MASSACHUSETTS INSTITUTE OF TECHNOLOGY VLSI PUBLICATIONS

VLSI Memo No. 89-509
March 1989

# Approaches to Multi-Level Sequential Logic Synthesis

Srinivas Devadas

Abstract

In this paper, we present approaches to multi-level sequential logic synthesis - algorithms and techniques for *the area and performance optimization of interconnected finite state machine descriptions.*

Interacting finite state machines are common in industrial chip designs. While optimization techniques for single finite state machines are relatively well developed, the problem of optimization across latch boundaries has received much less attention. Techniques to optimize pipelined combinational logic so as to improve area/throughput have been proposed. However, logic *cannot* be straightforwardly migrated across latch boundaries when the basic blocks are sequential rather than combinational circuits.

We present new techniques for the exploitation of *sequential don't cares* in arbitrary, interconnected sequential machine structures. Exploiting these don't care sequences can result in significant improvements in area and performance. We address the problem of migrating logic across state machine boundaries so as to make particular machines less complex at the possible expense of making others more complex. This can be useful from both an area and performance point of view. We present new optimization algorithms that *incrementally* modify state machine structures across latch boundaries. We discuss the use of more global state machine decomposition and factorization algorithms for area optimization. Finally, we present experimental results using these algorithms on sequential circuits.

Acknowledgements

To appear in *Proceedings, 26th Design Automation Conference*, Las Vegas, Nevada, June 26-29, 1989. This work was supported in part by the Defense Advanced Research Projects Agency under contract number N00014-87-K-0825.

Author Information

Devadas: Electrical Engineering and Computer Science, Room 36-848, MIT, Cambridge, MA 02139. (617) 253-0454.

# Approaches to Multi-Level Sequential Logic Synthesis

Srinivas Devadas

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology, Cambridge

## Abstract

In this paper, we present approaches to multi-level sequential logic synthesis – algorithms and techniques for *the area and performance optimization of interconnected finite state machine descriptions*.

Interacting finite state machines are common in industrial chip designs. While optimization techniques for single finite state machines are relatively well developed, the problem of optimization across latch boundaries has received much less attention. Techniques to optimize pipelined combinational logic so as to improve area/throughput have been proposed. However, logic *cannot* be straightforwardly migrated across latch boundaries when the basic blocks are sequential rather than combinational circuits.

We present new techniques for the exploitation of *sequential don't cares* in arbitrary, interconnected sequential machine structures. Exploiting these don't care sequences can result in significant improvements in area and performance. We address the problem of migrating logic across state machine boundaries so as to make particular machines less complex at the possible expense of making others more complex. This can be useful from both an area and performance point of view. We present new optimization algorithms that *incrementally* modify state machine structures across latch boundaries. We discuss the use of more global state machine decomposition and factorization algorithms for area optimization. Finally, we present experimental results using these algorithms on sequential circuits.

## 1 Introduction

Interacting finite state machines (FSMs) are common in chips being designed today. The advantages of a hierarchical, distributed-style specification and realization are many. While the terminal behavior of any set of interconnected sequential circuits can be modeled and/or realized by a lumped circuit, the former can be considerably more compact, as well as being easy to understand and manipulate.

The disadvantages of this form of specification from a CAD point of view are that sequential logic synthesis algorithms are generally restricted to operate on lumped circuits. State assignment algorithms (e.g. [1], [8], [3]), for instance, almost exclusively operate on single finite state machines. Given a set of interacting machines represented by State Transition Graphs, algorithms that encode the internal states of the machines, *taking into account their interactions*, do not exist to date. If indeed, the machines are encoded separately, disregarding their interconnectivity, a sub-optimal state assignment can result (and generally does).

Traditionally, the decomposition of an initial circuit specification into smaller, interacting sequential circuits has been performed by the logic designer. Once a decomposition has been performed, it is almost never changed and logic synthesis tools operate on separate logic blocks independently. Unfortunately, there are no guarantees regarding the quality of the initial decomposition, in terms of minimality of communication between the machines and/or complexities of the individual machines. There exist automatic techniques that can decompose lumped sequential circuits into smaller, interacting ones (e.g. [5]). These techniques are limited in the topology of interconnections that can be achieved and severely limited in their capabilities of handling circuits of large size. Flattening the initial, distributed specification can result in a *very* large lumped circuit.

Efficient and flexible algorithms for re-partitioning interacting sequential circuits for area and performance optimization have not been proposed in the past. Work has been done in re-partitioning pipelined combinational logic stages (e.g. [6]). There is no restriction on migrating logic across latch boundaries when the basic blocks are combinational, provided the latches are not observable – the functionality of the circuit is unchanged by moving say, one gate from before to after a latch. However, when sequential circuits are interconnected, as shown in Figure 1, one *cannot* arbitrarily move logic across pipeline latch boundaries (We refer to flip-flops that store state as state latches and flip-flops that store intermediate values as pipeline latches). The functionality and terminal behavior of the circuit will be changed, even though the latches are not observable.

One wishes to be able to migrate logic across pipeline latch boundaries for several reasons. The duration of the system clock has to be greater than the longest path between any two pipeline stages. If a machine, $A$, is significantly more complex than another machine $B$, the critical path/system clock may be unnecessarily long. The clock cycle could be shortened by making $A$ less complex at the possible expense of making $B$ more complex. In the best case, the complexities of both $A$ and $B$ would decrease.

