IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, May 9-10, 1991

A STATE MACHINE SYNTHESIZER WITH WEINBERGER ARRAYS

Aiman H. El-Maleh
Department of Electrical & Computer Engineering,
University of Victoria, P.O.Box 3055, Victoria, B.C. Canada V8W 3P6
Sadiq M. Sait
Department of Computer Engineering, KFUPM #673, Dhahran-31261, Saudi Arabia

Abstract

In this paper we describe the development of a digital circuit synthesis program. 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, an nMOS VLSI layout for a Weinberger array (WA) is generated. D flip-flops are assumed for memory elements. Using this tool, tedious manual calculations can be avoided and layouts are generated automatically from state table descriptions.

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[3].

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 is specified the final design and implementation be obtained automatically.

In VLSI design of state machines, generally Programmable Logic Arrays (PLAs) are used for combinational logic [1, 5]. In this paper, we synthesize the circuit using Weinberger Arrays (WAs) [10]. The structure of a WA, the advantages and disadvantages etc, are discussed in [6, 7, 8, 9].

Since the combinational logic is for a state machine, there is a constraint on the order of rows. This is to avoid the feed back lines through 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 where the flip-flops are placed. These constraints make the optimization of the generated WA difficult.

In this paper, we describe the development of a state machine synthesizer program. The program calculates and reduces the equations for the next state and output variables of the state table. In addition, it generates a WA for the given state machine if needed. Column folding is attempted but the optimization of the array with the above mentioned constraints is still an open problem. In the following sections, we present the algorithms and the implementation details of the state machine synthesizer program. Furthermore, the generation of the WA for the modelled state machine is discussed.

State Machine Synthesizer

In this section, we briefly present the algorithms used in the development of the state machine synthesizer program. Discussed are the notations used, the main algorithm, and a generalized Quine procedure for the generation of prime implicants. tautology [2, 4, 9].

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 $x_1\overline{x_2} + x_1x_3$ 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.

- An n-tuple $c = (c_1c_2...c_n)$ where $c_i \in \{0, 1, x\}$ is said to be a cube. A 0-cube is an n-tuple $c = (c_1c_2...c_n)$ where $c_i \in \{0, 1\}$. Given that C is a set of cubes,$K^\circ(C)$, 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. 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.

- SUBSUMING ($\Rightarrow$) : Let $a = (a_1a_2...a_n)$ and $b = (b_1b_2...b_n)$ 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. $K^\circ(a) \subseteq K^\circ(b)$).

- CONSENSUS (ç) : The consensus of two product terms 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 $\overline{x}_i$ is a literal in Q. In this case, we can write $P = x_i.P\prime$ and $Q = \overline{x}_i.Q\prime$, where $P\prime$ and $Q\prime$ are not functions of x_i, and therefore the consensus of P and Q is $P\prime Q\prime$.

- SHARP PRODUCT ($\sharp$) : The sharp 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 $P(a)\overline{P(b)}$. Equivalently, $a\sharp b$ is the set of all prime implicants for the complex $K^\circ(a) - [K^\circ(a) \cap K^\circ(b)]$.

CH2954-6/91-0000-753 $1.00

Components Of The Program

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

1. EXPAND MODULE : This module is used to expand the truth/state table entered by the user, if needed, from singular cover form into 0's and 1's form.
2. QUINE MODULE : This module is used to calculate the equations for the next state and output functions.
3. FCCOVER MODULE : This module is used to reduce the equations to reduce the size of the resulting circuit by forming an irredundant cover.
4. FNOR MODULE : This module is used to represent the equations in NOR form and to label the product terms for the functions.
5. WEINBERGER MODULE : This module is used to generate a WA for the given machine.

After the truth/state table for the machine is 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 don't care) is found in any row, the row containing it will be replaced by two rows one having 0 in place the 2 and one having 1. Following this way, the table will be expanded to 0-cubes in order to find the prime implicants for every function properly. More details of the algorithms are given in the next section.

Algorithms

The main difficulty with 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[2, 9].

Main Algorithm

number of functions = number of states + number of outputs

Step1 :

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 don't care vertices of f^i).

Step2 :

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

Step3 :

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

Step4 :

For each row r of CCOVER do
Raise column e_i of vector e by placing 2 instead of 1 in column e_i if the test below is positive :

1. Find the set R of rows other than r consisting of cubes c with 1 in column e_i. Let $g_1, ..., g_n$ be the terms corresponding to the rows of R.
2. let $h_1 + \cdots + h_m$ be the complement of the term represented by row r.
3. Test whether $h_1 + \cdots + h_m + g_1 + \cdots + g_n$ is a tautology.

Step5 :

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

Step6 :

[Generate the product terms for every function f_i]
For $i = 1$ to number of functions do
if $e_i = 1$ for row r then
$f^i = f^i \cup c$.

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

Algorithm For Generation Of Prime Implicants Z

Given below is the generalized Quine's procedure which is used for generating the prime implicants Z [2, 4].

Step1 $A_o = S[C_o \cup DC]$

Step2 For $r = 0, n$ we have :

a) 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.

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

c) If $A_{r+1} = \phi$, the process can be terminated since $Z^s = \phi$ for all $r + 1 \leq s \leq n$.

