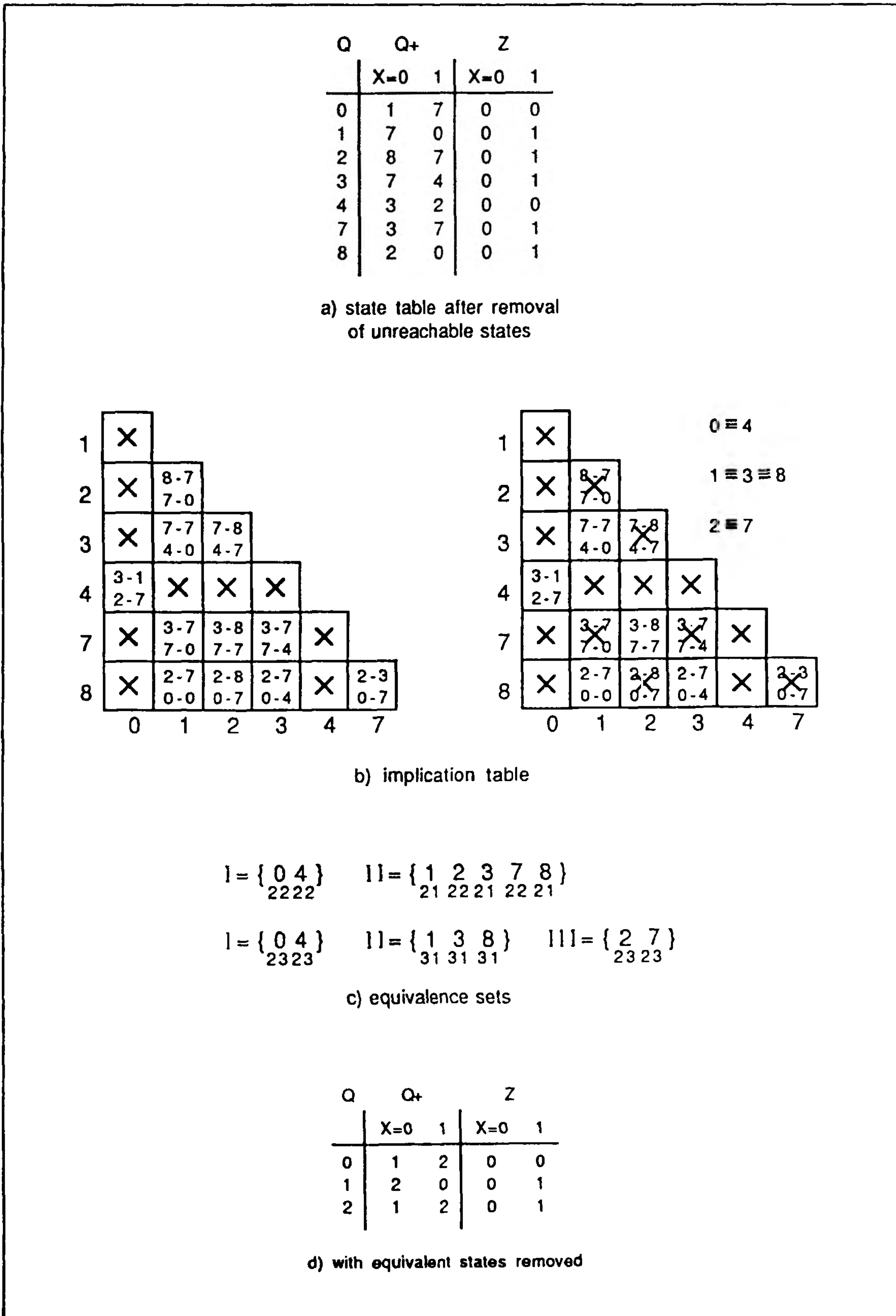
1. Removal of Unreachable States. Unreachable states are identified by forming the set of reachable states [4]. Initially, the only known reachable state, the initial state, is the sole member of this set. Then, in an iterative process, the next states of each member of the reachable set are added to the set if they are not already members. When no states are added on a pass, the set is complete. Any states not in the set are unreachable and are removed from the state table. References to these unreachable states as next states of reachable states need not be considered in this removal as there can be none.

```
Insert (Initial_State, Reachable_Set)
until No_States_Added
  No_States_Added := true
  for each Next_State of each State in Reachable_Set
    if Current_Next_State not in Reachable_Set
      Insert (Current_Next_State, Reachable_Set)
      No_States_Added := false
    end if
  end for
end until

for each State in State_Table
  if Current_State not in Reachable_Set
    Remove (Current_State, State_Table)
  end if
end for
```

Figure 3. Algorithm for Removal of Unreachable States

2. Removal of Equivalent States. Equivalent states can be identified by use of equivalence sets [1] or an implication table [3, 8]. In either case all states are at first considered to be equivalent and equivalences are then ruled out. When the equivalent states of the state table have been found, all but one of the states in each group of equivalent states are removed from the state table; in effect they are merged into one.



Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	7	0	0
1	7	0	0	1
2	8	7	0	1
3	7	4	0	1
4	3	2	0	0
7	3	7	0	1
8	2	0	0	1

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	2	0	0
1	2	0	0	1
2	1	2	0	1

Figure 4. Equivalent State Removal

To preserve the integrity of the state table, all references to removed states as next states are replaced by the id of the state kept.

a. Implication Tables. With an implication table (figure 4b), one entry exists for each possible pairing of states, without respect to order and excluding the pairing of a state with itself. An entry is marked when its pair of states is known not to be equivalent. The first of these marks are placed on the basis of differing outputs of the states' transitions, as states with differing outputs cannot be equivalent. The remaining entries are then checked in repeated passes of the table on the basis of the next states of each entry's pair of states. If the next states to be taken on a particular input for an entry's states have been found to be not equivalent, that entry's pair of states are not equivalent and it is marked. When a pass yields no additional marks, the remaining unmarked entries indicate equivalent states.

```
for each State in State_Table except last (Current_)
  for each State in State_Table beyond Current_State (Check_)
    if Check_State.Outputs ≠ Current_State.Outputs
     Table_Entry[Current_State, Check_State] := marked
    end if
  end for
end for

until No_Changes
  No_Changes := true
  for each State in State_Table except last (Current_)
    for each State in State_Table beyond Current_State (Check_)
      for each Input_Combination
        if Table_Entry[Current_State↑Next_State[Input_Combination],
              Current_State↑Next_State[Input_Combination]] is marked
        Table_Entry[Current_State, Check_State] := marked
        No_Change := false
      end if
    end for
  end for
end until
```

Figure 5. Algorithm for Removal of Equivalent States by Implication Table.

b. Equivalence Sets. When using equivalence sets (figure 4c), the states are first divided into separate sets according to the outputs of their transitions to next states. For the iterative part of this process, the states in each set are assigned a subscript for each transition indicating the set of which the transition's terminal state is a member. Each set is then broken down further into new sets for which the subscripts of all member states match. This is repeated, assigning new subscripts and dividing sets, until no more sets can be created. At this time, each set contains only equivalent states.

```
for each State in State_Table
  for each Equivalence_Set
    if Current_State.Outputs = Current_Set.Specs
      Insert (Current_State, Current_Set)
    end if
  end for
end for

until No_New_Sets
  No_New_Sets := true
  for each Input_Combination of each State of each Equivalence_Set
    for each Equivalence_Set (Current_)
      if Current_State↑Next_State[Current_Input_Combination] in Current_Set
        Current_State.Subscript[Input_Combination] := Current_Set.ID
      end if
    end for
  end for

  for each Equivalence_Set with Cardinality > 1
    for each State in Current_Set beyond first
      if Current_State.Subscripts ≠ First_State.Subscripts
        Remove (Current_State, Current_Set)
        Inserted := false
        for each New_Set split from Current_Set
          if Current_State.Subscripts = New_Set.Specs
            Insert (Current_State, Current_New_Set)
            Inserted := true
          end if
        end for

        if not Inserted
          Create (New_Set)
          Insert (Current_State, New_Set)
        end if
      end if
    end for
  end for
end until
```

Figure 6. Algorithm for Removal of Equivalent States by Equivalence Sets.

The use of equivalent sets was chosen over an equivalence table for the prototype program because the data structure grows less quickly. With n states in a machine, there will be exactly n entries in at most n equivalence sets while an implication table would have n^2 entries with $\frac{(n^2 - n)}{2}$ entries used.

B. STATE ASSIGNMENT

In the circuit implementation of a finite state machine, each state is represented by a binary n-tuple which is a concatenation of the values of the memory elements when the machine is in that state, n being the number of memory elements. The choice of these n-tuples, or state assignments, can affect the minimization of the combinational logic part of the circuit. For a given machine there are 2^n possible state assignments. Story [12] gives the number of possible combinations of assignments as

$$\frac{(2^n - 1)!}{(2^n - R)!n!}$$

where R is the number of states in the machine. Thus as the number of states grows large, the number of possible state assignments and their possible combinations grows very large.

Currently, there is no method for determining an optimal state assignment without comparing the results of assignments through trial and error. Story [11] does offer a method of reducing the number of assignments which need to be checked. His approach produces optimum combinations of state assignment columns. The number of distinct columns which need to be considered is

$$\frac{1}{2}\sum_{i=R-2^{n-1}}^{2^{n-1}}\binom{R}{i}$$

which still grows quickly. The prototype program uses the natural assignment method which consists of numbering the states sequentially starting with zero.

C. KARNAUGH MAP PRODUCTION

Karnaugh map representations of the machine outputs and next state signals are created to help in the production of the logic equations [3, 8]. Two maps are required for each JK or RS flipflop, or one for each D flipflop, and one is required for each state machine output. Figure 7 shows the production of J and K maps for one memory element of a finite state machine.

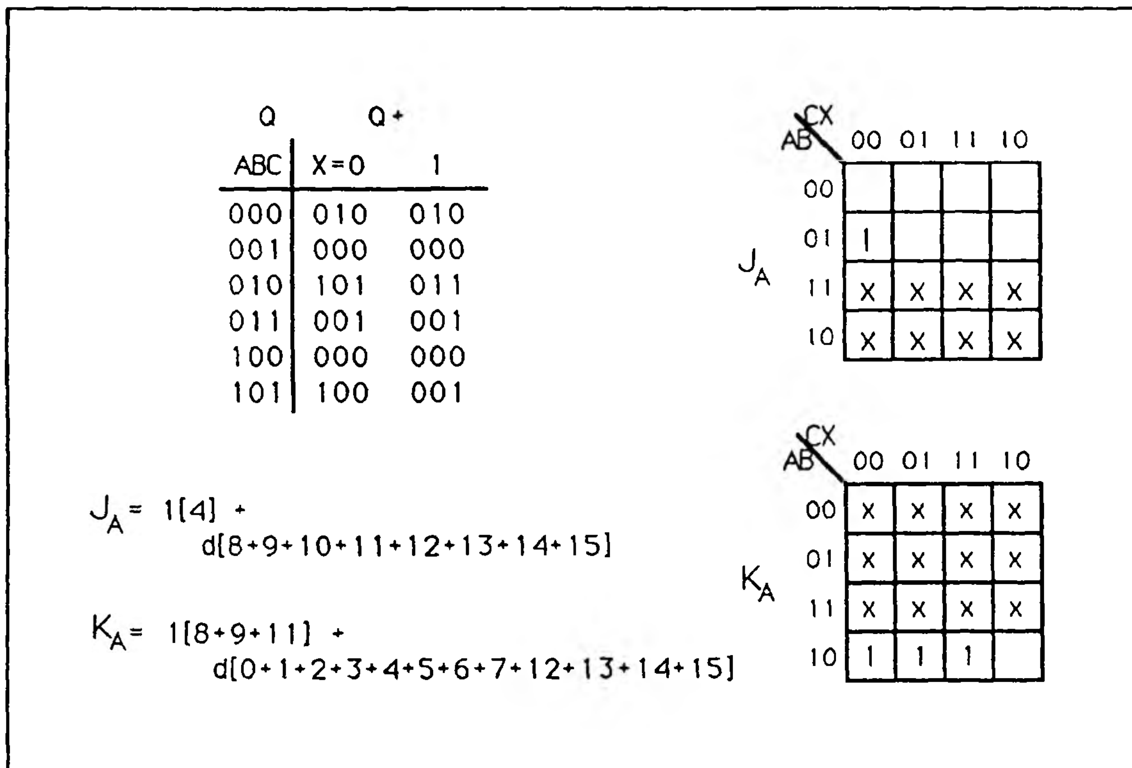


Figure 7. Karnaugh Maps

Story [12] gave formulas for finding on-cells and don't-care-cells for the Karnaugh maps for JK flipflops:

$$J = \sum_{j=0}^{R-1}(1 - y_j)Q_j\sum_{i=0}^{2^m} y_{ji}^{+}X_i + d\left[\sum_{j=0}^{R-1} y_jQ_j + \sum_{j=R}^{2^n} Q_j\right] \tag{1}$$

$$K = \sum_{j=0}^{R-1} y_jQ_j\sum_{i=0}^{2^m}(1 - y_{ji}^{+})X_i + d\left[\sum_{j=0}^{R-1}(1 - y_j)Q_j + \sum_{j=R}^{2^n} Q_j\right] \tag{2}$$