Another very important issue is the specification and exploitation of don't cares in interconnected FSM descriptions. For example, in Figure 1, certain binary combinations may never appear at the set of latches $L1$. This will correspond to an incompletely specified machine $B$. These don't cares can be exploited using standard state minimization strategies [9]. A more complicated form of don't care, referred to here as a *se-*

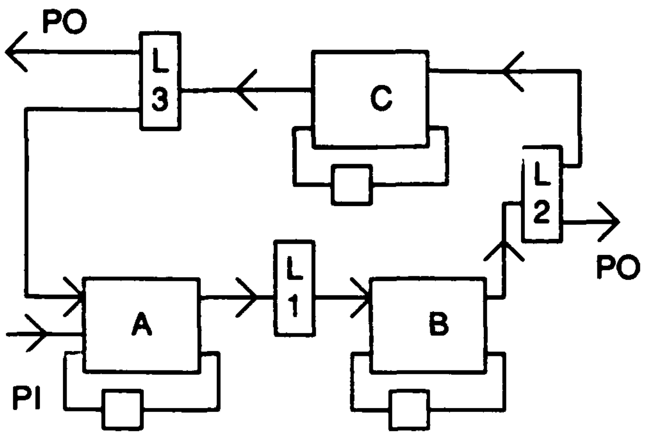

Figure 1: Interacting Finite State Machines

*quential don't care*, corresponds to an input sequence of vectors, say 1111, 1011, 1000 that does not appear at $L1$, though each of the separate vectors do appear. Sequential don't cares are more difficult to exploit. These don't cares are due to the limited controllability of $B$ and can be used to optimize $B$. There are also other don't cares related to the limited observability of $A$.

In this paper, we present new algorithms for the systematic *exploitation of sequential don't cares* resulting from the limited observability of a driving machine and the limited controllability of a driven machine. We show that exploiting either set of don't cares can significantly reduce the number of states and complexity of the driving and driven machines. A set of interacting machines can be *iteratively optimized* using these don't care sets.

We also present new techniques for the area and performance optimization of interacting machines, via the migration of logic across latch boundaries. If a machine $A$ drives machine $B$, our techniques can be used to reduce the number of states and complexity of $A$ at the possible expense of increasing the complexity of $B$ (the number of states in $B$ remains constant). Similarly, the number of states in $B$ can be reduced using complementary techniques. Re-encoding algorithms that minimize the areas of $A$ and $B$, by changing the encoding of the intermediate lines, have also been developed. These techniques are incremental, fast and have small memory requirements. They can be used to speed up the system clock and/or minimize area, in conjunction with the algorithms for don't care exploitation. We present experimental results on several examples that illustrate the efficacy of the proposed algorithms.

Basic definitions and notations used are given in Section 2. Different types of sequential don't cares are described in Section 3. Systematic methods for the exploitation of these don't cares are presented. Migration of logic across latch boundaries is the subject of Section 4. When a machine $A$ receives inputs from another machine $B$, modifications to the intermediate lines that carry information from $B$ to $A$ can change the complexities of $A$ and $B$. In Section 5, we present preliminary experimental results using these techniques on some examples.

## 2 Preliminaries

A **cube** in the Boolean n-space corresponding to a logic function is written as a bit vector on a set of variables with each bit position representing a distinct variable. The values taken by each bit can be 1, 0 or 2 (don't care), signifying the true form, negated form and non-existence respectively of the variable corresponding to that position. A **minterm** is a cube with only 0 and 1 entries. The **distance** between two minterms is defined as the number of bit positions they differ in. A cube $c_1$ is said to **cover** another cube $c_2$ (written as $c_1 \supseteq c_2$) if for each bit position, the entry in $c_1$ is equal to the entry in $c_2$ or is a 2.

A finite state machine, $M$, is represented by its **State Transition Graph** (STG), $G(V, E, W(E))$ where $V$ is the set of vertices corresponding to the set of states $S_M$, where $||S_M||$ is the cardinality of the set of states of the FSM, $E$ the set of transition edges in $M$ and $W(E)$ are the Boolean expressions corresponding the input and output combinations for $E$. The number of inputs and outputs are denoted $N_I$ and $N_O$ respectively. The input combination and present state corresponding to an edge are denoted $(i,\ s) \in E$, where $i$ and $s$ are cubes. The fanin and output of $(i,\ s)$ are denoted $fanin(i,\ s) \in V$ and $output(i,\ s)$ respectively. The complete set of fanin and fanout edges of a state $s$ are denoted $fanin(s)$ and $fanout(s)$. The fanin state, fanout state and output of an edge $e_1$ are denoted $e_1 -> fanin$, $e_1 -> fanout$ and $e_1 -> output$ respectively. The set of fanin (fanout) edges of a state, $q$, is denoted $E_{FI}(q)$ ($E_{FO}(q)$).

A **starting or initial state** is assumed to exist for a machine, $M$, also called the **reset state** and denoted $R_M$. A **distinguishing sequence** for a pair of states $q_1,\ q_2 \in S_M$ is a sequence of input vectors such that the last vector produces different outputs when the sequence is applied to $M$, when $M$ is initially in $q_1$ or when $M$ is initially in $q_2$. Two states $q_1,\ q_2$ in a machine $M$ are **equivalent** (written as $q_1 \equiv q_2$), if they do not possess a distinguishing sequence.

