Synthesis of Multiple-Input Change Asynchronous Finite State Machines

Maureen Ladd* William P. Birmingham
EECS Department
The University of Michigan
Ann Arbor, MI 48109
wpb@crim.eecs.umich.edu

Abstract

Asynchronous finite state machines (AFSMs) have been limited because multiple-input changes have been disallowed. In this paper, we present an architecture and synthesis system to overcome this limitation. The AFSM marks potentially hazardous state transitions, and prevents output during them. A synthesis tool to create the AFSM incorporates novel algorithms to detect the hazardous states.

1 Introduction

Operations in asynchronous, or self-timed [18], circuits are not controlled with an external clock. Computations begin when the inputs to the network arrive, instead of when a clock pulse asserts. Without a clock, however, the gate and line delays inherent in any design introduce hazards. To overcome these hazards, restrictions have been placed on asynchronous circuits. Developing an asynchronous finite state machine (AFSM) without restrictions will allow its full potential to be realized in a variety of designs. This paper describes an architecture that is both hazard-free and without input restrictions. A synthesis tool to automate the design of the architecture is detailed.

The paper is divided into the following sections. Section 2 highlights the new AFSM and its advantages over other machines. The model of the new hazard-free architecture is described in Section 3. An architectural description of the machine is found in Section 4. Section 5 details the synthesis tool, and the results of some FSM benchmarks are presented in Section 6. Section 7 compares our technique for providing a multiple-input change hazard-free AFSM to methods used by others. Finally, Section 8 concludes the paper.

2 A Hazard-free AFSM

Because AFSMs have no controlling clock, they must have some way to detect new inputs. The term "fundamental mode" [20] denotes a method of AFSM operation such that new inputs are accepted only when current inputs are assimilated. This requirement exists regardless of the model used for an AFSM. In addition, all FSMs must be hazard-free. A hazard is a possible deviation from expected operation caused by stray gate or line delays. A variety of hazard-free implementations exist [5, 10, 14], but they remove only one or two kinds of hazards. Our AFSM architecture, FANTOM, is free from all possible types of hazards.

2.1 Input Change Hazards

Assimilation of a new input vector can cause hazards. Different terminology is used to describe these hazards depending upon whether single-bit or multiple-bit input changes are involved. A gate output glitch due to a single-bit input change, is called a static, or combinational hazard. A dynamic hazard [20] causes a gate output to glitch if both x_i and $\overline{x}_i$ are input. The well-known technique of including all prime implicants in the logic equation (adding "consensus gates") resolves these hazards [20].

When the input transition involves a multiple-bit change, the term M-hazard is used [5]. An M-hazard can be either logic or function. The logic M-hazard is identical to the static hazard and is resolved the same way. A function M-hazard occurs if a state variable that should remain invariant changes during the input vector transition. This type of hazard is inherent in the flow-table representation, and cannot be eliminated using circuit additions. This seemingly unavoidable hazard is the reason why many architectures restrict the input vector to single-input changes. FANTOM uses a new technique, described in Section 5, to eliminate M-hazards, thereby removing input restrictions.

Other architectures allow multiple input-bit changes,

*Work done while on leave from Digital Equipment Corporation.

but only address a subset of the hazards discussed here. The methods used by these architectures to detect multiple versus single-bit changes involve complex input codings, source boxes, or time calculations [2, 6, 21]. FANTOM simply traps inputs with "self-synchronization", which uses internal signals to control events in a network [4]. These internal signals detect when the previous state change is stable before gathering new inputs.

2.2 Avoidance of Other Hazards

A steady-state hazard occurs when a sequential circuit enters the wrong internal state because of a static (logic) hazard or a critical race. A critical race condition exists if two or more state variables change due to an input transition, and the next stable state will depend upon the order in which the state variables change. To eliminate this hazard, state assignments that restrict the state vector to single bit or non-hazardous multi-bit changes are used [19].

Transient hazards, a special case of static hazards, affect the outputs. FANTOM avoids these using self-synchronization at the outputs. Thus, FANTOM allows multiple-output bit changes, as long as the output vector obeys the single-output-change (SOC) principle [20], i.e. bits can change only once per input transition.

Essential hazards are inherent to sequential circuits; they exist because of the possible race between a gate seeing an input change and a state variable change [5]. Essential hazards are avoided if two conditions are met. First, the inputs must reach all gates before the state variables can change. Second, the combinational logic must be hazard-free. The first condition can be restated as: the maximum line delay must be less than the minimum loop delay. This *loop delay assumption* also avoids the delay hazard, a principal obstacle for speed-independent (SI) circuits [20]. The conditions leading to an essential hazard can also cause a function M-hazard.

