Algorithms for the Optimal State Assignment of Asynchronous State Machines *

Robert M. Fuhrer
Dept. of Computer Science
Columbia University
New York, NY 10027

Bill Lin
IMEC Laboratory
Kapeldreef 75
B-3001 Leuven, Belgium

Steven M. Nowick
Dept. of Computer Science
Columbia University
New York, NY 10027

Abstract

This paper presents a method for the optimal state assignment of asynchronous state machines. Unlike state assignment for synchronous state machines, state codes must be chosen carefully to insure the avoidance of critical races *and* logic hazards*[29]. Two related problems are considered: (i)* optimal critical race-free state assignment*; and (ii)* optimal hazard-free and critical race-free state assignment *for* normal fundamental mode *machines. Analogous to a paradigm successfully used for the optimal state assignment of synchronous machines [17], each problem is formulated as an* input encoding problem*. Solutions are targeted to sum-of-products implementations. Initial results indicate output logic improvements up to 20% for the hazard-free algorithm, and more modest improvement for the optimal critical race-free algorithm.*

1 Introduction

There has been a renewed interest in asynchronous design, because of their potential for high-performance, modularity and avoidance of clock skew [30, 24, 14, 2, 10, 13]. This paper focuses on one class of asynchronous designs: asynchronous state machines. The design of asynchronous state machines has been an active area of research for the last 40 years (see [29]). However, asynchronous state machine design remains a subtle problem, since to insure correct dynamic behavior, *hazards* and *races* [29] must be eliminated.

Several recent methods have been introduced which demonstrate the practicality of asynchronous state machine synthesis [24, 20, 33, 4, 18]. Each method produces low-latency machines which are guaranteed hazard-free at the gate-level. These methods have been automated and applied to some significant industrial examples: an adaptive routing chip [6], a cache controller [19], an infrared communications chip [1] and a SCSI controller [23]. The design tools have benefited from a number of recent hazard-free optimization algorithms: exact two-level logic minimization [21], multi-level logic optimization [29, 9, 11], technology mapping [27] and synthesis-for-testability [8, 22]. However, none of these methods includes algorithms for optimal state assignment. The contribution of this paper is to present solutions to the optimal state assignment problem for asynchronous state machines.

There have been several important developments in optimal state assignment of synchronous machines. De Micheli [17] formulated and solved an *input encoding problem*, which approximates an optimal state assignment for PLA-based state machines. His CAD tool, *KISS*, first performs symbolic logic minimization, and then solves a resulting set of encoding constraints to produce a state assignment. Alternative formulations as an *output encoding* or *input/output encoding problem* have also been developed [16, 26, 3]. The *NOVA* program [31] produces PLA-based solutions with area reduction of *30%* over random assignments. Other recent methods have targeted *multi-level* [12, 7] and *low-power* [32] implementations.

*Acknowledgment: This work was supported in part by NSF under Grant no. MIP-9308810 and by the E.C. under Grant no. ESPRIT-6143 (EXACT).

Synchronous state assignment methods are inadequate for asynchronous designs, since the resulting machines may have critical races and logic hazards. In existing synthesis trajectories [34, 4, 18], the state assignment step is currently performed only to ensure critical race-free codes [28] without consideration for the optimality of the resulting logic, which may lead to unnecessarily expensive implementations. In this paper, we formulate and solve two related asynchronous state assignment problems: (i) *optimal critical race-free state assignment*; and (ii) *optimal hazard-free and critical race-free state assignment* for *normal fundamental mode* machines (*i.e.,* hazard-free for single-input changes [29]). As in KISS, each problem is formulated as an input encoding problem; solutions are targeted to a sum-of-products implementation. The requirement of critical race-free codes introduces some subtleties into both algorithms. As a result, our solution to (i) is quasi-optimal, while our solution to (ii) is exactly optimal with respect to output logic. As in KISS, both solutions are approximate with respect to next-state logic. Initial results indicate output logic improvements up to 20% for the hazard-free algorithm, and more modest improvement for the optimal critical race-free algorithm.

This work is a first step towards a general solution to the optimal state assignment problem for hazard-free and critical race-free machines (allowing multiple-input changes), cast in terms of the input/output assignment problem.

The paper is organized as follows. Section 2 gives background on optimal state assignment and asynchronous state machines. Section 3 introduces our new symbolic minimization and encoding algorithms for problem (i), and Section 4 describes related algorithms for problem (ii). Section 5 presents theoretical results on the optimality of the resulting solutions. Section 6 describes the program implementation, Section 7 presents experimental results, and Section 8 describes conclusions.

2 Background

2.1 Optimal State Assignment for Synchronous Machines

For the following, we assume basic familiarity with the terminology of multi-valued logic minimization (see [25]).

In KISS [17], De Micheli formulated the optimal state assignment problem as an *input encoding problem*. The goal is to find a binary encoding of symbolic inputs to insure an optimal sum-of-products implementation. The algorithm has three steps:

1. Generate a minimal symbolic cover
2. Generate a set of encoding constraints
3. Solve these constraints to produce a state assignment

The first step is symbolic logic minimization [17]. The next-state function is effectively treated as a *set* of functions, one for each possible next-state value, since no information is yet available as to the relation of the various next-state values to one another. As a result, the symbolic minimization problem can be formulated as a multiple-output multiple-valued-input minimization problem and solved using *espresso-mv* [17]. A minimal symbolic cover is formed, consisting of a set of symbolic implicants. Each implicant has four parts: binary inputs, symbolic present state, symbolic next state, and binary outputs. Present and next state can be represented using either symbolic or positional-cube notation.