A **differentiating sequence** for a pair of states $q_1,\ q_2 \in S_M$ is a sequence of input vectors such that some vector (or vectors) in the sequence produces different outputs when the sequence is applied to $M$ initially in $q_1$ or initially in $q_2$ and at the end of the sequence $M$ reaches the same final state. The pair of edges corresponding to each input vector in a distinguishing or differentiating sequence are called **co-edges**.

A sequence of vectors $VS_1$ is said to **contain** another sequence $VS_2$ (written as $VS_1 \supseteq VS_2$), if $VS_2$ appears in $VS_1$.

A **cascade** of two machines $A$ and $B$ is denoted $A \longrightarrow B$. $A$ is the **driving** machine and $B$ the **driven** machine.

## 3 Sequential Don't Cares

In Figure 1, we have a machine $A$ driving another machine $B$ via a set of latches $L1$ (We neglect $C$ for the moment). For the purposes of the discussion here, we assume that all the latches in $L1$ are not observable. In practice, a subset of the latches may be observable. However, the don't care exploitation techniques described here are easily modified to the general case.

We assume that a State Transition Graph description exists for both machines $A$ and $B$. Let the number of intermediate/pipeline latches in $L1$ be $N$. $A$ may or may not assert all $2^N$ possible output combinations. If a certain binary combination, $c_1$ never appears at $L1$, then $B$ will be incompletely specified – the transition edges corresponding to an input of $c_1$ need not be specified, whatever state $B$ is in (We don't care what happens when $B$ receives the input $c_1$). The more general case

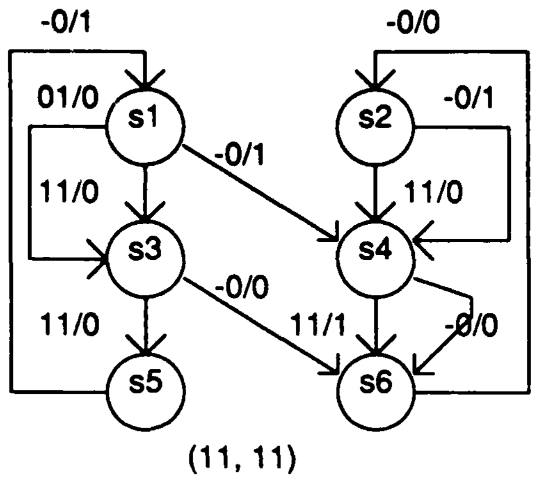

Figure 2: Sequential Don't Cares

is when a certain combination $c_2$ never appears at $L1$, when $B$ is in some set of states $Q_B \in S_B$ ($S_B$ is the set of all states $B$ can be in). It does appear when $B$ is in states other than $Q_B$. In this case, the states in $Q_B$ will have $c_2$ unspecified (If an edge on $c_2$ exists in $Q_B$, it can be removed). This type of don't care can be easily exploited via the use of standard state minimization algorithms that handle incompletely specified machines [9].

A more complicated sequential don't care is associated with vector *sequences* that never appear at $L1$, though all $2^N$ separate vectors appear. $A$ does not produce all possible output sequences. This type of don't care does *not* have a straightforward interpretation. Edges in the State Transition Graph of $B$ cannot be removed or left unspecified. In Figure 2, a State Transition Graph corresponding to a possible $B$ machine is shown. The machine is state minimal. We assume that each transition edge in $B$ is irredundant, i.e. $B$ makes every transition with appropriate input sequences. A don't care input sequence is shown below the Graph. Such a don't care sequence implies that certain *sequences of transitions* will not be made by $B$.

A don't care input sequence is assumed to have a length greater than 1. Given a don't care sequence $DC$, all sequences $SE$ such that $SE \supseteq DC$ are also don't care sequences. We define an **atomic don't care sequence** as one that does not contain any other don't care sequence. Thus, any subsequence of an atomic don't care sequence is a care sequence. In the sequel, we consider only atomic don't care sequences.

Given a set of sequences that a driving machine never asserts, our problem lies in exploiting this form of don't care, so as to optimize $B$. In the general case, we will have a set of don't care sequences. We can state the following lemma.

**Lemma 3.1** : *Given a machine $B$ and a set of don't care sequences $DC_j$ , $1 \leq j \leq N_A$, if two states in $B$, $s1$ and $s2$ have distinguishing sequences $I_i$ , $1 \leq i \leq N_D$ such that for each $k$, $I_k \supseteq DC_l$ for some $l$, then $s1$ and $s2$ are equivalent in $B$ under the $DC_j$.*

**Proof**: Since the $DC_j$ can never occur, it means the $I_i$ can never occur. Therefore, $s1$ and $s2$ in $B$ are equivalent under $DC_j$. **Q.E.D.**

An approach to exploit don't cares based on Lemma 3.1 would entail producing all distinguishing sequences for every pair of states in $B$ and checking for the containment condition. Pairs satisfying the condition can be merged. This is potentially very time consuming; a pair of states may have many distinguishing sequences and we have to find them for every possible pair. A more efficient approach is now outlined.

