Proceedings of ISCAS 85

A Finite-State Machine Synthesis System

R. Rudell
A. Sangiovanni-Vincentelli
G. De Micheli [1]
Department of Electrical Engineering
and Computer Sciences
University of California, Berkeley
Berkeley, California USA 94720

[1] IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

Finite-state machines are essential components of digital systems. A PLA-based finite-state machine synthesis system which optimizes each step of the synthesis process is described. A design example is used to illustrate the features of the system.

1. Introduction

Finite-state machines (FSMs) are used widely to implement control logic in applications such as microprocessors and specialized processors for speech synthesis, digital transmission, digital filters, and digital signal processing. The most straightforward implementation of a FSM consists of a ROM and a counter (the microcode approach). This can often be expensive in terms of the area and the speed of the machine. A better approach for many applications is a synthesis system which incorporates specific optimization steps to tailor the design for a given FSM.

In this paper, we address several of the optimization steps which occur in designing a FSM for a custom integrated circuit. We are especially interested in techniques which are applicable to both small and large machines, and hence heuristics play an important role. We describe a FSM synthesis system which we are working on at Berkeley. In particular, we address the issues which we see as important in the design of FSMs and the solutions (when we have them) for each of these problems. The design techniques discussed here are currently being used at Berkeley for the control logic in several integrated circuit projects. Several companies and Universities are also using the optimization programs which are part of our system.

2. Finite-State Machine Synthesis

2.1. Overview

Formally, we define a FSM as a 5-tuple consisting of: a set of primary inputs, a set of primary outputs, a set of states, a next-state function, and an output function. The next-state function maps the present-state and the primary inputs into a next-state, and the output function maps the primary inputs and the present-state onto the primary outputs. Either of these functions may be incompletely specified (i.e., the value of the function need not be well-defined for all possible inputs). We can represent any deterministic sequential function with this model. The FSM can be separated into two components: a combinational circuit and a memory. The combinational component implements the next-state equations and the output equations of the machine, and the memory stores the state vector between clock phases.

This formal definition is how the optimization tools view the FSM. The first, and in many ways the most difficult, part of FSM design is to capture the intentions of the designer and to transform them into the FSM model. The designer can represent the FSM in a formal design language which describes both the control and the data portions of the machine (such as Zeus [1]), or with a simpler language which describes only the control nature of the machine (such as SLIM [2]). Quite often, the designer finds state-diagrams or state-tables to be the best for describing a finite-state machine.

An important feature which has been often overlooked in the design of languages for finite-state machines is capturing dependencies which exist in the FSM. The designer must consider whether the primary inputs and the primary outputs to the FSM are independent or whether there are relationships among these signals. This information must be preserved and presented to the optimization tools. These dependencies often arise from the choices a designer has already made in partitioning his design into manageable pieces — the FSM does not exist in isolation, but is constrained by its surrounding environment. These relationships may spread over several states because surrounding logic often contains state information (e.g., a register in the datapath which feeds the FSM). Capturing this information results in a *don't-care set* which can greatly influence the optimization of the FSM.

Ideally, if the design proceeds automatically in a top-down fashion, the tools which perform the partitioning of the logic into FSMs can also automatically capture the types of dependencies described above which exist between the chosen partitions. Each of the smaller FSMs can then be optimally synthesized as leaf cells. Such a design system is an active research area at the current time.

2.2. Design Steps

Once the FSM has been captured in the formal model described above, we can perform many optimization steps. The optimal synthesis of finite-state machines has been an important part of digital design for many years. Classically, the first step is *state minimization* which reduces the number of states by recognizing those states in the machine which are equivalent. When this is possible, it is hoped that the resulting machine will be easier to build.

The complexity of the logic equations needed for the combinational component depends on the assignment of binary codes to each of the states of the machine. This encoding problem is referred to as the *state-assignment* problem.

The Boolean equations can be realized with a two-level structure (such as a PLA), or through random-logic by an interconnection of logic primitives. In either case, logical optimizations (such as Boolean minimization or logical partitioning and decomposition) are essential for an efficient realization.