A key goal in optimal state assignment is to insure the correctness of the symbolic cover after it is instantiated with binary state codes. To understand the problem, consider the state table of Figure 1, having 2 inputs, 4 symbolic states, and 1 output, and the given 2-variable state assignment. A minimal symbolic cover for the output consists of 2 symbolic implicants: p_1 =<0* {D}> and p_2 =<*1 {B,C}>. [1] Implicant p_1 contains a single symbolic state, D, and therefore can be

[1] For simplicity, we consider only single-output implicants in this example, though in general the method produces multiple-output implicants.

instantiated as binary product <0* 11>. However, implicant p_2 contains a pair of symbolic states, B and C, forming a *state group*. The smallest single binary cube, or *group face*, which contains the state codes for B and C is the *supercube* of the two codes: $**$. In this case, the resulting binary product, <*1 **>, is *invalid*, since it also contains an OFF-set minterm <11 00> corresponding to symbolic minterm <11 {A}>.

	00	01	11	10	
A	A,0	A,0	D,0	A,0	00
B	B,0	B,1	B,1	A,0	01
C	A,0	B,1	C,1	C,0	10
D	D,1	D,1	D,0	C,0	11

Figure 1. Example state table with state assignment

To avoid this problem, in the second step, *face embedding constraints* are imposed:

For each symbolic implicant p, with state group S_p, the corresponding group face must not intersect the code of any state s not in S_p. [17]

The third step is to find a state assignment satisfying these encoding constraints. A final step, after state assignment, is to produce a binary logic implementation. Typically, *espresso* or *espresso-exact* are used, since the resulting cover may have *smaller* cardinality than the symbolic cover (see [17]).

The above encoding constraints can be described using *dichotomies* [3, 28]. Given a set of states S, a dichotomy is a bipartition (U, V) of a subset T of states of S. In a given state assignment, a binary state variable y_i *covers* the dichotomy (U, V) if $y_i = 0$ for every state in U and $y_i = 1$ for every state in V (or vice-versa) [29, 28]. For the given problem, a set of *n-to-1 dichotomies* is formed, *i.e.,* between each state group S_p (containing n states) and each single disjoint state $s \notin S_p$. In the above example, dichotomies $(BC; A)$ and $(BC; D)$ are generated to prevent invalid state assignments with respect to the output. Exact dichotomy solvers have been developed which produce minimum-length assignments [3, 26].

A *1-hot encoding* [29, 17] always satisfies the above constraints, and can be used to implement the symbolic cover. This canonical state assignment has an important property:

> **Property #1:** The resulting binary cover using a 1-hot code has the same cardinality as the original symbolic cover.

Property #1 indicates that the cardinality of the symbolic cover is an upper bound on the size of a binary solution. In addition, for any state assignment satisfying the above constraints, the following property holds:

> **Property #2:** The resulting binary *output* cover is *exactly minimal* (if outputs are minimized separately from next-state).

Property #2 indicates that, if symbolic and binary minimization avoid the sharing of products between outputs and next-state, the resulting output cover is exactly minimal. That is, the optimal state assignment problem can be solved exactly with respect to outputs.

2.2 Asynchronous State Machines

An asynchronous state machine can be described by a flow table [29]. The key difference between an asynchronous and a synchronous flow table is that in the former, the *transient* behavior of the machine during input and state changes is considered; in the latter, it is not.

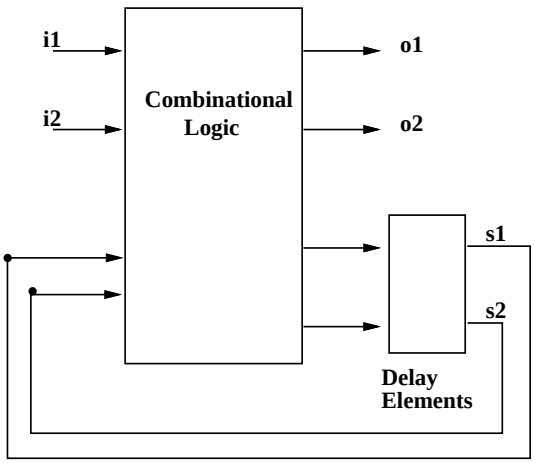


Figure 2. Block diagram of a Huffman machine

In this paper, two common assumptions are made on the form of an asynchronous flow table. First, every unstable state in the flow table must lead directly to a stable state; that is, no multi-cycling through different intermediate symbolic states is permitted. Second, during a state change, an output may change at most once, and may change only at the start of the state change. More formally, we assume a *single-output-change (SOC)* flow table in *standard form* [29].

An asynchronous flow table is shown in Figure 1. Each unstable state leads directly to a stable state; for example, $C-00$ is unstable, leading directly to stable state $A-00$. Moreover, an output change occurs only at the start of a state change; for example, the output is 1 in stable state $B-11$; if an input changes to $B-10$, the machine becomes unstable but will stabilize in $A-10$. The output changes from 1 to 0 in $B-10$ and remains unchanged throughout the state change.

An asynchronous flow table can be implemented as a *Huffman machine* (see Figure 2), with primary inputs, primary outputs, and fedback state variables. State is stored on the feedback loops, which may have attached delay elements. In any state, a Huffman machine may accept inputs, generate outputs, and change state. In a synchronous machine, inputs are processed at discrete clock ticks, and the transient behavior of the machine is irrelevant. In contrast, in a Huffman machine, there is no clock or latches. Therefore, the behavior of the machine depends on its transient operation during input and state changes.