In this approach, given a set of don't care sequences, $B$ is transformed into a new machine $B'$ which has a greater number of states, but is more incompletely specified than $B$. $B'$ is state minimized to obtain $B''$ ($||S_{B''}|| \leq ||S_B||$). The pseudo-code below illustrates the procedure.

```
exploit-input-dc( $B$, $DC$ ):
{
  $B' = B$ ;
  foreach ( don't care sequence $DC_i$ ) {
    foreach (depth-first path $P = e_1, .. e_K \in B'$ ) {
      if ( $P \supseteq DC_i$ ) {
        for( $i = 2$; $i \leq K$; $i = i + 1$ ) {
          $s_i = e_i -> fanout$ ;
          make states $s'_i$ and $s''_i$ ;
          $fanin(s'_i) = e_{i-1}$ ;
          $fanin(s''_i) = fanin(s_i) - e_{i-1}$ ;
          if ( $fanin(s''_i) = \phi$ ) delete $s''_i$ ;
          if ( $i < K$ )
            $fanout(s'_i) = fanout(s''_i)$
              $= fanout(s_i)$ ;
          else {
            $fanout(s'_i) = fanout(s_i) - e_{i-1}$ ;
            $fanout(s''_i) = fanout(s_i)$ ;
          }
          delete $s_i$ ;
        }
      }
    }
  }
  $B''$ = state-minimize ( $B'$ ) ;
}
```

The procedure is effectively producing a machine where the don't care sequences are *not* specified, but otherwise has the same functionality as the original machine. This means that if any two states in $B$ satisfy the conditions of Lemma 3.1, these two states will not possess a distinguishing sequence in $B'$ and will thus be *compatible* during state minimization. A smaller machine $B''$ will be obtained after state minimization. When $i = p < K$ in the *for* loop above, the fanout of $s_p$ is duplicated for the states $s'_p$ and $s''_p$ – the edge $e_p$ is also duplicated. Hence, at the next iteration, one of the $e_p$ fans into $s'_{p+1}$ and the other $e_p$ (as well as the remaining fanout edges from $s'_p$ and $s''_p$) into $s''_{p+1}$.

An illustrative example is given in Figures 2, 3 and 4. The machine and the don't care sequence of Figure 2 produce an expanded machine, shown in Figure 3. State minimizing this machine produces the result of Figure 4, which has one less state than the original machine of Figure 2. States $s3'$, $s4'$ and $s4''$ merge and so do $s1$ and $s2$.

The sequential don't cares discussed thus far are a product of the constrained controllability of the driven machine $B$ in a cascade $A \longrightarrow B$. There is another type of don't care due to the constrained observability of the driving machine $A$. We focus on the individually

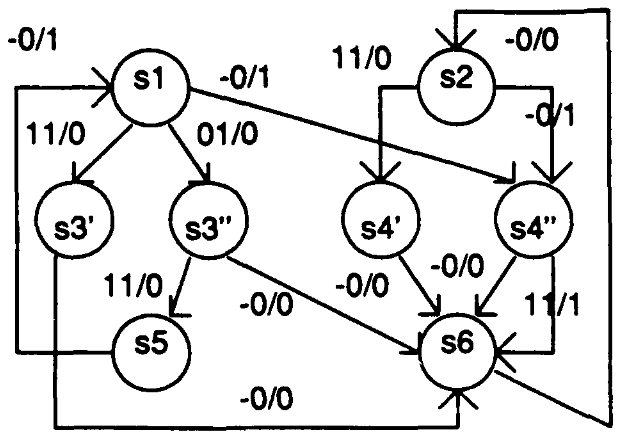

Figure 3: Expanding the Original Machine

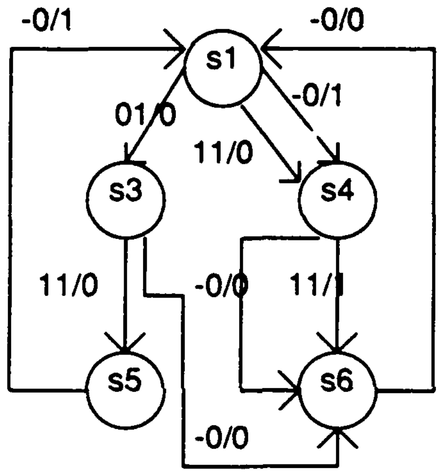

Figure 4: Machine after State Minimization

state minimized tables of Figure 5. The intermediate inputs/outputs have been given symbolic codes. Given that $A$ feeds into $B$, it is quite possible that for some transition edge $e_1 \in A$, it does not matter if the output asserted by this particular transition edge is, say, $INTi$ or $INTj$. In fact, in Figure 5, the 3rd transition edge can be either $INT1$ or $INT2$, ***without changing the terminal behavior of*** $A \longrightarrow B$ (We assume that there are no latches between $A$ and $B$, the starting state of $A$ is *sa1* and the starting state of $B$ is *qb1*). This is a don't care condition on $A$'s outputs. It is quite possible that making use of these don't cares can reduce the number of states in $A$. In fact, if one replaced the output of the 3rd edge in $A$ (Figure 5) by $INT1$ instead of $INT2$, we would obtain one less state after state minimization (*sa2* becomes equivalent to *sa3*).