Generating the actual integrated circuit layout of the FSM involves topological optimizations such as PLA folding and partitioning in the case of a PLA, or placement and routing in the case of a random logic approach.

Testing is becoming more and more important in VLSI design. A module generator should provide either testability information for the module, test patterns for the module, or additional circuitry to perform a self-test of the module. Test pattern generation can often be made efficient by exploiting the particular structure of the module being generated.

3. Berkeley Finite-State Machine Synthesis

3.1. FSM Design Style

To put together an effective design system for FSMs, we first target a particular design style. We are primarily concerned with the design of FSMs for the control part of custom

integrated circuits. We choose to use the Mealy model for the FSM, where we allow the output-equations to depend on both the state variables and the primary inputs. This requires a fully synchronous design which fits naturally into a precharged (or dynamic) CMOS PLA. We use a multiply-folded PLA for the combinational component, and D-type (or delay) latches for the memory. One could imagine using more general logic decomposition and layout techniques to implement the logic equations for the machine (such as cascaded PLAs or Domino CMOS), and these areas are being explored [3,4].

In our synthesis procedure we emphasize both logical and topological optimizations to reduce both the area and delay in the FSM. Because of this fixed design style, most of the optimizations which we perform are technology independent.

3.2. FSM Design Procedure

We first capture the intent of the FSM using the language MEG [5]. We next perform state-assignment using the program KISS [6] (which uses ESPRESSO as a subroutine), and the resulting two-level logic equations are again minimized with the program ESPRESSO-IIC [7]. Topological optimization is done using the program PLEASURE [8] which performs multiple-folding of PLAs. The PLA masks are generated with the program PANDA [9] which performs layout of multiply-folded PLAs suitable for the MAGIC graphics editor [10]. The program PLATPG [11] generates test patterns for the combinational component of the FSM.

3.2.1. Input language

MEG is a program which translates a simple language for describing FSMs into the standard format used by the remaining optimization tools. MEG has the flexibility to describe both outputs which are undefined for a particular transition in the machine, and transitions which are known not to occur. Unfortunately, it is still the work of the designer to manually specify this information. A better approach would be to have a design system where these types of dependencies can be extracted automatically from a complete description of the entire design. The best language to use is still an active research topic.

3.2.2. State Minimization

Even though we have a tool for state minimization [16], we do not include it in the design system for two reasons: (1) it is quite often the case that state minimization is unable to reduce the number of states in the machine, and (2) even when the number of states can be reduced, there is no assurance that the resulting FSM will be easier to implement. What is needed is a more general type of transformation on the FSM which, using knowledge of the surrounding environment, is able to transform the machine into an equivalent machine (with respect to the environment) that is easier to build. We consider this an open question which needs to be addressed for optimal FSM synthesis.

3.2.3. State Assignment

KISS solves the state-assignment problem by approximating the state-assignment as an *input-encoding problem*. The input-encoding problem is: given a Boolean function defined over binary variables and a set of symbols, code the symbols as binary vectors so as to minimize the logic required to implement the Boolean function. In this case, the primary outputs are considered to be a Boolean function of the present-state and the primary inputs, and the present-states are to be coded with binary vectors; the next-state function is not considered. The input-encoding problem is solved [6] by performing a multiple-valued minimization to produce a set of constraints. These constraints specify the relationships that should be satisfied among the codes chosen for the states. For example, a constraint may specify that two states are to be assigned adjacent codes (e.g., the codes 010 and 011 are adjacent), or that a group of states are to fit on a subspace of the Boolean space (e.g., the codes 000, 001, 100, and 101 belong to the subspace -0- of the Boolean space). The advantage of this technique over other heuristic constraint-based techniques is that we use the full power of our minimization tool to generate an optimal set of constraints. KISS will always determine a coding for the states which satisfies all of the constraints.

Experiments have shown that the approximation of solving the state-assignment problem as an input-encoding problem is good for many types of FSMs (primarily dense FSMs where there are a large number of arcs in proportion to the number of states). We are currently exploring how to extend our techniques to solve the complementary output-encoding problem which would improve the solutions for all types of FSMs (and in particular, sparse FSMs).