In FANTOM, a technique based on [1, 7], removes function hazards, and also eliminates essential, delay, and combinational hazards. This technique involves a single variable addition, allowing for a simple implementation. This variable marks potentially hazardous states, and prevents outputs during them. Combining both old and new methods, our AFSM is free of hazards and removes restrictions placed on inputs and outputs.

3 Extended SI Model for FANTOM

In SI circuits, all state transitions end in the same terminal class, the set of all stable states. It has been stated [16], however, that it is impossible to build truly SI circuits because they cannot react instantaneously to inputs, and thus cannot guarantee the terminal-class requirement. A subset of SI circuits, known as semimodular, can guarantee the terminal class requirement. These circuits have the following properties [14]. First, inputs are required to be persistent, which means that once changed, they remain invariant until the circuit has assimilated them. Second, the flow-table representation must be strongly connected, meaning that every stable state can be reached from every other stable state. Third, each state must have a unique bit-vector assignment. In addition, the allowed state sequence must be non-consecutive, to ensure detection of input assimilation. Thus, most circuits do not allow "like-successive" inputs, meaning that the same input vector can be used in succession, such as $< 0101 >$ preceding $< 0101 >$.

A general property of asynchronous circuits, regardless of the model, is that inputs and outputs are considered level. Therefore, a Huffman flow table can be used to represent circuit behavior. Persistence requires using some form of completion detection to define when the outputs are stable and the inputs can change. One method uses an external G (Go) signal that asserts when new inputs are available, and an internal R (Reply) signal that asserts when the outputs are ready [14]. Persistence is related to "fundamental mode", since the inputs do not change until the network is stable.

FANTOM's extended model removes the restriction on allowed sequences to include "like-successive inputs". The machine operates correctly given these inputs because completion detection is independent of the input sequence. To accomplish this, the G signal is generated internally when the circuit is stable *and* the inputs are ready. The R signal still asserts when the circuit, and hence output vector, is stable.

The delay assumption of the SI model considers gate delays to be unbounded, but finite, and wire delays to be negligible. Delay elements are not allowed in the feedback path, since the nature of the SI delay assumption makes it unnecessary to include them. Therefore, FANTOM does not include these elements, making a simpler state machine.

4 FANTOM Architecture

Figure 1 depicts the block diagram of a FANTOM state machine. It consists of two sets of positive, edge-triggered flip-flops, and combinational logic. $\hat{X}$ and $\hat{Z}$ denote the external inputs $(X_1, \ldots, X_j)$ and the external outputs $(Z_1, \ldots, Z_k)$, respectively. Internal signals include the input vector $\hat{x} = (x_1, \ldots, x_j)$, present state vector $\hat{y} = (y_1, \ldots, y_n)$, next state vector $\hat{Y} = (Y_1, \ldots, Y_n)$, and output vector $\hat{z} = (z_1, \ldots, z_k)$.

4.1 Self-synchronization Signals

Self-synchronization in FANTOM involves the three signals G, VOM (valid output marker), and VI (valid input), and the input and output flip-flops. VI is associated with $\hat{X}$,

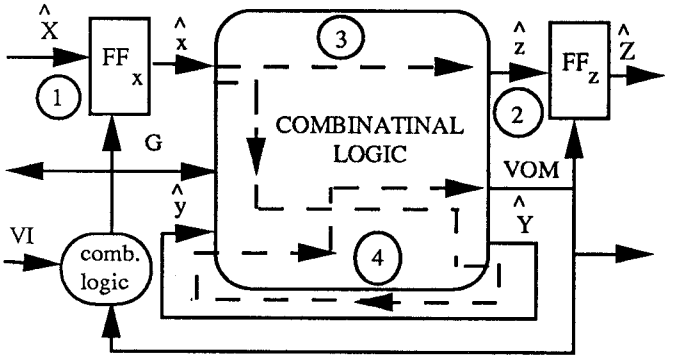

Figure 1: The FANTOM State Machine.