Given a cascade $A \longrightarrow B$, we give below a systematic procedure to detect this type of don't care, i.e. expand the output of each transition edge of $A$ to the set of all possible values that it can take while maintaining the terminal behavior of $A \longrightarrow B$. Standard state minimization procedures can exploit don't care outputs, represented as cubes. However, state minimization procedures have to be modified in order to exploit transition

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| i1 | sa1 | sa2 | INT1 | INT1 | qb1 | qb2 | out1 |
| i2 | sa1 | sa3 | INT2 | INT2 | qb1 | qb3 | out2 |
| i1 | sa2 | sa1 | INT2 | INT1 | qb2 | qb2 | out3 |
| i2 | sa2 | sa3 | INT1 | INT2 | qb2 | qb2 | out3 |
| i1 | sa3 | sa1 | INT1 | INT1 | qb3 | qb2 | out4 |
| i2 | sa3 | sa2 | INT1 | INT2 | qb3 | qb3 | out1 |

$$A \longrightarrow B$$

Figure 5: Output Expansion

edge outputs represented as arbitrary Boolean expressions (multiple cubes).

```
output-expansion( $A$, $B$ ):
{
  foreach ( edge $e_1 \in A$ ) {
    $OUT(e_1)$ = universe ;
    foreach ( state $q_1 \in S_B$ ) {
      if ( $B$ can be in $q_1$ as $A$ asserts $e_1$ ) {
        find largest set of output combinations
        $c_1 \ni c_1 \supseteq e_1 -> output$ && $fanin(c_1, q_1)$,
          $output(c_1, q_1)$ are unique ;
        $OUT(e_1) = OUT(e_1) \bigcap c_1$ ;
      }
    }
    $e_1 -> output = OUT(e_1)$ ;
  }
}
```

A transition edge $e_1$ in $A$ is picked. The set of states that $B$ can be in when $A$ makes this transition is found. Given this set of states, the largest cube (or set of output combinations) that covers the output of the edge and produces a unique next state and a unique output when $B$ is in *any* one of the possible states is found (corresponds to $OUT(e_1)$). The output of $e_1$ is expanded to the cube. The process is repeated for all edges in $A$.

The state minimization procedure proposed in [9] can be used for incompletely specified finite state machines. However, after output expansion, we may have a multiple-output FSM in which a transition edge has an output that can belong to a subset of symbolic or binary values, rather than the universe of possible values (as in the incompletely specified case). In the case of multiple cubes or Boolean expressions specifying the output combinations for fanout edges, an additional check has to be performed during state minimization during the selection of the compatibility pairs to see if three or more sets of states can, in fact, be merged, preserving functionality. This is because the pairwise intersection of the Boolean expressions corresponding to the fanout edges of these states may each be non-null, resulting in a compatibility relation between each pair of states, but the three-way intersection may be null, implying that the three states cannot be merged.

When we have a set of interconnected machines as in Figure 1, the don't cares corresponding to each cascade can be ***iteratively*** used. For instance, in Figure 1, $A$ drives $B$. The outputs of $A$'s edges can be expanded first. $A$'s output don't care sequences can be used to optimize $B$. Next, one can focus on $B \longrightarrow C$. Output expansion can be performed on $B$ and so on.

## 4 Optimization Across Latch Boundaries

### 4.1 Introduction

A set of interacting machines can be optimized using their associated don't cares as described in the previous section. If the initial decomposition is not an intelligent one, there will be a large set of don't cares associated with each pair of driving and driven machines. While exploiting don't cares has the effect of removing redundancy, the overall decomposition of logic functionality between the various circuits remains the same. As mentioned earlier, there are several attractions in being able to migrate logic from one machine to another. In this section, we will present incremental techniques that optimize cascaded pairs of machines via logic migration. These techniques are iteratively applied in the general case of interacting machines (like in Figure 1).

### 4.2 Re-encoding

Consider the cascaded pair of Figure 5. The intermediate line values have been represented by symbolic codes. The complexities of the machines are affected by the encoding of these lines. A good *output encoding* for $A$ will produce minimal complexity. However, a good output encoding for $A$ may not be a good *input encoding* for $B$ and vice versa. Thus, tradeoffs exist.

We propose *re-encoding* as a means of migrating logic between the two machines by exploring these tradeoffs. If the initial specification of the intermediate lines is binary (rather than symbolic), the specification is converted into a symbolic representation. For instance, one might view the machines of Figure 5 as being derived from a logic implementation where $INT1$ had a code 100, $INT2$ had a code 010 and so on. We can re-encode these lines in different ways to tune the complexities of $A$ and $B$. Re-encoding can be performed before or after state assignment.

If one wished to reduce $B$'s complexity, the intermediate symbolic implicants would be assigned binary values corresponding to an optimal input encoding. Strategies for optimal input encoding have been proposed [8]. Heuristics for output encoding to reduce $A$'s complexity, as in [7], can also be used.

It has been determined experimentally that re-encoding affects the relative complexities of the machines by as much as 25%. However, the number of states in the machines is unchanged.

