INT. J. ELECTRONICS, 1991, VOL. 71, NO. 1, 1–12

SADIQ M. SAIT, ELMALEH AH • STATE MACHINE SYNTHESIS WITH WEINBERGER ARRAYS • INTERNATIONAL JOURNAL OF ELECTRONICS 71 (1): 1-12 JUL 1991

State machine synthesis with Weinberger arrays

SADIQ M. SAIT† and AIMAN H. EL-MALEH‡

The development of a digital circuit synthesis program is described. The program accepts the transition table of a state machine and returns equations for an implementation that assumes a sum-of-product next-state and output functions. From the equations for the next-state and output functions, nMOS VLSI layout for a Weinberger array is generated. D flip-flops are assumed for memory elements. Using this tool, tedious manual calculations can be avoided and layouts can be generated automatically from state table descriptions.

1. Introduction

A state machine is a sequential circuit containing memory and combinational logic. The contents of memory define the state, and the logic defines the output and the next state as a function of the current state and the external inputs. State machines are widely used by engineers for digital circuit design. Synthesis of state machines is becoming an important part of VLSI design (Clare 1973).

Since much of the work in implementing a state machine involves tedious calculations, it is preferable that once the transition table of a state machine has been specified the final design and implementation are obtained automatically.

In VLSI design of state machines, in general, programmable logic arrays (PLAs) are used for combinational logic (Mead and Conway 1980, Ayres 1983). In this paper, we synthesize the circuit using Weinberger arrays (WAs) (Weinberger 1967). The structure of a WA, its advantages and disadvantages, are discussed by Ullman (1984), Mukherjee (1986), Sait and Al-Khulaiwi (1990) and Sait and Al-Rashed (1990).

Since the combinational logic is for a state machine, there is a constraint on the order of rows. This is to avoid the feedback lines through the memory from crossing each other. The other constraint arises from the fact that inputs for both the present-state and next-state signals must appear on the same side of the array as the flip-flops. These constraints make the optimization of the generated WA difficult.

The development of a state machine synthesizer program is discussed in this paper; it calculates and reduces the equations for the variables (that is, previous state, input, next state and output) of the state table. In addition, the program generates a WA for the given state machine if necessary. Column folding is attempted but the optimization of the array with the above constraints is still an

Received 30 January 1990; revision received 22 August 1990; accepted 22 August 1990.

†Department of Computer Engineering, King Fahd University of Petroleum and Minerals, 673, Dhahran-31261, Saudi Arabia.

‡Department of Electrical and Computer Engineering, University of Victoria, P.O. Box 3055, Victoria, B.C., Canada V8W 3P6.

open problem. The algorithms used and the implementation details of the state machine synthesizer program, also the generation of the WA for the modelled state machine are discussed.

2. State machine synthesizer

The algorithms used in the development of the state machine synthesizer program are briefly presented. The notation used, the main algorithm, a generalized Quine procedure for the generation of the prime implicants and the algorithm for testing tautology (Breuer 1972, McCluskey 1956 and Ullman 1984) are discussed, also the input/output format to the synthesizer and the data structure used in the implementation are discussed.

2.1. *Definitions*

Boolean switching functions can be represented in two equivalent forms: the normal form boolean switching expression and a cubical representation. For example, the expression $x_1\bar{x}_2+x_1x_3$ is represented in its cubical form as $10x$, $1x1$. The cubical notation is used in this paper because it is concise and lends itself to direct, easy computer implementation.

(*a*) An n-tuple $c=(c_1c_2\ldots c_n)$ where $c_i\in\{0,1,x\}$ is said to be a cube. A 0-cube is an n-tuple $c=(c_1c_2\ldots c_n)$ where $c_i\in\{0,1\}$. Given that C is a set of cubes, $K^0(C)$, the 0-complex defined by C, is the set of all 0-cubes, each of which is covered by some element of C. A cube c defines a boolean product term $P(c)$, and vice versa. For example, $P(10x0)=x_1\bar{x}_2\bar{x}_4$, where $\dot{x}_1,\dot{x}_2,\ldots,\dot{x}_n$ are the n variables of a switching function, and x_i and $\bar{x}_i$ are said to be the true and false literals associated with the variable $\dot{x}_i$. In general, if we set $\dot{x}_i^1=x_i$, $\dot{x}_i^0=\bar{x}_i$, $\dot{x}_i^x=1$, then $P(c)=\prod_{i=1}^{n}\dot{x}_i^{c_i}$.