In the equations, j is the state table row index, i is the input index, R is the number of states, n is the number of flipflops, m is the number of inputs. Where Story used r and W, y and y^{+} represent the current and next state values of the flipflop; QX (rather than SX used by Story) represents the cell number of the map (a concatenation of machine state and input), Q represents a grouping of cell numbers (two cells for a single input machine) for unused states when the input values do not matter, and d (Story uses 0.5) indicates don't-care-cells. The formulas simply define the maps. The summations can be thought of as listings of map cells; the multiplication of two summations as their intersection. For example, the equation for the Karnaugh map of the set signal of a JK flipflop specifies that the on-cells are those in which the current value of the flipflop y is 0 and the next value y^{+} is to be 1. The don't-care-cells are specified as those for which the values of both y and y^{+} are 1 and all those for unused states. Formulas for RS and D flipflops can be written similarly:

$$S = \sum_{j=0}^{R-1}(1 - y_j)Q_j\sum_{i=0}^{2^m} y_{ji}^{+}X_i + d\left[\sum_{j=0}^{R-1} y_jQ_j\sum_{i=0}^{2^m} y_{ji}^{+}X_i + \sum_{j=R}^{2^n} Q_j\right] \tag{3}$$

$$R = \sum_{j=0}^{R-1} y_jQ_j\sum_{i=0}^{2^m}(1 - y_{ji}^{+})X_i + d\left[\sum_{j=0}^{R-1}(1 - y_j)Q_j\sum_{i=0}^{2^m}(1 - y_{ji}^{+})X_i + \sum_{j=R}^{2^n} Q_j\right] \tag{4}$$

$$D = \sum_{j=0}^{R-1} Q_j\sum_{i=0}^{2^m} y_{ji}^{+}X_i + d\sum_{j=R}^{2^n} Q_j \tag{5}$$

The type of memory element chosen for a circuit can also affect the minimization of the combinational logic part of the circuit. The only method of determining which type will yield minimal results is trial and error. There are however only a limited number of common types available.

```
for each FlipFlop
  Mask := 2 ** #(Current_FlipFlop)
  for each Input_Combination of each State in State_Table
    Cell_ID := 2 ** #(Inputs) * Current_Input_Combination
    Y_Current := RShift (Current_State.ID and Mask, #(Current_FlipFlop))
    Y_Next := RShift (Current_State↑Next_State[Input_Combination] and Mask,
                      #(Current_FlipFlop))

    select (Y_Current || Y_Next)
      case '00':  Insert (Current_KMap, Cell_ID, don't_care)
      case '01':  Insert (Current_JMap, Cell_ID, on)
                  Insert (Current_KMap, Cell_ID, don't_care)
      case '10':  Insert (Current_JMap, Cell_ID, don't_care)
                  Insert (Current_KMap, Cell_ID, on)
      case '11':  Insert (Current_JMap, Cell_ID, don't_care)
    end select
  end for

  for each Input_Combination of each unused State_Assignment
    Cell.ID := 2 ** #(Inputs) * Current_State.ID + Current_Input_Combination
    Insert (Current_JMap, Cell_ID, don't_care)
    Insert (Current_KMap, Cell_ID, don't_care)
  end for
end for

for each Output
  for each Input_Combination of each State in State_Table
    Cell_ID := 2 ** #(Inputs) * Current_State.ID + Current_Input_Combination
    Mask := 2 ** #(Current_Output)
    if Current_State.Output[Input_Combination] and Mask ≠ 0
      Insert (Current_Output_Map, Cell_ID, on)
    end if
  end for

  for each Input_Combination of each unused State_Assignment
    Cell_ID := 2 ** #(Inputs) * Current_State.ID + Current_Input_Combination
    Insert (Current_Output_Map, Cell_ID, don't_care)
  end for
end for
```

Figure 8. Algorithm for each Karnaugh Map Production

The decision of which type of flipflop to use in the VHDL description is left to the user of the program, as other factors than just minimization may be relevant. No provision is made for mixing flipflop types in a single machine circuit. The prototype program can produce VHDL descriptions using JK, RS, or D type flipflops. Maps are

represented internally by a list of on-cells and a list of don't-care-cells. All cells not listed are off.

D. LOGIC EQUATION PRODUCTION

It is desirable that the logic equations describing a finite state machine have both a minimal number of gates and a minimal number of gate levels. Decreasing the number of gates decreases production costs while decreasing the number of gate levels increases speed of operation. Toward these goals the prototype program produces minimal two-level sum of products equations (disjunctive normal form) using only NOT, AND, and OR operations.

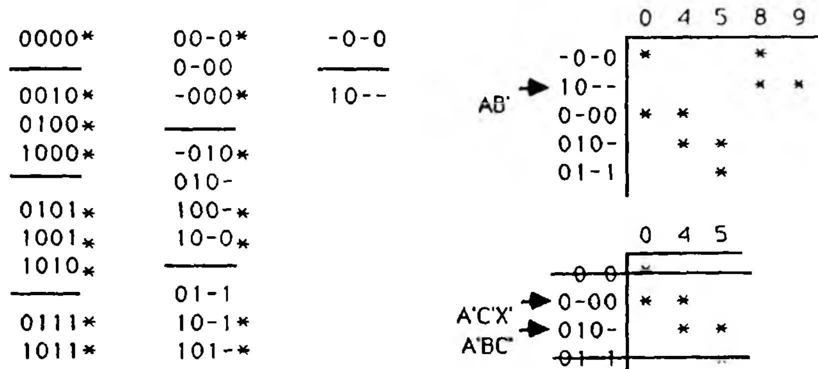
Two procedures were considered for the production of equations, the Quine/McCluskey and Prather Methods. Both start with the individual cells of the Karnaugh map and seek to combine them into the largest possible groupings. Larger cell groups can be represented in the equation by fewer terms with fewer literals, decreasing the number of gates and gate inputs necessary in the implementation of the circuit.

1. Quine/McCluskey Method.

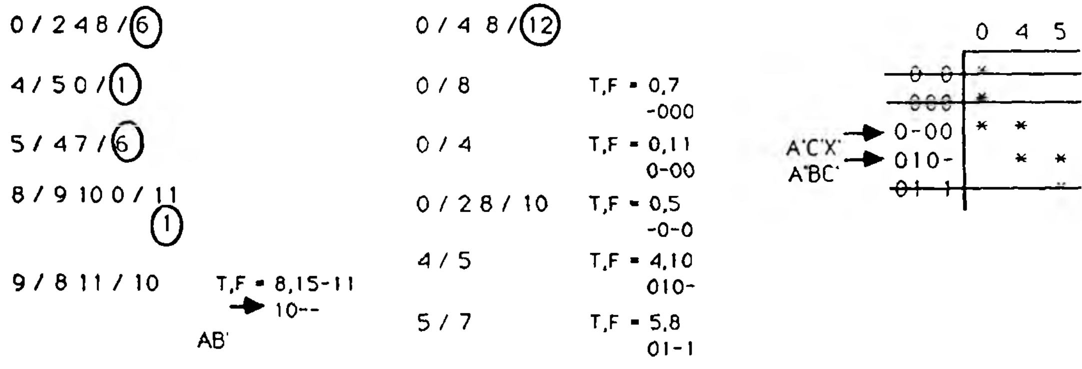
The standard procedure for producing logic equations from Karnaugh maps is the Quine/McCluskey method [5, 7] (figure 9b). With this method a list of the on-cells and don't-care-cells of the map, called implicants, is made. They are grouped according to the number of 1 bits in their binary representations. Each implicant in each group is then combined with as many implicants in the following group (those implicants with one more 1 bit) as possible, forming new implicants which are grouped separately , again according to number of 1 bits. The process is repeated with each list of new implicants until no more combinations are possible.

$S_C = 1[0+4+5+8+9] + d[2+7+10+11]$

a) Karnaugh Map Equation



b) Quine/McCluskey method



c) Prather Method

$S_C = AB' + A'C'X' + A'BC'$

d) Final Logic Equation

Figure 9. Logic Equation Production

An implicant may be combined with another if their binary representations match in all but one position (e.g. 0010 and 0110). The bit position in which the two differ is replaced by a don't-care-symbol (e.g. 0-10 or 0x10). In combinations involving implicants with don't-care positions, the don't-care positions must match exactly in both implicants (e.g. 0-10 and 0-11). The implicants which were combined to form new implicants are marked as such. When no new implicants can be formed, an implicant table is made from the implicants which have not been marked. Reduction of an implicant table to form an equation is explained below.

```
for each Map
  for each On_Cell and each Don't_Care_Cell of Map
    1Bits := 1Bit_Count (#(Cur_Cell))
    Insert (Cur_Cell, Imp_Group[1Bits])
  end for

  Cur_Imp_List := 1st_Imp_List
  until No_Combinations
    for each Implicant of each Imp_Group except last of Cur_Imp_List
      No_Combinations := true

      for each Implicant of Next_Imp_Group
        if Check_Implicant can combine with Current_Implicant
          New_Implicant := Combine (Check_Implicant, Current_Implicant)
          Insert (New_Implicant, New_Implicant_Group[Current_Group_1Bits])
          Mark (Current_Implicant_Group)
          No_Combinations := false
        end if
      end for
    end for

    Current_Implicant_List := New_Implicant_List
  end until

  for each Implicant of each Implicant_List
    if Current_Implicant not marked
      Insert (Current_Implicant, Implicant_Table)
    end if
  end for
end for
```

Figure 10. Algorithm for Quine/McCluskey Method

2. Prather Method.. A modification of the Quine/McCluskey method was given by Prather [6] (figure 9c). This technique identifies essential cells (prime implicants) by attempting to complete for each on-cell of the Karnaugh map the n-cell indicated

by adjacent on-cells and don't-care cells. If this n-cell can be completed, it is essential to the equation. If not, then the basic (nonessential) cells which cover the cell in question can be found by attempting to complete the n-cell without one or more of the original adjacent cells. First all essential n-cells are found and the cells they cover marked. Then all basic cells are found for those on-cells not yet covered and used to form an implicant table which is reduced in the manner explained below.

An n-cell is completed by checking to see if all the necessary cells are either on or don't-care. The on-cells and don't-care-cells adjacent to the cell to be covered are identified first. The number of these adjacent cells indicates the size of the n-cell and, as a power of two, the number of individual map cells covered (e.g. three adjacent cells indicate a 3-cell covering eight map cells, zero indicates a 0-cell covering one map cell). The next group of cells are identified by adding the delta (adjacent cell id minus original cell id) of each cell in the current group of the n-cell to each of the following adjacent cells. New groups of cells are found until one contains only a single map cell at which time the n-cell is complete, or until an indicated map cell is neither an on-cell nor a don't-care-cell. If the n-cell cannot be completed, an attempt to find basic cells can be made by omitting each of the original adjacent cells, one at a time, whose delta was involved in identifying the cell which failed to compete the n-cell.