2.3 Critical Races

Asynchronous state assignment differs from synchronous assignment because transient behavior must be considered. A *critical race* [29] exists in an encoded (non-symbolic) asynchronous flow table for a given state transition in a given input column if:

- 2 or more state variables change value in the transition;
- there is another state transition (possibly degenerate, *i.e.,* from a state to itself) in the same input column; and
- some assignment of feedback delays can cause the machine to enter a (possibly transient) state held in common by the two transitions

In the presence of a critical race, the machine may end up in the wrong internal state, even after the logic and feedback paths settle. It is well-known [29, 28] that critical races can be avoided by judicious choice of encoding. Specifically, the following two types of interaction among state transitions *within the same input column* must be avoided:

(i) An unstable transition contains a stable state as one of its transient states

(ii) An unstable transition contains a transient state of another unstable transition as one of its transient states

Example. These two conditions are illustrated in Figure 1. Case (i): Assume the state assignment: $A = 00$, $B = 01$, $C = 10$, $D = 11$. In column 11, there is a transition from $A = 00$ to $D = 11$; the remaining states are stable. Since both state bits change, the transient states 01 and 10 may be entered, and the machine may incorrectly stabilize in states B or C. Case (ii): Assume the state assignment: $A = 0000$, $B = 1100$, $C = 0101$, $D = 0110$. In column 10, there are two transitions: from B to A and from D to C. The B to A transition will not pass through either C or D, but both transitions intersect in a common transient state 0100. □

In general, we consider only *unicode single-transition time (USTT)* assignments, which insure that the machine moves directly from unstable to stable state [29]. The following theorem formalizes conditions (i) and (ii) in terms of dichotomies, to insure a USTT *critical race-free* assignment:

Theorem 3.1 (Tracey Conditions). A state assignment for an SOC flow table is a valid USTT assignment if and only if, for each pair of state changes $W \rightarrow X$ and $Y \rightarrow Z$ which appear in the same input column, where $X \neq Z$, the associated dichotomy (WX, YZ) is covered by at least one y-variable of the assignment. □

All non-trivial Tracey conditions can be described using *2-to-2* and *2-to-1 dichotomies*, that is, dichotomies between a pair of states (*i.e.,* WX) and either a singleton state (*i.e.,* case (i), where $Y = Z$) or another pair of states (*i.e.,* case (ii), where $Y \neq Z$). This is in contrast to face-embedding constraints, which are described using n-to-1 dichotomies.

Example. For the given example, the assignment $A = 111$, $B = 001$, $C = 100$, and $D = 010$ is critical race-free. For the A to D transition in column 11, dichotomies (AD, B) and (AD, C) are generated. For the B to A and D to C transitions in column 10, the dichotomy (AB, CD) is generated. Each of these dichotomies is covered by the given state assignment. □

3 Optimal Critical-Race Free Assignment

We can now define the first synthesis problem:

> **Problem #1: Optimal Critical Race-Free Assignment:** *Find a critical race-free STT assignment for an asynchronous flow table having a sum-of-products implementation of minimal cost.*

Synchronous state assignment methods are inadequate for this problem, since they do not take into account critical races and transient behavior. Our synthesis method follows the 3 basic steps of the KISS algorithm, but with modifications. In the first step, a constrained symbolic minimization problem is formulated. In the second step, modified encoding constraints are generated. These constraints are *not* the union of the KISS and Tracey constraints, but subsume both. Finally, a dichotomy solver is used to solve the constraints.

3.1 Symbolic Logic Minimization

Although it might appear that a standard symbolic minimization algorithm, such as espreso-mv, can be used for the first step, a problem arises which such algorithms cannot handle.

Consider an optimal symbolic logic cover as a starting point for a machine implementation. By optimal we mean having the fewest possible number of product terms. By symbolic, we mean that, lacking a state assignment at this stage, we treat the next-state function as a set of symbolic or multi-valued functions; likewise, the output function is multi-valued.

Roughly speaking, we wish to instantiate the symbolic logic cover with an appropriate state assignment to produce a binary implementation. Following the analogy with KISS [17], an important goal is to insure — as a minimum requirement — that the symbolic cover, when instantiated with binary codes, yield a valid implementation of the machine.

Asynchronous state tables are fundamentally different from synchronous state tables since, in the former, the transient behavior is defined. Therefore, to insure that the machine's output and next-state functions are *properly* implemented, these functions must be defined not only at the end points of each state transition, but during all intermediate points as well (if any). However, in our scheme, logic minimization is first performed symbolically; only the end points of a state transition can be identified. The intermediate points are inaccessible, as their present-state portion is between two symbolic points whose location in Boolean space is as yet undetermined. Nevertheless, all ON-set points must be covered, in order for the logic implementation represented by the symbolic cover to be correct when instantiated with a state encoding.

Example. Consider a flow table having a state transition in column $I1$ from state $S1$ to $S3$. Some subset of the state bits will change value as the machine moves from $S1$ to $S3$ in an encoded implementation. The following fragment illustrates the situation for $I1$ = `000`, $S1$ = `0100`, and $S3$ = `1101`, and one possible sequence of bit changes:

I1,S1	==	000,0100
		000,1100
I1,S3	==	000,1101

The intermediate point $000, 1100$ is not visible in symbolic space; hence, a symbolic minimization process which works by covering *points* in the Boolean/symbolic domain is guaranteed only to cover each of the two end points with some product term. Although the minimal cover might include a single term covering both end points (and hence the intermediate points as well), this is not true in general. □

Therefore, as long as logic minimization proceeds within the symbolic domain, the only solution is to insure that each state transition is *entirely contained* in some product of the symbolic cover. In the previous example, the state transition from $S1$ to $S3$ in input column 000 can be described by a symbolic cube: t =`<000 {S1,S3}>`. The symbolic minimization algorithm must be modified to insure that this entire *required cube* [2] is covered by some product of the cover. This product, when instantiated with a binary assignment, is guaranteed to contain all intermediate points of the state transition.

Note that this may result in a Boolean cover which is sub-optimal overall: if an encoding were available during logic minimization, the minimal cover might use several product terms to cover the cube spanned by a state transition, instead of being constrained to use a single term for that purpose. An example illustrates the preceding discussion.