If r elements of c are x's, we say that c is an r-cube. An r-cube is said to cover or contain 2^r 0-cubes, namely all those 0-cubes which can be obtained from c by replacing the x's by 0's and 1's.

(*b*) *Subsuming* ($\Rightarrow$). Let $a=(a_1a_2\ldots a_n)$ and $b=(b_1b_2\ldots b_n)$ be two arbitrary cubes. We say cube a subsumes cube b; written $a\Rightarrow b$ iff all the 0-cubes covered by a are also covered by b, i.e. $K^0(a)\subseteq K^0(b)$. For example, $11x$ subsumes $1xx$ and, equivalently, x_1x_2 subsumes x_1. If a subsumes b, then equivalently we can say that a is contained in or covered by b.

If C is a cover of a complex K^0, and if $a\Rightarrow b$ where $a,b\in C$, then $C-a$ is also a cover of K^0. Let C' be obtained from C by deleting all cubes $a\in C$ such that $a\Rightarrow b$ for some $b\in C$, where $a\neq b$. This operation of deriving C' from C is called subsuming, and it is denoted by $C'=S[C]$.

(*c*) *Consensus* (ç). The consensus of two product terms P and Q is the largest product R such that R does not imply either P or Q, but R implies $P+Q$. The consensus of P and Q is defined only for the case where there exists exactly one i such that x_i is a literal in P and $\bar{x}_i$ is a literal in Q. In this case, we can write $P=x_i.P'$ and $Q=\bar{x}_i.Q'$, where P' and Q' are not functions of $\dot{x}_i$, and therefore the consensus of P and Q is $P'Q'$.

The consensus operator is defined by the following coordinate table:

	ç	b_i: 0	1	x
a_i	0	0	y	0
	1	y	1	1
	x	0	1	x

(x and y are symbols used for the purpose of definition).

If $a_i \text{ ç } b_i = y$ for exactly one i, then

$a \text{ ç } b = (m(a_1 \text{ ç } b_1),\ m(a_2 \text{ ç } b_2), \ldots, m(a_n \text{ ç } b_n))$,

where $m(0)=0$, $m(1)=1$, and $m(x)=m(y)=x$

Otherwise $a \text{ ç } b$ is undefined.

$C \text{ ç } C$ is defined by the equation $C \text{ ç } C = \{c^i \text{ ç } c^j \mid c^i, c^j \in C, i<j\}$

Examples are

$011 \text{ ç } 111 = x11$

$x11 \text{ ç } x00 = \Omega$ (undefined)

$x01 \text{ ç } 1x0 = 10x$

(*d*) *Sharp product* (♯). The sharp product (♯-product) between two cubes a and b, denoted by $a \sharp b$ is defined to be the set of cubes such that $P(a \sharp b)$ is the set of all prime implicants of the function defined by $P(a)P(b)$. Equivalently, $a \sharp b$ is the set of all prime implicants for the complex $K^0(a) - [K^0(a) \cap K^0(b)]$. Algebraically, $a \sharp b$ is defined according to the following coordinate table:

	♯	b_i: 0	1	x
a_i	0	z	y	z
	1	y	z	z
	x	1	0	z

(y and z are symbols used for the purpose of definition)

$a \sharp b = a$ if $a_i \sharp b_i = y$ for any i;

$a \sharp b = \phi$ if $a_i \sharp b_i = z$ for all i

Otherwise,

$$a \sharp b = \bigcup_i (a_1 a_2 \ldots a_{i-1} \alpha_i a_{i+1} \ldots a_n), \text{ where}$$

$a_i \sharp b_i = \alpha_i \in \{0, 1\}$, and the union runs over all i.

