A STATE-MACHINE SYNTHESIZER — SMS

Douglas W. Brown
Del. Stat. 92-515
Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

ABSTRACT

Much of the work in implementing a state machine involves tedious calculations that require no creativity. This report describes the development of a digital-circuit synthesis program that helps reduce the tedium. SMS accepts a high-level description of a state machine and returns equations for implementation that assume a sum-of-products next-state and output functions and that also assume JK or D flip-flops for memory.

INTRODUCTION

A state machine has been defined by Clare[1] to be a circuit block containing memory and combinational logic. The memory defines the state of the circuit block. The logic defines the outputs and the next state as a function of the inputs and of the current state.

State machines have not been replaced by microprocessors. State machines are still needed to interface with microprocessors and for high-speed applications. Also, state-machine synthesis will become an important part of VLSI design.

Most of the creativity in designing state machines is in drawing the ASM (algorithmic state machine) chart, as shown by Clare,[1] and in assigning states. The rest of the work is well-suited for automation.

This paper describes the development of a state-machine synthesizer program named SMS. To use SMS, an engineer prepares a text file containing a high-level description of a state machine. SMS reads that file and prints another file that contains the equations for implementing the state machine. Implementation uses either JK or D flip-flops for state memory and assumes sum-of-products equations for the next-state and output functions. An example state-machine design using SMS is described in the Appendix of this paper. A different approach to a similar problem is described by Dietmeyer[2] and by Dietmeyer and Doshi[3].

The major engineering problem encountered in the project was the equation reducer. The method of calculating equations produced results that were not reduced enough for production. An algorithm had to be developed that could reduce equations for the size of state machines we had. A description of the algorithms that were developed is included in this paper.

GOALS OF THE PROJECT

The main purpose of the SMS project was to replace a similar program with one that could handle more variables and that had a friendlier user interface. The existing program had an exponential rise in computation time for the equation-reduction step as the number of variables increased and became impractical for state machines that had more than ten inputs. Engineers had state machines that exceeded this limit, so faster computation for the equation-reduction step was sought.

Engineers had also requested a friendlier interface. This request was satisfied by developing a manual, an input language, and an input language processor that provide the following features:

- The manual is small and easy to read. The engineer wants to spend minimum time reading the manual in order to learn how to describe a state machine.
- No ambiguous constructs are allowed in the input language. This constraint eliminates user traps and allows the engineer to quickly describe a state machine. The description will either be interpreted the way the engineer intended or it will not be a valid description.
- The preceding feature is unusable unless the input language processor gives error messages that enable the engineer to easily see the error and correct the state-machine description. Good error messages are a must.

STRUCTURE OF THE PROGRAM

The program is divided into modules, which are separate programs that communicate through text files. The advantages of using modules are:

- Any module can be easily replaced. For example, four different versions of the equation reducer have already been used.
- The most efficient language can be used for each module. It was felt that the proper choice of a language can have a dramatic effect on the simplicity of the program and, therefore, on the reliability. The advantage of the description of an algorithm in a well-suited language has to be weighed against the problem that arises when a language is so esoteric that few programmers can use the language or that the computer system staff cannot support the compiler. (For a discussion of the importance of simplicity, reliability, and adaptability of languages versus portability, efficiency, and generality, see Hansen[4].)
- Communication between the modules through text files has several advantages. Test data can be entered into a text file in order to test a module during initial development. A module can be re-run without the need for special code to save and restore the data for each run. This feature can be used to debug or to optimize a module. An error can be traced to a module by examining the text file, so the file saves some special data-printing code during debugging.

COMPONENTS OF THE PROGRAM

SMS is divided into four modules according to function:

- The Input Language Processor reads and checks the engineer's high-level description of a state machine.
- The Equation Calculator calculates next-state and output functions for flip-flops.
- The Equation Reducer reduces the equations to reduce the size of the resulting circuit.
- The Equation Printer prints the results.

Input Language Processor

The input-language processor was written in FLAK, which is a compiler-compiler developed at Tektronix for in-house use. The input processor checks for syntax errors along with the easier semantic errors, such as duplicate state assignments. There is also type-checking for names to avoid confusion between the names of states, state variables, state-machine inputs, and state-machine outputs. Error messages are written, when appropriate, along with some warning messages for possible errors, such as multiple transitions to the same state.