Example. Figure 3 gives a flow table and a critical-race free state assignment. The minimal symbolic cover for the single output is the pair of product terms `<0* {D}>` and `<*1 {B,C}>`. However, the transition $C \rightarrow B$ in column `01` may pass through total states `<01 000>` and `<01 101>`, neither of which is visible in the symbolic domain. Hence, the above cover, when instantiated with the given encoding, fails to implement two on-set minterms in the Boolean domain.

If, on the other hand, the next-state entry in total state `<10 B>` were changed to C, a different critical-race free assignment results: A = `101`, B = `010`, C = `110`, D = `000`. Using this new assignment, only one state bit changes during the transition $C \rightarrow B$ in column `01`. Thus, although there are no transient points in the given state transition, the symbolic cover is constrained to include the product term `<01 B,C>`, which is not part of the minimal solution. □

[2] This term is borrowed from algorithms for hazard-free logic minimization [21], which set up a similar cube covering problem.

	00	01	11	10	
A	A,0	A,0	D,0	A,0	111
B	B,0	B,1	B,1	A,0	001
C	A,0	B,1	C,1	C,0	100
D	D,1	D,1	D,0	C,0	010

Figure 3. State table with critical race-free assignment

Symbolic Logic Minimization Algorithm

Based on the above discussion, symbolic minimization must be modified. The ON-set is now described by a set containing both minterms *and* required cubes. For every output or next-state function, a required cube is generated for each state transition where the function has the value 1. Symbolic prime implicants are generated, and a modified symbolic covering problem is set up, where each specified ON-set minterm and required cube must be contained in some symbolic implicant in the cover. The resulting problem is solved using a standard unate covering algorithm, such as *mincov* [25].

3.2 Encoding Constraints

In step 2, encoding constraints are generated based on the symbolic cover. These constraints insure that the cover can be correctly instantiated.

The face embedding constraints used by KISS for synchronous machines are insufficient for our purposes, since they do not consider the transient behavior of an asynchronous state machine. We first review the KISS constraints, then indicate generalizations needed to handle asynchronous state machines.

Example. Consider a symbolic implicant p belonging to some minimal symbolic cover for some output z: `<011 {S1,S2}>`. This implicant has a binary instantiation `<011 xx>`, where `xx` is the supercube of the codes for `S1` and `S2`. If `S1` is assigned code `1010` and `S2` is assigned code `1100`, the binary implicant is `<011 1**0>`. Suppose that output z is specified at 0 in total state `011 S3`. If `S3` were assigned state code `1110`, the implicant could turn on in this total state, and z would be incorrectly implemented as 1 in total state `011 S3`. To prevent this encoding, a face embedding constraint $(\{S1, S2\}, S3)$ is generated. This *n-to-1* dichotomy insures that the group face for $\{S1, S2\}$ does not intersect the encoding for state $S3$. □.

For the asynchronous case, face embedding constraints must be generalized. A new condition concerns the correctness of the implementation of output and next-state functions in the presence of state transitions. During a transition of 2 or more state variables, transient points in the total input state space are reached which do not correspond to any valid encoded state. The possibility arises that the group face for some symbolic product term implementing a binary output may intersect such a transient point, thus inadvertently turning on the product term *during* the state transition. If the intended value of that output during the state transition is `0`, the output function will be incorrectly implemented.

Example. Consider a binary output symbolic implicant `<I1 {S0,S1,S2}>` for some output function z. Suppose there is a state transition `S3` $\rightarrow$ `S4` in input column `I1` during which the binary output should be held at `0`. Assume the following state assignment:

S0 0000
S1 1000
S2 1100
S3 0110
S4 0101

Using this assignment, the corresponding binary implicant is `<I1 **00>`. As a result, during the

S3 → S4 transition, the state variables can reach the transient value 0100 which would turn on the given implicant, incorrectly forcing the output value to 1. □

A similar problem occurs for the next-state function. In this case, the function requires a trivial generalization of the condition: if the value of the symbolic function (i.e. the destination state) during the transition differs from that which the product term implements, the machine could have an incorrect implementation.

The proposed solution is to add dichotomy constraints to avoid such problems resulting from state transitions. Unlike the face embedding constraints, these dichotomies are *n-to-2*: between (i) a state group of an symbolic product (*e.g.,* $\{S0, S1, S2\}$ in the preceding example) and (ii) a pair of states defining a state transition (*e.g.,* $\{S3, S4\}$). The resulting generalized embedding constraint, ($\{S0, S1, S2\}$,$\{S3, S4\}$), insures that the output will be correctly implemented after instantiation.

The above discussion only addresses constraints derived from a symbolic cover. It does not consider critical race-free encoding constraints. In Section 5, however, it will be shown that the above constraints subsume all Tracey constraints, and therefore insure a critical race-free assignment.

Constraint Generation Algorithm

In addition to the KISS face embedding constraints, we use the following algorithm:

```
for each implicant p in the symbolic cover {
 for each state transition t in an input column which p intersects {
   for each output o that p implements {
     if o = 0 in transition t
       generate dichotomy { stategroup(p); states(t) }
     }}}
```

3.3 Solving Constraints and Binary Logic Minimization

Since all constraints are described as dichotomies, they can be solved using a dichotomy solver. The resulting binary cover is then a valid binary instantiation of the original symbolic cover, which takes into account transient states. It can therefore be minimized using an existing binary-valued logic minimizer [25].

4 Optimal Hazard-Free and Critical-Race Free Assignment of Normal Fundamental-Mode Machines

The next problem is to consider a more restricted class of asynchronous state machines, but to perform state assignment to insure an optimal *hazard-free* [29] implementation. Therefore, we need to consider how an asynchronous state machine can be operated, and how hazards arise. A more detailed presentation can be found in Unger [29].