By definition,

$$C' \sharp b = S\left[\bigcup_{c' \in C'} c' \sharp b\right] \quad \text{and} \quad a \sharp C = S[(\ldots(a \sharp c^{i_1}) \sharp c^{i_2}) \ldots) \sharp c^{i_p})]$$

where $i_1, i_2, \ldots, i_p$ is any permutation of $1, 2, \ldots, p$ and $p = |C|$.

Also,

$$C' \sharp C = S\left[\bigcup_{c' \in C'} c' \sharp C\right]$$

Sharping can be thought of as a subtraction process because $C' \sharp C$ is a cover for all vertices in C' which are not in C (Breuer 1972).

2.2. *Components of the program*

The synthesizer program is divided into five main modules. The function of each module is briefly described below.

(i) *Expand module*: used to expand the truth/state table entered by the user, if needed, from singular cover form into 0's and 1's form

(ii) *Quine module*: used to calculate the equations for the next state and output functions

(iii) *FCCOVER module*: used to reduce the equations to reduce the size of the resulting circuit by forming an irredundant cover

(iv) *FNOR module*: used to represent the equations in NOR form and label the product terms for the functions

(v) *Weinberger module*: used to generate a WA for the given machine

When the truth/state table for the machine has been read, the expand module is called. Every column of the previous state and input variables in the table is checked. If a 2 (meaning a do not care) is found in any row, the row containing it will be replaced by two rows one having 0 instead of the 2 and one having 1. Following this method, the table will be expanded to 0-cubes in order to find the prime implicants for every function. More details of the algorithms are given in the next section.

2.3. *Algorithms*

The main difficulty with the simplification procedures is that they require the generation of the set of prime implicants Z which can be quite large. A technique for rapidly obtaining an initial connection cover $\hat{C}_o$ is used. Once $\hat{C}_o$ is obtained, redundancies in it can be eliminated, and an irredundant connection cover obtained. In this procedure each function f^i is simplified individually, hence the amount of computation grows only linearly with the number of functions (Breuer 1972 and Ullman 1984).

2.3.1. *Main algorithm*

Step 1

For $i=1$ to number of functions do
Generate an irredundant cover C^i for function f^i from initial covers C_o^i and DC^i. (C_o^i represents the true vertices of f^i, DC^i represents the do not care vertices of f^i.)

Step 2

$CCOVER = \phi$
For $i=1$ to number of functions do
$CCOVER = CCOVER \cup C^i$.

Step 3

For each $c \in CCOVER$ do
Obtain an element ce in $CCOVER$, where $e_i = 1$ iff $c \not\equiv C^i = \phi$
Otherwise
$e_i = 2$, that is, this cube c is not a product term for function i

Step 4

For each row r of *CCOVER* do

Raise column l of vector **e** by placing 2 instead of 1 in column e_i if the test below is positive:

(i) Find the set R of rows other than r consisting of cubes c with 1 in column e_i. Let $g_1, \ldots, g_n$ be the terms corresponding to the rows of R.

(ii) Let $h_1 + \ldots + h_m$ be the complement of the term represented by row r.

(iii) Test whether $h_1 + \ldots + h_m + g_1 + \ldots + g_n$ is a tautology.

Step 5

For every row r in *CCOVER* eliminate row r
if the **e** vector of that row consists of entirely 2's

Step 6

(Generate the product terms for every function i)
For $i = 1$ to numbers of functions do
if $e_i = 1$ for row r then
$f^i = f^i \cup c$
number of functions = number of states + number of outputs

Explanation of steps: Step 1 generates the irredundant cover C^i for every function i. Steps 2 and 3 are used for generating a connection cover for all the functions. Steps 4 and 5 are used for removing redundancies in the connection cover. Step 6 forms the product terms for every function.