### 4.3 Optimizing the Driven Machine

Consider again the cascaded pair of Figure 5. The symbolic implicants $INTi$ constitute the means of information flow from $A$ to $B$. It is conceivable that for some pair of states $(q_1, q_2) \in B$, a particular input vector $INTx$ is required as the first vector in each distinguishing sequence for the pair (For instance, $INT1$ is required to distinguish $qb1$ and $qb3$ in Figure 5). If one were to modify $A$ so as to produce $INTx' \neq INTx$ when $B$ is in $q_1$ and $INTx$ otherwise, the distinguishing sequences are invalidated. $q_1$ becomes equivalent to $q_2$ and $B$ can be reduced. This is the basic process behind the technique described in this section.

The algorithm identifies symbolic implicants which when split up result in state reductions in $B$. The number of states in $A$ remains constant. The complexity of $A$ may increase, since $A$ now asserts a larger number of distinct symbolic outputs. Even if a particular symbolic implicant appears in front of every distinguishing sequence of a pair of states in $B$, it is not always possible to reduce the number of states in $B$. The following theorem is a statement of the required conditions.

**Theorem 4.1** : *Given a cascade $A \longrightarrow B$, let the distinguishing sequences for a pair of states $q_1, q_2 \in Q_B$ be $DS_1, DS_2, .. DS_M$. Let the distinct first vectors in the $DS_i$ be $o_1, o_2, .. o_N$. When $B$ is in $q_1$ ($q_2$), let the possible transition edges that $A$ has just made be $E_{(A,\ B=q_1)}$ ($E_{(A,\ B=q_2)}$). $E_{j1} \in E_{(A,\ B=q_1)}$ and $E_{j2} \in E_{(A,\ B=q_2)}$ are the sets of edges that assert $o_j$, $\forall\ j$. If $E_{j1} \cap E_{j2} = \phi$, $1 \leq j \leq N$, then $q_1$ and $q_2$ can be merged in $B$.*

**Proof**: We make the outputs of $E_{j1}$ $o'_j \neq o_j$ (and distinct from all other symbolic implicants). This means that when $B$ is in $q_1$ it will never receive $o_j$, $1 \leq j \leq N$. Similarly, when $B$ is in $q_2$ it never receives $o'_j$, $1 \leq j \leq N$. The first vector in each distinguishing sequence $DS_i$, $1 \leq i \leq M$, is invalidated. Therefore, $q_1 \equiv q_2$. **Q.E.D.**

For states other than $q_1, q_2 \in Q_B$, $o'_j$ is made to produce the same next state and outputs as $o_j$, $\forall\ j$. $N = 1$ is the simplest case of state reduction.

This technique is essentially splitting the symbolic outputs of machine $A$ and *introducing new don't care sequences* to $B$. These don't care sequences are then used to reduce the complexity of $B$. The above theorem has a straightforward practical interpretation. For optimization purposes, we focus on symbolic implicants that appear most frequently as first vectors in distinguishing sequences for different pairs of states. The edge disjointness condition of Theorem 4.1 is checked for and the implicants split if the condition is satisfied, so as to reduce the number of states in $B$.

### 4.4 Optimizing the Driving Machine

A technique complementary to the technique described in the previous section can be used to decrease the complexity of the driving machine, $A$. Here, states in the driven machine $B$ are split. Splitting these states in $B$ results in *new degrees of freedom in expanding the outputs* of the edges in $A$. Output expansion results in reducing the number of states and the complexity of the driving machine $A$.

Again, the symbolic output implicants of $A$ cannot be arbitrarily merged, since one has to maintain the terminal behavior of $A \longrightarrow B$. The following theorem is a statement of the conditions required for implicant merging to be possible.

**Theorem 4.2** : *Given a cascade $A \longrightarrow B$, let a transition edge $e \in A$ assert the symbolic output $o_p$. When $A$ makes the transition $e$, let the possible states $B$ can be in be $Q_{(B,\ A|e)}$.*
$\forall\ q \in Q_{(B,\ A|e)}\ \exists\ o_q{}^e \mid ( fanin(o_q{}^e, q) = fanin(o_p, q)\ \&\&\ output(o_q{}^e, q) = output(o_r, q) )$
$\|\ ( E_{FI}(q) \cap E_{(B,\ A|E_q{}^e)} ) \bigcap ( E_{FI}(q) \cap E_{(B,\ A|e)} ) = \phi$,
*then $e->$ output can be expanded to $(o_q{}^e, o_r)$. $E_q{}^e$*

*is the set of transition edges asserting output $o_q{}^e$, $E_{(B,\ A|E_q{}^e)}$ $(E_{(B,\ A|e)})$ is the set of transitions $B$ can make when $A$ is making the transitions $E_q{}^e$ $(e)$.*

**Proof**: We split each $q \in Q_{(B,\ A|e)}$ for which an $o_q{}^e$ cannot be found such that ( $fanin(o_q{}^e,\ q) = fanin(o_p,\ q)$ && $output(o_q{}^e,\ q) = output(o_p,\ q)$ ), into two states $q'$ and $q''$, initially duplicating the fanout of $q$. $q'$ receives as fanin $E_{FI}(q)$ − ( $E_{FI}(q) \cap E_{(B,\ A|e)}$ ) and $q''$ receives ( $E_{FI}(q) \cap E_{(B,\ A|e)}$ ). When $B$ is in $q''$, it never receives $o_q{}^e$ from $A$. Those fanout edges from $q''$ can be deleted. This means that the condition for expanding the outputs of edge $e$ to $(o_q{}^e,\ o_p)$ is satisfied (Section 3). **Q.E.D.**

Splitting states in $B$ has the effect of introducing new output don't cares for $A$, which can reduce the complexity of $A$. If for each differentiating sequence for a pair of states $q_1,\ q_2 \in S_A$, each pair of co-edges, $e_1,\ e_2$ that assert different outputs are expanded so $e_1 -> output \supseteq e_2 -> output$ or $e_2 -> output \supseteq e_1 -> output$, then $q_1$ and $q_2$ can be merged in $A$.