The input processor passes information on to the equation calculator in an intermediate-level language (ILL). Although the ILL is still in human-readable form, it is much more computer readable because numbers have replaced all but the initial definition of names and reverse-polish notation has replaced (boolean) algebraic equations.

The choice of any general-purpose language to develop the input processor would have resulted in a much larger program (unless a set of routines for compiler-generation were available, which is what the compiler-compiler is). The input processor was able to achieve the described goals in just over 300 lines of FLAK code.

Equation Calculator

Another advantage of the input processor converting the input to an ILL, besides making the fields of data easier to read by the equation calculator, is allowing the equation calculator to assume that the syntax is correct at this point. This advantage eliminates the need for complicated error-recovery code. (This task is pushed off onto the input processor. Fortunately, the processor is written in a language that makes error recovery easy.)

The equation calculator was written in PASCAL. Because the calculator has to do boolean operations on boolean functions and various traversals of trees describing the state machine, a language that could handle unusual data types and that could do type checking was required. PASCAL was the best supported language available that fit this requirement.

Before the equation calculator calculates equations, it does some checking that is too complicated to do easily in the input processor. This checking includes seeing that all of the conditions for transitions to next states from a given state are disjoint. Then, the calculator writes out the flip-flop equations. The next-state and output tables are generated during data read and write. The calculation of the equations is a straightforward process, as described by Clare[1].

Equation Reducer

The equation reducer is the computation-intensive part of the program. The equation reducer was written in FORTRAN because that was the most efficient language on the machine.

The equation reducer reads a multiple-output sum-of-products function, with incomplete specification ("don't care's" on some outputs); performs a sub-optimal reduction; and writes out the function in a completely specified form. One requirement was that the reducer had to reduce both the product terms and the number of literals in the product terms. PLA-directed reduction would only require that the number of product terms be reduced. Four equation-reduction modules have been implemented so far.

The first reducer was based on Hong's MINI[5]. MINI uses heuristic methods to reduce the equations. The algorithm is able to handle multi-valued logic and is geared toward PLA's (reduces only the number of product terms). However, it was easy to implement the degenerate case of boolean logic and to add a final step that reduces the literals in the product terms. The resulting equations were quite acceptable for SMS, but the amounts of computer time and required memory were not.

The second implementation of the equation reducer module was Svoboda's PRESTO[6]. The algorithm is described in more detail later in this paper. The reduction in PRESTO was generally almost as good as in MINI, and the computer time and memory requirements were drastically reduced, making the package much more acceptable for SMS. The only drawback to the original PRESTO was a sharp rise in execution time for a large number of variables. This increased execution time was reasonable for all the state machines that had come along at that time, but there was a strong possibility that someone would try a state machine with too many input variables (say thirty) and would be unable to run the reducer in a practical amount of time.

The current reducer is an extended version of PRESTO (details follow). The original PRESTO contained a step that had to check every minterm of a product term, which made it sensitive to the number of variables, but extended PRESTO uses a tree algorithm for the same step and keeps the time more reasonable. This implementation provides the same reduction results with negligibly increased memory requirements. Extended PRESTO is so thrifty with computer time and memory (including the size of the program) that even a microprocessor version is practical.

Definitions. In the following discussions of algorithms, "function" is a boolean multiple-output sum-of-products. The function consists of a list of product terms, each representing an AND gate. It is convenient to let the outputs to which an AND gate is connected be associated with the product term for that gate. Each input literal can be present in either its true or its complemented form, or each input literal can be absent. If the literal is present, the AND gate is connected to that input. Each output can be present or absent. If the literal is present, the output of the AND gate is connected to the input of the OR gate that forms that output. Figure 1 demonstrates the relationship between inputs, outputs, product terms, and gates.

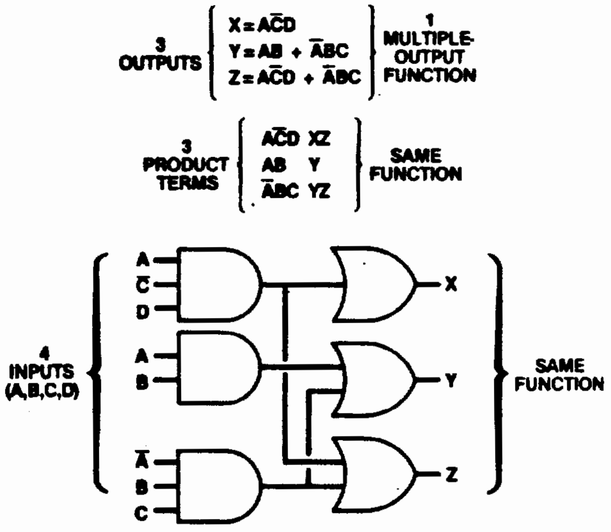

Figure 1. Relationship between inputs, outputs, product terms, and gates.