2.3.2. *Algorithm for generation of prime implicant Z.* Given below is the generalized Quine's procedure which is used for generating the prime implicants Z (Breuer 1972 and McCluskey 1956)

Step 1. $A_o = S[C_o \cup DC]$

Step 2. For $r = 0, 1, \ldots n$, we have

(i) Z^r is the set of all r-cubes $a \in A_r$ such that $a \cap C_o \neq \phi$ and no element in the set $a \varsigma A_r$ is an $(r+1)$ cube

(ii) $A_{r+1} = S[A_r \cup (A_r \varsigma A_r)] - Z^r$

(iii) If $A_r + 1 = \phi$, the process can be terminated since $Z^s = \phi$ for all $r + 1 \leqslant s \leqslant n$

Step 3

$$Z = \bigcup_{r=0}^{n} Z^r$$

2.3.3. *Tautology testing.* Testing for whether or not a boolean expression is a tautology is based on the idea that any expression $f(x_1, \ldots, x_n)$ can be written as $x_1 f_1(x_2, \ldots, x_n) + \overline{x_1} f_0(x_2, \ldots, x_n)$. Then f is a tautology iff both f_1 and f_0 are tautologous. There are many heuristics that can be used to tell in certain cases what the answer to a subproblem is. Two useful ones are (Ullman 1984):

(i) If a matrix has a row with all 2's, then surely it is a tautology, because this term covers all the points in the boolean cube.

(ii) Suppose a matrix has n columns, so its rows are terms representing points in the boolean n-cube. A term t with i_t 2's represents 2^{i_t} points, so we can easily obtain an upper bound on the number of points all the terms cover, by summing 2^{i_t} over all terms t. If this sum is less than 2^n, the matrix cannot represent a tautology. In this case, we not only answer 'no' for the particular matrix at hand; we also know that the original matrix is not a tautology. A recursive program has been developed for the above algorithm (Ullman 1984).

2.4. *Input/output format*

To describe a state machine, the following information must be supplied to the program in order.

(*a*) Number of rows of the transition/truth table

(*b*) Number of states in the machine

(*c*) Number of input variables

(*d*) Number of output variables

(*e*) Labels for the previous states, inputs, next states and outputs

(*f*) The transition/truth table of the machine

The program accepts both **completely specified** and **incompletely specified** circuits. A 2 is used to represent a 'do not care' in the table. The data can also be entered in a singular cover form and then it will be expanded by the program.

The program reads the input file with the above format for a state machine, and calculates the equations for the next state and output functions. The input data must be entered in the above order. The program accepts both combinational logic truth tables and sequential circuit state tables. If the table is of a combinational circuit, then the number of states must be entered as zero. The equations are printed for each function separately.

2.5. *Data structures used*

For storing the prime implicants and the product terms for every function, the following data structures are used. A one-dimensional array is used to store the number of product terms for every function. In addition, a three-dimensional array is used for storing the product terms or the cubes. These data structures are represented in Pascal as follows:

FC Array[1 ... L_2] of integer
cube Array[1 ... L_1] of char
F Array[1 ... L_2, 1 ... L_3] of cube
L_1 Upper limit on the number of (states + inputs)
L_2 Upper limit on the number of (states + outputs)
L_3 Upper limit on the number of product terms for a function

More details can be obtained from the two examples given in the next section.

3. Generation of a Weinberger array for a state machine

An algorithm for the generation of a WA for a state machine is given below.

A-	B-	X	A+	B+	Z
0	0	0	0	1	1
0	0	1	0	0	0
0	1	0	1	0	1
0	1	1	0	1	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	1	0

Figure 1. Input state table of Example 1.

A+		
A-	B-	X
0	1	0
1	0	2
1	2	1
B+		
A-	B-	X
2	0	0
2	1	1
Z		
A-	B-	X
2	2	0

(*a*)

	A+		
	A-	B-	X
M1	1	0	1
M2	0	1	2
M3	0	2	0
	B+		
	A-	B-	X
M4	2	1	1
M5	2	0	0
	Z		
	A-	B-	X
	2	2	0

(*b*)

Figure 2. (*a*) Irredundant covers for functions of example 1; (*b*) NOR representation of functions of Example 1.

Step 1

[Change the functions to NOR-representation]

Getting the equations of the next-state and output functions, they are first converted to NOR form.