The Prather method was the method chosen for the prototype program because it works at the integer level when dealing with cell id's rather than at the bit level. With the Prather method there is no need to count the bits in binary representations or check that all but one bit position of two numbers match.

3. Implicant Table Reduction. The rows of an implicant table are the implicants arranged so that priority is given to the number of on-cells covered and the number of don't-care positions (indicating fewer literals and thus fewer gate inputs). The

```
Find_1st_Group (Cell_To_Cover, NCell)
  for each Cell adjacent to Cell_To_Cover
    if Current_Cell is On or Don't_Care
      Insert (Current_Cell, 1st_Group)
    end if
  end for
end Find_1st_Group

Complete_NCell (Cell_To_Cover, NCell)
  Current_Group := 1st_Group
  until Current_Group has only one Cell or Failure
    Failure := false
    for each Cell in Current_Group except last (Current_)
      for each Cell following Current_Cell in Current_Group (Check_)
        Indicated_Cell := Map_Cell[Check_Cell_ID + Current_Cell.Delta]
        if Indicated_Cell is On or Don't_Care
          Insert (Indicated_Cell, Next_Group)
        else
          Failure := true
          Delta_History := Indicated_Cell_ID - Cell_To_Cover_ID
        end if
      end for
    end for

    Current_Group := Next_Group
  end until

  if Failure
    return (Delta_History)
end Complete_NCell

Find_Basic_Cells (NCell)
  Complete_NCell (Cell_To_Cover, 1st_Group, Delta_History)

  if Complete
    Insert (Implicant, Implicant_Table)
  else
    for each Delta in Delta_History
      Remove (Cell[Delta], 1st_Group)
      Complete_NCell (Cell_To_Cover, 1st_Group, Delta_History)
    end for
  end if
end Find_Basic_Cells

for each Map
  for each On_Cell in Map
    Find_1st_Group (Current_On_Cell, NCell)
    Complete_NCell (Current_On_Cell, NCell)

    if Complete
      Insert (Term (Min_Cell (NCell), Max_Cell (NCell)), Associated_Equation)
      Mark (Current_On_Cell)
    end if
  end for

  for each On_Cell not marked in Map
    Find_1st_Group (Current_On_Cell, NCell)
    Find_Basic_Cells (Current_On_Cell, NCell)
  end for
end for
```

Figure 11. Algorithm for Prather Method

columns of the implicant table are labeled by the on-cells of the map. Entries of a row which are in columns that represent on-cells covered by that row's implicant are marked

```
until Implicant_Table is empty
  sort Implicant_Table by Cell_Size within Columns_Covered

  for each Implicant in Implicant_Table (Current_)
    for each Implicant in Implicant_Table beyond Current_Implicant (Check_)
      if Current_Implicant dominates Check_Implicant
        Remove (Check_Implicant, Implicant_Table)
      end if
    end for
  end for

  Reduced := false
  for each Implicant in Implicant_Table
    if Current_Implicant alone covers a Column
      for each Column covered by Current_Implicant (Delete_)
        Remove (Delete_Column, Implicant_Table)
      end for

      Remove (Current_Implicant, Implicant_Table)
      Reduced := true
    end if
  end for

  if not Reduced
    Count := #(Implicants)
    for each Column in Implicant_Table
      if #(Implicants covering Current_Column) < Count
        Count := #(Implicants covering Current_Column)
        Select_Column := Current_Column
      end if
    end for

    for each Implicant in Implicant_Table until Reduced
      if Current_Implicant covers Select_Column
        for each Column covered by Current_Implicant
          Remove (Current_Column, Implicant_Table)
        end for

        Remove (Current_Implicant, Implicant_Table)
        Reduced := true
      end if
    end for
  end if
end until
```

Figure 12. Algorithm for Implicant Table Reduction

A prime implicant is one which alone covers an on-cell (is the only implicant with an entry in that column marked before any reduction is done). Prime implicants are essential to the equation and are removed from the table along with the columns they

cover and become the basis of the equation. All remaining columns are now covered by two or more implicants. With the Prather method prime implicants (essential cells) are recognized upon completion and not added to the implicant table but directly become a term of the equation.

If an implicant is dominated, it may be removed from the table without effect. One implicant dominates another if, for every column covered by the second, the first also covers that column. If two implicants dominate each other and one has fewer don't-care positions, it should be the one removed; otherwise the decision is arbitrary. If removing dominance from the table leaves columns which are covered by only one remaining implicant, those implicants should be selected--removed from the table along with the columns they cover and added to the equation. If no columns are covered by only one remaining implicant, then an implicant must be chosen by another method. Normally the implicant chosen is the one highest in the table covering a column having the least number of implicants covering it. The process of removing dominance and chosing implicants is repeated until the implicant table is empty. While the now complete equation may not be unique, it is minimal.

III. VHDL DESCRIPTION OF FINITE STATE MACHINES

As mentioned before, the circuit implementation of a finite state machine consists of inputs, outputs, a memory, and a combinational logic part. In a VHDL description of this circuit the inputs and outputs make up the entity declaration part, its interface. The memory and the combinational logic are defined by an the entity's architecture, the body of the description. The memory will be represented by flipflops for which standard, predefined descriptions exist that can be used. The combination logic part can be constructed from either discrete gates or a programmable logic array. If VHDL's behavioral type of description is used, the only difference is the number of inputs as a PLA does not require negated inputs. Thus the description of a finite state machine can be standardized, requiring only information concerning the number of inputs, outputs, and memory elements, and the necessary logic equations.

The VHDL code description of the state machine is produced with the use of a template file (appendix C) containing markers indicating where machine specific information is needed. Markers in the template are set off from the code by brackets. When, in copying the VHDL code file from the template to the output file, a marker is found, it is identified and replaced by the appropriate substitution string. Substitution strings, with the exception of the actual logic equations, are determined from parameters prior to writing the VHDL code file. The logic equations are formulated from their internal representation and written when the logic marker is found.

The prototype translation program produces two files as output. One is a trace of its operation including the initial state table, simplified state tables, Karnaugh map representations, essential cells and implicant tables for those equations with nonessential cells, complete equations, and timing of operation. The other file is the VHDL source code description of the finite state machine, a combination of structural

```
Set_Substitutions (Parameters)
Read (Text)
for each Marker in Text
  Replace (Current_Marker, Substitution_String[Current_Marker])
end for
Write (Text)
```

Figure 13. Algorithm for Producing VHDL Description

and behavioral descriptions. VHDL version 7.2 was used for this file. Syntax was checked for correctness with the VHDL Analyzer. Sample output for these files can be found in appendix D.

IV. CONCLUSIONS

The logic equations for the the finite state machines in the examples shown in appendix D were checked for correctness and if from a text, compared to the solution given where possible. The example solutions were also compared (see appendix E) with the output of Meg [9], a state machine equation generator.

The VHDL output file can be used as a source file for simulation or simply as a circuit description. The output of four of the examples in appendix D (examples 1, 5, 6, and 7) were run with the 1076/B VHDL Simulator. As the original VHDL code was version 7.2, some minor changes were required to make the machines run. They did, however, perform as expected.

Following are some possible extensions to the program. A graphical finite state machine editor used as an input interface would make input easier for the designer. The handling of asynchronous, multi-input/multi-output, and incompletely specified state machines would make the program more realistic in terms of use. Version 7.2 of VHDL was used for the prototype program as that was the latest version of the analyser available. The most recent version would be desired for actual use. Also, standard library components for the flipflops would make the designs more compatible with existing systems and allow greater device independence. The examination of various state assignments would ensure that the final logic equations were indeed the minimal possible. Interfacing the VHDL with EDIF [2] would allow for a standard graphical representation of the electrical circuit.

BIBLIOGRAPHY

1. Dietmeyer, Donald L. "Synchronous Sequential Networks", in Logic Design of Digital Systems. Allyn and Bacon, Inc., 2nd ed., 1978.

2. EDIF Electronic Design Interchange Format. Electronic Industries Association, Ver. 2 0 0, May 1987.

3. Hill, Fredrick J. and Gerald R. Peterson. Introduction to Switching Theory and Logical Design. John Wiley and Sons, 3rd ed., 1981, pp. 96-337.

4. Hopcroft, John E. and Jeffrey D. Ullman. "Simplification of Context-Free Grammars", in Introduction to Automata Theory, Languages, and Computation. Addison-Wessley Publishing Company, 1979.

5. McCluskey, Jr., E. J. "Minimization of Boolean Functions", The Bell System Technical Journal. Vol. 35, November 1956, pp. 1417-1444.

6. Prather, Ronald. "Computational Aids for Determining the Minimal Form of a Truth Function", Journal of the Association for Computing Machinery. Vol. 7, No. 4, October 1960, pp. 299-310.

7. Quine, W. V. "The Problem of Simplifying Truth Functions", The American Mathematical Monthly. Vol. 59, No. 8, October 1952, pp. 521-531.

8. Roth, Charles H. Fundamentals of Logic Design. West Publishing Company, 2nd ed., 1979, pp. 221-349.

9. Scott, Walter S., et. al. editors. "Meg", in Berkeley CAD Tools. University of California, 1986 ed., December 1985.

10. Stark, John Evan. "FSM, Source Listing", Internal Report, University of Missouri - Rolla, 1989.

11. Story, James R. "State Assignment Optimization for Synchronous Sequential Machines", Ph.D. dissertation, University of Alabama, Tuscaloosa, May 1971.

12. Story, James R. et. al. "Optimum State Assignment for Synchronous Sequential Circuits", IEEE Transactions on Computers. Vol. C-21, No. 12, December 1972, pp. 1365-1373.

13. VHDL Language Reference Manual. Intermetrics, Inc., Ver. 7.2, August 1985.

14. VHDL Language Reference Manual. CAD Language Systems, Inc., Ver. 1076/B, May 1987.

VITA

John Evan Stark (born January 20, 1965) attended secondary school in Chillicothe, Missouri, graduating in May 1983. He received a Bachelor of Science degree in Computer Science from Northeast Missouri State University in May 1987. He is currently a candidate for a Master of Science degree in Computer Science at the University of Missouri - Rolla, working as a graduate research assistant. While in school, he has been active in the local chapters of the Association for Computing Machinery and Kappa Mu Epsilon, an honorary mathematics society.

APPENDIX A

VHDL

VHDL (VHSIC Hardware Description Language) [13, 14] is a language that can be used for the design, description, and simulation of electrical systems and components. An entity is the basic design unit. It can be any object from a simple gate to an entire electrical system. Each entity description is composed of two parts, its interface and its architecture. More than one architecture for an entity, which share a single interface, can exist to allow for multiple descriptions of that entity.

The interface of an entity defines its inputs and outputs, both physical and logical, by direction and data type. Directions include in, out, bi-directional, buffered, and unknown. Data types can be standard predefined types (bit, boolean, integer, real, character) or user-defined types. Logical inputs, called generics, allow a single entity to model several identical and yet unique components of a design (e.g. the ROM chips of a memory board). The interface of an entity can also declare items visible only within the entity (e.g. data types, constants, subprograms).

An architecture is identified by its own name as well as by the name of the entity which it describes. The body of each architecture has a declarative part and a statement part. An entity can be described using one or more of three styles provided: structural, data-flow, and behavioral. Structural descriptions give a hierarchical arrangement of components, each of which is itself an entity with its own interface and architecture. Data-flow descriptions list concurrent signal assignments which represent the flow of data through the entity. Behavioral descriptions use sequential processes, similar to high level computer programs, to describe the operation of the entity.