Satisfying all of the constraints during the encoding will produce a PLA with the fewest rows possible (under the approximation mentioned above). Sometimes it is necessary to increase the number of state-bits used for the state vector in order to satisfy all of the constraints (which increases the number of columns in the PLA). The primary goal, of course, is minimum area in the PLA, and not the fewest number of rows. Another open problem we are investigating is how to have the state-assignment tool automatically trade-off the number of columns in the PLA (i.e., the number of state-bits) against the number of rows (which may increase as constraints are broken). We currently rely on the designer to assist the tool at this point in order to choose a state encoding which best optimizes the area of the PLA.

3.2.4. Boolean Minimization

Once the state assignment is fixed, we use ESPRESSO-IIC to perform a standard two-level Boolean minimization of the resulting logic equations. ESPRESSO is a powerful logic minimization program which has been shown to actually achieve the minimum solution for many problems.

We are currently exploring two other logical optimizations for the PLA. It is reasonable to assume that, without any cost, we can realize either a given function or its complement in the core of a PLA, and then correct the phase in the output buffer. Hence, we can choose, for each output individually, whether to implement the function or its complement. Solutions for this problem, which has been called the *output phase assignment* problem [12], have showed some promise, but more research is needed. We note that the output phase assignment problem is a special case of the output-encoding problem mentioned earlier, and we hope to apply results from the output encoding problem to the output phase assignment problem.

In a similar manner, it is easy to justify the cost of using *two-bit decoders* [13] in the input of the PLA in place of the normal input buffers. The dominant delay in the PLA is in the core of the PLA (with high capacitance signal lines), and the decoders add only a small area to the input buffer. Using the two-bit decoders can also only lead to a smaller and faster realization for the core of the PLA. The open problem here is how to best pair the input variables into groups of two.

3.2.5. PLA Folding

The program PLEASURE performs multiple-folding of PLAs to minimize the area required for the PLA. PLEASURE allows constraints to be placed on the signal lines in the PLA, so the designer has control over where the signals come out of the PLA. Especially in the design of finite-state machines, it is necessary to use constrained folding — it would be unwise to attempt to fold the PLA such that the state-variable feedback lines were on opposite sides of the PLA. Another novel feature of PLEASURE is allowing the individual columns in the AND-plane of the PLA to fold separately with the only constraint being that the two signal lines lie next to each other. This is referred to as *group-folding*.

3.2.6. PLA Layout

We use the program PANDA for the final assembly of the PLA. PANDA is driven from the output of PLEASURE to tile the layout for multiply-folded PLAs. PANDA minimizes the length of the polysilicon and metal signal lines, and allows the designer to choose a buffer sized for optimal delay through the PLA. We have a set of templates for 3 micron static CMOS PLAs (using p-channel current sources), and we are currently developing templates for dynamic PLA styles.

3.2.7. Testing

The program PLATPG generates tests for all crosspoint faults in a PLA structure. (A crosspoint fault corresponds to either an extra device or a missing device for each cell in the PLA matrix.) This can be used to assist testing the FSM after fabrication. PLATPG uses the algorithms presented in ESPRESSO-IIC in order to speed up the test generation process.

4. Design Example

We present the system in more detail by going over the design of a simple machine. We study the optimal design of one of the two FSMs built for the OPUS microprocessor [14]. OPUS was a simple 16-bit microprocessor (similar to a PDP-8) which was designed by four students in eight weeks as a class project in the VLSI System Design Class at the University of California, Berkeley, in the Fall of 1983. The microprocessor was fabricated and actually ran programs as a co-processor in a SUN workstation in the spring of 1984. The FSM design system described here was not used by the designers when OPUS was designed. Instead, the design was mapped straight from the transitions provided by the designers into an nMOS PLA which implemented the machine.

The control logic for OPUS consisted of two communicating FSMs. The first was referred to as the Master FSM and it controlled the datapath of the microprocessor. The second was called the Input/Output FSM (I/O-FSM), and it controlled the interface to the Multibus Design Frame [15]. We show how the design of the I/O-FSM would proceed with our design system, and then show a comparison between the optimized FSM and the original FSM.