The strategy used in optimization is to remove a particular symbolic output in $A$ by operating on *all* edges that assert this particular output.

1. The STG of $A$ is analyzed to find which of the symbolic outputs appear (as last vectors) in most differentiating sequences. One particular symbolic output is picked, namely $o_p$.

2. All the transitions in $A$, $E_p$, that assert $o_p$ are found. For each $e \in E_p$, the set of states $B$ can be in, after $A$ makes transition $e$, $Q_{(B,\ A|e)}$, is found.

3. The fanouts of states in $Q_{(B,\ A|e)}$ are analyzed to pick a symbolic output $o_q{}^e \neq o_p$ that produces the same next states and outputs as $o_p$, in a maximum number of $Q_{(B,\ A|e)}$.

4. For each $q \in Q_{(B,\ A|e)}$ for which $o_q{}^e$ and $o_p$ produce different next states or outputs, the fanin of $q$ is checked for a possible split as per Theorem 4.2. If so, go to Step 2 and pick a new edge $e$. Else, go to Step 4 and pick a new $o_q{}^e$. If all possible $o_q{}^e$ have been exhausted, go to Step 1 and pick a new $o_p$.

5. Split states in $B$ corresponding to the selected $o_p$ and $o_q{}^e$, $\forall\ e \in E_p$.

While this algorithm does not guarantee reduction in the number of states in $A$, it guarantees reduction in $A$'s complexity on completion, since $A$ now asserts a fewer number of symbolic outputs. Generally, a reduction in states is also obtained.

### 4.5 Partial Collapsing

When the driven machine $B$ in a cascade $A \longrightarrow B$ has multiple outputs, its complexity can be reduced by *collapsing* or *flattening* one or more outputs into $A$.

An output of $B$ is selected and two separate machines $B'$ and $B''$ operating in parallel are constructed, with $B'$ producing the single selected output and $B''$ the remaining. The STG of $B$ can be initially duplicated for $B'$ and $B''$ and then re-minimized after removing the appropriate outputs. Both $B''$ and $B'$ will be less complex than $B$. $B'$ can then be collapsed into $A$; a new machine corresponding to the *direct product* of $A$ and $B'$ will be obtained, that drives $B''$. If latches exist initially, between $A$ and $B$ then flattening is more complicated, since a latch itself represents a two-state finite state machine. The product of $A$, the latches and $B'$ has to be constructed.

Partial collapsing will result in a reduction of complexity in the driven machine in a cascade, but can significantly increase the complexity of the driving machine. It has limited uses, but is applicable in cases where the driven machine is significantly more complex than the driving machine.

The flattened machine can be *re-decomposed* in a cascade using the classical decomposition algorithms of [5]. General decomposition algorithms have been recently proposed [4], that produce two interacting submachines $A \longleftrightarrow B$ from the original description, attempting to minimize the complexities of the submachines. These algorithms are more powerful than those in [5], since the interaction between the submachines is two-way rather than uni-directional. Using these decomposition algorithms allows more global optimization at the expense of loss of control over the optimization and the ability to handle large circuits.

| Ex | pi | po | mac | int | states | | | lit |
|---|---|---|---|---|---|---|---|---|
| | | | | | M1 | M2 | M3 | |
| ex1 | 7 | 19 | 2 | 14 | 20 | 48 | - | 704 |
| ex2 | 11 | 9 | 2 | 8 | 10 | 16 | - | 338 |
| ex3 | 7 | 2 | 2 | 4 | 16 | 16 | - | 186 |
| ex4 | 8 | 11 | 3 | 13 | 20 | 32 | 19 | 926 |
| ex5 | 8 | 21 | 3 | 16 | 20 | 16 | 48 | 772 |

Table 1: Statistics of Examples

## 5 Results

We have run several examples to evaluate the optimization strategies and algorithms described in Sections 3 and 4. In Table 1, the statistics of the sequential circuits we experimented with are given. All these circuits were obtained by interconnecting the finite state machines of the MCNC 1987 Logic Synthesis Workshop benchmark set. In Table 1, the number of primary inputs (pi) and primary outputs (po), the number of separate machines (mac) and the number of states in each machine in the circuit (states) are indicated for each example. The number of intermediate, non-observable/non-controllable lines (int) and the total number of literals (lit) after state assignment using MUSTANG [3] and multi-level combinational optimization using MIS [2] are also given.

We first give results from using the don't care exploitation algorithms, in Table 2. For each circuit, the number of states in each of the machines and the total literal count is given, as well as the CPU time required for optimization on a VAX 11/8650 ($m$ stands for minutes). Significant reductions in circuit complexity have been obtained.