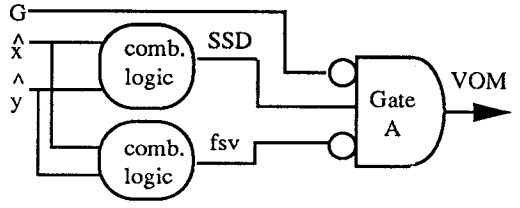

Figure 2: The VOM Block Diagram.

and is the VOM signal of the previous stage of a FANTOM state machine. As shown in Figure 1, G and VOM control FF_x and FF_z, respectively.

G allows new inputs into the network only if those inputs are stable (VI asserted) and the network has finished assimilating the previous inputs (VOM asserted). Because separate state machines are allowed to proceed at their own pace, $\hat{X}$ of the previous stage may be ready before the present stage needs them, or vice versa. Thus, G must remember if either VI or VOM asserted.

VOM asserts only after the circuit is in a stable state and $\hat{z}$ is ready. The circuit is stable when three signals, G, SSD (stable state detector), and fsv (fantom state variable) satisfy: $VOM = \overline{G} * \overline{fsv} * SSD$. Note that these signals are generated in the combinational logic part of the state machine. Figure 2 shows the block diagram for generating VOM. The signals fsv and SSD determine when the circuit is stable. The fsv signal hides circuit changes until $\hat{x}$ and $\hat{y}$ have settled, and SSD detects a new stable state. Once a new stable state is detected, $\hat{z}$ is latched to become the new $\hat{Z}$. Section 5 examines the synthesis procedures for generating fsv and SSD signals.

4.2 Implementation of Model Properties

Completion detection required for input persistence and fundamental mode operation is tightly coupled to the self-synchronization scheme described in the previous section. The R (Reply) signal of completion detection is implemented using VOM; the G signal implements "GO" [14]. The state sequence restriction described in Section 3 is overcome by permitting consecutive input vectors. These input vectors are allowed because VOM is deasserted when new inputs arrive, and reasserts when the circuit is stable and the outputs are ready.

4.3 Timing Considerations

As shown by the dashed and numbered paths in Figure 1, there are four critical paths in the FANTOM architecture. The signal dependencies in these paths must be considered to ensure proper operation. This discussion begins with the following definitions:

t_{su}^{FF} : setup time for a flip-flop
t_g^G : time needed to generate G
t_g^z : time needed to generate $\hat{z}$
t_g^{VOM} : time needed to generate VOM
t_d^A : delay time through Gate A
$\alpha : max(\hat{x}, \hat{y})$, time to generate $\hat{x}, \hat{y}$
t_g^{SSD} : time needed to generate SSD
t_g^{fsv} : time needed to generate fsv

Critical paths 1 and 2 involve the setup times of FF_x and FF_z. Critical path 3 involves the generation of $\hat{z}$. To operate correctly, the outputs must be stable t_{su}^{FFz} before VOM asserts. VOM depends upon critical path 4 which follows the path through the combinational logic needed to generate fsv.

To meet the setup requirements of FF_x, $t_{su}^{FFx} \leq t_g^G$. To meet the setup requirements of FF_z, $t_g^z + t_{su}^{FFz} \leq t_g^{VOM}$, where $t_g^{VOM} = t_d^A + min(t_g^G, min(\alpha + t_g^{SSD}, \alpha + t_g^{fsv}))$. This relationship for critical path 2 subsumes critical path 3.

Critical path 4 concerns the continued disabling of VOM by fsv or SSD before G deasserts. This must happen to ensure that false outputs are not captured by FF_z. The relationship is the following: $(\alpha + t_g^{fsv}) and (\alpha + t_g^{SSD}) < t_d^A + t_g^G$. The relationship between critical paths 3 and 4 is guaranteed because of the loop delay assumption explained in Section 2.2. The feedback loop involving fsv, and hence VOM, will take longer than that of generating the outputs. The derivation of all timing relationships is discussed in [9].

5 SEANCE Synthesis Program

The flow chart of Figure 3 shows the steps of the SEANCE synthesis tool, each of which is described below.