Figure 1 shows the specification of the I/O-FSM in the language MEG. It is important to note how, in this language, we capture the logic which surrounds the I/O-FSM. For example, two of the primary outputs of the machine, *MINIT* and *WAIT*, are not independent. Whenever *MINIT* is asserted high, the output *WAIT* is irrelevant. (While this is, of course, quite unobvious to the reader, it is apparent to the designer.) Hence, whenever the I/O-FSM asserts *MINIT*, the value of *WAIT* is given as ? which specifies the value as a *don't-care*.

Another important feature is shown in the state *op1*. The only way to reach *op1* is for the two-bit primary input *OP* to have the value *READ* or *WRITE*. Further, because we understand the surrounding logic, we know that the value of *OP* will not change until we return to *op0*. Hence, the input conditions of *OP* equal to *HALT* or *NOP* cannot occur in the state *op1*, and we are careful to specify this in the FSM description. This is done by using the pseudo-state *DCSET*. The extra transition given after the **endcase** statement forces all unspecified transitions from this state to be *don't-cares*, thus leading to an incompletely specified machine. We capture this information because the optimization tools which follow can make use of it.

Figure 2 shows part of the *symbolic implicant table* produced by MEG. Each row in the table is in one-to-one correspondence with the arcs in a state-diagram of the FSM. The primary inputs and primary outputs are represented by vectors of **0**, **1**, and **-** in a manner similar to those commonly used for PLA representations. The present state and next state fields are symbolic. It is possible to enter the design system at this point with a symbolic implicant table generated by other tools. For example, the symbolic implicant table could be easily extracted from a graphical system which allows the designer to draw a state-diagram, or from other high-level languages.

Figure 3a shows the constraints (and their weights) determined by the multiple-valued minimization, and the state-assignment chosen by KISS which satisfies these constraints. KISS required an extra bit to satisfy both constraints which leads to a PLA with 9 rows and 25 columns (8 inputs and 9 outputs). We decide to try breaking the second constraint in order to find an assignment which uses only two state bits. We run KISS again, and Figure 3b shows an encoding which satisfies just the first constraint. The resulting PLA with this assignment is minimized by the program ESPRESSO, and the output is shown in Figure 4. There are 10 rows and 22 columns (7 inputs and 8 outputs) in the minimized PLA.

Now that we are finished with the logical optimization, we seek to optimize the layout of the PLA. Figure 5 shows the results from folding the PLA with PLEASURE. The PLA was reduced from 22 columns to 16 columns by simple folding with the feedback signals constrained to lie on the same side of the PLA. The area reduction is about 18% even when we consider the overhead of providing buffers at both the top and the bottom of the PLA. With the PLA so small, the overhead incurred for multiple-folding of the PLA did not pay off.

We now have a PLA with 10 rows and 15 columns. Assuming a PLA cell size of 12 micron square, and a PLA buffer length of 60 micron at the top and bottom of the layout (which are typical figures for a 3 micron CMOS process), the resulting PLA has an area of 43,200 square microns. The original PLA used by OPUS had 24 rows and 28 columns for an estimated area of 137,000 square microns. The machine used in OPUS was specified as a Moore machine without any don't-care transitions. For a direct comparison with the original design, we note that the I/O-FSM, when cast as a Moore machine without using the don't-care information or PLA folding, required 16 rows and 28 columns after optimization for an estimated area of 105,000 square microns.

5. Results on Other Examples

Several other FSMs have also been designed using this system. Table 1 documents these examples and reports on the estimated area savings over straightforward implementations (i.e., without considering an optimal state-assignment, minimization, or folding).

6. Conclusions

We have presented a FSM design system that is being used at the University of California, Berkeley, for the optimal synthesis of finite-state machines. We have also identified several open questions to be solved. In particular: (1) the design of a description language which will automatically capture the environment of the FSM for the optimization tools; (2) state-minimization techniques which take into consideration the logic surrounding the FSM; and, (3) the output-encoding problem which will improve the effectiveness of the state-assignment for sparse FSMs and potentially have applications for the output-phase assignment problem.