The VHDL environment includes an analyzer, reverse analyzer, simplifier, simulator, design library, and design library manager. The analyzer checks VHDL source code for syntactic errors and translates it to an intermediate form which can be stored in the design library for future reference. The reverse analyser can reconstruct the VHDL code from the intermediate form of a unit in the design library. The simplifier reorganizes the hardware description, binding components to entities in preparation for simulation. The simulator computes successive signal values of a design, called waveforms, in a combination event-driven, continuous fashion. The design library manager integrates the elements of the VHDL environment.

APPENDIX B

FUNCTIONAL FLOW DIAGRAM

This appendix contains a functional flow diagram of the prototype program FSM.

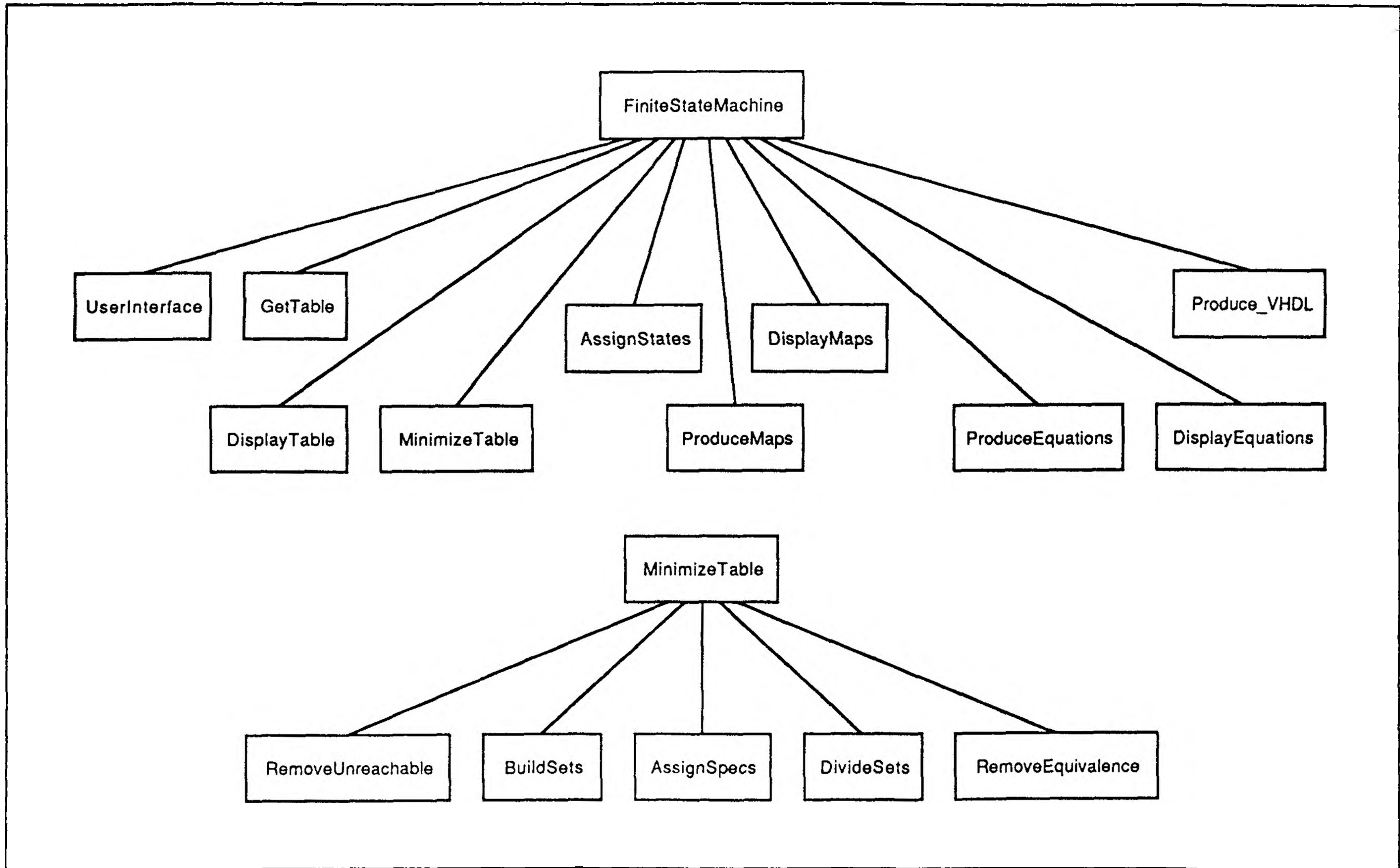


Figure 14. Functional Flow Diagram

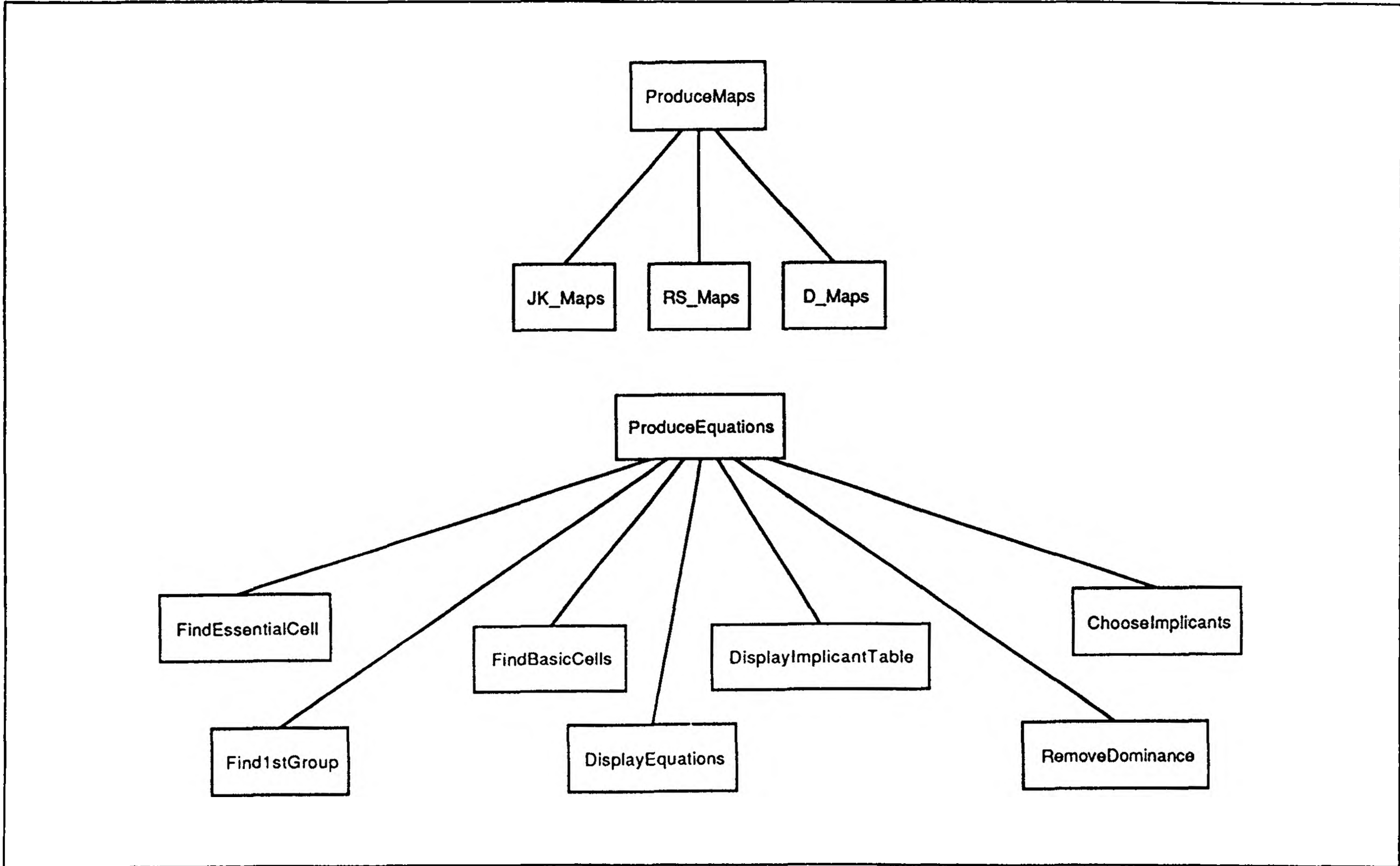


Figure 14. Functional Flow Diagram, cont.

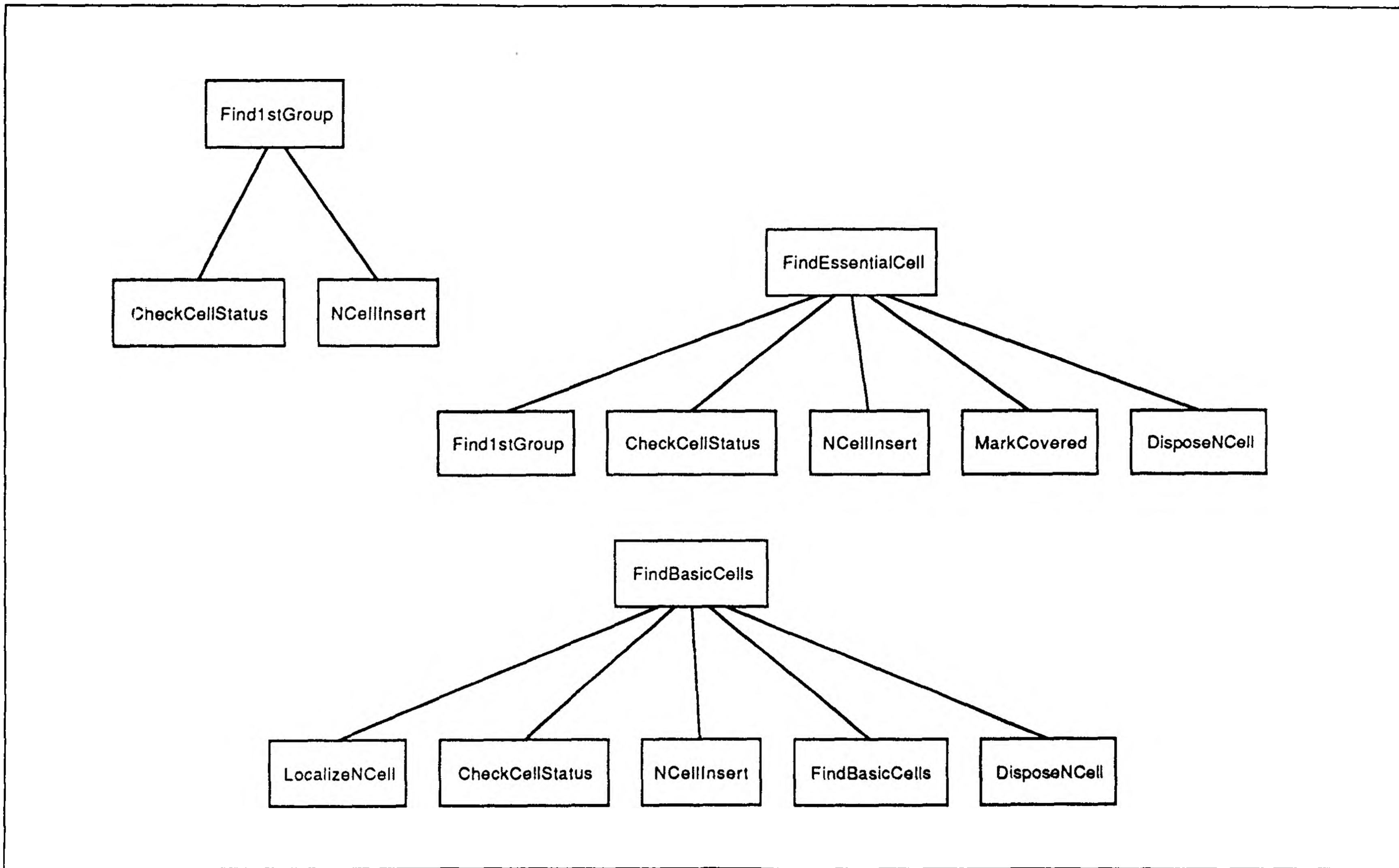


Figure 14. Functional Flow Diagram, cont.