Step3 $Z = \cup_{r=0}^{n} Z^r$.

Weinberger Array Generation

In this section we present an algorithm for the generation of a WA for a state machine

Step1 :

[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 are given in Fig. 2.

Step2 :

Every product term in a function needs a pull-up and a row in the WA. A 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 labeled from M_1 to M_i (i can be as large as needed). (i can be as large as needed). Before labeling 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. labeled at the end). This criteria allows overlapping of pull-ups from both directions of the WA (top & bottom) if possible, thus reducing area as in column folding.

Step3 :

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.

Top Pull-ups (STP) & Bottom Pull-ups (SBP):

For $i = 1$ to # of functions do
If $|f_i| = 1$ then there is no need for a pull-up since the function will be either an input , state or 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
If for all $M_j \in f_i$, $j > i - 1$ then
$SBP \leftarrow SBP \cup f_i$
Otherwise
$STP \leftarrow STP \cup f_i$

Step4 :

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 :

- Either of the symbols P_i or PU_i is used to represent a pull-up (i represents the number of the pull-up).
- The symbol '+' is used to represent the placement of a transistor.
- 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.
- Mapping of this array to stick diagram and hence to layout is straight forward.

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 Fig.3 and Fig.4 respectively.

Conclusion

A state-machine synthesis program that provides a useful aid in digital-circuit design and synthesis is developed. Stick diagrams and layouts of the WA for the input state diagram are generated automatically.
Referring to Fig.4, it is obvious that the area can be further reduced by optimization. As an example, interchanging rows for X and $\overline{A-}$ allows the folding of columns M3 and M4 without violating the earlier mentioned constraints. However, this was not done primarily in order to make the variable and its complement beside each other. Algorithm for row compaction [7] and column folding [8] can be modified and applied to reduce the area further. The system, as was explained earlier can also take only the truth table of combinational logic circuits and provide the VLSI layouts for WA automatically. However, in this case the earlier mentioned 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 by writing to the authors.

References

[1] Ayres, Ronald F. VLSI : Silicon Compiler and the Art of Automatic Chip Design, Prentice Hall, 1983.

[2] Breuer, Melvin, Design Automation of Digital Systems, Englewood Cliffs, New Jersey, 1972, ch. 2.

[3] Christopher R. Clare, Designing Logic Systems Using State Machines, McGraw Hill Book Company, New York, 1973.

[4] E. J. McCluskey, *'Minimization of Boolean Functions.'* Bell System Technical Journal, vol. 35 (1956), pp. 1417-1444.

[5] C. Mead and L. Conway, Introduction to VLSI Systems, Addison Wesley 1980.

[6] Mukherjee Amar, Introduction to nMOS and CMOS VLSI Systems Design, Prentice Hall International, Englewood Cliffs, New Jersey, 1986.

[7] Sadiq M. Sait and Fayez A. Al-Khulaiwi, *'Automatic Weinberger Array Synthesis from a UAHPL Description'* International journal of Electronics, vol. 69, no. 2 , pp. 211-224, 1990.

[8] Sadiq M. Sait and Muhammad Abdul-Aziz Al-Rashed, 'An Efficient Algorithm for Weinberger Array Folding,' International Journal of Electronics, vol. 69 , no. 4 , pp. 509-518, 1990.

[9] Ullman, J. D., Computational Aspects of VLSI, Computer Science Press.

[10] Weinberger A.,*'Large Scale Integration of MOS Complex Logic : A Layout Method,'* IEEE J. of Solid State Circuits SC-2:4,pp.182-190.

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

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

(Note : This symbol ~ means the complement of the variable)

Figure 3: Automatically generated WA for the state table of Example 1

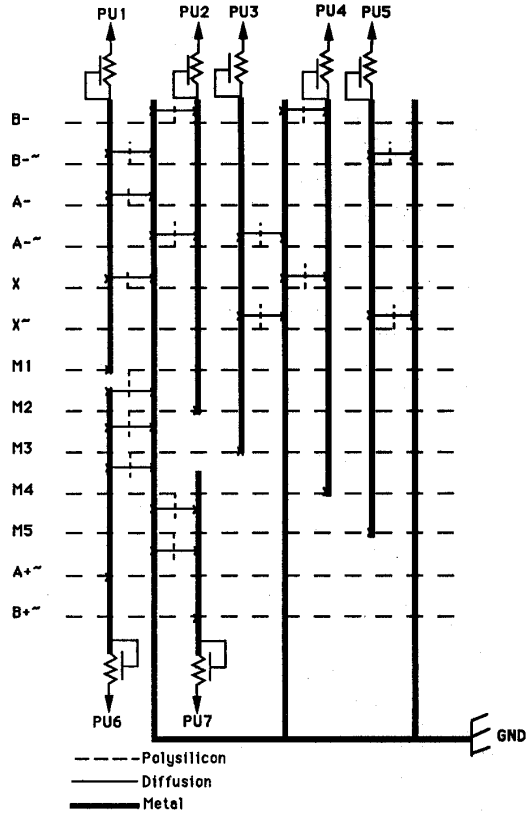

Figure 4: nMOS Stick Diagram for WA of Fig.3