4.1 Background

Operating Modes

The simplest operating mode for an asynchronous state machine is *single-input change (SIC)*: the machine receives one input change at a time. A more general operating mode, which we do not consider in this section, is *multiple-input change (MIC)*, where several inputs can change. [3] Fur-

[3] Our SIC optimal state assignment methods can be applied to MIC state machines, but optimality of results is not guaranteed.

thermore, it is assumed that the machine operates in *fundamental mode:* once an input change has occurred, no further inputs may change until the machine has stabilized. Along with the earlier assumptions (SOC flow table in standard form), such a machine is called *normal fundamental mode*.

Hazards

We continue to assume a critical race-free state assignment and target a Huffman machine, as in Figure 2. An *essential hazard* can occur if the next-state change occurs before the machine has fully absorbed the input change. Essential hazards can always be avoided by adding delays to the feedback path. In our approach, we add such delays as needed to separate an input change from its resulting state change. As a result, sequential operation is transformed into 2 distinct combinational operations: (i) the processing of the input change, and (ii) the subsequent processing of a resulting state change. Our goal is to synthesize combinational logic which is free of glitches for each possible input change and state change. Under the conservative assumption that gate and wire delays may assume any finite values, and the logic must always remain glitch-free, our goal is therefore to produce *hazard-free logic*.

In step (i), the logic receives a single-input change on a primary input, which corresponds to a horizontal or row transition in the flow table. In step (ii), several state bits may change, so the logic may receive a multiple-input change, corresponding to a vertical or column transition in the flow table. A $0 \rightarrow 0$ or $1 \rightarrow 1$ change in an output or next-state variable is called a *static transition*, and a $0 \rightarrow 1$ or $1 \rightarrow 0$ change is called a *dynamic transition*. Since the flow tables are assumed to be in standard form, SIC row transitions of an output or next-state variable may be either static or dynamic. However, MIC column transitions (where the next state changes) *must* be static. This holds, because in a state transition, outputs and next-state may only change at the start of the transition; they may not change during the transition.

Hazard Elimination

Since our optimal state assignment algorithms are targeted to sum-of-products realizations, our concern is with hazard elimination in such realizations.

First, consider an SIC row transition, such as in row B in the flow table of Figure 3. All four SIC transitions are possible for the output: $0 \rightarrow 0$ (from input state 00 to 10), $0 \rightarrow 1$ (from 00 to 10), $1 \rightarrow 0$ (from 11 to 10) and $1 \rightarrow 1$ (from 01 to 11). In any sum-of-products realization of the output, for the given state assignment, the first three of these transitions are guaranteed hazard-free. Only the fourth — $1 \rightarrow 1$ — requires special consideration. In particular, the supercube of this pair of binary minterms defines a product, $r = *1\,001$, which covers both minterms. A sum-of-products implementation is hazard-free for this transition *if and only if* some product p in the cover contains r [29]. Such a product is called a *required cube* [21], since it must be contained in an implicant of the cover to insure a hazard-free transition. Similar conditions apply to the next-state variables.

Next, consider a (possibly) MIC column transition. In column 10, the output makes a $0 \rightarrow 0$ transition from state $D = 010$ to state $C = 100$, where two state bits change. Since the next-state function remains at $C = 100$ throughout this transition, the first next-state bit makes a $1 \rightarrow 1$ transition and the last two bits each make a $0 \rightarrow 0$ transition. The encoding is critical-race free, so there is no problem with transient states where the output or next-state is otherwise defined (more formally, no *function hazard* can occur). In any sum-of-products realization, each $0 \rightarrow 0$ MIC transition is guaranteed hazard-free. For an MIC $1 \rightarrow 1$ transition, Eichelberger proved that, to eliminate a hazard in a sum-of-products implementation, the entire *required cube* for the transition must be contained in a product of the cover [29]. For example, for the $1 \rightarrow 1$ output transition in column 01 from state $C = 100$ to state $B = 001$. the entire required cube `<01 *0*>` must be contained in some product of the cover.

In summary, to eliminate all hazards for the given machine, it is necessary and sufficient to cover the required cube for each $1 \rightarrow 1$ transition (SIC or MIC). Therefore, a constrained covering problem must be solved.

4.1.1 Problem Statement and Overview

We can now define the second synthesis problem:

> **Problem #2: Optimal Hazard-Free/Critical Race-Free Assignment for Normal Fundamental Mode Machines:** *Find a critical race-free STT assignment for an normal fundamental mode flow table having a* **hazard-free** *sum-of-products implementation of minimal cost.*

Optimal synchronous assignment methods are inadequate, not only because they do not consider critical races and transient behavior, but because they do not target a hazard-free implementation.

Again, our synthesis method follows the 3 basic steps of the KISS algorithm. In the first step, it formulates a constrained symbolic covering problem. Unlike the algorithm of Section 3, this is a limited *symbolic hazard-free covering algorithm* (limited, since it does not deal with general MIC dynamic or symbolic transitions). In the second step, modified encoding constraints are generated exactly as in Problem #1. After solving encoding constraints in step #3, a *binary* hazard-free covering algorithm is used to find a hazard-free implementation.

4.2 Symbolic Logic Minimization Algorithm

Section 3 described a modified symbolic minimization algorithm, where an ON-set is described by a set containing both minterms and required cubes. Required cubes were used to specify transient states during state (column) transitions. Using a simple extension, this algorithm can be generalized to perform symbolic hazard-free minimization. Symbolic required cubes are not only generated for state (column) transitions, but also for input (row) transitions.

For every output or next-state function, a required cube is generated (i) for each vertical transition where the output is stable at 1, and (ii) for each SIC row transition, where the output is stable at 1 (*i.e.,* makes a $1 \rightarrow 1$ transition).