5.1 Flow Table Preparation

Desired circuit behavior is specified using a normal-mode flow table, which may be completely or incompletely specified. This table is directly generated from state diagrams, or

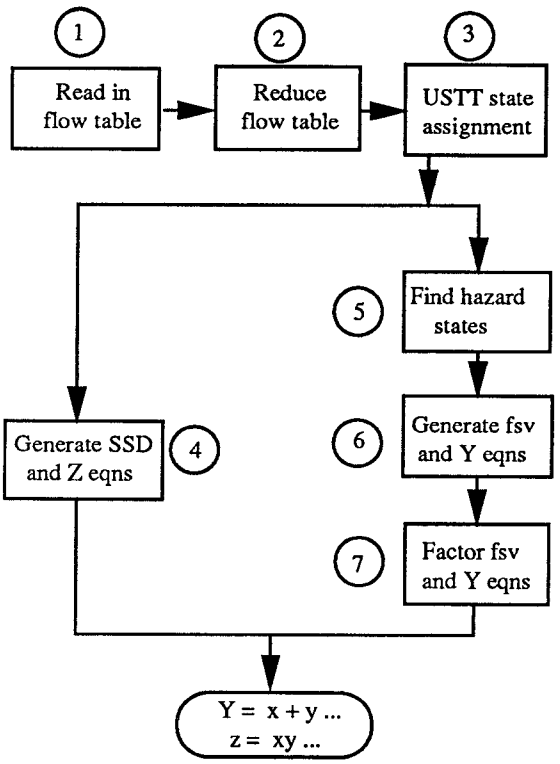

Figure 3: The SEANCE Synthesis Procedure.

can be easily derived from signal transition graphs (STG). "Normal mode" means that only one unstable transition is entered in going from one stable state to another. Because the program can handle incompletely specified flow tables, SEANCE's generality is enhanced. The program assumes that the generated flow table is strongly connected.

Large flow tables benefit from Step 2, table reduction. Redundant states within the flow table are removed using state machine minimization methods [8], thereby reducing the complexity of the state assignment process. The resulting flow table retains the normal mode characteristic.

Step 3 finds a valid unicode single-time transition (USTT) state assignment for the reduced flow table. A USTT assignment is a special case of the STT assignment where only one code is assigned per row of the flow table [20]. The procedure uses partition sets [19], and has two advantages. First, it works with incompletely or completely specified flow tables. Second, critical races are avoided because transitions move between states that differ in only one bit (the other bits are invariant). The synthesis program uses a general algorithm that will generate the smallest number of state variables [19]. A flow table given a state assignment is called a specified flow table.

5.2 Output Determination Stage

Step 4 of the synthesis program generates the $\hat{z}$ and the SSD part of the VOM signal. Canonical equations for $\hat{z}$ are generated by collecting all the minterms for each variable. The program then uses the Quine-McCluskey reduction technique to produce an essential SOP expression [12]. The use of self-synchronization at the outputs removes the possibility of transient hazards, thus it is not necessary to include all prime implicants in the expression.

The equation for SSD begins with a canonical expression involving the minterms where $\hat{y} = \hat{Y}$. The same reduction techniques as for $\hat{z}$ are used to reduce this to an essential SOP expression. By not using all of the prime implicants, SSD may glitch if there is a multiple-input change. This causes no problems, though, because the loop delay assumption assures that SSD will settle before fsv is stable.

5.3 Hazard Analysis

The specified flow table is subjected to a function hazard analysis in Steps 5 through 7 of SEANCE. The technique of function hazard removal using the fsv is based on [7].

The analysis begins with identifying the possible function hazards within the specified flow table. A hazard list for each state variable and fsv is composed from the hazard states found upon traversing each "stable-state transition". In a Huffman-type flow table, a stable-state transition begins in a stable state, moves horizontally to the input change, and then vertically to the new stable state. This flow table movement defines an input and state-transition space. The hazard list for $\hat{Y}$, denoted HL, contains states with function hazards that occur within the input transition space. Each possible hazard affects only one state variable because of the properties of the USTT assignment. The hazard list for fsv, denoted FL, includes all the states found for $\hat{Y}$. The algorithm for this process is shown in Figure 4, using the following notation:

T: the specified flow table

$\mathcal{S}(\hat{x}, \hat{y})$: the set of all states in the machine

$S(\hat{x}, \hat{y}) \in \mathcal{S} | \hat{y} = \hat{Y}$: the set of all stable states

$s(\hat{x}, \hat{y}) \in \mathcal{S} | \hat{y} \neq \hat{Y}$: the set of all transition states

$\phi : S(\hat{x}^a, \hat{Y}^a) \rightarrow S(\hat{x}^b, \hat{Y}^b)$: a Huffman table transition from input vector a to b

n: bit subscript for the j state variables

In the algorithm in Figure 4, subscripts represent bit positions, and superscripts represent input vectors.