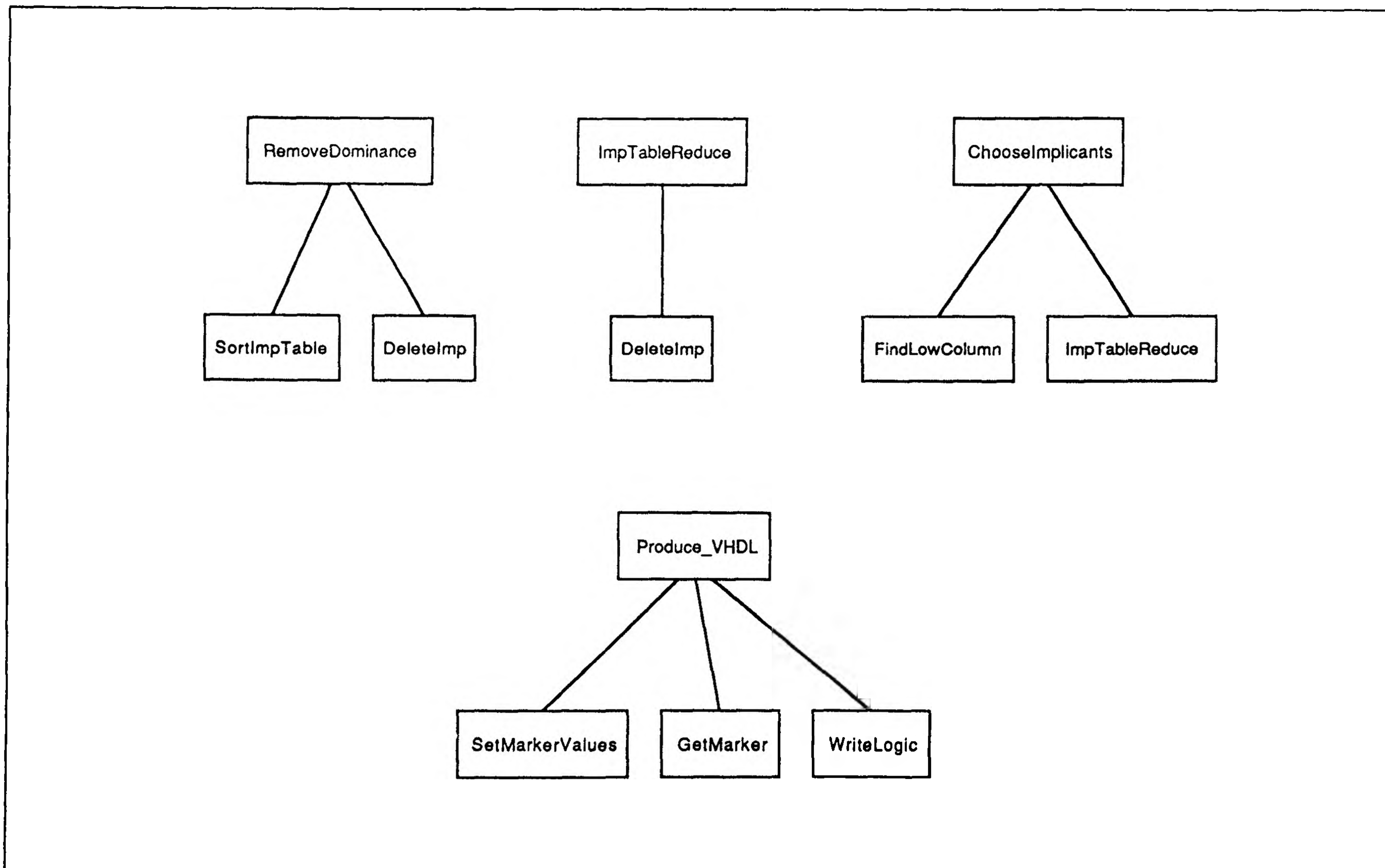


Figure 14. Functional Flow Diagram, cont.

APPENDIX C

VHDL TEMPLATE FILE

This appendix contains the template file used by the prototype program in producing the VHDL description of the finite state machine.

```
entity {Name}
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end {Name};


architecture {Arch} of {Name} is

  B1: block
    component {ff}_FlipFlop
      port ( {Ctrl}  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component {Comb}
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to {FlopIn});
    signal  Ynext:  Bit_Vector (0 to {FlopOut});

  begin
    Ycur({FlopInRange}) <= X({InRange});
    Z({OutRange}) <= Ynext({FlopOutRange});

    for I in 0 to {Flop} generate
      Mem: {ff}_FlipFlop
        port ( Ynext({NextI}), {Ctrl2} Ycur({CurI}), {Qnot}, Clk );
    end generate;

    Comb: {Comb}
      port ( Ycur, Ynext );
  end block;
end {Arch};


entity {ff}_FlipFlop
  ( {Ctrl}  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end {ff}_FlipFlop;


architecture Behavior of {ff}_FlipFlop is

  B1: block {Guard}
  begin
    P1: process {Sens}
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := {ff_Logic};
        Q <= Qhold after {ff_Time} ns;
        Qnot <= not Qhold after {ff_Time} ns;
      end if;
    end process;
  end block;
end Behavior;


entity {Comb}
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end {Comb};


architecture Behavior of {Comb} is

  B1: block
  begin {Logic}
  end block;
end Behavior;
```

APPENDIX D

SAMPLE OUTPUT

This appendix contains output from sample runs of the prototype program.

detect '101'

initial state table

Q	Q+ X= 0	1	Z X= 0	1
0	3	1	0	0
1	2	5	0	0
2	0	1	0	1
3	3	4	0	0
4	2	5	0	0
5	6	4	0	0
6	0	1	0	1

with unreachable states removed

Q	Q+ X= 0	1	Z X= 0	1
0	3	1	0	0
1	2	5	0	0
2	0	1	0	1
3	3	4	0	0
4	2	5	0	0
5	6	4	0	0
6	0	1	0	1

sets of equivalent states

```
1 = { 0 3 }
3 = { 1 4 5 }
2 = { 2 6 }
```

minimized state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	2	1	0	0
2	0	1	0	1

Karnaugh maps

```
Ja  1:  2
   dc:  7  6  5  4

Ka  1:  5  4
   dc:  7  6  3  2  1  0

Jb  1:  5  1
   dc:  7  6  3  2

Kb  1:  2
   dc:  7  6  5  4  1  0

Z0  1:  5
   dc:  7  6
```

complete equation

Ja = BX'

```
complete equation

   Ka = 1

complete equation

   Jb = X

complete equation

   Kb = X'

complete equation

   Z0 = AX

logic equations

   Ja = BX'
   Ka = 1
   Jb = X
   Kb = X'
   Z0 = AX

elapsed time:    0.66 sec

writing VHDL code file

elapsed time:    3.52 sec
```

```
entity Detect_101
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Detect_101;


architecture PLA_Structure of Detect_101 is

  B1: block
    component JK_FlipFlop
      port ( J, K:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Programmable_Logic_Array
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 2);
    signal  Ynext:  Bit_Vector (0 to 4);

  begin
    Ycur(2) <= X(0);
    Z(0) <= Ynext(4);

    for I in 0 to 1 generate
      Mem: JK_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(I), open, Clk );
    end generate;

    Comb: Programmable_Logic_Array
      port ( Ycur, Ynext );
  end block;
end PLA_Structure;



entity JK_FlipFlop
  ( J, K:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end JK_FlipFlop;


architecture Behavior of JK_FlipFlop is

  B1: block (Clk = '1' and not Clk'Stable)
  begin
    P1: process ( Guard )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := (J and not Qhold) or (not K and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Programmable_Logic_Array
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Programmable_Logic_Array;


architecture Behavior of Programmable_Logic_Array is

  B1: block
  begin
    Outputs(0) <= Inputs(1) and not Inputs(2) after 40 ns;
    Outputs(1) <= 1;
    Outputs(2) <= Inputs(2) after 40 ns;
    Outputs(3) <= not Inputs(2) after 40 ns;
    Outputs(4) <= Inputs(0) and Inputs(2) after 40 ns;
  end block;
end Behavior;
```

Dietmeyer, p313 M3

initial state table

Q	Q+ X= 0	Q+ X= 1	Z X= 0	Z X= 1
0	2	2	0	0
1	0	0	0	1
2	5	3	0	0
3	1	1	0	0
4	0	0	0	0
5	4	1	0	0
6	4	3	0	0
7	5	5	0	0

with unreachable states removed

Q	Q+ X= 0	Q+ X= 1	Z X= 0	Z X= 1
0	2	2	0	0
1	0	0	0	1
2	5	3	0	0
3	1	1	0	0
4	0	0	0	0
5	4	1	0	0

sets of equivalent states

1 = { 0 }
6 = { 4 }
5 = { 2 }
3 = { 3 }
4 = { 5 }
2 = { 1 }

minimized state table

Q	Q+ X= 0	Q+ X= 1	Z X= 0	Z X= 1
0	2	2	0	0
1	0	0	0	1
2	5	3	0	0
3	1	1	0	0
4	0	0	0	0
5	4	1	0	0

Karnaugh maps

```
Ja  1:  4
    dc: 15 14 13 12 11 10  9  8

Ka  1: 11  9  8
    dc: 15 14 13 12  7  6  5  4  3  2  1  0

Jb  1:  1  0
    dc: 15 14 13 12  7  6  5  4

Kb  1:  7  6  4
    dc: 15 14 13 12 11 10  9  8  3  2  1  0

Jc  1:  5  4
    dc: 15 14 13 12 11 10  7  6  3  2
```

```
   Kc  1: 10  3  2
      dc: 15 14 13 12  9  8  5  4  1  0

   Z0  1:  3
      dc: 15 14 13 12

complete equation

   Ja = BC'X'

complete equation

   Ka = C' + X

complete equation

   Jb = A'C'

complete equation

   Kb = X' + C

complete equation

   Jc = B

essential cells

   Kc = A'B'

implicant table

  implicant  |  columns covered
-------------+-------------------
        1--0 |  10
        -0-0 |  10

complete equation

   Kc = AX' + A'B'

complete equation

   Z0 = A'B'CX

logic equations

   Ja = BC'X'
   Ka = C' + X
   Jb = A'C'
   Kb = X' + C
   Jc = B
   Kc = AX' + A'B'
   Z0 = A'B'CX

elapsed time:    1.43 sec

writing VHDL code file

elapsed time:    4.56 sec
```

```
entity Dietmeyer_M3
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Dietmeyer_M3;


architecture Discrete_Structure of Dietmeyer_M3 is

  B1: block
    component JK_FlipFlop
      port ( J, K:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Discrete_Gates
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 6);
    signal  Ynext:  Bit_Vector (0 to 6);

  begin
    Ycur(6) <= X(0);
    Z(0) <= Ynext(6);

    for I in 0 to 2 generate
      Mem: JK_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(2*I), Ycur(2*I+1), Clk );
    end generate;

    Comb: Discrete_Gates
      port ( Ycur, Ynext );
  end block;
end Discrete_Structure;



entity JK_FlipFlop
  ( J, K:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end JK_FlipFlop;


architecture Behavior of JK_FlipFlop is

  B1: block (Clk = '1' and not Clk'Stable)
  begin
    P1: process ( Guard )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := (J and not Qhold) or (not K and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Discrete_Gates
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Discrete_Gates;


architecture Behavior of Discrete_Gates is

  B1: block
  begin
    Outputs(0) <= Inputs(2) and Inputs(5) and not Inputs(6) after 20 ns;
    Outputs(1) <= (Inputs(5)) or (Inputs(6)) after 40 ns;
    Outputs(2) <= Inputs(1) and Inputs(5) after 20 ns;
    Outputs(3) <= (not Inputs(6)) or (Inputs(4)) after 40 ns;
    Outputs(4) <= Inputs(2) after 20 ns;
    Outputs(5) <= (Inputs(0) and not Inputs(6)) or (Inputs(1) and Inputs(3)) after 40 ns;
    Outputs(6) <= Inputs(1) and Inputs(3) and Inputs(4) and Inputs(6) after 20 ns;
  end block;
end Behavior;
```