The concepts of "parts" and "distance" between product terms were taken from Hong, *et al*,[5] and concepts of "cdistance" and "jdistance" were added in order to simplify algorithm descriptions. The position for each input literal in a product term is a part. The positions for all output literals in a product term make one part. Figure 2 shows an example of parts.

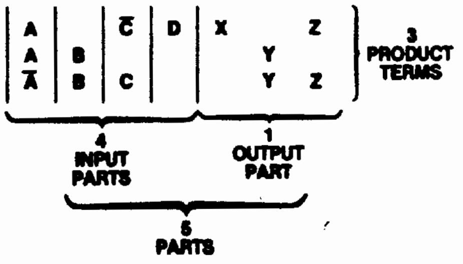

Figure 2. Example of "parts" in product terms.

Let A and B be product terms. An *input part* in A *covers* the corresponding part in B if the literal is absent from the part in A or if the literal is the same (both true or both complemented) in A and B. The *output part* in A *covers* the output part in B if, for each literal present in the output of B, the corresponding literal is also present in the output of A. An *input part* in A is *disjoint* from the corresponding part in B if the literal is present in the part and different (one true and the other complemented) in A and B. The *output part* of A is *disjoint* from the output part of B if, for each literal present in the output of A, the corresponding literal is absent in the output of B. A *part* of A *intersects* the corresponding part of B if the parts are not disjoint.

The distance between two product terms is the number of parts in which they differ. A distance of zero means the product terms are equal. A distance of one means the product terms can be merged. If the difference is in an input part, then the merged product term will have no literal in that part. If the difference is in the output part, then the merged product term will have all the literals that were present in either of the merged product terms.

The cdistance, or "cover distance," from product term A to B is the number of parts in A that do not cover the corresponding parts in B. If the cdistance from A to B is zero, then A covers B; otherwise, A does not cover B. The concept of how close a product term comes to covering another is valuable in the extended PRESTO algorithm.

The jdistance between product terms A and B is the number of parts in A that are disjoint from the corresponding parts in B. The jdistance is also the minimum distance between a pair of minterms of A and B. A jdistance of zero means two product terms intersect.

Function F intersects product term PT if any product term in F intersects PT. Function F1 intersects function F2 if any product term of F1 intersects any product term of F2. Coverage with functions is best described with minterms (the reason is involved in the step from PRESTO to extended PRESTO). A function F covers a product term PT if every minterm of PT appears in F. Function F1 covers function F2 if every minterm of F2 appears in F1.

<u>Details of PRESTO</u>. The theoretical details of PRESTO are described by Svoboda[6]. The following is a description of the algorithm. PRESTO accepts an incompletely specified multiple-output sum-of-products function. The incomplete specification means there are "don't cares" in the outputs. PRESTO forms two functions, F and FDC, from the original function. F is the original function with the "don't-cares" all set to zero, and FDC has them set to one. Thus, F has the minimum number of minterms for an acceptable final solution, and FDC has the maximum. PRESTO works with F by adding minterms from FDC that reduce the resulting circuit.

The main loop for PRESTO follows:

- First, try to eliminate each input literal. Take each input literal in each product term in order, remove the literal, and see if the product term is still covered by FDC. If the product term is still covered, then leave the literal out; if not, put the literal back.

- Next, try to eliminate each output literal. Take each output literal in each product term in order, remove the output literal, and see if the product term being uncovered by the removal of the output literal is still covered by the rest of F. If all the output literals of any product term are removed, then the product term can be removed, which reduces the size of the circuit by one AND gate or product term.

- Repeat the preceding two steps until there is no change in F. F is the result.

PRESTO is currently order dependent. Antonin Svoboda (the author of PRESTO) chose a minimal set of operations that were fast and that required little computer memory. Ordering of the product terms or of the parts within the product terms would result in greater reduction; however, for SMS, the extra reduction would probably not be worth the cost.