Example. In the given symbolic flow table in Figure refstate-tab-2, in row B, the output makes a $1 \rightarrow 1$ transition from input state 01 to 11, so the symbolic required cube `<-1 {B}>` is added to the output's ON-set. Similarly, the symbolic next-state value B, in positional cube notation, makes a $1 \rightarrow 1$ transition for the same SIC change; therefore, the same required cube is added to the ON-set corresponding to next-state B. □

As in Problem #1, the symbolic covering problem is solved using a standard unate covering algorithm.

4.3 Encoding Constraints

Dichotomy constraints are generated with respect to this new symbolic cover. The constraint generation algorithm is unchanged from the algorithm of Section 3. The resulting constraints insure that products can safely be instantiated with respect to both stable and transient points in the symbolic flow table.

4.4 Solving Constraints and Binary Logic Minimization

As before, since all constraints are in the form of dichotomies, they can be solved using a dichotomy solver, resulting in a state assignment. However, the final step of binary logic minimization must be altered. Since a *binary hazard-free* cover is desired, hazard-free minimization is required. The symbolic hazard-free algorithm in Section 4.2 can be used, but in binary form. Each symbolic required cube is instantiated as a binary required cube. The resulting unate covering problem is to cover all required cubes, and remaining minterms, using prime implicants. McCluskey [15] developed such an algorithm for SIC transitions. By allowing MIC required cubes, the same algorithm can be used. (Both algorithms are a special case of a general MIC hazard-free minimizer described in [21].)

5 Theoretical Results

We now sketch the basic theoretical results for each of the two algorithms. First, we propose a "canonical" assignment and binary instantiation of a symbolic cover, roughly analogous to the use of a 1-hot assignment in KISS. Second, we summarize results on cardinality and optimality of the binary cover. Finally, we summarize results on the soundness of the binary implementation.

5.1 Algorithm #1: Optimal Critical Race-Free Assignment

In [17], DeMicheli indicates that any symbolic minimized cover can be assigned a 1-hot encoding. The result is a $1 \rightarrow 1$ mapping of symbolic implicants to binary implicants, yielding a canonical cover whose cardinality is identical to that of the symbolic cover. For asynchronous machines, however, a 1-hot encoding is in general not critical race-free [29]. Therefore, such an assignment cannot be used. In general, no single encoding can be used for all asynchronous symbolic covers of N states. Therefore, we use a different approach: to obtain a "canonical" assignment for a given machine, the encoding constraints must first be solved. Any resulting assignment can then be used.

Binary Cover Instantiation

Given a minimal symbolic cover and a resulting *valid* assignment produced by Algorithm #1, symbolic products in the cover can be instantiated by substituting binary codes in the obvious way. For a symbolic implicant p (a 4-tuple $< input, present, next, output >$), a new binary implicant p' is generated, as follows:

$input(p')$	$\Leftarrow$	$input(p)$
$present(p')$	$\Leftarrow$	$supercube(encodings(stategroup(p)))$
$next(p')$	$\Leftarrow$	$encoding(next(p))$ if p implements next-state; 0's otherwise
$output(p')$	$\Leftarrow$	$output(p)$

For example, given the state assignment $S_0 = 000$, $S_1 = 011$, $S_2 = 100$, and $S_3 = 101$ the symbolic implicant `<011|1010 0010|100>` is mapped to the binary implicant `<011|-00 100|100>`.

Note that when instantiating a symbolic implicant which implements *only next-state*, an *empty* binary implicant may result. This occurs if the corresponding symbolic next-state is mapped to a binary code of all 0's. Such an implicant can be dropped from the binary cover. Therefore, the following property holds:

Property 1 (Opt-CRF). The cardinality of the original symbolic cover is an upper bound on the cardinality of the instantiated binary cover.

Correctness of Binary Cover

Lemma 5.1. Any state assignment satisfying the encoding constraints of Algorithm #1 is critical race-free.

Proof. We demonstrate that our encoding constraints subsume all Tracey critical race-free constraints [28]. In a given flow-table input column, I_1, 2 types of critical race-free constraints can arise, due to potential interference between:

- unstable vs. stable state transitions
- unstable vs. unstable state transitions

The first case requires a dichotomy such as $(S_0, S_1; S_2)$. We assume the unstable transition is from S_0 to S_1 (the same argument holds if the transition is from S_1 to S_0). Our symbolic cover is required to cover the entire vertical unstable state transition with some implicant, say P. P thus contains *at least* S_0 and S_1 in its state group, and implements next-state S_1. Meanwhile, S_2 is stable in I_1; hence our encoding constraints will include the constraint $(stategroup(P); S_2)$, which is identical to the Tracey constraint, or subsumes it, if the state group of P contains additional states.

The second case requires a dichotomy such as $[S_0, S_1; S_3, S_4]$. Again, the symbolic cover is required to cover the first unstable transition, $S_0 \rightarrow S_1$, with some implicant P. By an argument similar to the above, our encoding constraints will include $[stategroup(P); S_3, S_4]$, which is identical to or subsumes the Tracey constraint. □

We present, without proof, the following additional results.

Lemma 5.2. The binary products of B contain the ON-set of the encoded asynchronous machine.

Lemma 5.3. No binary product of B intersect the OFF-set of the encoded machine.

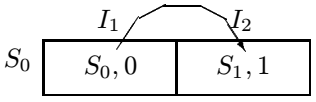
5.2 Algorithm #2: Optimal Hazard-Free/Critical Race-Free Assignment for Normal Fundamental Mode Machines

Binary Cover Instantiation

The above instantiation algorithm may allow hazards in the next-state logic in the final cover. Further, for hazard-free implementations, it turns out that a minimal symbolic cover instantiated with an optimal critical race-free code does not suffice either:

> Due to sharing of 1-bits in state codes, static-1 transitions for next-state may appear in the binary machine where dynamic transitions appeared in the symbolic machine.