7. Acknowledgements

We would like to acknowledge many interesting discussions on synthesis problems with Robert Brayton, Gary Hachtel, Curt McMullen, and Richard Newton. Richard Newton developed the first system for PLA design at Berkeley. We would like to thank all of the students who have worked on various pieces of this FSM system. We thank Duksoon Kay for improvements to PLEASURE, Ruey-sing Wei for his work on PLATPG, Grace Mah for her work on PANDA, and David Wood for his improvements to PEG (which is now known as MEG). We thank Robert Brayton of IBM Yorktown for providing the examples DK14 and DK15, and David Geiger of Carnegie-Mellon University for providing the examples MARK1 and SCF. DARPA under contract N00034-K-0251 and the MICRO program of the State of California supported most of this research. SRC supported the improvements to PLEASURE. Giovanni De Micheli was supported by an IBM Research Fellowship while he was at Berkeley, and Richard Rudell is currently supported by an IBM Research Fellowship.

8. References

[1] Karl J. Lieberherr, "Chip Design in Zeus and a Proposal for a Standard Benchmark Set for Hardware Description Languages," *Proc. 1984 Int. Conf. on Comp. Des.*, Rye, NY, Oct. 1984.

[2] J. L. Hennessy, "SLIM: a simulation and implementation language for VLSI microcode," *LAMBDA*, April 1981, pp. 20-28

[3] R. K. Brayton and C. McMullen, "Synthesis and Optimization of Multistate Logic," *Proc. 1984 Int. Conf. on Comp. Des.*, pp 23-30, Rye, NY, Oct. 1984

[4] M. Hofmann and A. R. Newton, "A Domino CMOS Logic Synthesis System," *Proc. 1985 Int. Symp. on Circ. and Syst.*, Kyoto, Japan, June 1985.

[5] D. Wood, "MEG User's Manual", Department of EECS, University of California, Berkeley.

[6] G. De Micheli, R. Brayton and A. Sangiovanni-Vincentelli, "KISS: A Program for the Optimal State Assignment of Finite-State Machines", *Proc. 1984 Int. Conf. on CAD*, pp. 209-212, Santa Clara, CA., Nov. 1984.

[7] R. Brayton, G. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* Kluwer Academic Publishers, 1984

[8] G. De Micheli and A. Sangiovanni-Vincentelli, "Multiple Constrained Folding of Programmable Logic Arrays: Theory and Applications," *IEEE Trans. on CAD of IC and Syst.*, Vol. CAD-2, N. 3, pp. 151-167, July 1983.

[9] G. H. Mah and A. R. Newton, "PANDA: A PLA Generator for Multiply-Folded PLAs," *Proc. IEEE Int. Conf. on CAD*, pp. 122-124, Santa Clara, CA., Nov. 1984.

[10] B. Mayo and J. Ousterhout, "Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool," *Proc. 20th Design Automation Conf.*, pp. 270-276, Miami Beach, FL., June 1983.

[11] R.S. Wei and A. Sangiovanni-Vincentelli, "PLATPG: A PLA Test Pattern Generation Program", *Proc. 1985 Design Automation Conf.*, Las Vegas, June 1985, to appear.

[12] T. Sasao, "Input Variable Assignment and Output Phase Optimization of PLAs", Draft, 1983.

[13] H. Fleisher and L. I. Maissel, "An Introduction to array logic," *IBM J. Res. and Dev.*, Vol 19., pp. 98-108, March 1975.

[14] R. Rudell, D. Wood, J. Pierret, and T. Mills, "OPUS: An nMOS 16-bit Microprocessor for the Multibus Design Frame", CS250 Project Report, December, 1983.

[15] G. Borriello, R. Katz, A. G. Bell, L. Conway, "VLSI System Design by the numbers", *IEEE Spectrum*, February 1985, pp. 44-50.