As an example, consider the state table given in Fig. 1 which is the input to the synthesizer. Irredundant covers for the functions and their corresponding NOR representation for $A+$, $B+$ and Z are obtained and are given in Fig. 2. For example,

$$A+ = (\overline{A-})(B-)(\overline{X}) + (A-)(\overline{B-}) + (A-)(X)$$

$$= (((A-) + (\overline{B-}) + (X))' + ((\overline{A-}) + (B-))' + ((A-) + (\overline{X}))')''$$

	PU1	PU2	PU3	PU4	PU5
B-		+		+	
B-~	+				+
A-	+				
A-~		+	+		
X	+			+	
X~			+		+
M1	+*				
M2	+	*			
M3	+		*		
M4		+		*	
M5		+			*
A+~	*				
B+~		*			
	PU6	PU7			

Figure 3. Automatically generated WA for the state table of Example 1. (~ symbol means the complement of the variable.)

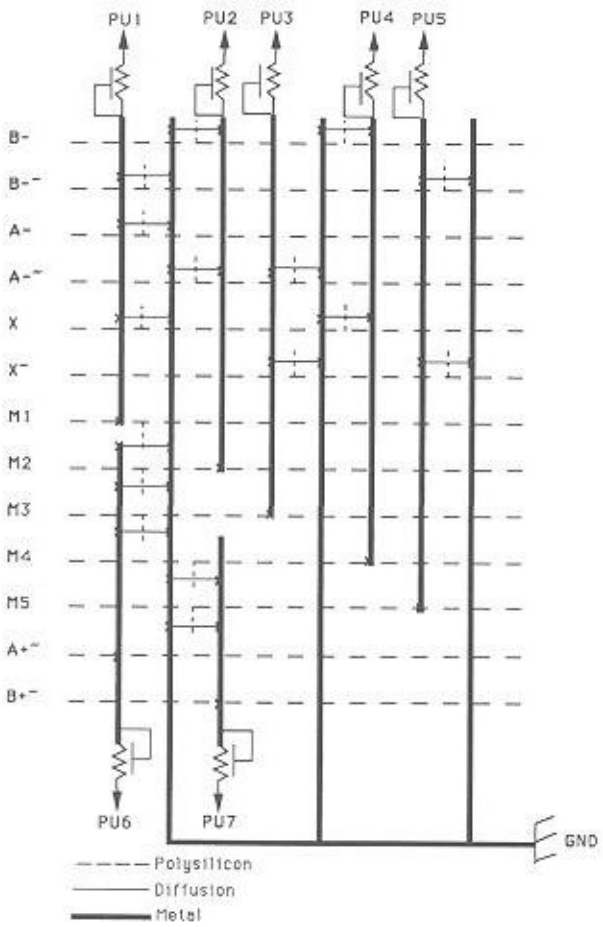

Figure 4. nMOS stick diagram for WA of Fig. 3.

y0-	y1-	c	t1	ts	y0+	y1+	st	h0	h1	f0	f1
0	0	0	2	2	0	0	0	0	0	1	0
0	0	0	2	0	2	0	0	0	0	1	0
0	0	1	1	2	0	1	1	0	0	1	0
0	1	2	2	0	0	1	0	0	1	1	0
0	1	2	2	1	1	1	1	0	1	1	0
1	1	1	0	0	2	1	0	1	0	0	0
1	1	0	2	2	1	0	1	1	0	0	0
1	1	2	1	2	1	0	1	1	0	0	0
1	0	2	2	0	1	0	0	1	0	0	1
1	0	2	2	1	0	0	1	1	0	0	1

Figure 5. Input state table of Example 2 (Mead and Conway 1980).

y0+				
y0-	y1-	c	t1	ts
1	2	2	2	0
2	1	2	2	1
y1+				
y0-	y1-	c	t1	ts
2	1	1	0	2
0	2	1	1	2
0	1	2	2	2
st				
y0-	y1-	c	t1	ts
0	0	1	1	2
0	1	2	2	1
1	1	0	2	2
1	1	2	1	2
1	0	2	2	1
h0				
y0-	y1-	c	t1	ts
1	2	2	2	2
h1				
y0-	y1-	c	t1	ts
0	1	2	2	2
f0				
y0-	y1-	c	t1	ts
0	2	2	2	2
f1				
y0-	y1-	c	t1	ts
1	0	2	2	2

Figure 6. Irredundant covers for functions of Example 2.