This requires extra covering effort: dynamic transitions are hazard-free in 2-level AND/OR implementations for SIC operation, but static-1 transitions must each be completely covered by some implicant. Consider the following transition in a machine with 1 output:



In this example, there is no symbolic implicant covering the transition supercube, since both output and next-state undergo dynamic transitions. An instantiated cover therefore fails to properly implement certain static-1 transitions for next-state. For example, suppose $S_0 = 011$ and $S_1 = 110$, where the 3 state bits are labelled y_0, y_1 and y_2, respectively. In this case, state bit y_1 makes a $1 \rightarrow 1$ transition from $I1$ to $I2$. No cube covers this entire transition in the instantiated binary cover, hence y_1 has a static-1 hazard. The solution is to add terms to the instantiated cover, 1 for each such static-1 SIC transition.

Property 1 (Opt-HFCRF). Let $|S|$ be the cardinality of the symbolic cover, $|B|$ be the cardinality of the binary instantiated cover, and k is the number of SIC input changes in the flow table; then $|B| = \mathrm{O}(|S| + k)$.

Hazard-Freedom and Optimality of Binary Cover

We present without proof the following results.

Lemma 5.4. The binary instantiated cover B is hazard-free for every specified input and state transition.

Property 2 (Opt-HFCRF). The binary instantiated *output* cover is *exactly* minimal (if outputs are minimized separately from next-state).

This property indicates that our algorithm produces assignments and hazard-free realizations which are exactly optimal with respect to output logic (if outputs and next-state are minimized separately).

6 Program Implementation

We have implemented the symbolic minimization and state encoding algorithms in portable C++, running on NeXT, Sun Sparc, and IBM RS/6000. It is currently integrated into the unclocked asynchronous sequential machine synthesis system `UCLOCK` [18], and can easily be integrated into other synthesis systems, e.g. 3D [34], MEAT [4], or `LCLOCK` [20][4]. We make use of existing highly optimized tools for certain steps, e.g. prime implicant generation (*espresso-mv* [25]), exact unate covering (*mincov* [25]), and dichotomy solution (*dichot* [26]).

The synthesis flow is as follows:

First, the synthesis system (e.g. `UCLOCK`) reads a description of the state machine, and performs state minimization. Next, a Berkeley PLA `.kiss` file describing the machine is written. Multi-output prime implicants are generated via `espresso -Dprimes`. The symbolic minimizer (algorithm #1 or #2) then sets up a unate "cube-covering" problem, and obtains an exactly minimal cover using *mincov*. The constraint generation algorithm is invoked with the symbolic cover and a description of vertical state transitions. *dichot* is then invoked to obtain an exactly minimal solution to the generated constraints.[5] The symbolic machine is then instantiated with the resulting encoding, and passed through a logic minimizer (`espresso -Dexact` for Algorithm #1, and a binary hazard-free minimizer for Algorithm #2) to produce the final machine implementation.

7 Experimental Results

Two sets of experiments were run using industrial examples ([5], [1]); one using the optimal critical race-free encoding of Algorithm #1, the second using the optimal SIC hazard-free critical race-free Algorithm #2. For each set, a parallel set of runs using a pure critical race-free encoding was done as well, for comparison with the optimal algorithms.

In the non-hazard-free case, the binary implementation step was performed using `espresso -Dexact`. For the hazard-free runs, a binary version of the "cube-covering" multi-output minimization Algorithm #2 was used for both the optimal and base critical race-free cases.

As a result of multi-output logic minimization, the optimality of the output logic is blurred by term sharing with next-state variables. Hence, to more clearly see the improvement in output implementation, we also performed a pair of experimental runs in which outputs and next-state were separately implemented; that is, multi-output prime implicants were used, but prevented from implementing both output and next-state (labelled *os-disjoint* in the tables).

Results appear in Figures 4 and 5. Improvements ranging from 0% $\rightarrow$ 20% in outputs are observed for the *os-disjoint* case, with a tendency toward greater improvements as design size increases.

8 Conclusions

In this paper, we have formulated and solved two related asynchronous state assignment problems: (i) optimal critical race-free state assignment; and (ii) optimal hazard-free and critical race-free state assignment for normal fundamental mode machines. New symbolic minimization algorithms and encoding constraints have been presented to account for the requirement of critical race-free codes for both problems. Our solution to (i) is quasi-optimal, while our solution to (ii) is exactly optimal with respect to output logic. As in the synchronous case, both solutions are approximate with respect to next-state logic. Initial experimental results are quite promising.

[4] This system does not require critical race-free codes, so it would benefit only from the hazard-free nature of the solution.

[5] *dichot* normally accepts only "face embedding constraints" (all $N \rightarrow 1$), while we also generate $N \rightarrow 2$ constraints. We modified *dichot*'s front-end to accept the latter constraints.

DESIGN	# I/S/O	*os-shared* # cubes OptCRF	*os-shared* # cubes CRF	*os-disjoint* # output cubes OptCRF	*os-disjoint* # output cubes CRF	code length OptCRF	code length CRF
sbuf-read-ctl	3/3/3	6	6	4	4	3	2
sbuf-send-ctl	3/4/3	10	9	7	7	4	2
rf-control	6/6/5	12	11	7	8	6	3
it-control	5/4/7	11	11	9	9	4	2
sd-control	8/13/12	24	25	19	21	9	4

Figure 4. Optimal Critical Race-Free Results

DESIGN	# I/S/O	*os-shared* # cubes OptHFCRF	*os-shared* # cubes HFCRF	*os-disjoint* # output cubes OptHFCRF	*os-disjoint* # output cubes HFCRF	code length OptHFCRF	code length HFCRF
sbuf-read-ctl	3/3/3	9	8	5	5	3	2
sbuf-send-ctl	3/4/3	15	13	9	11	4	2
rf-control	6/6/5	16	12	7	8	6	3
it-control	5/4/7	16	15	12	12	5	2
sd-control	8/13/12	34	35	20	25	10	4

Figure 5. Optimal Hazard-Free Critical Race-Free Results

Acknowledgments

The authors would like to thank Prof. Giovanni De Micheli (Stanford) for suggesting we consider the problem of optimal hazard-free state assignment.