Dietmeyer, p316 M4

initial state table

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	2	0	0
1	3	4	0	0
2	5	6	0	0
3	7	8	0	0
4	9	10	0	0
5	11	12	0	0
6	13	14	0	0
7	0	0	0	0
8	0	0	0	1
9	0	0	0	1
10	0	0	0	1
11	0	0	0	0
12	0	0	0	1
13	0	0	0	1
14	0	0	0	1

with unreachable states removed

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	2	0	0
1	3	4	0	0
2	5	6	0	0
3	7	8	0	0
4	9	10	0	0
5	11	12	0	0
6	13	14	0	0
7	0	0	0	0
8	0	0	0	1
9	0	0	0	1
10	0	0	0	1
11	0	0	0	0
12	0	0	0	1
13	0	0	0	1
14	0	0	0	1

sets of equivalent states

```
1 = {  0  }
8 = {  7 11  }
6 = {  1  }
7 = {  2  }
3 = {  3  5  }
4 = {  4  }
5 = {  6  }
2 = {  8  9 10 12 13 14  }
```

minimized state table

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	2	0	0
1	3	4	0	0
2	3	5	0	0
3	6	7	0	0
4	7	7	0	0
5	7	7	0	0

```
6 |   0   0 |   0   0
7 |   0   0 |   0   1
```

Karnaugh maps

```
Sa  1:  7  6  5  3
   dc: 11 10  9  8

Ra  1: 15 14 13 12
   dc:  4  2  1  0

Sb  1: 11 10  9  8  2  1
   dc:  7  6  4

Rb  1: 15 14 13 12  5
   dc:  3  0

Sc  1:  9  8  5  4  0
   dc: 11 10  7  2

Rc  1: 15 14  6  3
   dc: 13 12  1

Z0  1: 15
   dc:
```

essential cells

Sa = A'BX + A'BC

implicant table

```
  implicant   |  columns covered
--------------+------------------
        0-11 |   3
        -011 |   3
```

complete equation

Sa = A'CX + A'BX + A'BC

complete equation

Ra = AB

essential cells

Sb = B'C'X + AB'

implicant table

```
  implicant   |  columns covered
--------------+------------------
        0-10 |   2
        -010 |   2
```

complete equation

Sb = A'CX' + B'C'X + AB'

complete equation

Rb = BC'X + AB

```
essential cells

   Sc = AB'

implicant table

  implicant  |  columns covered
-------------+------------------
        -0-0 |   0
        -000 |   0
        0-00 |   0   4
        010- |   4   5
        01-1 |   5

complete equation

   Sc = A'BC' + A'C'X' + AB'


complete equation

   Rc = A'B'X + BCX' + AB


complete equation

   Z0 = ABCX


logic equations

   Sa = A'CX + A'BX + A'BC
   Ra = AB
   Sb = A'CX' + B'C'X + AB'
   Rb = BC'X + AB
   Sc = A'BC' + A'C'X' + AB'
   Rc = A'B'X + BCX' + AB
   Z0 = ABCX


elapsed time:    2.47 sec

writing VHDL code file

elapsed time:    5.54 sec
```

```
entity Dietmeyer_M4
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Dietmeyer_M4;


architecture PLA_Structure of Dietmeyer_M4 is

  B1: block
    component RS_FlipFlop
      port ( S, R:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Programmable_Logic_Array
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 3);
    signal  Ynext:  Bit_Vector (0 to 6);

  begin
    Ycur(3) <= X(0);
    Z(0) <= Ynext(6);

    for I in 0 to 2 generate
      Mem: RS_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(I), open, Clk );
    end generate;

    Comb: Programmable_Logic_Array
      port ( Ycur, Ynext );
  end block;
end PLA_Structure;


entity RS_FlipFlop
  ( S, R:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end RS_FlipFlop;


architecture Behavior of RS_FlipFlop is

  B1: block
  begin
    P1: process ( R, S )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := S or (not R and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Programmable_Logic_Array
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Programmable_Logic_Array;


architecture Behavior of Programmable_Logic_Array is

  B1: block
  begin
    Outputs(0) <= (not Inputs(0) and Inputs(2) and Inputs(3)) or
(not Inputs(0) and Inputs(1) and Inputs(3)) or
(not Inputs(0) and Inputs(1) and Inputs(2)) after 40 ns;
    Outputs(1) <= Inputs(0) and Inputs(1) after 40 ns;
    Outputs(2) <= (not Inputs(0) and Inputs(2) and not Inputs(3)) or
(not Inputs(1) and not Inputs(2) and Inputs(3)) or (Inputs(0) and not Inputs(1)) after 40 ns;
    Outputs(3) <= (Inputs(1) and not Inputs(2) and Inputs(3)) or
(Inputs(0) and Inputs(1)) after 40 ns;
    Outputs(4) <= (not Inputs(0) and Inputs(1) and not Inputs(2)) or
(not Inputs(0) and not Inputs(2) and not Inputs(3)) or (Inputs(0) and not Inputs(1)) after 40 ns;
    Outputs(5) <= (not Inputs(0) and not Inputs(1) and Inputs(3)) or
(Inputs(1) and Inputs(2) and not Inputs(3)) or (Inputs(0) and Inputs(1)) after 40 ns;
    Outputs(6) <= Inputs(0) and Inputs(1) and Inputs(2) and Inputs(3) after 40 ns;
  end block;
end Behavior;
```

Dietmeyer, p315 M5

initial state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	4	0	0
1	0	4	0	0
2	1	5	0	1
3	1	5	0	1
4	2	6	0	1
5	2	6	0	1
6	3	7	0	1
7	3	7	0	1

with unreachable states removed

Q	Q+ X= 0	1	Z X= 0	1
0	0	4	0	0
1	0	4	0	0
2	1	5	0	1
3	1	5	0	1
4	2	6	0	1
5	2	6	0	1
6	3	7	0	1
7	3	7	0	1

sets of equivalent states

```
1 = { 0 1 }
2 = { 2 3 }
3 = { 4 5 6 7 }
```

minimized state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	2	0	0
1	0	2	0	1
2	1	2	0	1

Karnaugh maps

```
Sa  1:  3 1
    dc: 7 6 5

Ra  1:  4
    dc: 7 6 2 0

Sb  1:  4
    dc: 7 6

Rb  1:  3 2
    dc: 7 6 5 1 0

Z0  1:  5 3
    dc: 7 6
```

complete equation

```
   Sa = X


complete equation

   Ra = X'


complete equation

   Sb = AX'


essential cells

   Rb = 0

implicant table

  implicant  |  columns covered
-------------+------------------
         0-- |   2   3
         -1- |   2   3
         --1 |   3

complete equation

   Rb = A'


complete equation

   Z0 = BX + AX


logic equations

   Sa = X
   Ra = X'
   Sb = AX'
   Rb = A'
   Z0 = BX + AX


elapsed time:    1.16 sec

writing VHDL code file

elapsed time:    4.18 sec
```

```
entity Dietmeyer_M5
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Dietmeyer_M5;


architecture Discrete_Structure of Dietmeyer_M5 is

  B1: block
    component RS_FlipFlop
      port ( S, R:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Discrete_Gates
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 4);
    signal  Ynext:  Bit_Vector (0 to 4);

  begin
    Ycur(4) <= X(0);
    Z(0) <= Ynext(4);

    for I in 0 to 1 generate
      Mem: RS_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(2*I), Ycur(2*I+1), Clk );
    end generate;

    Comb: Discrete_Gates
      port ( Ycur, Ynext );
  end block;
end Discrete_Structure;


entity RS_FlipFlop
  ( S, R:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end RS_FlipFlop;


architecture Behavior of RS_FlipFlop is

  B1: block
  begin
    P1: process ( R, S )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := S or (not R and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Discrete_Gates
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Discrete_Gates;


architecture Behavior of Discrete_Gates is

  B1: block
  begin
    Outputs(0) <= Inputs(4) after 20 ns;
    Outputs(1) <= not Inputs(4) after 20 ns;
    Outputs(2) <= Inputs(0) and not Inputs(4) after 20 ns;
    Outputs(3) <= Inputs(1) after 20 ns;
    Outputs(4) <= (Inputs(2) and Inputs(4)) or (Inputs(0) and Inputs(4)) after 40 ns;
  end block;
end Behavior;
```

Dietmeyer, p351 5.2-7a

initial state table

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	7	0	0
1	7	0	0	1
2	8	7	0	1
3	7	4	0	1
4	3	2	0	0
5	6	7	0	0
6	2	5	0	1
7	3	7	0	1
8	2	0	0	1

with unreachable states removed

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	7	0	0
1	7	0	0	1
2	8	7	0	1
3	7	4	0	1
4	3	2	0	0
7	3	7	0	1
8	2	0	0	1

sets of equivalent states

```
1 = { 0 4 }
2 = { 1 3 8 }
3 = { 2 7 }
```

minimized state table

Q	Q+ X=0	Q+ X=1	Z X=0	Z X=1
0	1	2	0	0
1	2	0	0	1
2	1	2	0	1

Karnaugh maps

```
Da  1:  5  2  1
   dc:  7  6

Db  1:  4  0
   dc:  7  6

Z0  1:  5  3
   dc:  7  6
```

complete equation

Da = B'X + BX'

complete equation

Db = B'X'

```
complete equation

   ZO = BX + AX


logic equations

   Da = B'X + BX'
   Db = B'X'
   ZO = BX + AX


elapsed time:    0.71 sec

writing VHDL code file

elapsed time:    3.68 sec
```

```
entity Dietmeyer_A
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Dietmeyer_A;


architecture PLA_Structure of Dietmeyer_A is

  B1: block
    component D_FlipFlop
      port ( D:      in  Bit;
             Q:      out Bit;
             Qnot:   out Bit;
             Clk:    in  Bit );

    component Programmable_Logic_Array
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 2);
    signal  Ynext:  Bit_Vector (0 to 2);

  begin
    Ycur(2) <= X(0);
    Z(0) <= Ynext(2);

    for I in 0 to 1 generate
      Mem: D_FlipFlop
        port ( Ynext(I),  Ycur(I), open, Clk );
    end generate;

    Comb: Programmable_Logic_Array
      port ( Ycur, Ynext );
  end block;
end PLA_Structure;


entity D_FlipFlop
  ( D:      in  Bit;
    Q:      out Bit;
    Qnot:   out Bit;
    Clk:    in  Bit ) is
end D_FlipFlop;


architecture Behavior of D_FlipFlop is

  B1: block (Clk = '1' and not Clk'Stable)
  begin
    P1: process ( Guard )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := D;
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Programmable_Logic_Array
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Programmable_Logic_Array;


architecture Behavior of Programmable_Logic_Array is

  B1: block
  begin
    Outputs(0) <= (not Inputs(1) and Inputs(2)) or (Inputs(1) and not Inputs(2)) after 40 ns;
    Outputs(1) <= not Inputs(1) and not Inputs(2) after 40 ns;
    Outputs(2) <= (Inputs(1) and Inputs(2)) or (Inputs(0) and Inputs(2)) after 40 ns;
  end block;
end Behavior;
```

Dietmeyer, p351 5.2-7b

initial state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	3	2	0	0
2	3	2	0	1
3	0	1	0	0

with unreachable states removed

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	3	2	0	0
2	3	2	0	1
3	0	1	0	0

sets of equivalent states

```
1 = { 0 3 }
3 = { 1 }
2 = { 2 }
```

minimized state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	0	2	0	0
2	0	2	0	1