In the original PRESTO, the test for a function covering a product term was done as a loop through all the minterms of the product term. This test was a simple algorithm and has been acceptable for all SMS problems seen to date. But the potential exists for a state machine that is too large to reduce practically with PRESTO, so a faster method of testing product-term coverage was sought.

<u>Extended PRESTO</u>. Extended PRESTO contains a tree method of checking to see if a product term is covered by a function rather than checking every minterm in the product term. Let the product term we are checking be PTO, and let the function be G. We will work with product term PT, which will start out equal to PTO. The stack starts out empty. The steps of the coverage test are:

1. If PT is covered by any product term of G and the stack is empty, then PTO is covered by G.

2. If PT is covered by any product term of G and the stack is NOT empty, then pop a new PT off the stack and go to step 1.

3. If PT does not intersect G, then PTO is not covered by G;

4. otherwise, push half of PT onto the stack and leave the other half in PT. Go to step 1.

"Splitting in half" is taking one product term and forming two that differ in one part. When splitting on an input part, the original product term will have the input literal absent in the splitting part, and the resultant product terms will have the literal and its complement in that part. When splitting on the output part, each output literal of the original product term goes either to one half or to the other but not to both.

Splitting in half is truly splitting the number of minterms into two equal groups on input splits. The worst-case of the tree method is to split PTO down to its individual minterms. How the product term is split determines how close we come to worst-case.

A "dumb" split would just take the next part of the product term with a literal absent and split there. This split was tested and found to be unacceptably slow. A "smart" split is necessary. The method chosen is to find a product term PTG in G that intersects PT and has the shortest cdistance to PT. Split along a part in PT that is not covered by the corresponding part in PTG.

An efficient output split is: Let H be the product terms whose input parts all cover the corresponding parts of PT. Remove from PT any output literals found in H. PT is the result, and nothing is pushed onto the stack. (The product term that would have been pushed onto the stack is already known to be covered.) This output split can be used only when no input splits are possible.

Let A, B, C, D be inputs and X, Y, Z be outputs.

Input split example:

$A\ \overline{C}\ D\ X\ Z$ ——> $A\ B\ \overline{C}\ D\ X\ Z$ and $A\ \overline{B}\ \overline{C}\ D\ X\ Z$.

Output split example:

$A\ \overline{C}\ D\ X\ Z$ ——> $A\ \overline{C}\ D\ X$ and $A\ \overline{C}\ D\ Z$.

Dumb split examples:

$A\ D\ X\ Y$ ——> $A\ B\ D\ X\ Y$ and $A\ \overline{B}\ D\ X\ Y$

$A\ B\ D\ X\ Y$ ——> $A\ B\ C\ D\ X\ Y$ and $A\ B\ \overline{C}\ D\ X\ Y$.

Smart split example:

PT = $A\ D\ X\ Y$

G = $\overline{A}\ D\ X\ Y,\ A\ \overline{C}\ D\ X\ Z,\ A\ \overline{B}\ C\ D\ X\ Z$.

product term of G	JDISTANCE to PT	CDISTANCE to PT
$\overline{A} D X Y$	1	1
$A \overline{C} D X Z$	0	2
$A \overline{B} C D X Z$	0	3

$\overline{A}DXY$ cannot be PTG because it does not intersect PT (JDISTANCE does not equal 0). $A\overline{C}DXZ$ has the minimum CDISTANCE of the remaining product terms. So

PTG = $A \overline{C} D X Z$.

Split the part containing literal C, because for all other input parts, the part in PTG covers the corresponding part in PT.

$A D X Y \longrightarrow A C D X Y$ and $A \overline{C} D X Y$

The current implementation of PRESTO is limited to 29 inputs, 59 outputs, and 1500 product terms. The first two can be extended with a small amount of coding, and the last is a single parameter. Table 1 lists example runs that give an idea of execution time.

Table 1.
Equation Reducer Timing Examples

RUN	INPUTS	OUTPUTS	PRODUCT TERMS BEFORE REDUCTION	PRODUCT TERMS AFTER REDUCTION	CYBER 175 CPU SECONDS
1	6	6	31	14	0.09
2	10	9	28	15	0.10
3	9	8	33	18	0.16
4	14	5	85	47	0.67
5	9	14	47	23	0.15
6	16	16	103	38	0.87
7	15	23	105	42	1.25
8	12	25	235	68	2.96

An experimental version of SMS uses Svoboda's absolute minimizer OPTIMA[7] for the reduction step. OPTIMA is a good complement to PRESTO because PRESTO quickly reduces equations that are too complex for any known technique to find the absolute minimum, while OPTIMA finds the absolute minimum for the simpler equations.