Step 6 of SEANCE generates the canonical sum-of-products (SOP) expressions for fsv and $\hat{Y}$. Each entry in the hazard list for fsv is a minterm in its SOP expression. The state variable expressions involve finding the minterms for when $fsv = 0$, and when $fsv = 1$. For the first case, any minterm that matches the hazard list is complemented. For the second case, all minterms are included

```
foreach ŷ_i ∈ T
    foreach S(x̂^a, Ŷ^a) ∈ S(x̂, ŷ)
        foreach φ | Hamming_distance(x̂^a, x̂^b) > 1
            k = (a + 1) to (b − 1)
            n = not_invariant(ŷ^a, Ŷ^b, Ŷ^k)
            if (n ≠ -1)
              then 1. HL_n = S(x̂^k, ŷ^a)
                   2. FL = S(x̂^k, ŷ^a)
        end_for; end_for; end_for
```

```
not_invariant ( ŷ^a, Ŷ^b, Ŷ^k )
        inv̂ = ŷ^a XOR Ŷ^b
        for n = 1 to j
            if (Ŷ^k_n ∧ inv̂_n) ∨ (Ŷ^k_n ∧ ŷ^a_n)
              return (n); end_for
        return (-1)
```

Figure 4: The Hazard Search Algorithm.

without change.

The equation for fsv is not a function of itself, and therefore cannot hold the value of the signal at one. Hence, we use the term "fantom" as a descriptive label for this variable. The effect of finding hazards in the machine doubles the state space, because the case when $fsv = 1$ must be handled.

In Step 7 the equations for fsv and $\hat{Y}$ are factored to prevent hazards. To avoid logic hazards, fsv is reduced to all its prime implicants using a technique such as Quine-McCluskey. Next, fsv is expanded to allow only "first-level gates" [1], which includes only true input variables and state variables. A term with complemented inputs is converted from an AND to an AND-NOR format. The resulting expression guarantees the first condition needed to avoid essential hazards, as explained in Section 2.2.

$\hat{Y}$ is factored according to the hazard factoring procedure of Figure 5. This factoring concept avoids delay and combinational hazards by substituting hazardous expressions with special subcube factorizations [1, 7]. The procedure first reduces each next-state equation to an essential SOP expression, for example, $Y_1 = \overline{fsv}(y_1x_1) + fsv(y_1x_1x_2) + fsv(y_2\overline{x}_1x_2)$. Then, common terms containing y_1 are extracted, producing an expression of the form $(L_1R_1 + fsv(y_2\overline{x}_1x_2))$, where L_1 contains the y_1 subcube and $R_1 = \overline{fsv} + fsv(x_2)$. The program then identifies the zero subcube within L_1, the term needed to make R_1 equal one. The expanded minterms of that zero subcube are called the set γ_1. Next, minterms of γ_1 that match the zero minterms of Y_1 are eliminated. The procedure substitutes the hazardous L_1R_1 with $L_1\overline{\gamma_1}$ in the SOP expression, and then converts the equation into a first-level gate expression.

```
given Y_i = fsv[∑ (minterms ∈ HL)]+
      fsv[∑ minterms = 1]
      standard reduction of Y_i
      factor common terms containing y_i to find L_i R_i
      ; identify zero subcube Z_i | R_i = 1
          Z_i = fsv β_i; where β_i ≡ remaining terms
          γ_i = ∑ minterms(Z)
          remove redundant minterms ∈ γ_i
          substitute R_i with γ_i in Y_i
      factor Y_i according to "first-level gate" definition
```

Figure 5: The Hazard Factoring Procedure.

Benchmark	fsv Depth	Y_i Depth	Total Depth
test example	3	5	9
traffic	3	5	9
lion	3	5	9
lion9	4	5	10
train11	2	5	8

Table 1: Results Using MCNC Benchmarks.

6 Experimental Results

Table 1 presents the results of running SEANCE on the MCNC benchmark suite [11]. The depth of fsv and the longest Y_i variable are used as a measure of the complexity of the resultant state machine. "Depth" refers to the number of levels in the logic equation. The last column "Total Depth" refers to the levels of logic that must be traversed in a worst-case, hazard-detected situation for the network to reach stability (assertion of VOM).

SEANCE takes about four seconds of CPU time on a Digital Equipment VAXStation 3100 to run an example.

Hackbart and Dietmeyer have commented in [7] on the possible slowed response of a network using a hazard detection variable. The experimental results in this section show that the levels of state variable logic can be high.