Karnaugh maps

```
Da  1:  5  3
   dc:  7  6

Db  1:  1
   dc:  7  6

Z0  1:  5
   dc:  7  6
```

complete equation

Da = BX + AX

complete equation

Db = A'B'X

complete equation

Z0 = AX

logic equations

```
   Da = BX + AX
   Db = A'B'X
   Z0 = AX

elapsed time:    0.33 sec

writing VHDL code file

elapsed time:    3.29 sec
```

```
entity Dietmeyer_B
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Dietmeyer_B;



architecture Discrete_Structure of Dietmeyer_B is

  B1: block
    component D_FlipFlop
      port ( D:      in  Bit;
             Q:      out Bit;
             Qnot:   out Bit;
             Clk:    in  Bit );

    component Discrete_Gates
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 4);
    signal  Ynext:  Bit_Vector (0 to 2);

  begin
    Ycur(4) <= X(0);
    Z(0) <= Ynext(2);

    for I in 0 to 1 generate
      Mem: D_FlipFlop
        port ( Ynext(I),  Ycur(2*I), Ycur(2*I+1), Clk );
    end generate;

    Comb: Discrete_Gates
      port ( Ycur, Ynext );
  end block;
end Discrete_Structure;



entity D_FlipFlop
  ( D:      in  Bit;
    Q:      out Bit;
    Qnot:   out Bit;
    Clk:    in  Bit ) is
end D_FlipFlop;



architecture Behavior of D_FlipFlop is

  B1: block (Clk = '1' and not Clk'Stable)
  begin
    P1: process ( Guard )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := D;
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;



entity Discrete_Gates
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Discrete_Gates;


architecture Behavior of Discrete_Gates is

  B1: block
  begin
    Outputs(0) <= (Inputs(2) and Inputs(4)) or (Inputs(0) and Inputs(4)) after 40 ns;
    Outputs(1) <= Inputs(1) and Inputs(3) and Inputs(4) after 20 ns;
    Outputs(2) <= Inputs(0) and Inputs(4) after 20 ns;
  end block;
end Behavior;
```

Kohavi, p291 detect '0101'

initial state table

Q	Q+ X= 0	1	Z X= 0	1
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	2	0	1

with unreachable states removed

Q	Q+ X= 0	1	Z X= 0	1
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	2	0	1

sets of equivalent states

```
1 = {  0  }
4 = {  1  }
3 = {  2  }
2 = {  3  }
```

minimized state table

Q	Q+ X= 0	1	Z X= 0	1
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	2	0	1

Karnaugh maps

```
Ja  1:  3
   dc:  7  6  5  4

Ka  1:  6  5
   dc:  3  2  1  0

Jb  1:  4  0
   dc:  7  6  3  2

Kb  1:  7  3
   dc:  5  4  1  0

Z0  1:  7
   dc:
```

complete equation

Ja = BX

complete equation

Ka = B'X + BX'

```
complete equation

   Jb = X'


complete equation

   Kb = X


complete equation

   Z0 = ABX


logic equations

   Ja = BX
   Ka = B'X + BX'
   Jb = X'
   Kb = X
   Z0 = ABX


elapsed time:    0.49 sec

writing VHDL code file

elapsed time:    3.46 sec
```

```
entity Detect_0101
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Detect_0101;


architecture PLA_Structure of Detect_0101 is

  B1: block
    component JK_FlipFlop
      port ( J, K:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Programmable_Logic_Array
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 2);
    signal  Ynext:  Bit_Vector (0 to 4);

  begin
    Ycur(2) <= X(0);
    Z(0) <= Ynext(4);

    for I in 0 to 1 generate
      Mem: JK_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(I), open, Clk );
    end generate;

    Comb: Programmable_Logic_Array
      port ( Ycur, Ynext );
  end block;
end PLA_Structure;


entity JK_FlipFlop
  ( J, K:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end JK_FlipFlop;


architecture Behavior of JK_FlipFlop is

  B1: block (Clk = '1' and not Clk'Stable)
  begin
    P1: process ( Guard )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := (J and not Qhold) or (not K and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Programmable_Logic_Array
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Programmable_Logic_Array;


architecture Behavior of Programmable_Logic_Array is

  B1: block
  begin
    Outputs(0) <= Inputs(1) and Inputs(2) after 40 ns;
    Outputs(1) <= (not Inputs(1) and Inputs(2)) or (Inputs(1) and not Inputs(2)) after 40 ns;
    Outputs(2) <= not Inputs(2) after 40 ns;
    Outputs(3) <= Inputs(2) after 40 ns;
    Outputs(4) <= Inputs(0) and Inputs(1) and Inputs(2) after 40 ns;
  end block;
end Behavior;
```

Kohavi, p295 modulo 8 counter

initial state table

Q	Q+ X= 0	Q+ X= 1	Z X= 0	Z X= 1
0	0	1	0	0
1	1	2	0	0
2	2	3	0	0
3	3	4	0	0
4	4	5	0	0
5	5	6	0	0
6	6	7	0	0
7	7	0	0	1

with unreachable states removed

Q	Q+ X= 0	Q+ X= 1	Z X= 0	Z X= 1
0	0	1	0	0
1	1	2	0	0
2	2	3	0	0
3	3	4	0	0
4	4	5	0	0
5	5	6	0	0
6	6	7	0	0
7	7	0	0	1

sets of equivalent states

```
1 = {  0  }
8 = {  1  }
7 = {  2  }
6 = {  3  }
5 = {  4  }
4 = {  5  }
3 = {  6  }
2 = {  7  }
```

minimized state table

Q	Q+ X= 0	Q+ X= 1	Z X= 0	Z X= 1
0	0	1	0	0
1	1	2	0	0
2	2	3	0	0
3	3	4	0	0
4	4	5	0	0
5	5	6	0	0
6	6	7	0	0
7	7	0	0	1

Karnaugh maps

```
Da  1: 14 13 12 11 10  9  8  7
   dc:

Db  1: 14 13 12 11  6  5  4  3
   dc:

Dc  1: 14 13 10  9  6  5  2  1
   dc:
```

```
   Z0  1: 15
      dc:


complete equation

   Da = A'BCX + AB' + AC' + AX'


complete equation

   Db = B'CX + BC' + BX'


complete equation

   Dc = C'X + CX'


complete equation

   Z0 = ABCX


logic equations

   Da = A'BCX + AB' + AC' + AX'
   Db = B'CX + BC' + BX'
   Dc = C'X + CX'
   Z0 = ABCX


elapsed time:    1.21 sec

writing VHDL code file

elapsed time:    4.40 sec
```

```
entity Modulo_8
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Modulo_8;


architecture PLA_Structure of Modulo_8 is

  B1: block
    component D_FlipFlop
      port ( D:     in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Programmable_Logic_Array
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 3);
    signal  Ynext:  Bit_Vector (0 to 3);

  begin
    Ycur(3) <= X(0);
    Z(0) <= Ynext(3);

    for I in 0 to 2 generate
      Mem: D_FlipFlop
        port ( Ynext(I),  Ycur(I), open, Clk );
    end generate;

    Comb: Programmable_Logic_Array
      port ( Ycur, Ynext );
  end block;
end PLA_Structure;


entity D_FlipFlop
  ( D:     in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end D_FlipFlop;


architecture Behavior of D_FlipFlop is

  B1: block (Clk = '1' and not Clk'Stable)
  begin
    P1: process ( Guard )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := D;
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Programmable_Logic_Array
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Programmable_Logic_Array;


architecture Behavior of Programmable_Logic_Array is

  B1: block
  begin
    Outputs(0) <= (not Inputs(0) and Inputs(1) and Inputs(2) and Inputs(3)) or
(Inputs(0) and not Inputs(1)) or (Inputs(0) and not Inputs(2)) or
(Inputs(0) and not Inputs(3)) after 40 ns;
    Outputs(1) <= (not Inputs(1) and Inputs(2) and Inputs(3)) or
(Inputs(1) and not Inputs(2)) or (Inputs(1) and not Inputs(3)) after 40 ns;
    Outputs(2) <= (not Inputs(2) and Inputs(3)) or (Inputs(2) and not Inputs(3)) after 40 ns;
    Outputs(3) <= Inputs(0) and Inputs(1) and Inputs(2) and Inputs(3) after 40 ns;
  end block;
end Behavior;
```

Kohavi, p299 parity bit generator

initial state table

Q	Q+ X= 0	1	Z X= 0	1
0	1	2	0	0
1	3	4	0	0
2	4	3	0	0
3	5	6	0	0
4	6	5	0	0
5	0	0	0	0
6	0	0	1	1

with unreachable states removed

Q	Q+ X= 0	1	Z X= 0	1
0	1	2	0	0
1	3	4	0	0
2	4	3	0	0
3	5	6	0	0
4	6	5	0	0
5	0	0	0	0
6	0	0	1	1

sets of equivalent states

```
1 = {  0  }
7 = {  5  }
5 = {  1  }
6 = {  2  }
3 = {  3  }
4 = {  4  }
2 = {  6  }
```

minimized state table

Q	Q+ X= 0	1	Z X= 0	1
0	1	2	0	0
1	3	4	0	0
2	4	3	0	0
3	5	6	0	0
4	6	5	0	0
5	0	0	0	0
6	0	0	1	1

Karnaugh maps

```
Sa  1:  7  6  4  3
   dc: 15 14  9  8

Ra  1: 13 12 11 10
   dc: 15 14  5  2  1  0

Sb  1:  8  2  1
   dc: 15 14  7  5

Rb  1: 13 12  6  4
   dc: 15 14 11 10  9  3  0
```

```
   Sc  1:  9  5  0
     dc: 15 14  6  2

   Rc  1: 11 10  7  3
     dc: 15 14 13 12  8  4  1

   Z0  1: 13 12
     dc: 15 14


complete equation

   Sa = A'CX + A'BX'


complete equation

   Ra = AC + AB


complete equation

   Sb = A'C'X + A'B'CX' + AB'C'X'


essential cells

   Rb = BX'

implicant table

  implicant  |  columns covered
-------------+-------------------
        11-- |  13
        1--1 |  13

complete equation

   Rb = AB + BX'


complete equation

   Sc = A'B'X' + A'BC'X + AB'C'X


essential cells

   Rc = CX

implicant table

  implicant  |  columns covered
-------------+-------------------
        1-1- |  10
        1--0 |  10

complete equation

   Rc = AC + CX


complete equation

   Z0 = AB


logic equations

   Sa = A'CX + A'BX'
```

```
    Ra = AC + AB
    Sb = A'C'X + A'B'CX' + AB'C'X'
    Rb = AB + BX'
    Sc = A'B'X' + A'BC'X + AB'C'X
    Rc = AC + CX
    Z0 = AB

elapsed time:    1.76 sec

writing VHDL code file

elapsed time:    4.83 sec
```

```
entity parity
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end parity;


architecture PLA_Structure of parity is

  B1: block
    component RS_FlipFlop
      port ( S, R:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Programmable_Logic_Array
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 3);
    signal  Ynext:  Bit_Vector (0 to 6);

  begin
    Ycur(3) <= X(0);
    Z(0) <= Ynext(6);

    for I in 0 to 2 generate
      Mem: RS_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(I), open, Clk );
    end generate;

    Comb: Programmable_Logic_Array
      port ( Ycur, Ynext );
  end block;
end PLA_Structure;


entity RS_FlipFlop
  ( S, R:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end RS_FlipFlop;


architecture Behavior of RS_FlipFlop is

  B1: block
  begin
    P1: process ( R, S )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := S or (not R and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Programmable_Logic_Array
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Programmable_Logic_Array;


architecture Behavior of Programmable_Logic_Array is

  B1: block
  begin
    Outputs(0) <= (not Inputs(0) and Inputs(2) and Inputs(3)) or
(not Inputs(0) and Inputs(1) and not Inputs(3)) after 40 ns;
    Outputs(1) <= (Inputs(0) and Inputs(2)) or (Inputs(0) and Inputs(1)) after 40 ns;
    Outputs(2) <= (not Inputs(0) and not Inputs(2) and Inputs(3)) or
(not Inputs(0) and not Inputs(1) and Inputs(2) and not Inputs(3)) or
(Inputs(0) and not Inputs(1) and not Inputs(2) and not Inputs(3)) after 40 ns;
    Outputs(3) <= (Inputs(0) and Inputs(1)) or (Inputs(1) and not Inputs(3)) after 40 ns;
    Outputs(4) <= (not Inputs(0) and not Inputs(1) and not Inputs(3)) or
(not Inputs(0) and Inputs(1) and not Inputs(2) and Inputs(3)) or
(Inputs(0) and not Inputs(1) and not Inputs(2) and Inputs(3)) after 40 ns;
    Outputs(5) <= (Inputs(0) and Inputs(2)) or (Inputs(2) and Inputs(3)) after 40 ns;
    Outputs(6) <= Inputs(0) and Inputs(1) after 40 ns;
  end block;
end Behavior;
```