Equation Printer

The last module of the SMS program is the equation printer. This module prints a list of all the product terms in the solution, a list of the product terms summed in each output, and a list of the sum-of-products for each output. Like the equation calculator, the equation printer processes complex data structures, so PASCAL was a good language for this application.

The initial project called for printing the results. A program has been written to link the SMS output to a PROM programmer for programming FPLA's.

Manual

Although not a program module, the manual is an important part of the project. The manual, the input language, and the input processor must all work together to form an attractive tool for the engineer. The approach used in the manual is a tutorial that is quick to read. The manual is not necessarily a comprehensive reference work. The tutorial consists of a series of examples starting with a basic state machine, with each example adding another feature. The input language is designed to make this possible, and the input processor has to give good feedback to make up for lack of detail in the manual. Regarding the latter case, the manual does not even mention maximum name lengths, because the input processor's error feedback is sufficient to notify the engineer.

RESULTS OF PROJECT

SMS has about five regular users and about twenty users who have tried it. We expect the number of users to gradually increase judging from experience with previous circuit-design-aid programs. Most engineers have not yet gained complete faith in SMS. They use it as a check on, instead of as a replacement for, their work. SMS has caught a few errors made by the engineers, typically a product term lost in the equation-reduction step. Finding these errors has saved considerable time, because the errors were corrected before a prototype was built. Occasionally, a lost product term can be difficult to detect.

Feedback from engineers has been encouraging. The basic goals of the project have been achieved satisfactorily. No engineer has been unable to design a state machine using SMS due to the state machine size, and no engineer has complained about the interface. Negative feedback has been in the form of requests for more features. Graphic input and output is a favorite request. Automatic state assignment plus hazard and race detection are also requested. There are scattered needs for state machines that use something other than JK or D flip-flops as state memory, e.g., shift registers.

CONCLUSION

A state-machine-synthesis program (SMS) was developed that provides the friendly interface and fast equation reduction required to be useful. A new, non-optimal, multiple-output-logic-function reducer was also developed. Engineers have used SMS successfully and have suggested areas for improvement.

ACKNOWLEDGMENTS

Philip White, Ron Bohlman, Allen Hollister, Graeme Boyle, and Henry Alward all made important contributions to SMS. Antonin Svoboda's PRESTO algorithm is currently an indispensable part of SMS. The author wishes to thank the reviewer who pointed out several key typographical errors and who helped further by improving a confusing paragraph.

APPENDIX—EXAMPLE STATE MACHINE

The example state machine shown in Figure 3 has four states: ALPHA, BETA, GAMMA, and ERROR that have state assignments 00, 01, 10, and 11, respectively. The two inputs are X and Y, and the three outputs are PHASE1, PHASE2, and ERROROUT. The next state after ALPHA is BETA if X is high; otherwise, the next state is GAMMA. The next state after BETA is ALPHA if Y is high; otherwise, the state remains BETA. GAMMA goes to ALPHA if both X and Y are high; otherwise, GAMMA goes to ERROR. ERROR never goes to another state. BETA outputs PHASE2 if Y is high; otherwise, BETA outputs PHASE1. ERROR unconditionally outputs ERROROUT.

Figure 3 shows an ASM chart for the state machine:

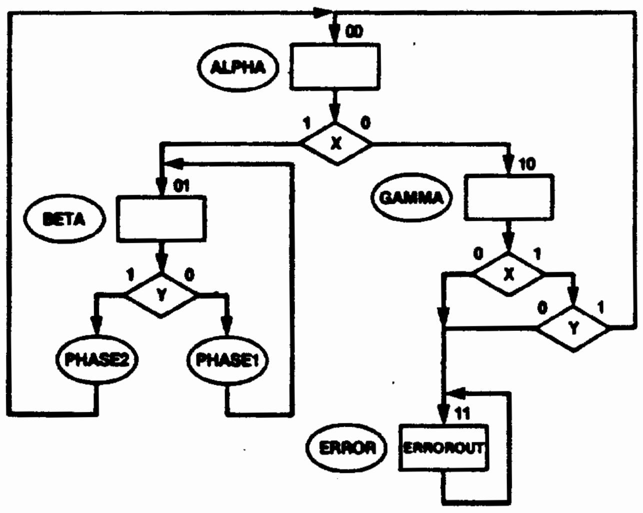

Figure 3. ASM chart, as described by Clare[1], for example state machine.