References

[1] B. Coates A. Marshall and P. Siegel. The design of an asynchronous communications chip. *Design and Test*, June 1994.

[2] T.-A. Chu. Synthesis of self-timed vlsi circuits from graph-theoretic specifications. Technical Report MIT-LCS-TR-393, Massachusetts Institute of Technology, 1987. Ph.D. Thesis.

[3] M.J. Ciesielski, J.J. Shen, and M. Davio. A unified approach to input-output encoding for fsm state assignment. In *DAC-91*, June 1991.

[4] A. Davis, B. Coates, and K. Stevens. Automatic synthesis of fast compact self-timed control circuits. In *1993 IFIP Working Conference on Asynchronous Design Methodologies (Manchester, England)*, 1993.

[5] Al Davis. The mayfly parallel processing system. Technical Report HPL-SAL-89-22, Hewlett-Packard Systems Architecture Laboratory, 1989.

[6] A.L. Davis, B. Coates, and K. Stevens. The post office experience: Designing a large asynchronous chip. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 409–418. IEEE Computer Society Press, January 1993.

[7] S. Devadas, H.-K. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. MUSTANG: State assignment of finite state machines targeting multi-level logic implementations. *IEEE Transactions on CAD*, CAD-7(12):1290–1300, December 1988.

[8] K. Keutzer, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis for testability techniques for asynchronous circuits. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, November 1991.

[9] D.S. Kung. Hazard-non-increasing gate-level optimization algorithms. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 631–634. IEEE Computer Society Press, November 1992.

[10] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, pages 302–308. Association for Computing Machinery, June 1991.

[11] B. Lin and S. Devadas. Synthesis of hazard-free multi-level logic under multiple-input changes from binary decision diagrams. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*, November 1994.

[12] B. Lin and A. R. Newton. Synthesis of multiple level logic from symbolic high-level description languages. In *IFIP Conference on VLSI*, pages 187–196, August 1989.

[13] B. Lin, C. Ykman-Couvreur, and P. Vanbekbergen. A general state graph transformation framework for asynchronous synthesis. In *Proceedings of the 1994 European Design Automation Conference*, September 1994.

[14] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Institute on Concurrent Programming, pages 1–64. Addison-Wesley, Reading, MA, 1990.

[15] E.J. McCluskey. *Introduction to the Theory of Switching Circuits*. McGraw-Hill, New York, NY, 1965.

[16] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level logic macros. *IEEE Transactions on CAD*, CAD-5(4):597–616, October 1986.

[17] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment for finite state machines. *IEEE Transactions on CAD*, CAD-4(3):269–285, July 1985.

[18] S.M. Nowick and B. Coates. UCLOCK: automated design of high-performance unclocked state machines. In *Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, October 1994.

[19] S.M. Nowick, M.E. Dean, D.L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 419–427. IEEE Computer Society Press, January 1993.

[20] S.M. Nowick and D.L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proceedings of the 1991 IEEE International Conference on Computer-Aided Design*, pages 318–321. IEEE Computer Society Press, November 1991.

[21] S.M. Nowick and D.L. Dill. Exact two-level minimization of hazard-free logic with multiple-input changes. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*, pages 626–630. IEEE Computer Society Press, November 1992.

[22] S.M. Nowick, N.K. Jha, and F. Cheng. Synthesis of asynchronous circuits for stuck-at and robust path delay fault testability. In *Proceedings of VLSI Design 95*, January 1995.

[23] S.M. Nowick, K.Y. Yun, and D.L. Dill. Practical asynchronous controller design. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 341–345. IEEE Computer Society Press, October 1992.

[24] Steven M. Nowick. Automatic synthesis of burst-mode asynchronous controllers. Technical report, Stanford University, 1993. Ph.D. Thesis.

[25] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued optimization for PLA optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(5):727–750, September 1987.

[26] A. Saldanha, T. Villa, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. A framework for satisfying input and output encoding constraints. In *DAC-91*, June 1991.

[27] P. Siegel, G. De Micheli, and D. Dill. Technology mapping for generalized fundamental-mode asynchronous designs. In *30th ACM/IEEE Design Automation Conference*, June 1993. To appear.

[28] J.H. Tracey. Internal state assignments for asynchronous sequential machines. *IEEE Transactions on Electronic Computers*, EC-15:551–560, August 1966.

[29] S.H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York, NY, 1969.

[30] Kees van Berkel. Handshake circuits: an intermediary between communicating processes and VLSI. Technical report, Eindhoven Institute of Technology, 1992. Ph.D. Thesis.

[31] T. Villa and A. Sangiovanni-Vincentelli. NOVA: State assignment of finite state machines for optimal two-level logic implementations. In *Design Automation Conference*, pages 327–332, June 1989.

[32] K.-H. Wang, W.-S. Wang, T.T. Hwang, A.C.H.Wu, and Y.-L. Lin. State assignment for power and area. In *Proceedings of the 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. IEEE Computer Society Press, October 1994.

[33] K.Y. Yun and D.L. Dill. Automatic synthesis of 3D asynchronous finite-state machines. In *Proceedings of the 1992 IEEE/ACM International Conference on Computer-Aided Design*. IEEE Computer Society Press, November 1992.

[34] K.Y. Yun, D.L. Dill, and S.M. Nowick. Synthesis of 3D asynchronous state machines. In *Proceedings of the 1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 346–350. IEEE Computer Society Press, October 1992.