benchmark, Modulo12

initial state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	1	2	0	0
2	2	3	0	0
3	3	4	0	0
4	4	5	0	0
5	5	6	0	0
6	6	7	0	0
7	7	8	0	0
8	8	9	0	0
9	9	10	0	0
10	10	11	0	0
11	11	0	0	1

with unreachable states removed

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	1	2	0	0
2	2	3	0	0
3	3	4	0	0
4	4	5	0	0
5	5	6	0	0
6	6	7	0	0
7	7	8	0	0
8	8	9	0	0
9	9	10	0	0
10	10	11	0	0
11	11	0	0	1

sets of equivalent states

```
 1 = { 0 }
12 = { 1 }
11 = { 2 }
10 = { 3 }
 9 = { 4 }
 8 = { 5 }
 7 = { 6 }
 6 = { 7 }
 5 = { 8 }
 4 = { 9 }
 3 = { 10 }
 2 = { 11 }
```

minimized state table

Q	Q+ X= 0	1	Z X= 0	1
0	0	1	0	0
1	1	2	0	0
2	2	3	0	0
3	3	4	0	0
4	4	5	0	0
5	5	6	0	0
6	6	7	0	0
7	7	8	0	0

```
  8  |    8    9  |    0    0
  9  |    9   10  |    0    0
 10  |   10   11  |    0    0
 11  |   11    0  |    0    1
```

Karnaugh maps

```
   Sa  1: 15
      dc: 31 30 29 28 27 26 25 24 22 21 20 19 18 17 16

   Ra  1: 23
      dc: 31 30 29 28 27 26 25 24 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0

   Sb  1:  7
      dc: 31 30 29 28 27 26 25 24 14 13 12 11 10  9  8

   Rb  1: 15
      dc: 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16  6  5  4  3  2  1  0

   Sc  1: 19 11  3
      dc: 31 30 29 28 27 26 25 24 22 21 20 14 13 12  6  5  4

   Rc  1: 23 15  7
      dc: 31 30 29 28 27 26 25 24 18 17 16 10  9  8  2  1  0

   Sd  1: 21 17 13  9  5  1
      dc: 31 30 29 28 27 26 25 24 22 18 14 10  6  2

   Rd  1: 23 19 15 11  7  3
      dc: 31 30 29 28 27 26 25 24 20 16 12  8  4  0

   Z0  1: 23
      dc: 31 30 29 28 27 26 25 24
```

complete equation

```
   Sa = BCDX
```

essential cells

```
   Ra = 0
```

implicant table

```
  implicant  |  columns covered
-------------+-------------------
       1-111 |  23
       -0111 |  23
```

complete equation

```
   Ra = ACDX
```

complete equation

```
   Sb = A'B'CDX
```

complete equation

```
   Rb = BCDX
```

complete equation

```
   Sc = C'DX
```

```
complete equation

   Rc = CDX


complete equation

   Sd = D'X


complete equation

   Rd = DX


complete equation

   Z0 = ACDX


logic equations

   Sa = BCDX
   Ra = ACDX
   Sb = A'B'CDX
   Rb = BCDX
   Sc = C'DX
   Rc = CDX
   Sd = D'X
   Rd = DX
   Z0 = ACDX


elapsed time:    2.19 sec

writing VHDL code file

elapsed time:    5.21 sec
```

```
entity Modulo_12
  ( X:    in  Bit_Vector;
    Z:    out Bit_Vector;
    Clk:  in  Bit ) is
end Modulo_12;


architecture Discrete_Structure of Modulo_12 is

  B1: block
    component RS_FlipFlop
      port ( S, R:  in  Bit;
             Q:     out Bit;
             Qnot:  out Bit;
             Clk:   in  Bit );

    component Discrete_Gates
      port ( Inputs:   in  Bit_Vector;
             Outputs:  out Bit_Vector );

    signal  Ycur:  Bit_Vector (0 to 8);
    signal  Ynext:  Bit_Vector (0 to 8);

  begin
    Ycur(8) <= X(0);
    Z(0) <= Ynext(8);

    for I in 0 to 3 generate
      Mem: RS_FlipFlop
        port ( Ynext(2*I),  Ynext(2*I+1), Ycur(2*I), Ycur(2*I+1), Clk );
    end generate;

    Comb: Discrete_Gates
      port ( Ycur, Ynext );
  end block;
end Discrete_Structure;



entity RS_FlipFlop
  ( S, R:  in  Bit;
    Q:     out Bit;
    Qnot:  out Bit;
    Clk:   in  Bit ) is
end RS_FlipFlop;


architecture Behavior of RS_FlipFlop is

  B1: block
  begin
    P1: process ( R, S )
      variable  Qhold:  static Bit := '0';

    begin
      if Guard then
        Qhold := S or (not R and Qhold);
        Q <= Qhold after 50 ns;
        Qnot <= not Qhold after 50 ns;
      end if;
    end process;
  end block;
end Behavior;


entity Discrete_Gates
```

```
  ( Inputs:   in  Bit_Vector;
    Outputs:  out Bit_Vector ) is
end Discrete_Gates;


architecture Behavior of Discrete_Gates is

  B1: block
  begin
    Outputs(0) <= Inputs(2) and Inputs(4) and Inputs(6) and Inputs(8) after 20 ns;
    Outputs(1) <= Inputs(0) and Inputs(4) and Inputs(6) and Inputs(8) after 20 ns;
    Outputs(2) <= Inputs(1) and Inputs(3) and Inputs(4) and Inputs(6) and Inputs(8) after 20 ns;
    Outputs(3) <= Inputs(2) and Inputs(4) and Inputs(6) and Inputs(8) after 20 ns;
    Outputs(4) <= Inputs(5) and Inputs(6) and Inputs(8) after 20 ns;
    Outputs(5) <= Inputs(4) and Inputs(6) and Inputs(8) after 20 ns;
    Outputs(6) <= Inputs(7) and Inputs(8) after 20 ns;
    Outputs(7) <= Inputs(6) and Inputs(8) after 20 ns;
    Outputs(8) <= Inputs(0) and Inputs(4) and Inputs(6) and Inputs(8) after 20 ns;
  end block;
end Behavior;
```

APPENDIX E

MEG OUTPUT COMPARISON

Meg [9] is a finite state machine equation generator. It translates a Mealy model description of a finite state machine into logic equations in several formats, including truth tables and boolean equations. In comparing FSM output to Meg output, some differences must be accounted for. Meg does not attempt to minimize the input state machine so it must be input in simplified form if the results of Meg and FSM are to be comparable. Meg does not consider unused state assignments as don't-care states so only state machines without unused states assignments will produce the similar results from both Meg and FSM. Also, Meg itself does not minimize the equations it produces. This must be done by another program such as Espresso.

The following is an example run with Meg. It corresponds to the third example in appendix D. Given first is the machine description used as input. Meg then produces a state table and logic equations. In the equations produced, symbols generated by Meg end with an asterisk, an exclamation mark preceding a symbol indicates negation, the ampersand signifies conjunction, and the vertical bar signifies disjunction. Following the equations of is a PLA map produced by Espresso. Logic equations for this PLA map in the form used by FSM are also given.

```
 INPUTS:  X;
OUTPUTS:  Z;

Q0:  IF X THEN Q1 ELSE Q0;
Q1:  IF X THEN Q2 ELSE Q1;
Q2:  IF X THEN Q3 ELSE Q2;
Q3:  IF X THEN Q4 ELSE Q3;
Q4:  IF X THEN Q5 ELSE Q4;
Q5:  IF X THEN Q6 ELSE Q5;
Q6:  IF X THEN Q7 ELSE Q6;
Q7:  IF X THEN Q0(Z) ELSE Q7;


SUMMARY INFORMATION GENERATED BY MEG FROM FILE fsm08.meg

INPUTS:
   i00:  X
   s00:  StBit0* (msb)
   s01:  StBit1*
   s02:  StBit2* (lsb)

OUTPUTS:
   n02:  StBit2* (lsb)
   n01:  StBit1*
   n00:  StBit0* (msb)
   o00:  Z

State Table
      i  s  s  s  n  n  n  o
      0  0  1  2  2  1  0  0
   ------------------------
      0  0  0  0  0  0  0  0    Q0
      1  0  0  0  1  0  0  0    Q0

      0  0  0  1  1  0  0  0    Q1
      1  0  0  1  0  1  0  0    Q1

      0  0  1  0  0  1  0  0    Q2
      1  0  1  0  1  1  0  0    Q2

      0  0  1  1  1  1  0  0    Q3
      1  0  1  1  0  0  1  0    Q3

      0  1  0  0  0  0  1  0    Q4
      1  1  0  0  1  0  1  0    Q4

      0  1  0  1  1  0  1  0    Q5
      1  1  0  1  0  1  1  0    Q5

      0  1  1  0  0  1  1  0    Q6
      1  1  1  0  1  1  1  0    Q6

      0  1  1  1  1  1  1  0    Q7
      1  1  1  1  0  0  0  1    Q7


INORDER=
   X
   StBit0*
   StBit1*
   StBit2*;
OUTORDER=
   StBit2*
   StBit1*
   StBit0*
   Z;
StBit2*=
      (!X& StBit0*& StBit1*& StBit2*)|
      ( X& StBit0*& StBit1*&!StBit2*)|
      (!X& StBit0*&!StBit1*& StBit2*)|
```

```
        ( X& StBit0*&!StBit1*&!StBit2*)|
        (!X&!StBit0*& StBit1*& StBit2*)|
        ( X&!StBit0*& StBit1*&!StBit2*)|
        (!X&!StBit0*&!StBit1*& StBit2*)|
        ( X&!StBit0*&!StBit1*&!StBit2*);
StBit1*=
        (!X& StBit0*& StBit1*& StBit2*)|
        ( X& StBit0*& StBit1*&!StBit2*)|
        (!X& StBit0*& StBit1*&!StBit2*)|
        ( X& StBit0*&!StBit1*& StBit2*)|
        (!X&!StBit0*& StBit1*& StBit2*)|
        ( X&!StBit0*& StBit1*&!StBit2*)|
        (!X&!StBit0*& StBit1*&!StBit2*)|
        ( X&!StBit0*&!StBit1*& StBit2*);
StBit0*=
        (!X& StBit0*& StBit1*& StBit2*)|
        ( X& StBit0*& StBit1*&!StBit2*)|
        (!X& StBit0*& StBit1*&!StBit2*)|
        ( X& StBit0*&!StBit1*& StBit2*)|
        (!X& StBit0*&!StBit1*& StBit2*)|
        ( X& StBit0*&!StBit1*&!StBit2*)|
        (!X& StBit0*&!StBit1*&!StBit2*)|
        ( X&!StBit0*& StBit1*& StBit2*);
Z=
        ( X& StBit0*& StBit1*& StBit2*);


.ilb X StBit0* StBit1* StBit2*
.ob StBit2* StBit1* StBit0* Z
.i 4
.o 4
.p 10
1011 0010
1111 0001
01-- 0010
1-01 0100
-1-0 0010
0-1- 0100
1--0 1000           Z0 = XABC
-10- 0010           Da = XA'BC + X'A + AC' + AB'
0--1 1000           Db = XB'C + X'B + BC'
--10 0100           Dc = XC' + X'C
.e
```