This example uses JK flip-flops S1 and S2 for the state memory. The following is the SMS input for this state machine:

```
SMS EXAMPLE FOR THE 18TH DAC

STATE ALPHA 00
        ON X    GOTO BETA
        ON -X GOTO GAMMA

STATE BETA 01
        ON Y
                GOTO ALPHA
                OUT PHASE2
        ON -Y
                GOTO BETA
                OUT PHASE1

STATE GAMMA 10
        ON X*Y GOTO ALPHA
        ON -(X*Y) GOTO ERROR

STATE ERROR 11
        UGOTO ERROR
        UOUT ERROROUT

VARDEF  S1  S2

TYPEFF  JKFF
```

The following is the SMS output for the example state machine:

```
P1=X*-S1
P2=-X*-S2
P3=Y*-S1*S2
P4=-Y*-S1*S2
P5=X*Y*-S2
P6=-X*S1
P7=-Y*S1
P8=S1*S2

S1(J)=P2
S1(K)=P5
S2(J)=P1+P6+P7
S2(K)=P3
PHASE2=P3
PHASE1=P4
ERROROUT=P8

S1(J)=-X*-S2
S1(K)=X*Y*-S2
S2(J)=X*-S1+-X*S1+-Y*S1
S2(K)=Y*-S1*S2
PHASE2=Y*-S1*S2
PHASE1=-Y*-S1*S2
ERROROUT=S1*S2
```

Figure 4 shows an implementation of the state machine:

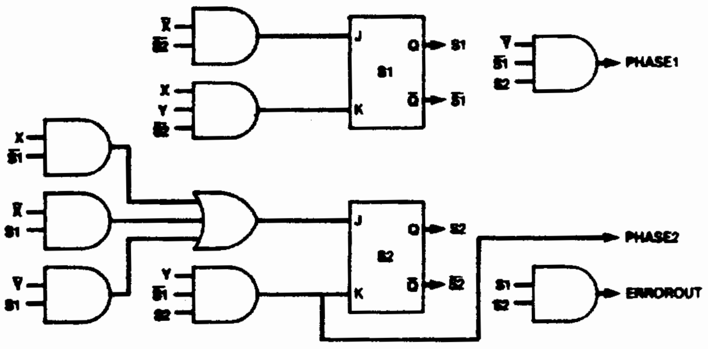

Figure 4. Implementation of example state machine.

REFERENCES

[1]Christopher R. Clare, *Designing Logic Systems Using State Machines*, (McGraw-Hill Book Company, New York, 1973).

[2]D.L. Dietmeyer, "Connection arrays from equations," *Journal of Design Automation and Fault-Tolerant Computing*, vol. 3, pp. 109-125, (April 1979).

[3]D.L. Dietmeyer and M.H. Doshi, "Automated PLA synthesis of the combinational logic of a DDL description," *Journal of Design Automation and Fault-Tolerant Computing*, vol. 3, pp. 241-257, (Winter 1979).

[4]Per Brinch Hansen, *The Architecture of Concurrent Programs*, (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977), ch. 1.

[5]S.J. Hong, R.G. Cain, D.L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM Journal of Research and Development*, pp. 443-458, (September, 1974).

[6]A. Svoboda, "Fast multiple output logical circuit minimization." Svoboda died before he could publish this work. Douglas W. Brown, the author of this paper, has the preliminary version of Svoboda's final report on PRESTO to Tektronix, Inc. This preliminary version is based on: A. Svoboda, "The concept of term exclusiveness and its effect on the theory of boolean functions," *Journal of the Association of Computing Machinery*, vol. 22, no. 3, pp. 425-440, (July 1975) and on R.C. DeVries and A. Svoboda, "Multiple output minimization with mosaics of boolean functions," *IEEE Transactions on Computers*, vol. C-24, no. 8, pp. 777-785, (August 1975).

[7]A. Svoboda and Donnamaie E. White, *Advanced Logical Circuit Design Techniques*, (Garland Press, New York, 1979), ch. 4 and 5.