$\underline{A+}$			$\underline{A+}$ (NOR representation)		
$A-$	$B-$	X	$A-$	$B-$	X
0	1	0	1	0	1
1	0	2	0	1	2
1	2	1	0	2	0

Step 2

Every product term in a function needs a pull-up and a row in the WA. A

	y0+				
	y0-	y1-	c	t1	ts
M1	0	2	2	2	1
M2	2	0	2	2	0
	y1+				
	y0-	y1-	c	t1	ts
M3	2	0	0	1	2
M4	1	2	0	0	2
M5	1	0	2	2	2
	st				
	y0-	y1-	c	t1	ts
M6	1	1	0	0	2
M7	1	0	2	2	0
M8	0	0	1	2	2
M9	0	0	2	0	2
M10	0	1	2	2	0
	h0				
	y0-	y1-	c	t1	ts
	1	2	2	2	2
	h1				
	y0-	y1-	c	t1	ts
M5	1	0	2	2	2
	f0				
	y0-	y1-	c	t1	ts
	0	2	2	2	2
	f1				
	y0-	y1-	c	t1	ts
M11	0	1	2	2	2

Figure 7. NOR representation of functions of Example 2.

product term consisting of one variable need not have a pull-up and a specific row in the array since it can be taken from the (previous state or input variables) rows.

These rows (of WA) are labelled from M_1 to M_i (i can be as large as needed). Before labelling the rows (corresponding to the product terms) of the functions, they are ordered using a certain criteria. This criteria is that the common product terms are pushed down (i.e. labelled at the end).

This criteria allows overlapping of pull-ups from both directions of the WA (top and bottom) if possible, thus reducing area as in column folding.

Step 3

This step consists of forming the set of pull-ups to be placed in the top side of the array and the set of pull-ups to be placed in the bottom side of the array.

The set of top pull-ups (STPs) and bottom pull-ups (SBPs)

All product terms are labelled with M_i labels

For $i=1$ to number of functions do

If $|F_i|=1$ then there is no need for a pull-up since the function will be either an (input or state variable) or an M_i variable.

If the function has a product term consisting of one variable and $|F_i|>1$ Then

$STP \leftarrow STP \cup F_i$

Otherwise

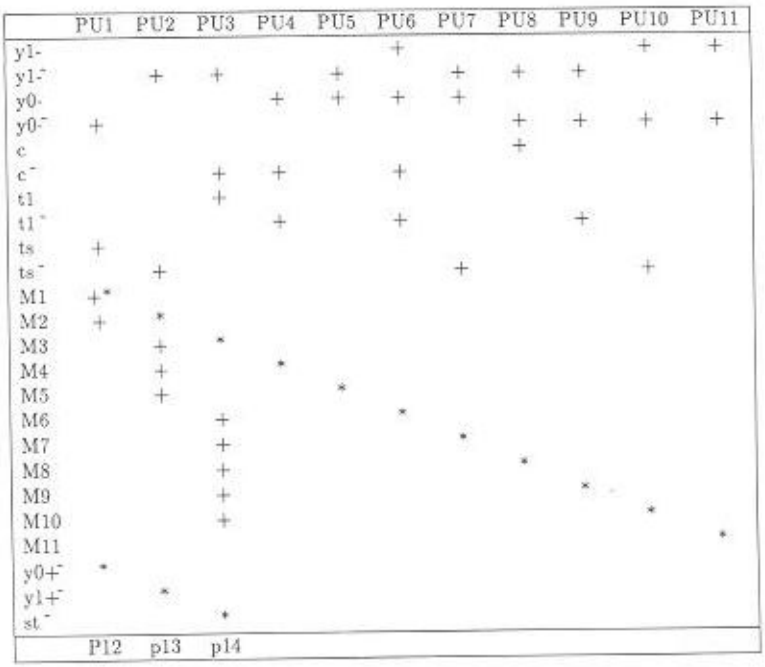

Figure 8. Automatically generated WA for state table of Example 2.