7 Discussion

The preceding sections have explained how multiple-input change, hazard-free AFSMs are created based on the FANTOM model and using the procedures in SEANCE. This section examines the difference between this method and another which provides for multiple-input change AFSMs.

STGs have been used in other architectures to allow multiple-input changes [3, 13, 17]. The STG, based on Petri Nets [15], assigns input changes to directed arcs. Hazardous input changes are avoided by adding arcs so that inputs remain persistent as the graph is tranversed one bit (arc) at a time [13]. Hence, the input space has been expanded to move in single-bit steps to avoid the hazards associated with multiple-input changes. In this paper, the

hazards which restrict inputs to single-bit changes are removed by expanding the state variable space. The variable fsv implements this expansion. Essentially, a FANTOM machine moves through at most two state changes regardless of the number of bit changes in the input. This simplifies several steps of the synthesis process, such as finding and neutralizing hazards.

8 Conclusions

This paper has described a new architecture and synthesis tool for the implementation of a hazard-free, multiple-input and multiple-output change AFSM. The machine works by detecting hazardous states, and preventing output during them. In addition, the machine ensures that the hazard does not affect proper state transitions. The resultant state machine has some overhead, but there is greatly increased flexibility. In addition, the circuit implementations are robust since hazards are removed without relying on the insertion of complex hardware, such as decoding boxes or delay elements.

A synthesis tool, SEANCE, has been developed that automatically creates FANTOM state machines from a completely or incompletely specified normal-mode flow table. SEANCE employs a number of unique techniques for finding and eliminating hazards.

Acknowledgements

The authors wish to thank Michael Butler comments on FANTOM, and Troy Brandel and Andrew Ladd for writing some of the software. We also thank Bob Colwell for helpful suggestions.

References

[1] D.B. Armstrong, A.D. Friedman, and P.R. Menon. Realization of asynchronous sequential circuits without inserted delay elements. *IEEE Transactions on Computers*, C-17(2), February 1968.

[2] D.B. Armstrong, A.D. Friedman, and P.R. Menon. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers*, C-18(12), December 1969.

[3] T-A. Chu. Synthesis of self-timed VLSI circuits from graph-theoretic specifications. In *Proc. of the International Conference on Computer Design*. IEEE, 1987.

[4] H. Chuang and S. Das. Multiple-input change asynchronous machines using controlled excitation and flip-flops. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory*. IEEE, October 1973.

[5] E.B. Eichelberger. Hazard detection in combinational and sequential switching circuits. In *Proc. of the 5th Annual Symposium on Switching Circuit Theory and Logical Design*, November 1968.

[6] A.D. Friedman and P.R. Menon. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers*, C-17(6), June 1968.

[7] R. Hackbart and D. Dietmeyer. The avoidance and elimination of function hazards in asynchronous sequential circuits. *IEEE Transactions on Computers*, C-20(2), February 1971.

[8] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, New York, 1978.

[9] M. Ladd and W. Birmingham. Synthesis of multiple-input change asynchronous finite state machines. Technical report, University of Michigan, Dept. of Electrical Engineering and Computer Science, Forthcoming 1991.

[10] G.G. Langdon. Analysis of asynchronous circuits under different delay assumptions. *IEEE Transactions on Computers*, C-17(12), December 1968.

[11] R. Lisanke. Finite-state machine benchmark set. In *MCNC Logic Synthesis Workshop*, 1987.

[12] M. Mano. *Digital Logic and Computer Design*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[13] T. Meng, R. Brodersen, and D. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11), November 1989.

[14] R.E. Miller. *Switching Theory*. Wiley, New York, 1966.

[15] D. Misunas. Petri nets and speed independent design. *Communications of the ACM*, 16(8), August 1973.

[16] D.E. Muller and W.S. Bartky. A theory of asynchronous circuits. In *Proc. of the International Symposium on the Theory of Switching*, volume 1, Cambridge, MA, 1957. Harvard University Press.

[17] C.L. Seitz. Asynchronous machines exhibiting concurrency. In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970.

[18] C.L. Seitz. System timing. In *Introduction to VLSI Systems*. Addison-Wesley, Reading, MA, 1980.

[19] J.H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15(4), August 1966.

[20] S.H. Unger. *Asynchronous Sequential Switching Circuits*. Krieger, Melbourne, FL, 1969.

[21] S.H. Unger. Asynchronous sequential switching circuits with unrestricted input changes. *IEEE Transactions on Computers*, C-20(12), December 1971.