[16] Villa, T., "A Program for State Minimization of FSMs," EECS 244 Project Report, May 1984.

```
-- OPUS Multibus controller, Version 2.0, 2/15/85
INPUTS:          INIT OP1 OP2 SWR MACK;
OUTPUTS: WAIT MINIT MRD SACK MWR DLI;
-- Define the encoding for the processor operations
#define OP OP1 OP2
#define NOP     0 0
#define HALT    0 1
#define READ    1 0
#define WRITE   1 1
#define XOP     ? ?

-- describe the reset logic
reset on INIT to swr0(MINIT WAIT=?);

-- wait for slave write from multibus to activate us
swr0:if SWR then swr1(MINIT WAIT=? DLI SACK)
     else swr0(MINIT WAIT=?);

-- wait for slave write to go away
swr1:if SWR then swr1(MINIT WAIT=? SACK)
     else op0;

-- wait for request from processor
op0:  case (SWR MACK OP)
           0 ? NOP => op0;
           0 ? HALT => swr0(MINIT WAIT=?);
           0 0 READ => op1(WAIT MRD);
           0 0 WRITE => op1(WAIT MWR);
           0 1 READ => op0(WAIT);
           0 1 WRITE => op0(WAIT);
           1 ? XOP => swr1(MINIT WAIT=? DLI SACK);
     endcase => DCSET;

-- after starting transaction, wait for acknowledge
op1:  case (SWR MACK OP)
           0 0 READ => op1(MRD);
           0 0 WRITE => op1(MWR);
           0 1 READ => op0(DLI);
           0 1 WRITE => op0;
           1 ? READ => swr1(MINIT WAIT=? DLI SACK);
           1 ? WRITE => swr1(MINIT WAIT=? DLI SACK.
     endcase => DCSET;
```

Figure 1. MEG language description of the OPUS I/O-FSM

```
.inlabel INIT OP1 OP2 SWR MACK
.outlabel WAIT MINIT MRD SACK MWR DLI

#primary   present    next     primary
 #inputs     state    state    outputs
1----      ANY       swr0     -10000
0--0-      swr0      swr0     -10000
0--1-      swr0      swr1     -10101
0--0-      swr1      op0      000000
0--1-      swr1      swr1     -10100
00000      op0       op0      000000
```

Figure 2. Symbolic implicant table produced by MEG

```
.i 7
.o 8
.p 10
01000-0 10001000
01100-0 10000010
0-0-110 00000001
-01--0- 01010000
0--1--0 00000001
0--1-0- 00000001
-1---0- 00100000
-----01 01010000
1------ 01010000
0--1--- 11010100
.e
```

Figure 4. PLA Matrix after ESPRESSO

```
.top INITbar INIT OP2
   OP2bar ps1 ps1bar ns0
   ns1 MINIT MRD
.bottom OP1 OP1bar SWR
   SWRbar MACKbar MACK
   ps0bar ps0 WAIT MWR DLI
   SACK
.and    11    4
.row      10
 XXXX****X**   ++++
X-0----1-0--   ~ II~X
X--------0!-   ~ II~X
X-----!-----   ~ II~X
X1--000-0--0   I~~ iX
X--1-0------   IiiIX
X1--000!---0   iI~~X
X----0-10-10   ~~I~X
X--1-0-----0   ~~I~X
X--1-0---0--   ~~I~X
X1-------0--   I~~~X
  ****X**X**X  ++++
```

Figure 5. Folded PLA Matrix after PLEASURE

```
/* constraints are */
/* weight=    3 0011 */
/* weight=    1 1011 */
#define swr0 110
#define swr1 111
#define op0 000
#define op1 100
#define ANY ---
/* 9 product terms */
```

Figure 3a. KISS State-Assignment

```
/* constraints are */
/* weight=    3 0011 */
#define swr0 01
#define swr1 11
#define op0 00
#define op1 10
#define ANY --
/* 10 product terms */
```

Figure 3b. KISS State-Assignment after Designer interaction

				Original			After State assignment			After PLA Folding		
Name	in	out	states	rows	cols	area	rows	cols	area	rows	cols	area
CSE	7	7	16	91	33	3003	47	36	1692	47	25	1175
DK14	3	5	7	56	20	1120	23	23	529	23	18	414
DK15	3	5	4	32	17	544	16	23	368	16	17	272
MARK1	5	16	15	22	38	836	19	38	722	19	24	456
SCF	27	56	121	166	134	22244	140	134	18760	140	70	9800

Table 1. PLA area for the combinational component after each step of the synthesis
(Area is measured by the number of PLA cells)