We used the re-encoding algorithms of Section 4.2 on the cascaded pairs. In Table 3, the literal counts

| Ex | mac | states | | | lit | CPU time |
|---|---|---|---|---|---|---|
| | | M1 | M2 | M3 | | |
| ex1 | 2 | 17 | 44 | - | 641 | 8.1m |
| ex2 | 2 | 9 | 13 | - | 241 | 4.0m |
| ex3 | 2 | 11 | 15 | - | 151 | 2.1m |
| ex4 | 3 | 19 | 13 | 17 | 781 | 11.1m |
| ex5 | 3 | 17 | 10 | 48 | 642 | 8.8m |

Table 2: Results Via Don't Care Exploitation

| Ex | ORIGINAL | | INPUT | | OUTPUT | |
|---|---|---|---|---|---|---|
| | M1 lit | M2 lit | M1 lit | M2 lit | M1 lit | M2 lit |
| ex1 | 141 | 563 | 181 | 492 | 112 | 601 |
| ex2 | 220 | 118 | 251 | 101 | 206 | 151 |
| ex3 | 118 | 68 | 149 | 51 | 99 | 87 |

Table 3: Results using Re-encoding

| Ex | M1 | | M2 | | M3 | | CPU time |
|---|---|---|---|---|---|---|---|
| | sta | lit | sta | lit | sta | lit | |
| ex1 | 17 | 146 | 36 | 391 | - | - | 2.2m |
| ex2 | 9 | 194 | 12 | 68 | - | - | 1.7m |
| ex3 | 11 | 126 | 11 | 47 | - | - | 2.6m |
| ex4 | 19 | 142 | 11 | 351 | 16 | 198 | 4.1m |
| ex5 | 16 | 134 | 10 | 78 | 40 | 362 | 4.2m |

Table 4: Optimizing the Driven Machine

for each of the two machines in the circuit originally (orig.) and the extreme cases of re-encoding (input and output) are given. As before, the literal counts are after state assignment and logic optimization. As can be seen, re-encoding affects the complexity of the individual machines by as much as 25%. Cascade **ex1**, for instance, would be best implemented using an input encoded driven machine so as to make the complexities of the driven and driving machines comparable.

Finally, we present results using the logic migration algorithms of Section 4.3 and 4.4. The states in the individual machines of a sequential circuit can be reduced or increased using these algorithms. In Tables 4 and 5, the number of states in the optimized machines and the new literal counts are given using the strategies of Section 4.3 and 4.4, respectively. As with re-encoding, solutions in between these extremes can be obtained – however, the numbers of Tables 4 and 5 illustrate the range in capabilities of the proposed algorithms.

## 6 Conclusions

We presented algorithms and techniques for the area and performance optimization of interconnected finite state machine structures. These algorithms include don't care exploitation techniques as well as logic migration techniques across latch boundaries in interacting sequential structures. The results we have obtained using these algorithms thus far are encouraging.

| Ex | M1 | | M2 | | M3 | | CPU time |
|---|---|---|---|---|---|---|---|
| | sta | lit | sta | lit | sta | lit | |
| ex1 | 14 | 91 | 50 | 561 | - | - | 3.6m |
| ex2 | 7 | 136 | 17 | 102 | - | - | 2.8m |
| ex3 | 8 | 72 | 21 | 73 | - | - | 4.1m |
| ex4 | 17 | 109 | 13 | 423 | 18 | 216 | 3.8m |
| ex5 | 14 | 112 | 16 | 47 | 54 | 463 | 7.1m |

Table 5: Optimizing the Driving Machine

## 7 Acknowledgements

The interesting discussions with Tony Ma and Richard Newton on sequential circuit optimization are acknowledged. Pranav Ashar and Robert Brayton offered constructive criticisms on an early draft of this manuscript. This research was supported in part by the Defense Advanced Research Projects Agency under contract N00014-87-K-0825.

## References

[1] D. B. Armstrong. A programmed algorithm for assigning internal codes to sequential machines. In *IRE Transactions on Electron Computers*, pages 466–472, August 1962.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Mis: a multiple-level logic optimization system. In *IEEE Transactions on CAD*, pages 1062–1081, November 1987.

[3] S. Devadas, H-K. T. Ma, A. R. Newton, and A. Sangiovanni-Vincentelli. Mustang: state assignment of finite state machines targeting multi-level logic implementations. In *IEEE Transactions on CAD*, pages 1290–1300, December 1988.

[4] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite state machines. In *Int'l Conference on Computer-Aided Design*, November 1988.

[5] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice-Hall, Englewood Cliffs, N. J., 1966.

[6] C. E. Leiserson, F. M. Rose, and J. B. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. of Third CalTech Conference on VLSI*, March 1983.

[7] G. De Micheli. Symbolic design of combinational and sequential logic circuits implemented by two-level macros. In *IEEE Transactions on CAD*, pages 597–616, September 1986.

[8] G. De Micheli, R. K. Brayton, and A. Sangiovanni-Vincentelli. Optimal state assignment of finite state machines. In *IEEE Transactions on CAD*, pages 269–285, July 1985.

[9] M. C. Paul and S. H. Unger. Minimizing the number of states in incompletely specified sequential circuits. In *IRE Transactions on Electronic Computers*, pages 356–357, September 1959.