If for all $M_j \in F_i$, $j > i-1$ then

$SBP \leftarrow SBP \cup F_i$

Otherwise

$STP \leftarrow STP \cup F_i$

Step 4

The rows in the WA used for previous state variables are ordered to avoid the crossing of wires while connecting the next-state rows, through the memory, to the previous-state rows. This is done easily by looking at the order of representation of the next-state variables in the WA, and ordering the previous states rows in an opposite order.

By performing the above four steps, an irredundant WA is generated for the state machine.

Output format of WA

(*a*) Either of the symbols P_i or PU_i is used to represent a pull-up (i represents the number of the pull-up)

(*b*) The symbol '+' is used to represent the placement of a transistor

(*c*) The symbol '*' represents the end of the metal line running from the output of the top pull-up, and it also represents the placement of a contact cut

(*d*) Mapping of this array to stick diagram and hence to layout is straightforward

The automatically generated WA in the above-discussed format and its corresponding stick diagram in mixed notation for the state-table in Fig. 1 are given in Figs 3 and 4 respectively.

Another example of a state table is given in Fig. 5. This is the traffic light controller (Mead and Conway 1980). The generated irredundant covers and their corresponding NOR representation are given in Figs 6 and 7 respectively. The generated WA in '$-+*$' notation is given in Fig. 8.

4. Conclusions

A state-machine synthesis program that provides a useful aid in digital-circuit design and synthesis has been developed. Stick diagrams and layouts of the WA for the input state diagram are generated automatically.

From Fig. 4, it is obvious that the area can be further reduced by optimization. For example, interchanging rows for X and $\overline{A-}$ allows the folding of columns M3 and M4 without violating the constraints mentioned earlier. However, this was not done primarily to place the variable and its complement beside each other. An algorithm for row compaction (Sait and Al-Khulaiwi 1990) and column folding (Sait and Al-Rashed 1990) can be modified and applied to reduce the area further. As was explained earlier, the system can also take the truth table of combinational logic circuits and provide the VLSI layouts for WA automatically. However, in this case the optimization constraints will be relaxed.

The algorithms are coded in Pascal and the software runs on IBM PC. Copies of the software can be obtained from the authors.

Acknowledgment

The authors acknowledge support from the King Fahd University of Petroleum and Minerals and the University of Victoria, Canada.

References

Ayres, R. F., 1983, *VLSI: Silicon Compiler and the Art of Automatic Chip Design* (Englewood Cliffs, NJ: Prentice-Hall).

Breuer, M., 1972, *Design Automation of Digital Systems* (Englewood Cliffs, NJ: Prentice-Hall).

Clare, C. R., 1973, *Designing Logic Systems Using State Machines* (New York: McGraw-Hill Book Company).

McCluskey, E. J., 1956, Minimization of Boolean functions. *Bell System Technical Journal*, **35**, 1417–1444.

Mead, C. and Conway, L., 1980, *Introduction to VLSI Systems* (Addison Wesley).

Mukherjee, A., 1986, *Introduction to nMOS and CMOS VLSI Systems Design* (Englewood Cliffs, NJ: Prentice-Hall).

Sait, S. M. and Al-Khulaiwi, F. A., 1990, Automatic Weinberger array synthesis from a UAHPL description. *International Journal of Electronics*, **69**, 211–224.

Sait, S. M. and Al-Rashed, M. A.-A., 1990, An efficient algorithm for Weinberger array folding. *International Journal of Electronics*, **69**, 509–518.

Ullman, J. D., 1984, *Computational Aspects of VLSI* (Computer Science Press).

Weinberger, A., 1967, Large scale integration of MOS complex logic: a layout method. *I.E.E.E. Journal of Solid State Circuits*, **2**, 182–190.